Homework 05 - Water Fountain

Authors: Ryan Miles, Emma Barron, Manny Sonubi, Peter Dong, Allan Nguyen, Shishir Bhat

Problem Description

The water fountain company that you work for wants to re-write the code for their water fountains. The production team has given you all the information that needs to be tracked by each water fountain. The sales team has also given you a few requirements for special features to help sell the updated models. Your job is to make a Java class to represent a water fountain that contains all the information and requirements you have received. Unfortunately, you are the only software developer at your job so you have to write everything yourself.

Solution Description

For this assignment create a file named `WaterFountain.java`. You will have to fill in the required instance variables and methods listed below.

Instance Variables:

- `String modelName`
  - `modelName` should have private access and be used to hold the model name of the water fountain
- `boolean requiresMaintenance`
  - `requiresMaintenance` should have private access and be used to hold if the water fountain requires maintenance or not.
- `int cupsPoured`
  - `cupsPoured` should have private access and be used to hold the number of cups of water the water fountain has poured.

Static Variable:

- `int totalWaterFountains`
  - `totalWaterFountains` should have private access and be used to hold the number of water fountains that have been created. It should be set to `0` by default.

Constant Variable:

- `String SOFTWARE_VERSION`
  - `SOFTWARE_VERSION` should have public access, not be able to be changed, and should be able to be referenced statically. The variable should be set to `"2.0.0"`.

The Constructor:

The constructor should only take in values for `modelName` and `cupsPoured` and set each of them respectively. `requiresMaintenance` should be initialized to `false` and `totalWaterFountains` should be incremented by one.

The Methods:

All getters and setters should have public access.

- `Proper Getter & Setter for modelName`
- `Proper Getter & Setter for requiresMaintenance`
- `Proper Getter & Setter for cupsPoured`
- `Proper Static Getter for totalWaterFountains`
- `void pourCup()`
  - This method takes no parameters, returns nothing, and has public access. If the water fountain does not require maintenance, `cupsPoured` should be incremented by one. If the water fountain does require maintenance, do nothing.
- `boolean equals(WaterFountain other)`
  - This method takes in a `WaterFountain`, returns true if the other passed in water fountain is logically equal to this water fountain and false otherwise, and has public access. Water fountains are logically equal if they have the same `modelName`, `cupsPoured`, and `SOFTWARE_VERSION`.
- `String toString()`
  - This method takes no parameters, returns a `String`, and has public access. This method will return a string that contains the water fountain's `modelName`, `requiresMaintenance`, `cupsPoured`, and `SOFTWARE_VERSION`.
  - If the water fountain needs maintenance the returned string should match this: `"[modelName] has poured [cupsPoured] cups, requires maintenance, and is running version: [SOFTWARE_VERSION]"`
  - If the water fountain does not need maintenance then the returned string should match this: `"[modelName] has poured [cupsPoured] cups, does not require maintenance, and is running version: [SOFTWARE_VERSION]"`
  - Example: `"A-222 has poured 987 cups, does not require maintenance, and is running version: 2.0.0"`

Note: Each method requires proper JavaDocs (see below for more information)

Testing your code

If you want to test out your own code before submitting we recommend creating a separate java file and implementing a main method there. You can then create new water fountains and test your methods. Below is some sample testing code. **These tests are not comprehensive!**

```
WaterFountain wf = new WaterFountain("A-222", 987)
WaterFountain wf2 = new WaterFountain("B-5", 1)
wf.toString()
-> A-222 has poured 987 cups, does not require maintenance, and is running version: 2.0.0
wf.pourCup()
wf.getCupsPoured()
-> 988
wf.equals(wf2)
-> false
wf.equals(wf)
-> true
WaterFountain.totalWaterFountains
-> 2
```

Feel free to create your own tests in the `main` method! Try to write tests for the remaining methods in the file!

Rubric

- [15] Method Variables
  - [5] Correct visibility and type for instance variables
  - [5] Correct visibility, modifiers, and type for `totalWaterFountains`
  - [5] Correct visibility, modifiers, and type for `SOFTWARE_VERSION`
- [20] Constructor
  - [15] Correct setting of instance variables
  - [5] Increments `totalWaterFountains`
- [20] Getters & Setters
  - [5] Correct Getter & Setter for `modelName`
  - [5] Correct Getter & Setter for `requiresMaintenance`
  - [5] Correct Getter & Setter for `cupsPoured`
  - [5] Correct Getter for `totalWaterFountains`
- [10] Correct pourCup() method
- [20] Correct equals(WaterFountain other) method
- [15] Correct toString() method

Allowed Imports

To prevent trivialization of the assignment, you are *not* allowed to import any classes or packages.

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

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 overview for them is extremely 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 include 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 1.0
 */
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) {
    ...
    }

}
```

A more thorough tutorial for Javadocs can be found here.

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 `@verion` tag included. The comments for a class should 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.

Checkstyle can check for Javadocs using the -a flag, as described in the next section.

Checkstyle

For this assignment, we will be enforcing style guidelines with Checkstyle, a program we use to check Java style. Checkstyle can be downloaded here.

To run Checkstyle, put the `jar` file in the same folder as your homework files and run

```
java -jar checkstyle-6.2.2.jar -a *.java
```

The Checkstyle cap for this assignment is **25 points**. This means that up to 25 points can be lost from Checkstyle errors.

For this homework we will not count off for the checkstyle issue of `"Definition of 'equals()' without corresponding definition of 'hashCode()'."` or `"covariant equals without overriding equals(java.lang.Object)"`

Collaboration

Collaboration Statement

To ensure that you acknowledge collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one java file that you submit**. That collaboration statement should say either:

I worked on the homework assignment alone, using only course materials.

or

In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].

Recall that comments are special lines in Java that begin with `//`.

Turn-In Procedure

Submission

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

- `WaterFountain.java`

Make sure you see the message stating "HW05 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:

1. Prevent upload mistakes (e.g. forgetting checkstyle, non-compiling code)
2. 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.

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
- Submit every file each time you resubmit
- Read the “Allowed Imports” and “Restricted Features” to avoid losing points
- Check on Piazza for all official clarifications