Pre-Lab 6

These exercises are meant to help prepare you for Lab 6 (or help with review). They are completely optional. Some information provided here may be redundant given the information in the lab, but none of the actual lab problems will be reproduced here.

Main concept: Inheritance

Inheritance - If you can, you should do the Head First Java reading, chapters 7 and 8. An especially useful page is page 168. You can also look at the official Java tutorial on inheritance https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html. And here are some questions and exercises: https://docs.oracle.com/javase/tutorial/java/IandI/QandE/inherit-questions.html. However, there's a lot there in both cases, so here's a slightly more condensed review and explanation, which will hopefully get you ready for Lab 6.

Underlying problems:

- Given a particular class, can we write a similar class with some added functionality without copying and pasting all the code from the first class?
- If we have a group of related classes, all of which have some shared behavior and some unique behavior, we may want to change the shared behavior in all the classes at once. Can we do that without having to identically edit every class?
- Let's say we want to write a method that uses a particular type of class - say, it's a method climbTree(Tree t). Then, suddenly, we are struck with the brilliant idea of creating more than one type of tree so as to preserve the biodiversity of our computer ecosystem - we want to have both Oaks and Douglas Firs. Suddenly, however, we realize that climbTree only takes a Tree argument, and if we create the classes Oak and DouglasFir (and other types of tree), we will be forced to create more and more identical copies of climbTree - climbTree(Oak t), climbTree(DouglasFir t), and on and on as we increase the number of tree types. Is there an available approach to avoid this awful arboreal aggregation?

The solution to these problems is a thing called inheritance. Inheritance means that, given a sort of "parent" class, you can create a "child" class that steals all the code from the parent class without you ever re-writing it. So, if you had a class named "Tree", you could write a class that was only the following code:

```
public class Oak extends Tree {

}
```

Now, you have a thing called Oak which works EXACTLY THE SAME WAY AS TREE (as long as it can find the Tree.class file, that is). It copies all the methods, instance variables, constructors, and behavior from Tree. So if your Tree has a method Tree.growLeaf(), you can also write Oak.growLeaf(), even though there is no explicit method growLeaf() in the Oak class. Not only that, but Oak **is a** type of Tree, so any method that takes a Tree as an argument will also take an Oak! (It does **not** work the other way around, because not all Trees are Oaks).

The real magic happens when you start writing stuff in the Oak class. Here are some things you can do:

You can add instance variables and methods that weren't there before. For example, you could have a new instance variable called **numberOfAcorns** and a new method called **growAcorn()**. You could also add a new version of an old method that takes different arguments, like **growLeaf(Oak partner)** or something.

You can **override** methods by writing new versions of them. So, if you want to make sure Oak tree grows a new, special, oak-type leaf, you can write a new version of the **growLeaf()** function. It must have the same return type as the Tree class had, or you will get a compiler error. You can still secretly access the function you have overriden by calling **super.methodName(args)**.

You can **override** the constructor by writing a new constructor. This constructor will automatically and secretly call the parent class's (called the **superclass**) no-argument constructor unless you explicitly call a different superclass constructor. You can do this by writing **super(arguments)**. Super works kind of like the keyword this. It's a bit confusing at first, but you'll get the hang of it after writing inheritance constructors a few times.

You can overwrite instance variables. This is extremely confusing and weird and I am not going to try to explain it. Just know that if you ever re-declare an instance variable in a new version of a class, the gods of Java may punish you with really complicated confusing behavior. You will learn way more about this than you ever wanted to know in the next few weeks.

Inheritance has lots of crazy complicated rules. You can easily find any specific instance of behavior online, so it's not worth going through all of them. You will run into almost all of them at least once in the next few weeks.

Practice Problem:

Write a class called "Cat." Cat should have a constructor with a single String argument that sets the cat's name as a private variable. It should also have a method called "sayName" that prints out "I am ____" where ___ is the cat's Name.

Now, write a second class called “FuzzyCat” FuzzyCat should work the same as Cat, except instead of saying “I am (name)” it says “I am a fuzzy cat. I am (name).” Use inheritance. You will probably need to use the line **super.sayName();** at some point.

Solution:

```
public class Cat {

        private String name;

        public Cat(String n) {
                name = n;
        }

        public void sayName() {
                System.out.println(“I am “ + name);
        }

}

public class FuzzyCat extends Cat {

        public void sayName() {
                System.out.print(“I am a fuzzy cat. “);
                super.sayName();
        }
}
```

Other Concepts

Inheritance is the first (and fundamental) of the four concepts you will learn in Lab 6. The others are **polymorphism, abstract classes,** and **interfaces**, but these are all just extensions of the concepts of inheritance. If you really get the idea of inheritance, you’ll get the other ideas quickly. Here are some brief introductions to the other concepts:

- **Polymorphism** is the behavior of classes when you have a bunch of related classes pretending to be each other. For example, if **Oak** extends **Tree**, you can actually write the line of code “**Tree confusing = new Oak();**”, because **Oak** is a **Tree**. Then, you can say something like **Oak what = (Oak) confusing** to remind the computer that even though you said **confusing** was a **Tree** it’s actually an **Oak**. More on this soon.
- **Abstract classes** are versions of classes where you can leave some methods blank. Later, you or other people will create classes that inherit from your abstract class and finish those methods. You can’t actually create an instantiation of an abstract class.

- **Interfaces** are like abstract classes, but every single method is blank. Interfaces are like blueprints for classes that will be created later. This is useful for polymorphism.

One Final Note

While working with Inheritance, a bunch of stuff will not work for completely unpredictable reasons. Do not assume anything. Enter your most peaceful state of mind and become one with Google. Some queries you may at some point use are:

java subclass access private instance variable defined in superclass (answer is no)
java can i access superclass of superclass (answer is no)
java can interface extend interface (answer is yes)
java is polymorphism awful and confusing (answer is yes)