Programming Exercise 04 – Coffee Shop

Authors: Brittney, Ariel, Akul, Jacob, Aqdas, Vince

Problem Description

This programming exercise is to help you get comfortable with Enums, static methods, and arrays. To do this, we want you to help the Willage Starbucks. Most days, there is only one poor barista to tackle rush hour, and a lot of coffee orders. We want your help to prove to the managers that more than one barista is needed.

Solution Description

You will have two files: `Order.java` and `CoffeeShop.java`. The `Order.java` will be an enumeration of all the possible orders and how long it takes to make each of them. `CoffeeShop.java` will take in a string of orders, convert the input to an array of `Order` enumeration values, and calculate the time it will take x number of baristas to complete them.

Order.java

1. Create an enumeration (`enum`) named Order with these values:
    a. LATTE
    b. COFFEE
    c. ICED_COFFEE
    d. FRAPPE
    e. PASTRY

CoffeeShop.java

1. Create a static method called `createOrderArray`. This function should take in a `String` as a parameter and return an array of `Orders`.
    a. The string will be formatted like “LCIFPLLP”, each letter representing a different order. There will always be at least 1 letter in the input. For example, “LCIFPLLP” would mean: **L**ATTE, **C**OFFEE, **I**CED_COFFEE, **F**RAPPEE, **P**ASTRY, **L**ATTE, **L**ATTE and **P**ASTRY.
    b. Create an array of `Orders`, and fill it with the respective `Order enum` values according to
        - L – LATTE
        - C – COFFEE
        - I – ICED_COFFEE
        - F – FRAPPE
        - P – PASTRY

For the example above (the string "LCIFPLLP"), the array should contain eight orders, with a LATTE first, COFFEE second, ICED_COFFEE third, FRAPPEE fourth, PASTRY fifth, LATTE sixth, etc.

c. Return the newly created `array`.

2. Create a static method named `lookupMakeTime` that takes in an `Order enum` value and **using a switch statement** returns a `double` representing how long that type of coffee takes to make.

   a. LATTE: 3.0

   b. COFFEE: 0.5

   c. ICED_COFFEE: 2.0

   d. FRAPPE: 6.0

   e. PASTRY: 3.0

3. Create a static method called `computeOrderMakeTime`. This function should take in an `array` of `Orders` and an `int` called `numBaristas` representing the number of baristas. It will not return anything.

   d. Calculate the amount of time it will take the passed in number of baristas to make the orders using the following formula:

      i. (totalMakeTime / numBaristas) + (numOrders % numBaristas)

      ii. Where `totalMakeTime` is the sum of every Order's make, `numBaristas` is the number of baristas passed in, and `numOrders` is the number of orders.

      iii. This function should call `lookupMakeTime` to calculate the `totalMakeTime`

   e. Print out this result as "`It will take {time} minutes for {numBaristas} baristas to make these orders.`" where `{time}` is the double value calculated from the formula above and `{numBaristas}` is the value of the function's int parameter.

2. In the main method:

   a. Welcome your user to the coffee shop by printing "`Welcome to your local coffee shop! What does the rush look like today?`" followed by a new line to the console.

   b. Create a `Scanner` to take in their order input using the `nextLine()` method.

   c. Call the `createOrderArray` method to get an array of Orders.

   d. Call the `computeOrderMakeTime` method three times with 1, 2, and 3 as their respective parameters.

Example Outputs

User input is **bolded**. Please make sure to follow the exact formatting as shown in the pdf.

```
Welcome to your local coffee shop! What does the rush look like today?
```

LCPPFIC

```
It will take 18.0 minutes for 1 baristas to make these orders.
```

```
It will take 10.0 minutes for 2 baristas to make these orders.
It will take 7.0 minutes for 3 baristas to make these orders.
```

Checkstyle and Javadocs

You must run checkstyle on your submission. The checkstyle cap for this assignment is 3. If you don't have checkstyle yet, download it from Canvas. Place it in the same folder as the files you want checkstyle. 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 points we would take off (limited to the amount we specified 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!

Allowed Imports

To prevent trivialization of the assignment, you may only import `java.util.Scanner`.

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`

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.

Turn-In Procedure

Submission

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

- Order.java
- CoffeeShop.java

Make sure you see the message stating "PE04 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:

- Prevent upload mistakes (e.g. non-compiling code)
- 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
- Run Checkstyle on your code to avoid losing points
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications