Homework 03

Authors: Jacob, Angela, Ishuma, Ariel, Nicolas

Problem Description

Hello, and congratulations on your selection as a NASA intern! More specifically, you've been assigned to the Jet Propulsion Laboratory in Pasadena, California, home to the facilities for the Deep Space Network in the United States. Since you're just getting started, we have a deeply exciting assignment for you: converting old satellite classification data from its original format into something more readable by modern software, so that the American people can peruse our satellite catalog from a website your fellow interns are writing. You'll be responsible for reading the traditional two-line element set (TLE) format from plain text files, using polymorphic methods on the objects representing the satellites, and then saving the data to a new file, in an easier to read format. For those who are interested, an explanation of the TLE format can be found in the Appendix of this PDF. Good luck: the people of our nation are counting on you.

Solution Description

In this assignment you will get practice with **file I.O., polymorphism, and exception handling**. You will be provided a folder named `satellite_data` with several files in it. This is the test data set you will use to verify your code works. In addition, you will be provided with a java class called `SatelliteTLEParser`, which we are providing to you to handle the messier work of reading the TLE format. You will write your program in a Java class called `SatelliteDataConverter` and it should use Java's built in File I/O classes and methods to iterate through the folder, open each file, load the data within, pass the data to the parser method, and then write that data out to a new file. Your code should be able to handle any valid data we give it – do not hardcode to the provided test data.

Satellite Classes

In addition, you will write three Java classes to represent the satellite objects: `Satellite`, `ClassifiedSatellite`, and `GeostationarySatellite`. Since classified and geostationary satellites are satellites, your class structure should reflect that "is-a" relationship.

`Satellite.java`

- Variables

  All three satellite classes should have the following fields, either explicitly or via inheritance:

  - `String name`
  - `int catalogNumber`
  - `int launchYear`
  - `int launchDay`
  - `double inclination`
  - `double meanMotion`

  All fields should be private.

- Constructor
  - Include a constructor that takes in `name`, `catalogNumber`, `launchYear`, `launchDay`, `inclination`, and `meanMotion` in this exact order.

- Methods

  In addition, you will be responsible for writing the following methods. All methods should be public unless otherwise specified. You may write other methods that you find useful as well.

  - `orbitTime()`
    - The "mean motion" of a satellite is defined as the number of orbits around earth per day. This mathematical definition is useful for astronomers and rocket scientists, but we'd prefer to offer a more easily digestible number: the number of minutes it takes for the satellite to complete one orbit. Since the mean motion is the number of orbits per day, we can calculate the orbit time in minutes with the following formula:

      `time = (24 * 60) / meanMotion`
    - This method should return the result of this calculation as a `double`.
  - `toString()`
    - This method should return a string with the following format with a newline at the end: "`{name}: #{catalogNumber}. Launched on day {launchDay} of {launchYear}. Inclination: {inclination} degrees. {meanMotion} orbits per day - {orbitTime()} minutes per orbit.`"
    - The curly braces {} denote values that should be replaced with the corresponding variable(s), do not include the curly braces in your output.
    - Do not round any of the floating-point values in your output.
  - `encodeCSV()`
    - This method should also return a string with a newline at the end, but it should be in a comma-separated format, as shown below: "`{name},{catalogNumber},{launchYear},{launchDay},{inclination},{meanMotion},{orbitTime()}`"
    - The curly braces {} denote values that should be replaced with the corresponding variable(s), do not include the curly braces in your output.
    - Do not round any of the floating-point values in your output.

ClassifiedSatellite.java

Under the TLE format, there are three supported levels of classification: unclassified, classified, and secret. To keep things simple, you will create an additional Java class, `ClassifiedSatellite`, to handle both classified and secret satellites, which will extend `Satellite`. This class will support the same functions as the parent class but will require authentication by the user to invoke any of its methods that output information (`toString()` and `encodeCSV()`).

- Include a constructor that takes in `name`, `catalogNumber`, `launchYear`, `launchDay`, `inclination`, and `meanMotion` in this exact order.
  - Ensure that you call the constructor located in the `Satellite` class from this constructor to maximize code reuse
- To facilitate authentication for `toString()` and `encodeCSV()`, you should read in user input, and check if the input matches the password.
  - Prompt the user for the password by printing "`Password:`" to the terminal with a newline at the end.
  - If the user enters the correct password, return the normal value for the method called.

    - If the user does not enter the correct password, you should not return the normal value, but rather the String "`INCORRECT PASSWORD`".
    - The password you should check against is "`f8ee89496da476b3849f4de45a4528b4`".
- **Hint:** we strongly recommend writing a helper method to handle the password check.
- **Note:** the mainframe on which this code will be running has dangerously limited storage capacity, and so any duplication of code is a hindrance to the mission. For full credit, you should reuse code whenever possible – using the parent methods from `Satellite` is a requirement.

`GeostationarySatellite.java`

You'll also write a third class: `GeostationarySatellite`, which will also extend `Satellite`. Geostationary satellites are those which orbit perfectly in sync with the rotation of the Earth, meaning that they remain over the same location in perpetuity.

- Include a constructor that takes in `name`, `catalogNumber`, `launchYear`, `launchDay`, and `inclination` in this exact order.
    - Since they all orbit once per day, every `GeostationarySatellite` should have a `meanMotion` of 1.0.
    - Ensure that you call the constructor located in the `Satellite` class from this constructor to maximize code reuse.

`SatelliteDataConverter.java`

This class will be responsible for the execution of your program. Use methods from existing Java File I/O libraries to help you—do not reinvent the wheel. If you aren't sure where to start, look at the Java API to find methods that might be useful to you. Additionally, remember you must make necessary imports and handle any associated Exceptions when working with File I/O in Java.

