CSC-223
Lab 4
Groups of 2 or 3, ideally 3.

JSON Parsing of Geometry Figures

As with all more complex software, we need a sub-system that can handle reading and processing of input. Our project will use the JavaScript Object Notation (JSON) as our input language format for encoding our geometry figures. After completing this assignment, we hope you will agree that the JSON language is simple and powerful.

Background: Encoding Geometry Figures in JSON

To work with more interesting geometry figures, we need to implement a robust input system. Knowing that we need an input system requires that we carefully consider the format of our input data and the API (or external libraries) to easily interpret the input so that it is useful downstream in our larger system. We will use the `org.json` library for JSON processing; with your implementations, you will need to reference the API found here.

The best way to understand how we will encode and decode geometry figures in JSON is by example. Consider Figure 1 as a whole figure, but also consider some of the components: triangles, segments, and points. We can easily view how a figure has many constituent segments that are constructed with points. We will use this idea when encoding a geometry figure in JSON.

An example of corresponding JSON representation of the geometry figure in Figure 1 is given in Figure 2. Observe that there is an intersection point in Figure 1 that is not explicitly named nor is it defined in the JSON representation; we will compute and name this `implicitly` defined point in subsequent assignments.

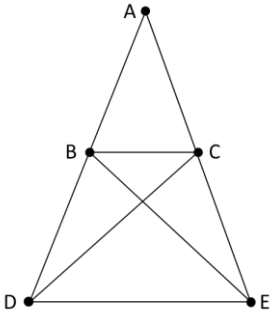

Figure 1: A sample geometry figure.

```
{ "Figure" :
  { "Description" : "Crossing symmetric triangle construction.",
    "Points" : [
                  { "name" : "D", "x" : 0, "y" : 0 },
                  { "name" : "E", "x" : 6, "y" : 0 },
                  { "name" : "B", "x" : 2, "y" : 4 },
                  { "name" : "C", "x" : 4, "y" : 4 },
                  { "name" : "A", "x" : 3, "y" : 6 }
                ],
    "Segments" : [
                    {"A" : ["B", "C"] },
                    {"B" : ["C", "D", "E"]},
                    {"C" : ["D", "E"] },
                    {"D" : ["E"]}
                  ]
  }
}
```

Figure 2: A JSON encoding of the geometry figure in Figure 1.

JSON is built on two structural ideas: (1) `name / value` pairs (commonly thought of as an *object*) and (2) an ordered lists of values (in an *array* structure).

An *object* is an unordered set of `name / value` pairs and begins with `{` and ends with `}`. At the top-level in Figure 2, we observe a *figure object*. We also observe that each figure object is defined by:

1. A string-based description (`Description: String`),

2. A list of points (`Points: List` of named points in the cartesian plane), and

3. A list of segments (`Segments: List` of adjacency lists).

FigureNode. Similar to other languages like Java or English, a language grammar formally defines the rules for expected structures defined using the language. With our geometry fact computer, we are interested in encoding geometry figures as a set of points and segments. JSON is a language and thus facilitates encoding an *Abstract Syntax Tree* (AST) for bottom-up construction of our *figure expression trees*; this technique is similar to how compilers encode statements and expressions. We consider in Figure 3 the top-level class we will use to represent our geometry figures: `FigureNode`.

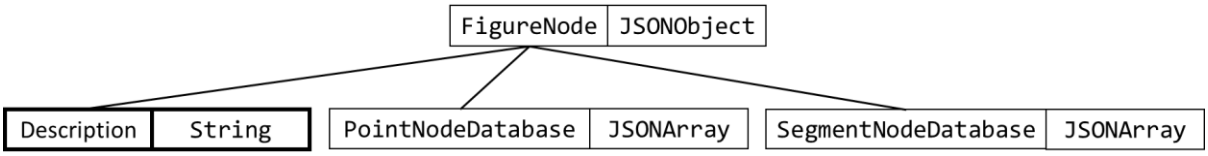

Figure 3: The top-level node in our geometry figure syntax tree structure: a `FigureNode`. A `FigureNode` consists of a terminal `Description` node and a sequence of `Points` and `Segments`.

As depicted in Figure 3, each figure consists of:

- An input `JSONArray` object containing a set of points that will be represented and returned as a `PointNodeDatabase` object,

- An input `JSONArray` object containing a set of segments that will be represented and returned as a `SegmentNodeDatabase` object, and

- A string describing the geometry figure. The description node is bolded with a thicker outline to indicate that it is a *terminal node*. In parsing parlance, a terminal node refers to a node that does not contain any other node (i.e., a leaf node in an abstract syntax tree).

Points (vertices in a graph). A fundamental object in a geometry figure is a point; conveniently, we implemented the `PointNode` class in a previous assignment. As can be observed in the JSON representation in Figure 2, we define our named points in the Cartesian plane consistent with our earlier `PointNode` implementation. The three attributes of each point (name and coordinates) can be seen in the abstract syntax tree representation in Figure 4 as *terminal nodes*.

