CodeCrunch

Home | My Courses | Browse Tutorials | Browse Tasks | Search | My Submissions | Logout   Logged in as: **e1053625**

CS2030 (2320) Practical Assessment #1

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Barista Ro'Bro

Our favourite Cool-Spot at COM2 has decided to ride on the robot-barista bandwagon by engaging Ro'Bro as an additional helper. Ro'Bro's job is to serve up a variety of hot drinks to students and staff who require a daily fix to maintain their function (and also sanity) throughout the day. Moreover, in line with our government's Nutri-Grade initiative to "grade" our drinks, each drink will be labeled with its nutrition content.

Task

Your task is to write a program to create various types of drinks using different ingredients and toppings, while also determining the corresponding nutrition content of each drink.

Take note!

There are altogether five levels. You are required to complete ALL levels.

You should keep to the constructs taught in class, as well as the programming discipline instilled throughout the course.

- Write each `class/interface` in its own file; you are NOT allowed to use `enum` or `record`.
- Ensure that ALL declarations of object properties and class constants include the `private` and `final` modifiers.
- Ensure that your classes are NOT cyclic dependent.
- You are NOT allowed to use java libraries, other than those from `java.lang`, `java.util.List`, `java.util.Comparator` and `java.util.Iterator` or as specified in the question.
- If necessary, use only the `ImList` class and `Pair` class provided; do NOT create your own.
- `instanceof` can only be used in a class to check `instanceof` the same class, e.g. within a typical `equals(Object)` method.
- You are NOT allowed to use java keywords/literals: `null`.
- You are NOT allowed to use Java reflection, i.e. `Object::getClasses` and other methods from `java.lang.Class`
- You are NOT allowed to use `*` wildcard imports.
- You are NOT allowed to use explicit typecasts to convert between types.
- You are NOT allowed to define anonymous inner classes or lambdas.
- You are NOT allowed to define array constructs, e.g. `String[]` or using ellipsis, e.g.`String...`

You may assume that all tests provide valid arguments to the methods; hence there is no need to validate method arguments.

Level 1

The following `Drink` interface is provided. You are NOT ALLOWED to modify this interface.

```
interface Drink {
    public Nutri value();
}
```

The `Drink` interface specifies a `value` method that takes no argument, and returns the nutrition content of the drink encapsulated in a `Nutri` object. Define the `Nutri` class with one constructor that takes in a `String` followed by an integer so as to create a `Nutri` object comprising one nutrient with its corresponding amount in milligrams.

Define the classes `Tea` and `Coffee` to represent plain black tea and plain black coffee. Include the `value` method to return the nutrition content in milligrams: 100mg of caffeine for coffee; 50mg of caffeine for tea.

Include an appropriate `toString` method that returns a `String` representation of the drink in lowercase.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Nutri("caffeine", 10)
$.. ==> [(caffeine, 10)]

jshell> new Coffee()
$.. ==> coffee

jshell> new Coffee().value()
$.. ==> [(caffeine, 100)]

jshell> Drink drink = new Tea()
drink ==> tea

jshell> Nutri nutri = new Tea().value()
nutri ==> [(caffeine, 50)]
```

In this level, your program will be tested against other drinks. You may assume that each drink contains only one nutrient (not necessarily caffeine) with its corresponding amount in milligrams.

Level 2

We can add sugar to either coffee or tea.

Write a `Sugar` class that takes in a coffee or tea (or any base drink) and returns the drink with sugar added. The amount of sugar added is a constant 8 grams (or 8000 mg) per drink. Note that while sugar is an ingredient, it also contains sugar as a nutrient.

Since tea (or coffee) with sugar is itself a drink, it will have a nutrition content that includes both sugar and caffeine.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Sugar(new Coffee())
$.. ==> sugar

jshell> new Sugar(new Coffee()).value()
$.. ==> [(caffeine, 100), (sugar, 8000)]

jshell> Drink drink = new Sugar(new Tea())
drink ==> sugar

jshell> drink.value()
$.. ==> [(caffeine, 50), (sugar, 8000)]
```

The nutrition content should be output by alphabetic order of the nutrient. In the above example, `"caffeine"` is placed before `"sugar"` in the output nutrient list. You may assume that all nutrients are lowercase.

Moreover, for simplicity, the `String` representation of `new Sugar(..)` should remain as only `"sugar"`, leaving out the rest of the components of the drink.

In this level, your program will be tested against drinks with or without sugar added. The base drinks will not contain sugar.

Level 3

Other than sugar as an ingredient, we can add further ingredients such as milk. Using coffee as an example, we can now make additional varieties of coffee.

- plain coffee with no milk and no sugar
- coffee with milk only
- coffee with sugar only
- coffee with milk and sugar

To construct additional drinks out of various ingredients, writing a class for each combination can become cumbersome. Instead, we need only write classes for each ingredient, but create combinations of drinks on-the-fly (or dynamically), e.g. `new Milk(new Sugar(new Coffee()))`.

