Space exploration

Programming Fundamentals 2

Goals

- ✯ Understanding the concept of *inheritance*.
- ✯ Use subtype polymorphism with the mechanism of *overriding*.
- ✯ **Relevant video**:
  - Principles of object-oriented programming.

Deliverables

1. The code on your Github repository generated by clicking here: `https://classroom.github.com/a/3mihXf-r`
2. **Reviewer**: Maria Hartmann (mhart01 on Github).
3. **Documentation**: Comment the methods you write using JavaDoc.

Exercise 1 – Using a build automation tool (Maven)

For this laboratory, you compile and execute your code with *Maven*. Install Maven with `sudo apt install maven` on Windows or Ubuntu in the terminal, and with `brew install maven` on OSX. We already provide the file `pom.xml` at the root of the project, which describe how the code is compiled and executed.

- To compile, write `mvn compile` at the root of your project.
- To run the program, write `mvn exec:java -Dexec.mainClass="exploration.Exploration"`.

Exercise 2 – To explore strange new worlds

You are a sentient computer on a small spaceship of explorers travelling around a newly discovered planetary system. Because of complicated trajectory dynamics, the order in which planets are visited is fixed, but you just cannot figure out the reasoning behind your crews' exploration missions once they get to each planet. Sometimes they barely take one look before moving on, and sometimes they insist on exploring every bit of the planet - humans are so confusing! You decide to make a model of your crew to figure out what they might do next.

1. You start off by modelling the planets. You make a class `Planet` and you give it two numerical attributes: `danger` and `novelty`. You believe that these are the attributes based on which your crew members decide whether to explore a planet. Write a constructor initialising these attributes. Also write two methods `public int dangerLevel()` and `public int noveltyLevel()` in this class, returning the respective value.

2. Then, you make three classes modelling the role of each crew member: the captain, the scientist and the medic. (Each class goes into a separate file, i.e. Captain.java, Scientist.java and Medic.java.) Each of them has a method `public boolean isInterested(Planet planet)` that decides for a given planet whether or not they would like to explore it. The crew members make these decisions differently:

   - The scientists do not care about the danger of exploring the planet; their level of interest depends only on the novelty of the planet.
   - The captain is a bit of a thrill-seeker, so more danger on a planet actually makes them more interested in exploring it.
   - The medics are, in your opinion, the most reasonable of the three: they are more interested in higher-novelty and lower-danger planets.

   The output of this method should be a yes or no (Boolean) value. You can for example model the behaviour of the different crew members by introducing `dangerThreshold` and `noveltyThreshold` variables for each of them (the values of these should be easily changeable across instances).

3. In the file Exploration.java, instantiate all five crew members (1 captain, 2 medics, and 2 scientists) and a planet in the `main` method. Call the `isInterested` method on all five crew members and print the result.

4. You notice that the crew member classes have something in common: they all share the same method `isInterested`. Therefore, you introduce a new abstract class `CrewMember`, move the code common to all crew members into this class and have the separate types of crew member inherit it.

5. You decide to extend this model to allow for a variable number of crew members on the ship. With more crew members, more of a command structure is needed, so you introduce a Boolean attribute `isOfficer`.
   All captains have officer rank, but some of the scientists or medics may be officers as well.

   It is a good opportunity to use *overloading* by having two constructors in the classes `Medic` and `Scientist`: a default one, and one specifying the attribute `isOfficer`. For instance, we will have:

   ```
   public Scientist(int danger, int novelty) { ... } // initialize 'isOfficer' to 'false'.
   public Scientist(int danger, int novelty, boolean isOfficer) { ... }
   ```

   So calling `scientist=new Scientist(10, 10)` will invoke the first constructor, and calling `scienceOfficer=new Scientist(10, 10, true)` will invoke the second one.

6. Now you create a class `Crew` to represent the entire crew of the spaceship. This class should be usable as follows to decide whether the crew will explore a given planet:

   ```
   Crew crew = new Crew(8);
   if(crew.wantToExplorePlanet(planet)) { ... }
   ```

   where 8 is the total number of crew members (but could be any other value), and `wantToExplorePlanet` takes an object of type `Planet` and returns a Boolean value indicating the joint decision of all eight crew members about exploring the given planet. The decision pattern is as follows:

   - The officers are in command. Because your crew is exceptionally good at working together, decisions within the officer class are made essentially democratically. This means that whenever a majority of the officers is interested in exploring a planet, the crew will explore it.
   - If more than two thirds of the full crew is interested, they will convince the officers and the planet will be explored.
   - If neither of these conditions is satisfied, the crew will not explore the planet.

7. Implement an example instantiation of the model using at least two different planets and a crew of seven or more explorers. Have the crew decide whether to explore the different planets and print the results in the console. Think about what your implementation does in case of edge cases, e.g. if the crew consists of only scientists, has no officers, and so on.