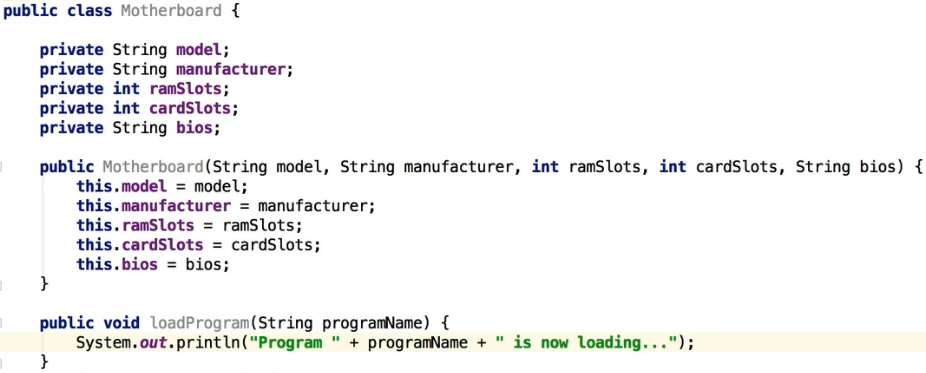
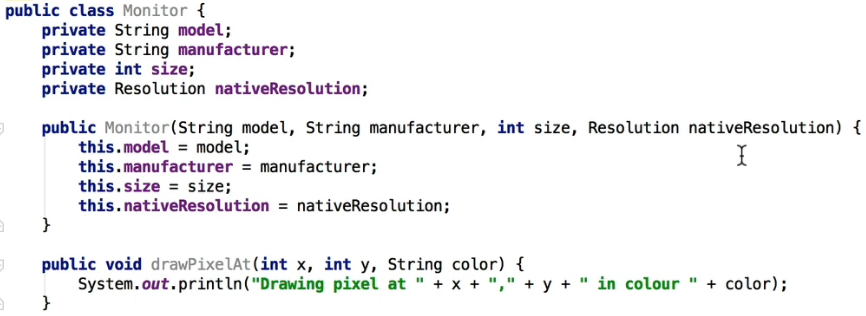
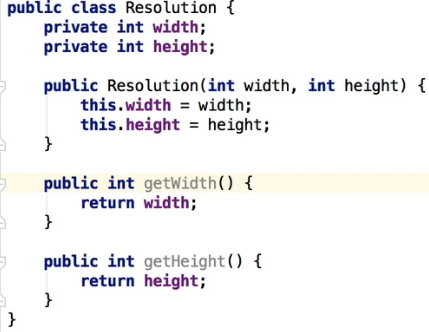
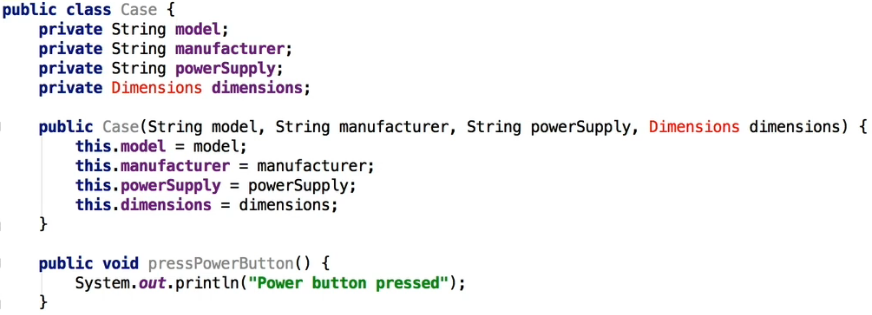
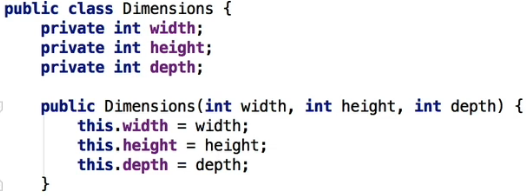