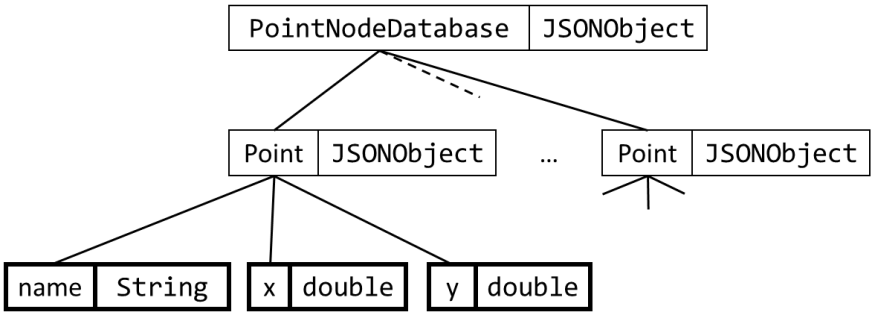

Figure 4: A figure defines a set of named Cartesian points; we will represent them using a `PointNodeDatabase`.

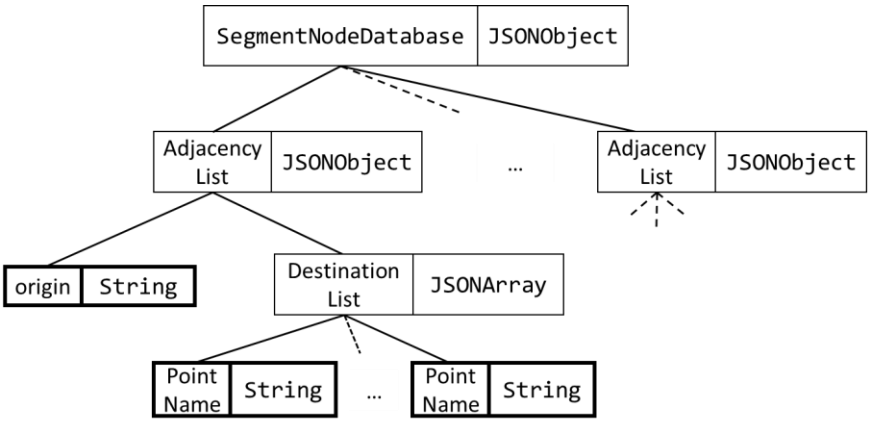

Figure 5: A figure also defines a set of segments using adjacency lists; we will represent them using a `SegmentNodeDatabase`.

Segments (undirected edges in a graph). Connecting the points to form a geometry figure are the segments. Just as with a previous assignment, we will represent our segments in JSON using an adjacency map (a list of adjacency lists). As an example, it is clear in Figure 1 that point `B` is part of 4 segments (`BA`, `BC`, `BE`, `BD`). The corresponding adjacency list in JSON is defined as `{"B" : ["C", "D", "E"]}`; we note how the JSON representation takes a minimalist approach by not repeating segments (since AB is represented in the adjacency list under `"A"`). The abstract syntax tree class structure along with corresponding JSON objects for segments is shown in Figure 5. Observe that the constituent points for each segment are terminal nodes; this includes the origin point and each destination point.

What You Need to Do: `JSONParser` Implementation

Your task is to implement functionality that will read a JSON data file and create an abstract syntax tree structure for a geometry figure. As one last example, we can view the abstract syntax tree object in Figure 6 which corresponds to the geometry figure in Figure 2.

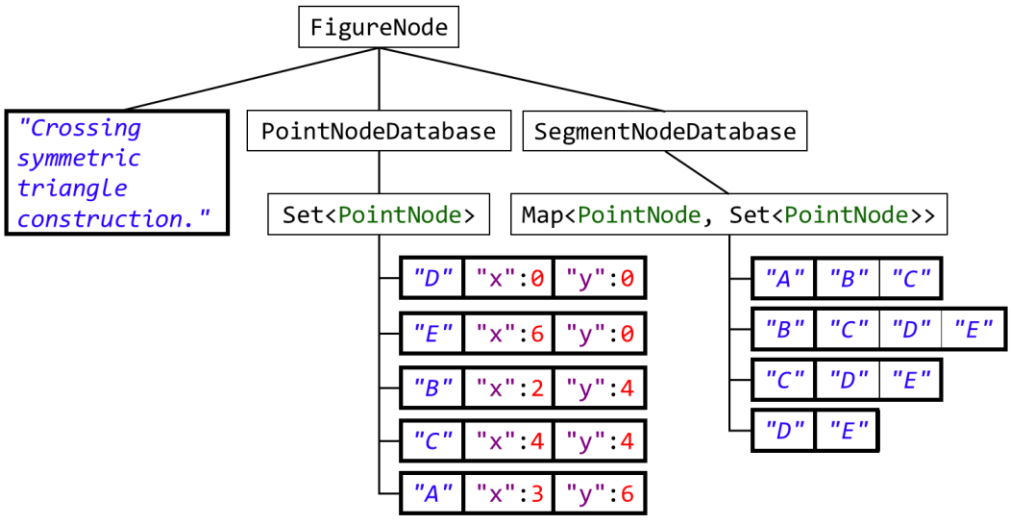

Figure 6: Abstract syntax tree object structure for the JSON data in Figure 2.

