Programming Assignment #6: PotionMaster

COP 3330, Fall 2019

Due: Wednesday, December 4, *before* 11:59 PM

Abstract

This assignment is designed to give you some practical experience working with lists, sets, and maps in Java. This one is designed to be fairly short, given our time constraints as we head into final exams.

Deliverables

PotionMaster.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

> *In mythology and literature, a potion is usually made by a magician, dragon, fairy or witch and has magical properties. It is used for various motives including the healing, bewitching or poisoning of people. For example, love potions for those who wish to fall in love (or become deeply infatuated) with another; sleeping potions to cause long-term or eternal sleep (in folklore, this can range from the normal REM sleep to a deathlike coma); and elixirs to heal/cure any wound/malady.*
>
> *In modern fantasy, potions are often portrayed as spells in liquid form, capable of causing a variety of effects, including healing, amnesia, infatuation, transformation, invisibility, and invulnerability.*
>
> – *Potions (Wikipedia)*

1. Overview

Magical potions abound in all types fantasy fiction, from books and movies like *Harry Potter* to online games like *World of Warcraft*. Most contemporary works of fantasy share the foundational premise that brewing a particular type of potion is a rather formulaic affair that requires a certain set of ingredients (sometimes called "reagents"). The formula for a potion is much like a recipe used in cooking: to brew a particular potion, one needs to gather a particular set of ingredients – some magical, some mundane – and then follow the prescribed process to create the desired potion. We will work with that premise for the rest of this assignment.

In a fantasy setting, if we were to give the name of a common potion to any respectable potion master, he or she would be able to quickly rattle off the required ingredients for that potion. In that way, the potion master's mind is like a map: the name of a potion is a *key* that maps to the set of ingredients required to make that potion.

Similarly, if we were to give the name of a reagent to a potion master, he or she would be able to list off any known potions that require that particular ingredient (and perhaps a variety of other uses for that reagent, as well). Here, the potion master's mind is again acting as a map, but this time in the opposite direction: the name of a reagent acts as a *key*, and the *value* produced by that key is the set of all known potions that require that particular reagent.

In this assignment, you will write a method to construct each of those types of maps from a list of *PotionInfo* objects. The *PotionInfo* class (defined in *PotionInfo.java*, included with this assignment) has three primary fields (one of which – the *description* – is just flavor text that can be ignored for the purposes of this assignment):

```java
public class PotionInfo
{
   public String name;            // The name of this potion.
   public String description;     // Description of potion's effect on the user.
   public List<String> reagents;  // Reagents required to brew this potion.
}
```

Searching through a list of such objects to find the reagents necessary for a particular potion would be a slow endeavor. Using maps like the ones described above will help speed up that process.

Of course, a sharp potion master will also be able to look at the set of reagents he or she has on hand and quickly list off all known potions that can be brewed from those reagents (i.e., without need for any additional ingredients). You will write a method to accomplish that task – and a few related tasks – as well.

2. Miscellaneous Guarantees Related to Potions and Reagents (*Important!*)

Here are a few guarantees we will make about the data passed to your methods:

★ Throughout this assignment, you do not need to worry about variations in capitalization of strings. For example, if the string "12-ounce Can of Root Beer" occurs in a test case, it will never appear elsewhere in that same test case with different capitalization (e.g., "12-ounce can of root beer").

★ None of the reagent lists for a given potion will contain duplicate items. We will assume that if a potion requires some reagent, it needs only one of that particular reagent. (You might wonder, then, why we don't just use a set for the *reagents* field in the *PotionInfo* class. The main reason for that is that we don't want anyone to simply copy any set references passed to the required methods for this assignment. See the warning/note/rant in *PotionInfo.java* for further information about why that would be dangerous.)

3. Method and Class Requirements

Implement the following methods in a public class named *PotionMaster* in a file named *PotionMaster.java*. Please note that they are all **public** and **static**. You may implement helper methods as you see fit. Please include your name, the course number, the current semester, and your NID in a header comment at the top of your source file.

```
public static Map<String, Set<String>> potionToReagentsMap(List<PotionInfo> potionsManual)
```

Description: This method takes a bunch of potion formulas (in the form of a *PotionInfo* list called *potionsManual*) and maps potion names to the reagents required to make those potions. Effectively, you are transferring all the information from that *potionsManual* into a map that will allow for fast and easy lookup of the ingredients required to make any particular potion. The name of a potion will act as a *key* to this map, and the *value* it produces will be the set of reagents necessary to make that particular potion (in the form of a set of strings).

Each *PotionInfo* object will get its own entry in the map you create. We guarantee that there won't be any duplicate potion names in the *potionsManual.* Furthermore, we guarantee the *potionsManual* reference will not be *null*, and there won't be any *null* references anywhere in the *potionsManual* list or in the *PotionInfo* objects it contains.

