**Used Cars**

Having already accomplished your goal of watching every Youtube video ever, you decide to embark on a new quest; an exciting journey into the world of used car sales. Knowing how to code, you're confident that a paper and pencil inventory tracking system will not be sufficient. You decide to write software to track your inventory, such that you can better understands sales trends and build yourself a used car sales empire.

Along with the powerpoints obviously, [this](https://www.youtube.com/watch?v=gQTzUpqeLH4) video is a very good introduction to the concept used in this lab called ***inheritance.*** It might be useful to watch prior to starting the lab!

Begin by writing two classes that will represent vehicles in your inventory:

1. The Car class will have the following (all instance variables private):
   * String type – type of car, e.g. "Honda Civic"
   * int year – the year the car was made
   * double price – price of the car, in dollars
   * double mpg – fuel efficiency, in miles per gallon
   * A constructor to initialize all the instance variables.
   * boolean greatGasMileage() – returns true if the mpg of that Car is >= 36
   * String getInfo() – method that will return printable info for that Car, in the following fashion (don't worry about decimal formatting):

1998 Honda Civic, 32.0 mpg, $6499.0

1. The Truck class will have the following (all instance variables private):
   * String type – type of truck, e.g. "Ford F-150"
   * int year – the year the car was made
   * double price – price of the car, in dollars
   * int towing – the truck's towing capacity, in pounds
   * A constructor to initialize all instance variables
   * boolean canTowBoat() – returns true if the towing for that Truck is >= 2,000.
   * String getInfo() – method that will return printable info for that Truck, like the following:

2004 Ford F-150, 3200 lbs. towing, $8999.0

1. Next, write a class called Inventory that will store all the vehicles in your inventory. The Inventory class has the following:
   * ArrayList<Car> cars – list of all the cars in your inventory
   * ArrayList<Truck> trucks – list of all the trucks in your inventory
   * a constructor that initializes the ArrayList objects
   * void addCar(Car c) – add a Car object to the list (the lists should be private)
   * void addTruck(Truck t) – add a Truck object to the list
   * void listInventory() – iterates through all the cars and trucks in the inventory and prints their information (calls their getInfo() method). Use for-each loops for this.
2. In the main() method in a Runner class, create a least 2 Car objects and 2 Truck objects, and test that all your methods work (add to inventory, print information for all, etc.). If you have an error, make sure you check the **FAQ** (in the lab folder) first!

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. 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 the Inventory class has (essentially) duplicate lists and methods.

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 new 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, parent) class that contains everything these two have in common. Car and Truck will then ***inherit*** from this "super" 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 that are going into the Vehicle class from Car and Truck. If these variables exist in both the super-classes and 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 methods for all the instance variables in Vehicle – even though these variables will be inherited by Car and Truck (they will exist for every object of these types), they are private and can't be accessed directly outside the declaring class.
3. Note that Car and Truck both have a getInfo() method, but each works slightly differently. In the Vehicle class, add a getInfo() method that returns only the year and type as a String (e.g. "2016 Toyota Prius"). In the Car and Truck classes, keep the getInfo() method that works as described previously. When an inheriting class declares the same method as the super-class, the sub-class' method ***overrides*** the parent class' method (more info to come).
   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's getInfo() method (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.

1. Recall that constructors are special methods with two main purposes – to initialize that class' instance variables and make objects of that class. As such, it makes sense that constructors are not inherited – sub-classes really have no business making objects of the super-class (that's what the super-class' constructor does!). So how can a sub-class initialize the instance variables that are declared in (and private to) the super-class?

This is done using the super keyword. The super keyword is used to refer to the super-class (and can only be used in inheritance). Because constructors are not inherited, you must use the super keyword to call the super-class' constructor.

The Vehicle class declares three instance variables (type, year, and price), so its constructor must take parameters for those to create an object of that type (shown below), even though we don't intend to make Vehicle objects:

public Vehicle(String t, int y, double p)

{

type = t;

year = y;

price = p;

}

The Car class will inherit type, year, and price from Vehicle, and define its own variable, mpg. Modify the constructor of the Car class as follows:

public Car(String t, int y, double p, double m)

{

**super(t, y, p);**

mpg = m;

}

The super(t, y, p) line MUST BE THE FIRST STATEMENT IN THE CONSTRUCTOR. This piece of code calls the super-class' constructor, letting it do its job of initializing type, year, and price. These variables can't be initialized in Car, because they are declared in Vehicle! Even though Car will inherit these variables, they are private, and therefore can't be accessed directly outside the class they're declared in.

In Java, a sub-class MUST call the super-class constructor (and it must be first statement in the constructor!). If you do *not* write a super-class constructor call, the Java compiler will insert a super-class call automatically, to ensure that the super-class' fields get initialized. The inserted call is equivalent to adding a no-parameter super() call.

**This automatic insertion by the compiler only works if the super-class HAS a constructor without parameters** - the compiler cannot guess what parameter types should be used. If the super-class *doesn't* have a no-parameter constructor, an error will occur.

It is best practice to always include explicit super-class constructor calls in your sub-class' constructors, even if it is one that the compiler would generate automatically. This practice is considered good style, because it avoids the possibility of confusion - especially for people that may not know about automatic code addition.

You should now be able to successfully compile your code – if you're using BlueJ, you should see arrows pointing from Car and Truck to Vehicle, indicating that these classes extend Vehicle.