What You Need to Do: Add JSON to the Eclipse Project

Provided with this assignment is a Java archive file (JAR) containing the `org.json` code we will use for JSON processing. You may have to add this external library to the project. To do so:

- Created a `lib` folder in the project and copy (click and drag) the provided JSON JAR file into the project folder.
- Right-click on the file and select `Build Path > Add to Build Path` as shown in Figure 7.
- A copy of the file will be created in the `Reference Libraries` project folder as shown in Figure 8.

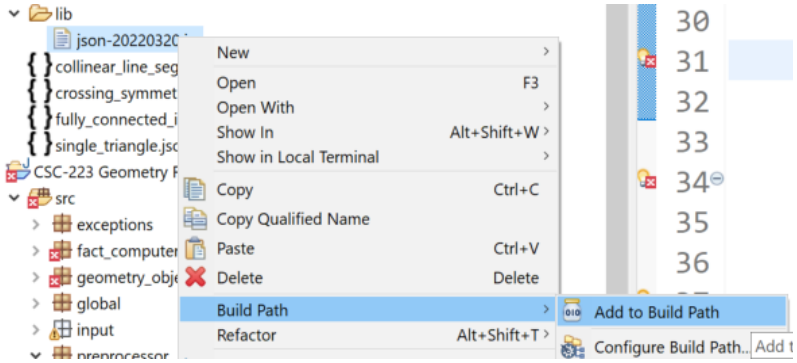

Figure 7: Add a JAR to the Build Path

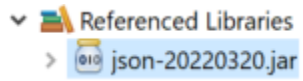

Figure 8: The result of adding a JAR to the Build Path.

What You Need to Do: Verifying Syntax Tree Construction with Unparsing

Parsing a syntax tree structure is a challenging endeavor, but Object-Oriented languages like Java help ease the burden. With Java, we will exploit the fact that we can define an interface for which all tree-based classes will inherit. Specifically, each of the classes in Figure 9 will implement the `ComponentNode` interface specified in Figure 10.

`FigureNode`
`PointNode`
`PointNodeDatabase`
`SegmentNodeDatabase`

Figure 9: Classes that will implement the `ComponentNode` interface.

```
public interface ComponentNode
{
    void unparse(StringBuilder sb, int level);
}
```

Figure 10: The `ComponentNode` interface.

In order to verify the construction of a syntax tree representing a geometry figure, we will implement common functionality for tree construction techniques called *unparsing*. The idea of unparsing is to traverse the tree and output the contents of the tree as a string so we can verify that our input matches the object-based representation. In this case, our goal will be to unparse the syntax tree so that the input string will *correspond* to the output string (it does not have to be exact). Unparsing is not considered strictly to be a pre-order, post-order, or in-order operation since, depending on the desired output, unparsing may function using all of these techniques, although it may be helpful to think of it as in-order traversal.

For our implementation of unparsing indicated in Figure 10, we will build a string using a given `StringBuilder` object. The second argument, `level`, is an indication of how many levels of indentation we wish to communicate to the next node in the tree; differing levels of indentation can be observed in Figure 11.

A reasonable unparsing of the JSON code in Figure 2 is shown below in Figure 11. Observe that the input JSON representation does not match exactly the unparsed output, but we get a strong sense of correctness of the tree structure object-based tree structure. Of course, one giant downfall of this approach is that it does not facilitate automated testing, but that is the price we will pay.

```
Figure
{
    Description : "Crossing symmetric triangle construction."
    Points:
    {
        Point(D)(0.0, 0.0)
        Point(E)(6.0, 0.0)
        Point(B)(2.0, 4.0)
        Point(C)(4.0, 4.0)
        Point(A)(3.0, 6.0)
    }
    Segments:
    {
        A : B  C
        B : A  C  D  E
        C : A  B  D  E
        D : B  C  E
        E : B  C  D
    }
}
```

Figure 11: Unparsing of the JSON representation in Figure 2.

What You Need to Do: Develop Geometry Figures

Some sample geometry figures encoded in our JSON format have been provided; however, you are strongly encouraged to create more simple and more complex figures. We will use any of these figures in future assignments.

What You Need to Do: junit Testing

Your junit tests should effectively test construction of a figure by checking for the correctness of points and segments existing in their corresponding databases.

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*.