Map and Set Restrictions: You must return either a *TreeMap* or a *HashMap* from this method. Also, the sets contained within the map must all be of one type – either *TreeSet* or *HashSet*. It's up to you to determine which map type and set type will be more appropriate given how this map will be used by other methods in this assignment.

Return Value: Return a map as described above.

```
public static Map<String, Set<String>> reagentToPotionsMap(List<PotionInfo> potionsManual)
```

Description: This method takes a bunch of potion formulas (in the form of a *PotionInfo* list called *potionsManual*) and maps reagents to the potions that can be made with those reagents. Effectively, you are transferring all the information from that *potionsManual* into a map that will allow for fast and easy lookup of all the potions call for a certain reagent that one has on hand. The name of a reagent will act as a *key* to this map, and the *value* it produces will be a set of potion names (in the form of a set of strings).

Each regent found in the *potionsManual* will get its own entry in the map you create. As in the previous method, we guarantee that there won't be any duplicate potion names in the *potionsManual.* Furthermore, we guarantee the *potionsManual* reference will not be *null*, and there won't be any *null* references anywhere in the *potionsManual* list or in the *PotionInfo* objects it contains.

Map and Set Restrictions: You must return either a *TreeMap* or a *HashMap* from this method. Also, the sets contained within the map must all be of one type – either *TreeSet* or *HashSet*. It's up to you to determine which map type and set type will be more appropriate given how this map will be used by other methods in this assignment.

Return Value: Return a map as described above.

```
public static boolean canBrewPotion(PotionInfo potionInfo, Set<String> reagentsOnHand)
```

Description: Given a *PotionInfo* object (*potionInfo*) that describes a potion we want to brew, as well as a list of reagents we have on hand (*reagentsOnHand*), this method determines whether we have all the necessary reagents to brew the potion in question. If so, return *true*. Otherwise, return *false*.

The following are guaranteed not to be *null*: *potionInfo*, *potionInfo.name*, *potionInfo.reagents*, all the string references contained within *potionInfo.reagents*, *reagentsOnHand*, and all the string references contained in the *reagentsOnHand* set. Basically, you shouldn't encounter any *null* references in this method.

Return Value: Return a boolean as described above.

```
public static boolean canBrewPotion(String potionToBrew,
                                    Map<String, Set<String>> potionToReagentsMap,
                                    Set<String> reagentsOnHand)
```

Description: This is an overloaded version of the *canBrewPotion()* method. This version takes the following input parameters: (1) the name of a potion we want to brew (*potionToBrew*), (2) a map (*potionToReagentsMap*) that maps potion names to sets of required reagents for those potions (i.e., a map in which a potion name is a *key* and the *value* it maps to is the set of reagents required to brew that potion), and (3) a set of strings indicating what reagents we have on hand (*reagentsOnHand*). The method returns *true* if we have all the reagents required to brew the *potionToBrew*. If we do not have all the necessary reagents, or if *potionToBrew* is not in the *potionToReagentsMap*, return *false*.

Recall that *potionToReagentsMap.get(key)* could return *null* if we pass it a key that is not mapped to anything. Other than that, you do not need to worry about encountering any *null* references anywhere in the parameters passed to this method. Furthermore, we guarantee that any potion in the *potionToReagentsMap* will require at least one reagent to be brewed, which means that we will not have any empty sets in that map.

Return Value: Return a boolean as described above.

```
public static Set<String> allPossiblePotions(Map<String, Set<String>> potionToReagentsMap,
                                             Map<String, Set<String>> reagentToPotionsMap,
                                             Set<String> reagentsOnHand)
```

Description: This method takes three parameters: (1) a map (*potionToReagentsMap*) that maps potion names to sets of required reagents for those potions, (2) a map (*reagentToPotionsMap*) that maps the names of reagents to the potions that require those reagents, and (3) a set of strings (*reagentsOnHand*) indicating what reagents we currently have on hand. Based on those parameters, this method creates and returns a set containing the names of all the potions we can brew using just the reagents we have on hand, without any need to acquire any additional ingredients.

If we cannot brew any potions with the given set of reagents on hand, this method should return an empty set (**not** a *null* reference).

Set Restriction: You must return either a *TreeSet* or a *HashSet* from this method. It's up to you to determine which one of those is most appropriate in this context.

