Homework 5 - Coffee Shop

Problem Description

After a long night as band manager, you decide you want coffee, but the closest coffee shop is wildly inefficient. Therefore, you decide to build a Cup class, a CoffeeMachine class, and a Drink enum.

Solution Description

The Drink Enum

- Contains 'CHOCOLATE', 'COFFEE', 'TEA' and 'EMPTY' as possible values.
- The above should have the following `name` and `price` (instances variables with the correct types): CHOCOLATE is named "hot chocolate" with price 1.50 COFFEE is named "coffee" with price 2.00 TEA is named "tea" with price 1.00 EMPTY is named "nothing" with price 0.00
- Be sure to use proper convention when creating getters!

More information about enums:

- Enums are like classes with a set number of instances. For instance, if you made a color enum with values RED, GREEN and BLUE, you would not be able to create a new 'YELLOW' instance of color.
- Because enums are types of classes, we can write constructors for enums. However, all instances are created within the enum class, so the constructor will be private. So, we must call the constructor when declaring the instances of the enum.
- Enums can also have instance fields and methods, written as you would for any other classes.

Here's an example:

```
public enum Size {
    S("small"), M("medium"), L("large");
    String name;
    private Size(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
}
```

On line 2 above you see constructor calls after each instance of the enum is declared. This enum has three instances called S, M and L that each have a 'name' instance field with values "small", "medium" and "large" respectively. Note that the syntax for the enum is exactly the same as that for any class after the line of instance declarations.

The Cup Class

Instance fields:

- A cup has a `drink` (from the Drink enum) and a `stamp` (indicating where that cup is from). The stamp should be immutable. Examples of stamps would be "Starbucks" or "Blue Donkey". These instance fields should have appropriate getters and setters (think about what immutable means).
- An `equals` method which checks if two cups are equals. Two cups are equal if their drink and stamp are the same.
- A `toString` method which returns a string with the cup's stamp and drink (using the Drinks name) in the format "A cup of [drink name] from [stamp]". For example: "A cup of coffee from Java".

The CoffeeMachine Class

- All members of this class should be STATIC. That means we have static variables and static methods!

- Keeps track of the number of cups, `cupsUsed`, the coffee machine has dispensed
- Keeps track of how much money, `sales`, has been made using the price of the drinks dispensed
- Has a method named `stats` which prints that amount in the format "Today we made [amount earned] and used [number of cups used]"
- Stores up to 10 Cups at a time, `cups`, which is restocked when all the cups are given out (hint: how do you store multiple instances of a Class in a list without imports).
- Contains a `pour` method, which takes in a cup and returns that same cup with whatever drink the user requested. If the user does not give the coffee machine a cup and only requests a drink, the coffee machine should take a cup from its 'stock' of cups and return that cup with whatever drink the user requested. If that stock is empty, it should be refilled. When refilling, the coffee machine's cups should all have the stamps "Java".

Remember

- For both the Cup and CoffeeMachine classes, think about private versus public modifiers (i.e. something should only be public if it ABSOLUTELY has to be). When do we use public and when do we use private? How do we ensure that we can access those private members of the class?
- Be efficient and avoid duplicate code whenever possible!
- Don't be afraid to add methods beyond the ones we require. This lets you to add a level of abstraction to your code. (Abstraction is the idea that a user should know what a method does and what it may give/return, but doesn't need to know how this is done). In this case, it'll probably be easier for you to write a program when everything isn't inside one method. Plus, this helps you reuse code!
- You will probably benefit from writing a tester class (i.e. Tester.java) with a main method that interacts with your CoffeeMachine class (and tests your other classes) to make sure all methods are functioning correctly.
- Use proper convention when creating getters and setters!

Allowed Imports

To prevent trivialization of the assignment, you are not allowed to import anything.

Feature Restrictions

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

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

Rubric

[27] **Drink**

- [14] Is an enum with only the values: `EMPTY`, `TEA`, `COFFEE`, and `CHOCOLATE`
- [3] Constructor
- [4] `price` and `name` fields
- [4] Getters for all fields
- [2] Fields are private

[29] **Cup**

- [4] Constructor that takes in a `Drink` and a `String`
- [6] `drink` and `stamp` fields
- [6] Getters for all fields
- [3] Setter for `drink`
- [5] `toString`

- [4] `equals`

[44] **CoffeeMachine**

- [3] Only static fields
- [6] `sales`, `cups`, and `cupsUsed` fields
- [5] `cups` holds 10 cups
- [20] Static `pour` method
  - [10] Works with `Cup` and `Drink` parameters
  - [10] Works with just `Drink` parameter
- [10] Static `stats` method
  - [5] Works in basic case
  - [5] Reports correct stats after repeatedly calling `pour`

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:

```
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. Just like Checkstyle, **one point will be deducted for each Javadoc error detected by running checkstyle**. These deductions will also be limited by the checkstyle cap, which applies to the sum of checkstyle and javadoc errors.

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
```

Make sure to include the -a flag included for testing both checkstyle and javadocs

The message above means there were no Checkstyle or javadoc 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!

There are two unavoidable errors for this assignment. You will not lose points for the following:

- Definition of `equals()` without corresponding definition of `hashCode()`
- Covariant equals without overriding equals(java.lang.Object)

Collaboration Statement

To ensure that you acknowledge a 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].

Submission

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

- `CoffeeMachine.java`
- `Cup.java`
- `Drink.java`

Make sure you see the message stating "HW## 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. non-compiling code)
2) Provide basic formatting and usage validation

In other words, the test cases on Gradescope are not 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 and run your code; you can do that locally on your machine.

Other portions of your assignment are also autograded once the submission deadline has passed. The autograders used are often dependent on specific output formats, so **make sure that your submission passes all test cases marked "FORMAT:"**.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Ensure you pass all "FORMAT:" tests
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications