**Used Cars, part 3**

Recall that constructors are special methods with two main purposes – to initialize that class' instance variables and allocate memory for 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 – in 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 toString() 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.toString(); //calls Car's overridden toString() method, rather than Vehicle's

The compiler will always run overridden methods in sub-classes (should they exist)! If a sub-class hasn't overridden a 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 toString() method for whatever sub-type it happens to be. The listInventory() should now have just a single for-each loop, and when the toString() method gets called on the Vehicle variable, the sub-class' overridden method will be called (if it exists).

* + Add a Scooter class that also extends Vehicle, with a unique instance variable of your choosing and an overridden\* toString() 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 \*/

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.toString() will call *this* *object's* toString() 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.toString() will (preferentially) call the super-class' toString() 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 toString() 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 one 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).