- Methods
    - `createSatelliteArray()`
        - Private static helper method that takes in the File object representing the `satellite_data` folder.
        - This method should determine how many files belong to the folder – remember, one satellite per file – and pass each file's data to the `SatelliteTLEParser.parseData()` method, which expects the entire contents of one file as a String.
            - ◊ **Note:** `SatelliteTLEParser.parseData()` expects the passed-in String to include the newlines from the original file. As such, you must take care to preserve the newline characters when you read in each file.
        - `SatelliteTLEParser.parseData()` will return the appropriate Satellite object, which you should store in an array of Satellite objects that you will return.
            - ◊ Your code won't know (and doesn't need to know) which kind of satellite object was loaded – polymorphism in action!
    - `main()`
        - Your program should expect the first command line argument passed to it to be the relative path to the folder containing the satellite data and should then create a `File` object using this command line argument.
            - ◊ You may assume that folder will always exist and will be accessible from the same directory as `SatelliteDataConverter.java`.

        - ◊ **Note:** When you are running your program, you will need to pass the path to the "`satellite_data`" folder as a command line argument. If you are running your program from the terminal, simply run "`java SatelliteDataConverter {relative path to satellite_data}`". If you are running your program from IntelliJ, please see this guide for instructions on how to add command line arguments.
    - ▪ Your program should then open each file within the folder and parse through its data to create a `Satellite` object for each file.
        - ◊ Utilize the `createSatelliteArray()` helper method to help you.
    - ▪ After all the text files have been loaded into the program, decoded, and the appropriate satellite objects have been stored in an array, your program should open a new file named "`satellite_output.csv`". It should fill this file with the string from each satellite object's `encodeCSV()` method, with a new line between each satellite.
        - ◊ **Hint:** Iterating over the array returned from `createSatelliteArray()` is the simplest way to do this.
    - ▪ Whenever working with `File` objects, be prepared to handle checked exceptions, particularly `FileNotFoundExceptions` and `IOExceptions`. Use try/catch blocks to handle any potential errors. If you encounter an exception, you should print out the exception's message field followed by a newline.

Your efforts will enhance our nation's scientific and defense capabilities, and we thank you!

Hints

- The format of your output must match character-for-character. This is because your output, if done correctly, will be able to be viewed in any spreadsheet program, or by any spreadsheet-processing code. (Try this out in your tests!)
- If you can find out what text I used to generate the test password, I'll nominate you for a Turing Award. Hint: Google "md5 hash".

Appendix

While you do not need to interface directly with the TLE data, it may be of interest. The TLE format is easy for computers to read, as it was originally designed for 80-column punch cards. A sample TLE entry is shown below:

```
ISS (ZARYA)
1 25544U 98067A   08264.51782528 -.00002182  00000-0 -11606-4 0  2927
2 25544  51.6416 247.4627 0006703 130.5360 325.0288 15.72125391563537
```

- Line 1
    - `ISS (ZARYA)` is the satellite `name`
- Line 2
    - The first character will always be 1
    - `25544U` is the combination of the `catalogNumber`, which will always be 5 characters, and the classification level (U = Unclassified, C = Classified, S = Secret)
    - `98067A` is the international designator
    - `08264.51782528` contains the epoch `launchYear` (08) and the epoch `launchDay` (the rest of the token).

        - The epoch year will always be exactly two characters. If the epoch year is within the range 57-99, inclusive, it represents a year in the twentieth century (e.g. 58 = 1958). If the epoch year is within the range 00-56, it represents a year in the twenty-first century (e.g. 21 = 2021).
    - The rest of the information in the second line is not needed for the `Satellite` class
- Line 3
    - The first character will always be 2
    - `25544` is the `catalogNumber`
    - `51.6416` is the `inclination`
    - The fourth, fifth, sixth, and seventh tokens in the third line are not needed for the `Satellite` class
    - `15.72125391` is the `meanMotion` and the rest of the eighth token is not needed for the `Satellite` class

Clarifications and Example Output

Please refer to the pinned Piazza thread for HW03 Clarifications for any necessary clarifications about the requirements outlined in this pdf. We may also post some example output that you can use to check your code against on the same Piazza post.

Allowed Imports

You will need to import many Java classes to complete this assignment. At minimum, `java.io.File`, `java.io.FileNotFoundException`, `java.io.PrintWriter`, `java.util.Scanner`, and/or `java.io.FileReader`, and will likely prove useful. However, you can use any class from the standard library you choose.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`

Checkstyle and Javadocs

You must run Checkstyle on your submission. The Checkstyle cap for this assignment is **9 points**. If you don't have Checkstyle yet, download it from Canvas -> Modules -> Checkstyle Resources (Section B & C) or General Information (Section D) -> checkstyle-8.28.jar. Place it in the same folder as the files you want Checkstyled. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the Checkstyle cap mentioned above). The Java source files we provide contain no Checkstyle errors. In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

For additional help with Checkstyle see the CS 1331 Style Guide.

Collaboration

No collaboration is allowed on this assignment. See syllabus for more details.

In addition, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- Satellite.java
- ClassifiedSatellite.java
- GeostationarySatellite.java
- SatelliteDataConverter.java

Please only submit the individual files—do not put them into a package or you will risk getting a zero on the assignment. Make sure you see the message stating "HW03 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine. In addition, the Gradescope autograder may not yet be available for a period of time after the assignment is released so it is important that you learn techniques to test your own code locally.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Run Checkstyle on your code to avoid losing points
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Feature Restrictions" to avoid losing points
- Check on Piazza for a note containing all official clarifications