1. At this point, at least a couple features of inheritance should be obvious – it minimizes code duplication when multiple classes have similar functionality, makes code easier to maintain (making one change in a super-class, rather than in many potential sub-classes), and allows reuse of code. This may come as a shock to you, but there are *even more* reasons to use inheritance. Try to remain calm.
   * When one class extends another class, it has what is called an ***is-a*** relationship with the super-class. A Car *is-a* Vehicle, and an object of type Car can be used anywhere an object of type Vehicle is expected. A sub-class is a more specialized version of a super-class, but it still has everything the super-class has, and therefore satisfies the *is-a* relationship.

Back in the Inventory class, recall that we had two separate methods (and two separate lists) to add Car and Truck objects to the inventory. This was unfortunate but necessary, because we had no Vehicle super-class yet.

Refactor the two methods into one, void addVehicle(Vehicle v), and use just a single list, ArrayList<Vehicle> inventory. Because Car and Truck extend Vehicle, they *are* Vehicles; you can pass an object of either type into this method without issue. Make this change and run the main() method in the Inventory class to make sure your code still works.

* + Note that the Vehicle, Car, and Truck classes all have a getInfo() method. This is a problem, right? *It must be a problem* (it's not). When a sub-class declares the same method (same in name and parameter list) as a super-class, the super-class' method is ***overridden*** by the sub-class. Example:

Vehicle v = new Car("Honda Civic", 2000, 6500.0, 32.0);

v.getInfo(); //calls Car's overridden getInfo() method, rather than Vehicle's

The compiler will always run overridden methods in sub-classes if they exist! If a sub-class doesn't override a super-class method, the compiler will continue "up" and look for the method in super-class(es). If the method still isn't found, the compiler will check the Object class that every class inherits from automatically (more on this later!). If the method can't be found there, the code won't compile.

In Java, variables that store references to objects are ***polymorphic***. Polymorphism *literally* means "many shapes", and refers to how a super-class type variable can store many different types of objects – either the declared type or any sub-class of the declared type.

Refactor the listInventory() method in the Inventory class, such that it iterates through all the Vehicles in inventory and *polymorphically* calls the getInfo() method for whatever sub-type it happens to be. The listInventory() should now have just a single for-each loop, and when the getInfo() method gets called on the Vehicle variable, the sub-class' overridden method will be called (if it exists).

So you don't miss it, make sure you understand what you just accomplished. Having a list of Vehicle objects and having the getInfo() method dynamically decide what info to return is based on the actual type of the objectis **polymorphism**. This is also known as *late binding* because it wasn't known until run-time that (for example) inventory.get(1) really stored a Car object.

* + Add a Scooter class that also extends Vehicle, with a unique instance variable of your choosing and an overridden\* getInfo() method. Add a Scooter object to your inventory, run the main() method, and revel in the fact you *didn't have to do nothin'* to make the listInventory() method work. Hopefully you can start to see how the inheritance model makes project extensibility much, much easier.

/\* You may have seen the ***@Override*** annotation online before – adding this before a method is not required, but it ***asserts*** to the compiler that you are overriding a method. If you're not actually overriding a method (e.g. you put the wrong parameters or name), the compiler will let you know \*/

* + Complete the "Object type assignment" worksheet, found in the lab folder. You'll need to download the file to your computer first - it can't be completed using the web preview.

Recall that a super-class variable may store a sub-class reference, but not vice-versa. Every instance of a sub-class *is-a* object of the super-class, but not all super-class instances will be an instance of a particular sub-class.

1. If you've spent any time on Stack Overflow or elsewhere, you might have noticed a curious keyword frequently used in constructors: this. The this keyword is used in a similar fashion to super – while super refers to the super-class, this refers to *this* object – whatever object is calling the method.

The this keyword is frequently used in constructors to differentiate instance variables from parameter (local) variables. Consider a Person class, that has String name and int age instance variables. Its constructor could look like this:

public Person(String name, int age) {

name = name;

age = age;

} //the above is going to cause problems!

However, a curious situation arises: how does the compiler know which name and which age you're referring to? The compiler can't read your mind. By default, Java will assume you're referencing the variable with the smallest scope (similar to the inherited variable "hiding" issue referenced previously). The *instance variables* name and age are being "hidden" by the local (parameter) variables. To get around this, you must specify *which* name and age you're referring to, like this:

public Person(String name, int age)

{

this.name = name;

this.age = age;

}

Here, you're saying *this* *object's* name and age instance variables should have the value of the name and age local (parameter) variables. While there is no rule that you *must* name the parameters the same as instance variables, this is generally considered best practice (once you get used to it, it's much faster and easier to read).

The super and this keywords can also refer to methods; this.getInfo() will call *this* *object's* getInfo() method. Note that if you omit this., the compiler will add it for you. Whether or not you add this. is a matter of personal preference; I tend to prefer it, as it is more explicit that you're calling this object's method or referring to this object's instance variable.

Similarly, super.getInfo() will (preferentially) call the super-class' getInfo() method. Remember that if you omitted super. in this case, the compiler will add this. and look in the sub-class for an overridden method, and call that method first (if it exists). Again, explicitly using super and this is good practice, even when not necessarily required, as it prevents mistakes that can occur when you accidentally call an overridden method when you intended to call a super-class method.

Reminder - crafty programmers can use super to re-use some of the code in Vehicle's getInfo() method in Vehicle's sub-classes.

Despite your sincerest wishes, you can only "directly" inherit from one class (though the class you inherit from may also inherit from one other class, which itself can inherit from on other class, and so on). Though it may be tempting, you can't use super.super.something(). A class can only have one super-class (not counting the Object class that every class implicitly inherits from).

There is a lot to learn with inheritance, and the initial learning curve can be steep. Do some research, read the powerpoints, write some classes that use inheritance – with practice you'll get it!