JSON + X

Assigned: 2022-01-26
Due Date: 2020-02-01 by 11:59PM U.S. Central Time

Assignment

Get your copy of the assignment here

You may encounter "Non-managed pom.xml file found". You should select "Add as Maven Project"

The goal of this assignment is to get you comfortable with JSON in preparation for the next assignment, in which you will be utilizing JSON to as one component of a more complex project. This week, you'll get to choose what JSON dataset to work with, and you'll have some freedom over what you do with that dataset.

Out of the following 4 tasks, you must complete 2 (any additional tasks completed will be counted as extra credit):

- 4 filtering functions
- 4 analysis functions
- A visualization (e.g. charts, graphs, playoff brackets, a periodic table, episode catalog)
- Some other interesting use of your data (e.g. encoding and playing a song)

Remember that you still have to write tests to ensure that all of the functionality you've implemented works correctly.

Gradescope

There is no autograder for this assignment, but you still must submit your GitHub repository to Gradescope.

Goals

- The concept of separating data from code
- Processing structured data in the form of JSON
- Explore your interests in a cool and unique way

Background Info

Serializing (or marshalling) is the process of converting objects in code into a different representation for storage, whether that's JSON, YAML, binary data, or something else. **Deserializing** (or unmarshalling) is the reverse: converting serialized data into actual objects in your code. In this case, you are transmitting serialized data about a topic of your choosing, so that you can deserialize it into Java objects for processing.

A **schema** is a definition of how to represent some data. For example, if we want to describe a course at UIUC in words we might say the following:

- A course at UIUC is an object
  - It has a department, which is a string
  - It has a number, which is an integer
  - It might be difficult, which is represented as a boolean
  - It might be time consuming, which is represented as a boolean
  - It has some professors who teach the course, represented as an array of strings

There is a natural translation between a schema describing objects stored in JSON and objects in Java.

JSON

We require that your data is stored as JSON, a data format popularized by the JavaScript programming language. Here's an example of some JSON, representing a group of courses at UIUC:

```
{
    "courses": [
        {
            "department": "Computer Science",
            "number": 126,
            "isDifficult": false,
            "isTimeConsuming": true,
            "professors": ["Craig Zilles", "Carl Evans", "Michael Woodley"]
        },
        {
            "department": "Computer Science",
            "number": 421,
            "isDifficult": true,
            "isTimeConsuming": false,
            "professors": ["Elsa Gunter", "Mattox Beckman"]
        }
    ]
}
```

Things to note about syntactically valid JSON:

- The contents are wrapped in an outermost pair of curly braces: `{` and `}`
- Keys are **always** quoted strings
- Values can be numbers, strings, arrays, booleans, or objects
- For arrays and objects, you *must* omit the very last comma

If you want to check that your JSON is valid, you can use a JSON validator like this one.

Parsing

How you parse the JSON is up to you. The office hours staff and your code mod will be familiar with the following libraries, so it is recommended that you choose one of them. However, if you prefer a different method, feel free to implement it that way.

GSON

If we have some JSON like above, we need to define some Java classes that match the schema so that GSON can deserialize the JSON into instances of those objects.

First, we define the outermost class with its fields:

```
public class UiucCourses {
    private List<Course> courses;

    // -- Getter methods omitted --
}
```

Then we define the inner class, with its own fields (Note that this class should have its own file):

```
public class Course {
    private String department;
    private int number;
    private boolean isDifficult;
    private boolean isTimeConsuming;
    private String[] professors;

    // -- Getter methods omitted --
}
```

Finally, we deserialize our JSON into an instance of `UiucCourses`:

```
String myJson = "....."; // Read in JSON from somewhere...

Gson gson = new Gson(); // Create an instance of a GSON parser
UiucCourses courses = gson.fromJson(myJson, UiucCourses.class);
```

Note that you can define your collections as either `Object[]` or `List<Object>` at your preference, and GSON will convert it to the appropriate format.

If you need more documentation and examples on how to use GSON, see the User's Guide.

Jackson

You can also use Jackson to deserialize your JSON. Jackson is already included in your project; all you have to do is `import` and use it.

If we have some JSON like above, we need to define some Java classes that match the schema so that Jackson can deserialize the JSON into instances of those objects.

