Homework 2 – The Kitchen

Authors: Lucas, David, Michael, Mridul, Nicolas, Dongjae

Problem Description:

This assignment will test your knowledge of classes, instance data, encapsulation, methods, and constructors.

Wonderful aromas stream from the kitchen (note: not GT Dining) but what's the secret? Only the chefs know and they're not telling you their recipes. You're in charge of overseeing the kitchen, and you must ensure this secrecy-filled arrangement continues to run seamlessly.

Solution Description:

Note: you must follow the rules of encapsulation for all instance fields of all classes you write.

You will first create an `Ingredient` class. For a chef to keep track of his recipes, he must know two qualities of each ingredient: its `kind` and its `quantity`. An ingredient's kind can be any string of characters as long as it is not just spaces, the empty `String`, or `null`. As a hint, the String class has a method which returns a string that omits leading and trailing whitespaces (try checking the API)! An ingredient's quantity can be any positive number of grams. When creating a new ingredient for a recipe, we can let chefs specify both a kind and quantity or if they specify neither, we will just assume they mean to create an ingredient with kind "`Salt`" and quantity `0.2`. If a chef tries to create an ingredient with an illegal name or quantity, replace the illegal value(s) with "`Salt`" and `0.2`, respectively. Keep in mind that chefs may need to view and change an ingredient's kind and quantity once they are created (hint: what kinds of methods allow us to accomplish this while maintaining encapsulation?). If they attempt to change either quality into an illegal value, do not change that value.

You will also create a `Recipe` class. A recipe is more than just a collection of ingredients! A recipe has a `name`, `prepTime`, `numServings`, and finally `ingredients`. A recipe's name can be anything except just spaces, the empty `String`, or `null`. A recipe's prep time can be any positive number of whole minutes. Its number of servings, which represents how many people this recipe can serve, should also be a positive whole number. Finally, the recipe's ingredients, a collection of `Ingredient` objects, must contain at least one ingredient and can contain a maximum of 10 ingredients at a time. A chef must be able to create a recipe by giving it a name, prep time, number of servings, and an array of ingredients. However, if none of these are specified, they should be assumed to be creating a recipe with name "`Lamb Sauce`", prep time of `60` minutes, `8` servings, and with `ingredients` containing only one ingredient that is of kind "`Salt`" and quantity `0.2`. If a chef tries to create a recipe with any illegal values, replace those illegal value(s) with the values specified in the previous sentence. Additionally, since chefs can also view or change a recipe's qualities through specific methods, if they ever try to change any quality into an illegal value, do not change that value.

Thankfully, storing your recipes in Java brings many extra benefits. Among these is the ability to determine which ingredient is most prevalent in a recipe by just calling the `dominantIngredient` method on it and the specific ingredient with highest quantity in this recipe should be returned. In the

event of a tie, return the ingredient which was first considered most dominant. Additionally, a recipe can be scaled by a positive whole number factor to serve more people even though it will cost more ingredients and prep time. For example, if I `scaleRecipe` by 3 on a recipe that originally takes 15 minutes, serves 2 people, and contains ingredients Salt with quantity 0.2 grams and Lamb with quantity 4 grams, after the method call, the recipe should take 45 minutes to prep, serve 6 people, and contain 0.6 grams of Salt and 12 grams of Lamb. If a chef tries to scale a recipe by an impossible factor (see constraints above), do not change the recipe.

Furthermore, and most excitingly, you will also create a `Chef` class. A chef has a `name`, `specialty`, and keeps a collection of `recipes`. The chef's name may be any `String` which is not-empty, not comprised solely by whitespaces, and not `null`. Likewise, the chef's specialty may be any `String` which is not-empty, not comprised solely by whitespaces, and not `null`. Any chef's collection of recipes will contain at least one recipe and can contain a maximum of 10 recipes at a time. When initializing a chef, a default constructor should create a chef with name "`Gordon Ramsay`", specialty "`Lamb Sauce`", and an array of recipes which only contains the earlier described default recipe. A three-arg constructor will take in values for each of the above mentioned instance fields. In the event that *any* of the arguments received by this constructor are invalid, the default constructor values will be used for all instance fields.

Chefs can be asked to prepare any recipe in their repertoire! Given an index of a recipe in the chef's known recipes, the `cook` method prints:

"Bon Appetit! Using

<first ingredient's quantity> grams of <first ingredient's kind>

<second ingredient's quantity> grams of <second ingredient's kind>

… (for all used ingredients)

I, <chef's name>, finished cooking <recipe's name> after <recipe's prep time> minute(s). This can serve <recipe's number of servings> people."

Note: The cook method, shown above, will only be tested on arrays where the data has been stored contiguously (no gaps between entries), you may assume this property holds true

Additionally, if the chef is asked to cook a recipe which matches their specialty, the recipe's prep time reported should be half the one listed in the recipe (it is okay for this to cause prep time to become 0). If the chef is asked to cook a recipe whose index is out of bounds of his recipes or if the index requested does not contain a recipe, this method should print:

"What is this nonsense? Get out of my kitchen, I won't make that!"

Building upon the chef's previous abilities, they can also be asked to prepare food for a certain number of people, or cater. The `cater` method causes the chef to look through their recipes and cook the first (and only the first) recipe which properly serves at least the number of people given as an argument. If the chef does not find a recipe that meets the requirements, this method should print:

"My gran could do better… than me?!"

Lastly, you will create a driver `Kitchen` class. We highly encourage you to test your code in this driver class. Think about any edge cases that may cause your code to "break." The only firm requirement is that you make use of the main method to create at least one chef.

Submitting

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

- `Ingredient.java`
- `Recipe.java`
- `Chef.java`
- `Kitchen.java`

Make sure you see the message stating, "Homework 2 submitted successfully." We will only grade your last submission be sure to **submit *every file* each time you resubmit.**

Checkstyle

You must run Checkstyle on your submission. (To learn more about Checkstyle, examine our course's external website Java Resources page under "CS 1331 Style Guide".) The Checkstyle cap for this assignment is **10 points**. If you don't have Checkstyle yet, download it from Canvas → Files → Resources → checkstyle-8.28.jar. You can also find it on the course website under "Java Info." Place it in the same folder as the files you want to run Checkstyle on. 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 assignments, we will be increasing this cap, so get into the habit of fixing these style errors early!

Import Restrictions

You may not use any imports for this assignment.

Feature Restrictions

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

- var (the reserved keyword)
- System.exit

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. Only post code on Ed Discussion in a **private** post.

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 "Import Restrictions" and “Feature Restrictions" to avoid losing points
- Check on Ed Discussion for all official clarifications
    - Make sure to test your program manually based on the assignment instructions and expected outputs.

The Gradescope autograder visible to you is **NOT** comprehensive. A few test cases will be hidden when you submit. This is done intentionally so that you learn to write your own test cases for your assignments.