**Used Cars, Part 2**

Even though this is by no means a complete application (it has no user interface, for example), we have defined the core logic of the program in Part 1. It works, and it is fairly straightforward. However, there are several fundamental problems with the application as it is now. First and foremost is an egregious case of ***code******duplication***. The Car and Truck classes share much of their source code, and everything in the Inventory class is doubled.

Now, imagine you wanted to expand your business and include other types of vehicles (e.g. motorcycles) – you would have to repeat all the code for every type of vehicle. The more classes that are added, the more code is duplicated, and the harder it is to make changes later. For example, if you wanted to change the type of the year variable to a String, for formatting purposes, you would have to make changes in every single class, in multiple places. Not good!

Luckily, there is a concept in **object-oriented** programming languages that can help with this issue. Rather than having two separate classes (Car and Truck) that define everything independently, first make a ***super*** (i.e. above, over) class that contains everything these two have in common. Car and Truck will then ***inherit*** from this class, but maintain the ability to add their own unique code.

To do this, make a new class called Vehicle that will store everything Car and Truck have in common.

1. Refactor the "common" variables found in both Car and Truck into the new Vehicle class.
   1. Make sure to ***remove*** the common variables from Car and Truck that are going into the Vehicle class. If these variables exist in both the super-classes and the sub-classes, the variables with the smallest scope (in this case, the variables in Car and Truck) will "hide" (or "shadow") the variables in the super-class.
2. Write getter and setter methods for all the instance variables in Vehicle – even though these variables will be inherited by Car and Truck, they are private and can't be accessed directly outside the declaring class. They can only be accessed outside the class by public getters and setters.
3. Note that Car and Truck both have a toString() method, but each works slightly differently. In the Vehicle class, add a toString() method that returns only the year and type as a String (e.g. "2016 Toyota Prius"). In the Car and Truck classes, keep the toString() method that works as described previously.
   1. Remember that even though the fields in Vehicle are inherited, they are declared private and can't be accessed directly outside of the declaring class – you'll have to use getter methods for instance variables declared in Vehicle. Crafty programmers can use the super keyword to re-use some of what is already done in Vehicle (more on super later).
4. In the Car and Truck classes, add the extends keyword that indicates these classes inherit from another class, like this:

public class Car extends Vehicle {

The phrase extends Vehicle indicates that Car is a ***sub-class*** of the Vehicle class, and will inherit all its fields (variables) and methods. A Car will have (and be able to use) all of Vehicle's fields and methods, yet it will still maintain the ability to add code unique to Cars.

Turn in your Vehicle.java, Car. java, and Truck.java files even if they aren’t compiling. You will have some compile errors as we build this bridge to our final product!