**Vehicle Empire**

Interfaces are a strange concept at first – a class that doesn't do anything?! Interfaces are purely abstract types and contain *only* abstract methods (unlike abstract classes, which could have either).

An interface's job is to specify a set of abstract methods (without implementation / method bodies) that an implementing class *must* override. Think of an interface as a contract – if a class ***implements*** an interface, it is signing a contract to implement all the methods in the interface to satisfy the *is-a* relationship.

**The Vehicle interface**

Think back on the Used Cars lab; there is no compelling reason for Vehicle to be a "regular" (concrete) class. You don't sell vehicles, you sell cars and trucks (which *are* vehicles, but actual instances - real things).

You could make a case that Vehicle should be an *abstract* class. However, abstract classes are preferred when some of their implementation is already known or required; this is not necessarily the case now, as you'll see below. It might be better as an ***interface****,* with more specialized classes as abstract classes.

1. Create a new interface with the following:

public interface Vehicle

{

/\*\* should return the final price of this vehicle, after applicable price mods \*/

double getPrice(); //same as: public abstract double getPrice();

/\*\* should return a String listing all the pertinent info for this Vehicle \*/

String getInfo();

}

Interfaces are types, just like classes, but they are purely abstract – they do not have instance variables, and therefore do not have any constructors. They can't be instantiated or contain any concrete methods! Essentially, every vehicle you sell should have the ability to return its price (after adjustments) and get a String containing its pertinent info.

Interfaces *can* have public static final variables (class constants, like Math.PI), but we can worry about that later.

1. Your vehicle sales empire is flourishing! You now sell *new* cars and trucks, in addition to used airplanes and helicopters. With your burgeoning knowledge of inheritance and abstract classes, it should hopefully be immediately apparent to you that writing four concrete classes that implement Vehicle is not the best approach – think of all the code that would be repeated!

Add two abstract classes, Automobile and Aircraft, that will serve as parent classes for cars/trucks and airplanes/helicopters, respectively. These classes should have the following:

* 1. Automobile
     1. double MSRP (manufacturer's suggested retail price, i.e. list price of a new vehicle), String type, and int year
     2. A constructor, getter methods, and an overridden getInfo() method in the form of:

"<year> <type>"

* 1. Aircraft
     1. double price, String type, int year, and int passengers
     2. A constructor, getter methods, and an overridden getInfo() method in the form of:

"<year> <type>, <passengers> passengers"

Note that Automobile and Aircraft override only one of the methods in the Vehicle interface - this is acceptable as these classes are abstract (they will never be instantiated). (Concrete) sub-classes will be required to implement the getPrice() method (note that they *can* override the getInfo() method again if they choose, though it won't be required by the compiler).

1. Add the Car/Truck and Airplane/Helicopter classes that inherit from Automobile/Aircraft, respectively. These classes will now satisfy the *is-a* relationship with Automobile/Aircraft *and* Vehicle, and should have the following (utilize code from super-classes when possible):
   1. Car
      1. double mpg
      2. A constructor, an overridden getPrice() method (cars with gas mileage of 30 mpg or more get 10% off of MSRP), and an overridden getInfo() method in the form of:

"<year> <type>, <mpg> mpg, $<final price>"

* 1. Truck
     1. int towing
     2. A constructor, an overridden getPrice() method (large trucks with towing 5,000 lbs. or more get 15% off of MSRP), and an overridden getInfo() method in the form of:

"<year> <type>, <towing> lbs. towing, $<final price>"

* 1. Airplane
     1. boolean hasFirstClass
     2. A constructor, an overridden getPrice() method (Airplanes that have a first class receive a 10% markup for the extra upholstery work required), and an overridden getInfo() method in the form of:

"<year> <type>, <passengers> passengers, $<final price>"

then (on the next line) either "Has first class" or "No first class"

* 1. Helicopter
     1. boolean canShootMissiles
     2. A constructor, an overridden getPrice() method (Helicopters that can shoot missiles receive a 25% markup - missiles are really expensive), and an overridden getInfo() method in the form of:

"<year> <type>, <passengers> passengers, $<final price>"

then (on the next line) either "Shoots missiles" or "Does not shoot missiles"

1. (Riddle) I went into the woods and got it. I sat down to seek it. I brought it home with me because I couldn't find it. What is it?
2. Recall that Java only allows single inheritance (to prevent problems when two super-classes define the same method / implementation). Because an interface has no "concrete" elements (instance variables or method implementations), it is completely fine to implement any number of interfaces.

Create another interface, called Leaseable. Classes that implement this interface will be "lease-able", meaning they can be leased from the dealership, rather than purchased outright.

The Leaseable interface declares the following method:

double getMonthlyPayment(); //public and abstract not required in interfaces

1. Because why not, you will lease helicopters to rich people. Assume that your company only does 60-month leases, and there is no interest (such that monthly payment would be total cost divided by 60).

The Helicopter class will now extend Aircraft *and* implement the Leaseable interface. Helicopter objects now satisfy the *is-a* relationship for BOTH Vehicle and Leaseable, and can be used wherever a variable of that type is expected. Examples:

Vehicle v = new Helicopter();

Leaseable lease = new Helicopter();

1. A complete inheritance hierarchy is shown below. Test your code with the Runner.java file in the lab folder; your output should match the contents of the **"output.txt"** file in the lab folder.