First, we define the outermost class with its fields:

```
public class CourseExplorer {
    private List<Course> courses;

    // -- Constructor and getter/setter methods omitted for brevity --
}
```

Then we define the inner class, with its own fields (Note that this class should have its own file) :

```
public class Course {
    private String department;
    private int number;
    private boolean isDifficult;
    private boolean isTimeConsuming;
    private String[] professors;

    // -- Constructor and getter/setter methods omitted for brevity --
}
```

Finally, we deserialize our JSON into an instance of `CourseExplorer`:

From a file (`courses.json`):

```
File file = new File("src/main/resources/courses.json");
CourseExplorer explorer = new ObjectMapper().readValue(file, CourseExplorer.class);
```

From a string:

```
String myJson = "...";
CourseExplorer explorer = new ObjectMapper().readValue(myJson, CourseExplorer.class);
```

Note that you can define your collections as either `Object[]` or `List<Object>` at your preference, and Jackson will convert it to the appropriate format.

If you need more documentation and examples on how to use Jackson, see the User's Guide.

JUnit's `@Before`

As with the previous assignment, we expect you to write unit tests as you develop. We're going to introduce a new feature of JUnit to make writing some kinds of tests easier for you: the `@Before` annotation.

Here's the motivation: your unit tests should be self contained so that running one test doesn't influence the results of another. When you have objects that are shared between tests, that can be hard to guarantee. When you apply the `@Before` annotation to a function in your testing class, it tells JUnit that it should run that function once before each and every one of your unit tests, so we can use an `@Before`-annotated method to reset any necessary state between test cases.

Specifically, this is great for tests involving GSON:

```
public class UiucCoursesTest {
    private Gson gson;

    @Before
    public void setUp() {
        gson = new Gson();
    }

    @Test
    public void testCaseOne() {
        gson.fromJson(/* ... */);
    }

    @Test
    public void testCaseTwo() {
        gson.fromJson(/* ... */);
    }
}
```


Now we don't have to repeat the line `Gson gson = new Gson()` in all of our tests, and the GSON instance will be reset before each and every one of our unit tests. For more information, see JUnit's documentation.

Part 0: Choose a JSON Dataset

Here's some sample JSON: https://github.com/jdorfman/awesome-json-datasets, and here's some API's that may return JSON: https://github.com/public-apis/public-apis. Feel free to find your own JSON as well! Just make sure there's enough data to fulfill the requirements.

You may download a copy of this data to use in your development process and for testing, but you should avoid copy-pasting the JSON into your code.

Part 1: Deserialization

In order to do anything meaningful with the data in Java, you have to convert it into Java objects.

Specifically you should:

- Look at the JSON data to determine what kinds of Java classes you will need to define to represent the schema
- At runtime, read and deserialize the JSON (either from a local file or from the Internet)

Part 2: Processing and Analyzing the Data

Once you have your data as Java objects, you need to be able to filter and process it to get more interesting results.

If you choose to do the filtering functions, you'll need at least **four** different functions that filter collections of information based on some criteria. Some examples (from some soccer JSON):

- A list of all goals scored after a certain timestamp during the game
- A list of all goals scored by a single player
- Given a player, a list of all the people that player has passed to

For the analysis option, you'll need at least **four** different functions that do some analysis and produce a computed result. Some examples (also from the soccer JSON):

- The average jersey number of all the players who scored a goal
- A sorted list of players by the number of times they received the ball

For the purposes of this assignment, a filtering function takes in a collection and produces another collection with the same number or fewer elements. An analysis function gives you some kind of result, whether that result is a list or a number or a boolean.

Your functions should not mutate any of the data they operate on; they should return new collections/numbers/objects as necessary.

Options 3 & 4 are for those who are already comfortable with JSON. If you're unsure if your idea qualifies, don't hesitate to ask your moderator or check on Campuswire.

This assignment is deliberately open-ended with regards to how you choose to implement the requirements; design decisions like what kinds of classes and functions to write are left up to you. Be creative! We have avoided giving you any starter code to encourage you to reason about how to organize your project.

Testing

As always, testing is essential. However, because you're deciding the functionality, it is up to you to make sure that you're thoroughly testing it.

Note -- You may have to mark the test directory as "Test Sources Root". This can be done by right clicking on the directory and selecting "mark directory as".

Deliverables and Grading

As always, try to use the best code design and coding style. Continue to focus on writing good names (you have a lot of freedom in naming for this assignment) and think about how to cleanly lay out your code so that it's easy for an outsider (your moderator) to follow. You might find Chapter 4 of the textbook and Section 4 of the Google Java Style Guide to be good resources for this.

As you work on this project, you should commit small pieces of functionality as you write them. Don't make a single mega-commit at the very end! We expect your commit history to demonstrate you following some sort of iterative design process. For example: writing some tests, writing some code, changing some tests, fixing a test, writing more code, etc. Commits are cheap. Make lots of them!