Return Value: Return a set as described above.

```
public static double difficultyRating()
```

Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
public static double hoursSpent()
```

Return a realistic and reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

4. Special Requirement: No Compile-Time Warnings (*Super Important!*)

Here are some special restrictions regarding compile-time warnings (the same as in *BlackBoxOfChaos*):

- ★ Your code must not produce any warnings when compiled on Eustis. For this particular assignment, it's especially important not to have any -*Xlint:unchecked* warnings. Please note that some systems don't produce -*Xlint:unchecked* warnings! The safest way to check whether your code is producing such warnings is to compile and run your program on Eustis with the *test-all.sh* script.
- ★ You cannot use the *@SuppressWarnings* annotation (or any similar annotations) to suppress warnings in this assignment.

Please do not give your code to classmates and ask them to check whether their compilers generate compile-time warnings for your code. Remember, sharing code in this course is out of bounds for assignments.

Most of the information on the following pages is repeated from other assignments and is included merely for ease of reference. But please be sure to check out the special restrictions (pg. 7) and the grading notes (pg. 9).

5. Style Restrictions (*Same As Always*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.

★ Any time you open a curly brace, that curly brace should start on a new line.

★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.

★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.

★ Please avoid block-style comments: /* *comment* */

★ Instead, please use inline-style comments: // *comment*

★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"

★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed ***above*** your import statements.

★ Use end-of-line comments sparingly. Comments longer than three words should always be placed ***above*** the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.

★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.

★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.

★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)

★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.

★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.

★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

★ Do not use *var* to declare variables.

6. Special Restrictions (*Super Important!*)

You must abide by the following restrictions in this assignment. Failure to abide by certain of these restrictions could result in a catastrophic loss of points.

- ★ (***Incredibly Important!***) Your *PotionMaster* class cannot have any member variables (i.e., fields). Every variable you create for this assignment must be declared within a method.
- ★ File I/O is forbidden. Please do not read or write to any files.
- ★ Do not write malicious code. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

7. Compiling and Running All Test Cases (and the *test-all.sh* Script!)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. At the command line, whether you're working on your own system or on Eustis, you need to use the *cd* command to move to the directory where you have all the files for this assignment. For example:

   ```
   cd Desktop/potionmaster_assignment
   ```

   Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type, or the entire name needs to be wrapped in double quotes. For example:

   ```
   cd potionmaster\ assignment
   ```

   ```
   cd "potionmaster assignment"
   ```

   It's probably easiest to just avoid file and folder names with spaces.

2. To compile your program with one of my test cases:

   ```
   javac PotionMaster.java TestCase01.java
   ```

3. To run this test case and redirect the program's output to a text file:

   ```
   java TestCase01 > myoutput.txt
   ```

4. To compare your program's output against the sample output file I've provided for this test case:

```
diff myoutput.txt sample_output/TestCase01-output.txt
```

If the contents of *myoutput.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

```
seansz@eustis:~$ diff myoutput.txt sample_output/TestCase01-output.txt
seansz@eustis:~$ _
```

Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

5. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *PotionMaster.java* and all the test case files and typing:

```
bash test-all.sh
```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting. Note that this script might have limited functionality on Mac OS systems or Windows systems that aren't using the Linux-style bash shell.

8. Transferring Files to Eustis

When you're ready to test your project on Eustis, using MobaXTerm to transfer your files to Eustis isn't too hard, but if you want to transfer them using a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*PotionMaster.java*, *test-all.sh*, the test case files, and the *sample_output* folder).

2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

```
scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
```

Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

9. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *PotionMaster.java*, via Webcourses. The source file should contain definitions for all the required methods (listed above), as well as any helper methods you've written to make them work. Be sure to include your name and NID at the top of the file you submit.

10. Grading Criteria and Miscellaneous Requirements

Important Note: When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

100%	Passes test cases with 100% correct output formatting and abides by all restrictions listed above. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods. See note about additional deductions below.

(*Important!*) Additional point deductions may be imposed for violating any of the restrictions listed above or failing to meet any of the requirements set forth in this document. You should also still abide by the style restrictions and other general restrictions from the previous assignments. While points will not be *awarded* for following all restrictions and requirements listed in this PDF, points may be *deducted* for failure to do so.

(*Important!*) Your program must be submitted via Webcourses, and it must compile and run on Eustis in order to be eligible for credit. Programs that do not compile and run on Eustis will automatically receive zero credit.

11. Final Thoughts (*Abbreviated*)

★ Please be sure to submit a *.java* file (not a *.class* file, and certainly not a *.doc* or *.pdf* file).

★ After submitting to Webcourses, please consider downloading your source code, re-compiling it, and re-testing it in order to ensure that you uploaded the correct version.

★ Please remove *main()* from *PotionMaster.java*, and then double checking that your program still compiles before submitting.

★ Please do not articulate a *package* in your *PotionMaster.java* file.

★ Please be aware that the test cases included with this assignment are not comprehensive. Please be sure to create your own test cases and thoroughly test your code. Sharing test cases with other students is allowed (and encouraged!), as long as those test cases don't include any solution code for the assignment.

Start early! Work hard! Ask questions! Good luck!