Inheritance & Abstract Class

Exercise 1

You are hired by a 5\* restaurant as their software architect (i.e., you design the whole software system, not just implement code). The first task you need to do is to create a collection of classes/objects to manage their food management and producing processes.

All food contains different amounts of ingredients. So, creating an ingredient class is needed.

Note: even for experienced software designers (not graphics designers) and business analysts, it's almost impossible to know in advance all details of the systems they are developing. So, keep in mind that your designed classes/objects will be changed in the future. As an architect, try to design your system so that future changes do not require you to change much current implementations.

Create an **Ingredient** class. You need to store the name (String) and measurement unit (String) of each ingredient.

Some example instances of Ingredient:

* name: water, measurement unit: spoon
* name: water, measurement unit: teaspoon
* name: sugar, measurement unit: mg
* name: chicken, measurement unit: kg

You need to decide the class' required methods/fields to support the above needs. Remember to encapsulate the data fields.

Implement a **Recipe** class to manage food recipe for the restaurant. A recipe has a name (String) and an array of components. Each component is an instance of **RecipeComponent** class. Each RecipeComponent instance, in turn, contains an Ingredient instance and a floating number amount.

For example, this Recipe instance represents a scone recipe (don't ask me how I know it, I got it from Google)

scone (Recipe)

{

[c1, c2, c3]

}

---

c1 (RecipeComponent)

{

ingredient: {name: "self-rising flour", measurement unit: "g"}

amount: 500

}

---

c2 (RecipeComponent)

{

ingredient: {name: "chilled lemonade", measurement unit: "cup"}

amount: 1

}

---

c3 (RecipeComponent)

{

ingredient: {name: "thickened cream", measurement unit: "ml"}

amount: 300

}

Exercise 2

At the moment, to create an ingredient instance, your program needs to have knowledge about your ingredients' internals. Later, if the restaurant owner decides to add more fields to the ingredient class, then you need to change not only the Ingredient class but also the programs that use it. That's not a good design.

Redesign your Ingredient class so that programs that use it do not need to know its structure (in other words, those programs do not need to know that each ingredient has a name and a measurement unit).

To do that, you need to implement 2 methods: createIngredient() and displayIngredient() for Ingredient class. Programs that use Ingredient class only need to know how to use these 2 new methods.

Exercise 3

Now, the restaurant owner decides to add a new field to provide a useful comment (String) about an ingredient instance. This can help chefs (especially new ones) in their preparation process.

An example of a new ingredient instance is {name: "sugar", unit: "mg", comment: "if no sugar is available, use salt instead"}

Whenever you create a new Recipe object, you want to keep track of how many recipes you have created so far. As you can see, this information is not related to any particular recipe, so it should not be stored as an instance variable. Add a new static variable "count" to the Recipe class to handle this task. Moreover, it is nonsense to modify this value from the outside. So, make sure that outside programs can only know how many recipes are there, but not able to change it.

Exercise 4

Now, we need to display the recipes to viewers. Create a class RecipeDisplay. This class is responsible for displaying recipe information. After doing market research, you know that there are 3 attractive ways to display one recipe object:

* For casual viewers, display the recipe components in the recipe in any order. In other words, you just need to iterate the RecipeComponent array list and display the components in that order. Name it RecipeCasualDisplay
* For serious viewers, display the recipe components in the recipe based on the sorted order of amount values. In other words, the component with smallest 'amount' value must be displayed first. Name it RecipeSeriousDisplay
* For hackers (like us), display the recipe components in the recipe based on the sorted order of the first digit of 'amount' values. In other words, the recipe component whose amount is 123 must be displayed before another recipe component whose amount value is 99. Name it RecipeHackerDisplay

Implement the parent class RecipeDisplay and the child classes RecipeCasualDisplay, RecipeSeriousDisplay, and RecipeHackerDisplay according to the specifications.

Hint: to sort objects, you can use the ArrayList.sort() or Arrays.sort() or Collections.sort() method. In this method call, you provide a Comparator object. The comparator object is responsible for determining which object in the array list is bigger than other according to its own rule.

The comparator must implement the Comparator interface. For now, that means you need to implement the method compare() that returns a positive, negative, or zero (integer). The return value decides if the first object is greater, smaller, or equal to the second object respectively.

For example, assume you have a class Student. And you want to sort students based on their gpa. You can use the sample code below

class Student {

  public String studentName;

  public double gpa;

}

class StudentByGPA implements Comparator<Student> {

  public int compare(Student s1, Student s2) {

    if (s1.gpa < s2.gpa) {

      return -1;

    }

    if (s1.gpa > s2.gpa) {

      return 1;

    }

    return 0;

  }

}

If you have an array list students pointed to by the variable 'students', you can sort them as

students.sort(new StudentByGPA());

Exercise 5

After doing another market research (yes, you have a big budget for market research), you learn that for each type of users, there are 3 preferred ways to separate the recipe components when displaying them:

* Space separator: use a blank line to separate 2 recipe components
* Dash separator: use an all-dashed line (i.e., -------) to separate 2 recipe components
* Binary separator: use '0' and '1' digits alternatively (i.e., 010101) to separate 2 recipe components

How can you extend the code in exercise 4 to include the new requirements?

You can implement 9 child classes (3 types of users x 3 types of separators), but it is a lot of work.

Can you update the parent class RecipeDisplay to accept a Separator object? Then extend the Separator to some child classes? Use the diagram below to think about that technique

Diagram

Description automatically generated with medium confidence

As the RecipeDisplay and Separator class is not a concrete class, make them and the corresponding methods abstract to prevent users from instantiating them as well as to force the subclasses to override those abstract methods.

After doing those exercises, you can see that OOP can help you manage the changes/new requirements easier. Even though you can use IF statements for the above exercises, it's very difficult to update all the IF logic when there are many new/updated requirements.

## Exercise 6 (Additional)

Write a regular expression and give the corresponding automata for each of the following sets of binary strings. Use only the basic operations.

a) 0 or 11 or 101

b) only 0s

c) all binary strings

d) all binary strings except empty string

e) begins with 1, ends with 1

f) ends with 00

g) contains at least three 1s

h) contains at least three consecutive 1s

i) contains the substring 110

j) doesn't contain the substring 110

k) has at least 3 characters, and the third character is 0

l) starts and ends with the same character

m) starts with 0 and has odd length, or starts with 1 and has even length

n) length is at least 1 and at most 3