Specific things we will be checking for:

- Do you meet at least two of the four JSON handling requirements?
- Is the JSON data loaded at runtime (i.e, not copy-pasted into your code)? (Bonus: if you wanted to fetch the JSON from a different source, could you do it *without* changing your code?)
- Does the code have comprehensive unit tests?
- Is the naming clear and idiomatic? Does it follow the Google Java Style Guide?
- Is the code responsive with regards to handling different data inputs? If we gave you a different JSON file (same schema, different data), would the code still work?
- Do the functions avoid mutating the input parameters?

Assignment Rubric

Note that this rubric is not a comprehensive checklist; it'd be impossible to list out every single thing that you should be considering while writing your code.

▼ Click here to view

Readability and flexibility of code

- Modularity: each method should perform one distinct task
- It should be easy to read through each method, follow its control flow, and verify its correctness
- The code should be flexible/ready for change (no magic numbers, no violations of DRY)

Object decomposition

- Member variables stored by each class
  - Classes should store their data using the most intuitive data structure
  - No "missing" member variables
  - No member variables which should be local variables
  - No redundancy / storing multiple copies of the same data in different formats
- Encapsulation
  - Appropriate access modifiers
  - Member variables should generally only modified by member functions in the same class
  - The interface of a class should be intuitive/abstract, and external code should only interact with the class via the interface
    - By intuitive, we mean that it should be easy to understand and use the class, and there shouldn't be any hidden assumptions about how the class should be used
    - By abstract, we mean that an external client shouldn't need to worry about the internal details of the class
  - No unnecessary getters/setters
  - In Java, getters should not return a mutable member variable

Documentation

- Specifications
  - Specifications are required for all functions which are part of the public interface of a class
  - Specifications should precisely describe the inputs and outputs of a function, and should also describe what the function does (e.g. mutating state of object)
  - Specifications should also be formatted properly (e.g. follow Javadoc style for Java assignments)
- Inline comments should not describe things which are obvious from the code, and should describe things which need clarification

Naming

- Semantics: names should effectively describe the entities they represent; they should be unambiguous and leave no potential for misinterpretation. However, they should not be too verbose.
- Style: names should follow the Google Java/C++ Style Guide

Layout

- Spacing should be readable and consistent; your code should look professional
  - Vertical whitespace should be meaningful
    - Vertical whitespace can help create paragraphs
    - Having 2+ empty lines in a row, or empty lines at the beginning or end of files, is usually a waste of space and looks inconsistent
  - Horizontal whitespace should be present where required by the Google Style Guide
- Lines should all be under 100 characters; no horizontal scrolling should be necessary

Testing

- You should make sure all types of inputs and outputs are tested.
  - If category A of inputs is likely to cause different behavior than category B, then you should test inputs from both categories
- Boundary/edge cases often cause different/unexpected behavior, and thus, they should be tested
- Your tests should cover all of the functionality that you've implemented. In other words, every line of code should be exercised by some test case, unless the assignment documentation says otherwise
- Each individual test case should only serve one coherent purpose. Individual test cases should not have assertions testing unrelated things
- Your tests, like your code, should be organized and easy to understand. This includes:
  - Easy to verify thoroughness / all possibilities covered
  - Easy to verify the correctness of each test case
  - Clear categories of test cases, where similar tests are grouped together
    - In Java, this can be accomplished by inserting comments to separate groups
  - Test case names make the purpose of each test case clear
    - Here is one possible way to name test methods in JUnit

Process

- Commit modularity
  - Code should be checked-in periodically/progressively in logical chunks
  - Unrelated changes should not be bundled in the same commit
- Commit messages
  - Should concisely and accurately describe the changes made
  - Should have a consistent style and look professional
    - First word of the message should be a verb, and it should be capitalized

Presentation

- Arrived on time with all necessary materials and ready to go
- Good selection of topics to focus on
- Logical order of presentation
- Appropriate pacing and engagement of the fellow students
- Speaking loud enough and enunciating clearly

Participation

- Each student should contribute at least one meaningful comment or question for every other student who presents in his/her code review
- Students must behave respectfully to moderator and other students

Weightings

Your grades for each section of the rubric will be weighted as follows:

- Readability and flexibility of code (15%)
- Object decomposition (15%)
- Documentation (5%)
- Naming (12.5%)
- Layout (12.5%)
- Testing (20%)
- Process (10%)
- Presentation (5%)
- Participation (5%)