Now include the `Milk` class. The portion of milk mixed into coffee or tea has a nutrition content of 2680 mg of fat and 4340 mg of sugar. Since milk has more than one nutrient, include an `update` method in the `Nutri` class that allows the nutrient content to be updated. It will also be useful to consider updating the same nutrient by adding the amounts.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Nutri("fat", 1).update("sugar", 2).update("caffeine", 3).update("sugar",4)
$.. ==> [(caffeine, 3), (fat, 1), (sugar, 6)]

jshell> new Milk(new Coffee())
$.. ==> milk

jshell> new Milk(new Coffee()).value()
$.. ==> [(caffeine, 100), (fat, 2680), (sugar, 4340)]

jshell> Drink drink = new Milk(new Tea())
drink ==> milk

jshell> drink.value()
$.. ==> [(caffeine, 50), (fat, 2680), (sugar, 4340)]

jshell> new Sugar(new Milk(new Coffee()))
$.. ==> sugar

jshell> new Sugar(new Milk(new Coffee())).value()
$.. ==> [(caffeine, 100), (fat, 2680), (sugar, 12340)]

jshell> new Milk(new Sugar(new Tea()))
$.. ==> milk

jshell> new Milk(new Sugar(new Tea())).value()
$.. ==> [(caffeine, 50), (fat, 2680), (sugar, 12340)]
```

In this level, your program will be test against more types of drinks other than just combinations of tea, coffee, sugar and milk. Specifically,

- each base drink may contain multiple nutrients;
- each ingredient may contain multiple nutrients;
- base drinks and ingredients may contain overlapping nutrients;
- each base drink and ingredient will not be used more than once.

Level 4

Just like ingredients, we can add toppings to a drink. We shall illustrate with two toppings: cinnamon and cream.

- cinnamon: 220mg of fat; 150mg of sugar
- cream: 10000mg of fat; 2000mg of sugar

Write the `Cinnamon` and `Cream` classes to represent the two toppings. We can now create further combination of drinks. You may assume that each ingredient/topping will not be used more than once.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> new Milk(new Cinnamon(new Tea())).value()
$.. ==> [(caffeine, 50), (fat, 2900), (sugar, 4490)]

jshell> new Cream(new Sugar(new Coffee()))
```

```
$.. ==> cream

jshell> new Cream(new Sugar(new Coffee())).value()
$.. ==> [(caffeine, 100), (fat, 10000), (sugar, 10000)]

jshell> new Cinnamon(new Milk(new Cream(new Sugar(new Coffee()))))
$.. ==> cinnamon

jshell> new Cinnamon(new Milk(new Cream(new Sugar(new Coffee())))).value()
$.. ==> [(caffeine, 100), (fat, 12900), (sugar, 14490)]
```

Although toppings and ingredients look similar, during preparation ingredients need to be added first into an empty cup, then the drink is poured into the cup, and finally topped up with toppings.

We now upgrade the `Drink` interface to the following.

```
interface Process {
    public String prep();
}

interface Drink extends Process { // please modify Drink.java with "extends Process"
    public Nutri value();
}
```

DO NOT modify these two interfaces from now on.

We illustrate with an example of `Tea` with `Milk` and `Cinnamon` where milk is added first, followed by tea, and then topped up with cinnamon. Note that the preparations below are the same.

```
jshell> new Milk(new Cinnamon(new Tea())).prep()
$.. ==> "milk>[tea]<cinnamon"
jshell> new Cinnamon(new Milk(new Tea())).prep()
$.. ==> "milk>[tea]<cinnamon"
```

Specifically, append (add to the back) each addition of an ingredient with `>`, and prepend (add to the front) each addition of a topping with `<`. The base drink is surrounded by square brackets `[..]`.

The example below illustrates the order of adding multiple ingredients and toppings. Ingredients are added from left to right, but toppings are added from right to left.

```
jshell> new Cinnamon(new Milk(new Cream(new Sugar(new Coffee())))).prep()
$.. ==> "milk>sugar>[coffee]<cream<cinnamon"

jshell> new Cream(new Milk(new Cinnamon(new Sugar(new Coffee())))).prep()
$.. ==> "milk>sugar>[coffee]<cinnamon<cream"
```

Level 5

With the introduction of Ro'Bro, Cool-Spot now sees a never-ending queue of students. University Health Service is now deeply concerned about the nutritional content of drinks and their effects on the health of students. They have decided to conduct an audit to ascertain the health effects.

Write an `Audit` class with a method `add` that takes in a `Drink` and tracks the grade of the drink using a frequency table. The grade of a drink is determined as follows:

Grade	Sugar	Fat
A	up to 2400mg	up to 1680mg
B	>2400 to 12000mg	>1680 to 2880mg
C	>12000 to 24000mg	>2880 to 6720mg
D	>24000mg	>6720mg

Every drink added to `Audit` will be tracked as a data point in a frequency table. This table is returned via the `toString` method of `Audit`.

```
$ javac your_java_files
$ jshell your_java_files_in_bottom-up_dependency_order
jshell> Drink drink = new Cinnamon(new Milk(new Cream(new Sugar(new Coffee()))))
drink ==> cinnamon

jshell> drink.value().grade()
$.. ==> "D"

jshell> new Audit().add(new Coffee()).add(new Tea()).add(drink)
$.. ==> [(A, 2), (B, 0), (C, 0), (D, 1)]
jshell> /exit
```

In the above, two grade "A" drinks, and one grade "D" drink have been added.