Exercise for Chapter 11 And 12

Introductory Programming 2020

This week there are a lot of exercises. So you probably won't be able to finish all of them, which is okay. This is a recommendation of exercises we think you should look at, if you have the time.

Chapter 11

When solving these exercises open your modified version of the `network-v2` from last week's exercises (week 40). If you have not done any modifications, just open the code provided.

Exercise 11.1

Open your last version of the `network project`. (You can use `network-v2` if you do not have your own version yet.) Remove the `display` method from class `Post` and move it into the `MessagePost` and `PhotoPost` classes. Compile. What do you observe?

Exercise 11.2

In your `network project`, add a `display` method in class `Post` again. For now, write the method body with a single statement that prints out only the username. Then modify the `display` methods in `MessagePost` and `PhotoPost` so that the `MessagePost` version prints out only the message and the `PhotoPost` version prints only the caption. This removes the other errors encountered above (we shall come back to those below).

You should now have a situation corresponding to Figure 11.4, with `display` methods in three classes. Compile your project. (If there are errors, remove them. This design should work.)

Before executing, predict which of the `display` methods will get called if you execute the news feed's `show` method.

Try it out. Enter a message post and a photo post into the news feed and call the news feed's `show` method. Which `display` methods were executed? Was your prediction correct? Try to explain your observations.

Exercise 11.3

Modify your latest version of the `network project` to include the super call in the `display` method. Test it. Does it behave as expected? Do you see any problems with this solution?

Own defined exercise

What is the purpose of the `toString()` method? What can it be used for?

Exercise 11.6

The version of `display` shown in Code 11.2 produces the output shown in Figure 11.9. Reorder the statements in the method in your version of the `network project` so that it prints the details as shown in Figure 11.10. (check the book for the figures and code)

Exercise 11.7

Having to use a superclass call in `display` is somewhat restrictive in the ways in which we can format the output, because it is dependent on the way the superclass formats its fields. Make any necessary changes to the `Post` class and to the `display` method of `MessagePost` so that it produces the output shown in Figure 11.11. Any changes you make to the `Post` class should be visible only to its subclasses. Hint: You could add protected accessors to do this.

Exercise 11.11

Assume that you see the following lines of code:

```
Device dev = new Printer();
dev.getName();
```

`Printer` is a subclass of `Device`. Which of these classes must have a definition of method `getName` for this code to compile?

Exercise 11.12

In the same situation as in the previous exercise, if both classes have an implementation of `getName` , which one will be executed?

Exercise 11.13

Assume that you write a class `Student` that does not have a declared superclass. You do not write a `toString` method. Consider the following lines of code:

```
Student st = new Student();
String s = st.toString();
```

Will these lines compile? If so, what exactly will happen when you try to execute?

Exercise 11.14

In the same situation as before (class `Student`, no `toString` method), will the following lines compile? Why?

```
Student st = new Student();
System.out.println(st);
```

Exercise 11.15

Assume that your class `Student` overrides `toString` so that it returns the student's name. You now have a list of students. Will the following code compile? If not, why not? If yes, what will it print? Explain in detail what happens.

```
for(Object st : myList) {
    System.out.println(st);
}
```

Chapter 12

If you are unsure about the `foxes-and-rabbits` project you can either read through the code or the explanation in chapter 12. Also be sure to read the `README.txt` in each folder.

For the following exercises use the code in folder `foxes-and-rabbits-v1`.

Exercise 12.28

Identify the similarities and differences between the `Fox` and *Rabbit* classes. Make separate lists of the fields, methods, and constructors, and distinguish between the class variables (static fields) and instance variables.

Exercise 12.29

Candidate methods for placement in a superclass are those that are identical in all subclasses. Which methods are truly identical in the `Fox` and `Rabbit` classes? In reaching a conclusion, you might like to consider the effect of substituting the values of class variables into the bodies of the methods that use them.

