Homework 6

1 Introduction

In this homework, you will practice translating a written description into multiple classes using inheritance and overriding.

Please be sure to read the **entire document** as there are things even at the very end that you will not want to miss for this and future assignments!

2 Solution Description

`PlanetarySystemSim.java` (**DO NOT CHANGE THIS FILE**) is a driver and will be used for you to test your homework and interact with the simulation.

`PlanetarySystem.java` (**DO NOT CHANGE THIS FILE**) is a management file to keep track of the star and planets your system has.

`Color.java` (**DO NOT CHANGE THIS FILE**) is an enum that represents the colors a star may have, and provides you with methods to get and set that value for your star instances.

You will personally be creating the following classes:

- `AstronomicalObject.java`
- `Star.java`
- `Planet.java`
- `RockyPlanet.java`
- `GasGiant.java`

The specification for each class is provided below:

2.1 AstronomicalObject

- a `name` of astronomical object
- A constructor that takes in the name of the object.

- A setter method to change the `AstronomicalObject`s name.
- A properly overridden toString method that returns a string representation of the `AstronomicalObject`. This should just return the name + ”.”

2.2 Star

- This class has an ”is-a” relationship with `AstronomicalObject`.
- Instance variables:
  - An appropriately-typed variable `color`, which will represent the color of the star.
  - An appropriately-typed variable `isSun`, that represents whether or not this instance of Star can be a Sun.
- A constructor that takes the name of the star and the id of the color of the star. This constructor should not repeat code from its parent.
- A method that uses the methods provided in the Color enum to get the Color by the id passed in. After assigning the color, remember to set the `isSun` variable accordingly.
- A properly overridden `toString` method that returns the String representation of the Star instance. Should have the format of ”Star [name]. Color: [color]. [statement on whether it cant or cannot have planets]”.
- Appropriate getters and setters.

2.3 Planet

- This class has an ”is-a” relationship with `AstronomicalObject`.
- Instance variables:
  - An appropriately-typed variable `radius`, which will represent the radius in km. of the planet. Remember this is a value that cannot be changed, so consider that in your typing.
  - An appropriately-typed variable `orbitCount`, that represents how many orbits this planet has made.
- A constructor that takes in the name of the planet, the radius of the planet and the current `orbitCount`. This constructor should not repeat code from its parent.
- A properly overridden `toString` method that returns the String representation of the Planet instance. Should have the format of ”Planet [supers toString]. Radius: [radius]. Orbits completed: [orbitCount]”. If your star cannot have planets, state it instead of listing the planets.
- An `orbit` method that will make the planet orbit once around its Sun by incrementing the `orbitCount` variable.
- Appropriate getters and setters.

2.4 GasGiant

- This class has an "is-a" relationship with `Planet`.
- Instance variables
  - An appropriately-typed variable `numStorms`, that represents the number of anticyclonic storms this instance of `GasGiant` has (like Jupiters Great Red Spot!).
- A constructor that takes in and sets all instance data. This constructor needs to make use of code in its parent.
- A properly overridden `orbit` method that has the same functionality as `Planets` method, and adds three storms to the Planet by incrementing the `numStorms` variable.
- A properly overridden `toString` method that returns the String representation of the Planet instance. Should have the format of "Gas Giant [supers toString]. Number of storms: [numStorms]".
- Appropriate getters and setters.

2.5 RockyPlanet

- This class has an "is-a" relationship with `Planet`.
- Instance variables:
  - An appropriately-typed variable `hasLife`, that represents whether this instance of RockyPlanet has the conditions to for living beings to exist.
- A constructor that takes in and sets all instance data. This constructor needs to make use of code in its parent.
- A properly overridden orbit method that has the same functionality as `Planets` method, and toggles the `hasLife` variable (if true set to false, and vice versa.).
- A properly overridden `toString` method that returns the String representation of the Planet instance. Should have the format of "RockyPlanet [supers toString]. [statement on whether it can or cannot have life]".
- Appropriate getters and setters

3 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```java
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

    /**
     * Creates an awesome dog (NOT a dawg!)
     */
    public Dog(){
        ...
    }

    /**
     * This method takes in two ints and returns their sum
     * @param a first number
     * @param b second number
     * @return sum of a and b
     */
    public int add(int a, int b){
        ...
    }
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

3.1 Javadoc and Checkstyle

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

4 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **20** points.
Review the Style Guide and download the Checkstyle jar. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.2.jar -a *.java
Audit done. Errors (potential points off):
0
```

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.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **20** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

5 Turn-in Procedure

Non-compiling or missing submissions will receive a zero. NO exceptions

Submit all of the Java source files you modified and resources your program requires to run to T-Square. **Do not submit** any compiled bytecode (`.class` files) or the Checkstyle jar file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code through Checkstyle!

5.1 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!