In-class exercises (lecture 5)

Exercise 1: Overloading the Triangle constructor

Open up the shapes package for this week and open the new Triangle class. You will see that it already has two constructors, one that uses the default reference point and one that allows the client to specify a reference point.

Currently, both constructors require the client to specify the length of all three sides and the size of all three internal angles. If you remember your school geometry class, you'll know that you don't need to specify all six of those parameters to define a valid triangle. Only three of the six (must include at least one side) are actually required—the remaining sides / angles can be worked out using geometry.

Your task is to design some overloaded constructors that would allow the client to create a triangle object by providing fewer arguments e.g. one side and two angles and any other permutations you can think of.

When you have some ideas, try implementing the overloaded constructors in IntelliJ—just add the constructor signatures (constructor name, parameters) but leave each constructor empty. The goal here is to explore constructor overloading, not refresh your geometry knowledge. You ***will*** run into problems as you work. Try to figure out the cause of these problems!

Exercise 2: Static vs. dynamic types

- Create a test class for the `Animal` interface.
- Create one test object for each species (or at least a couple of them). Each test object should have `Animal` as the static type. E.g. `Animal testCat;`
- Write tests to verify that the `talk` method returns the appropriate String for each animal.

Exercise 3: Sub type polymorphism

In the lecture 5 code, `animals` package, there are five concrete animal classes: `Cat`, `Dog`, `Cow`, `Chicken`, and `Tiger`. There is also a `PetOwner` class, which stores a single `Pet`. Currently, the pet can be any instance of `Animal`, including species that aren't appropriate pets, like tigers. Modify the code so that:

1. `PetOwner` can only have appropriate subclasses of `Animal` as pets and,
2. `PetOwner` can have an unlimited number of pets, which can be a mix of species e.g. 5 `Cats` and a `Dog`.
3. `PetOwner` has a method, `adoptPet`, that adds an animal to the owner's collection.

For the purposes of this exercise, assume that, of the current concrete animal subclasses, only cats and dogs would be classified as appropriate pets but additional pet species may be added in future (e.g. Hamster, Rabbit, Goldfish). To keep things simple, you can also assume that, if the animals were to be further categorized (e.g. farm animals, wild animals), a species would only belong to one category.

Excluding `equals` methods, you should have no conditional statements and you should not use `instanceof` or `getClass`!

It is highly recommended that you work in groups. Spend around 10 - 15 minutes planning your design, then start coding. Revisit your design as necessary.

Exercise 4: Sub type polymorphism ctd.

We may not have time for this during class. If that's the case, you can use this as optional practice outside of class. A sample solution will be posted to lecture-code.

The starter code for this exercise is in the `securitysystem` package, which contains part of a (not very secure) security system for an office building. The package already contains a class representing a `Door` in the building, and an interface, `IStaff`, which outlines common functionality for all staff that work in the building. Take a moment to read through the code.

- Implement `IStaff` as you see fit. There are two types of staff member: those that automatically have keys to all doors in the building, and those that have restricted access—they only have keys for doors that they're specifically allowed to access. `IStaff` has a required method, `hasKey`, which should return true if the staff member has access to a given door and false otherwise.
- All staff members (classes inheriting `IStaff`) should have a String ID and a String name, along with getters and setters as you see fit. You may like to have an abstract class in between the interface and concrete classes to handle these common properties and methods.
- In the `Door` class, complete the unlock method as described in its Javadoc. If the `IStaff` object passed to the method has a key for the `Door`, the method should return true (even if the door was already unlocked) and set the `locked` and `lastUnlockedBy` fields appropriately. If the `IStaff` object passed to the method does not have a key, return false.

As with exercise 3, design your solution to take advantage of sub type polymorphism. This means you should not need to use `instanceof` or `getClass` outside of equals!