Exercise 12.30

In the current version of the simulation, the values of all similarly named class variables are different. If the two values of a particular class variable (`BREEDING_AGE`, say) were identical, would it make any difference to your assessment of which methods are truly identical?

Exercise 12.32

The `Randomizer` class provides us with a way to control whether the "random" elements of the simulation are repeatable or not. If its `useShared` field is set to `true`, then a single `Random` object is shared between all of the simulation objects. In addition, its `reset` method resets the starting point for the shared `Random` object. Use these features as you work on the following exercise, to check that you do not change anything fundamental about the overall simulation as you introduce an `Animal` class.

Create the `Animal` superclass in your version of the project. Make the changes discussed above. Ensure that the simulation works in a similar manner as before. You should be able to check this by having the old and new versions of the project open side by side, for instance, and making identical calls on `Simulator` objects in both, expecting identical outcomes.

Exercise 12.33

How has using inheritance improved the project so far? Discuss this.

For the following exercises use the code in folder `foxes-and-rabbits-graph`.

Exercise 12.45

Move the `age` field from `Fox` and `Rabbit` to `Animal` . Initialize it to zero in the constructor. Provide accessor and mutator methods for it and use these in `Fox` and `Rabbit` rather than in direct accesses to the field. Make sure the program compiles and runs as before.

Exercise 12.46

Move the `canbreed` method from `Fox` and `Rabbit` to `Animal` , and rewrite it as shown in Code 12.8. Provide appropriate versions of `getBreedingAge` in `Fox` and `Rabbit` that return the distinctive breeding age values.

Exercise 12.47

Move the `incrementAge` method from `Fox` and `Rabbit` to `Animal` by providing an abstract `getMaxAge` method in `Animal` and concrete versions in `Fox` and Rabbit.

Exercise 12.48

Can the `breed` method be moved to `Animal` ? If so, make this change.

For the following exercises no code is needed.

Exercise 12.35

Is it necessary for a class with one or more abstract methods to be defined as abstract? If you are not sure, experiment with the source of the `Animal` class in the `foxes-and-rabbits-graph` project.

Exercise 12.36

Is it possible for a class that has no abstract methods to be defined as abstract? If you are not sure, change `act` to be a concrete method in the `Animal` class by giving it a method body with no statements.

Exercise 12.37

Could it ever make sense to define a class as abstract if it has no abstract methods? Discuss this.

Exercise 12.58

Which methods do `ArrayList` and `LinkedList` have that are not defined in the List interface? Why do you think that these methods are not included in List ?

Exercise 12.68

Can an abstract class have concrete (non-abstract) methods? Can a concrete class have abstract methods? Can you have an abstract class without abstract methods? Justify your answers.

For the following exercises use the code in folder `foxes-and-rabbits-graph`.

These are challenge exercises

Exercise 12.51

Define a completely new type of animal for the simulation, as a subclass of `Animal` . You will need to decide what sort of impact its existence will have on the existing animal types. For instance, your animal might compete with foxes as a predator on the rabbit population, or your animal might prey on foxes but not on rabbits. You will probably find that you need to experiment quite a lot with the configuration settings you use for it. You will need to modify the `populate` method to have some of your animals created at the start of a simulation.

When you have defined your new animal class and added it to the population you should look in the simulator at line 65 and 71. Here you will see the following code out-commented:

```
// view.setColor(Your.class, Color.GREEN);
```

Replace the *Your* with the name of your class. For example if you have made new type of animal called `Tiger`, then it would look like this:

```
view.setColor(Tiger.class, Color.GREEN);
```

Now you should be able to run the simulation and see your custom defined class interacting with the others.

Exercise 12.53

Introduce the `Actor` class into your simulation. Rewrite the `simulateOneStep` method in `Simulator` to use `Actor` instead of Animal. You can do this even if you have not introduced any new participant types. Does the `Simulator` class compile? Or is there something else that is needed in the `Actor` class?

Exercise 12.54

Redefine as an interface the abstract class `Actor` in your project. Does the simulation still compile? Does it run? Make any changes necessary to make it runnable again.