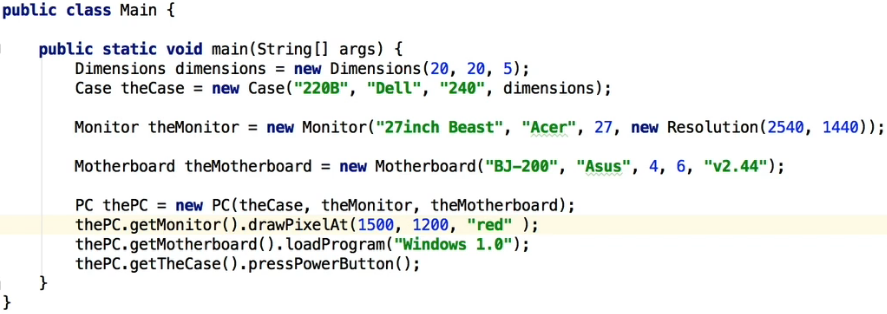
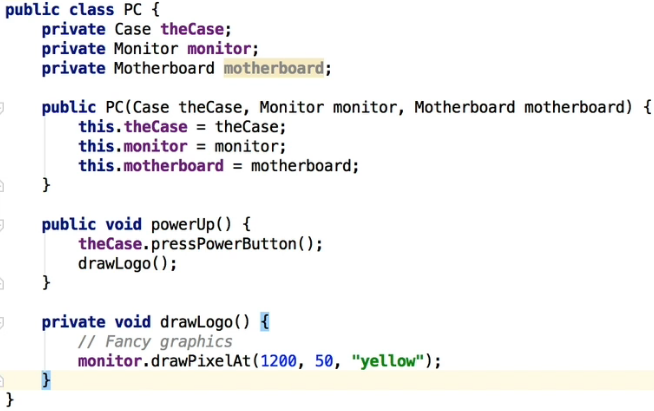
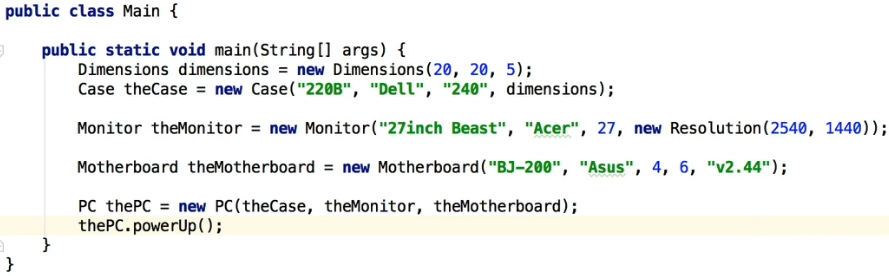
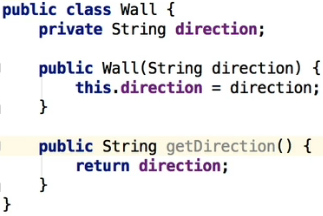
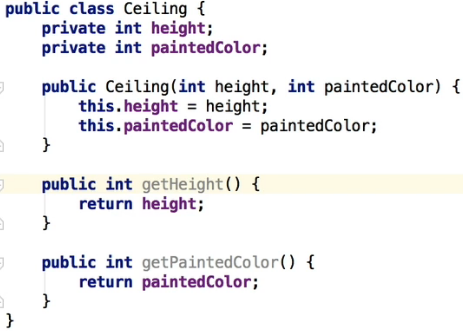
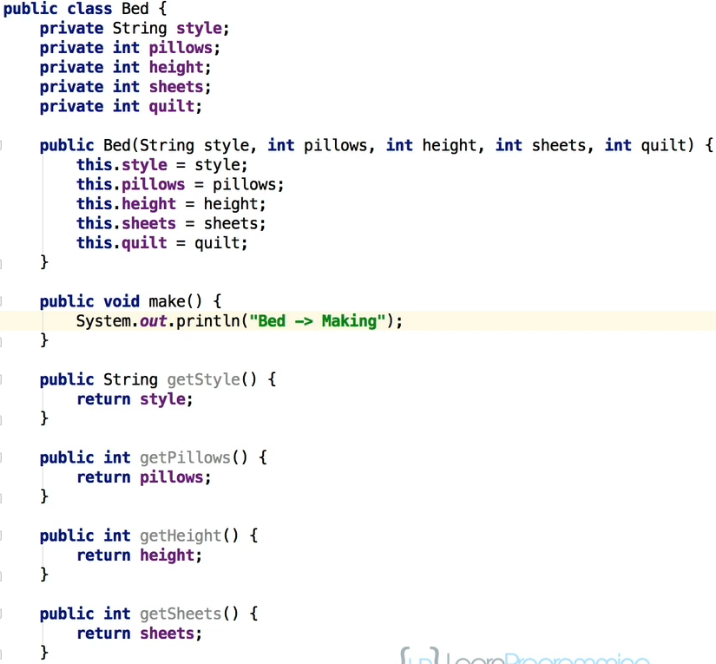
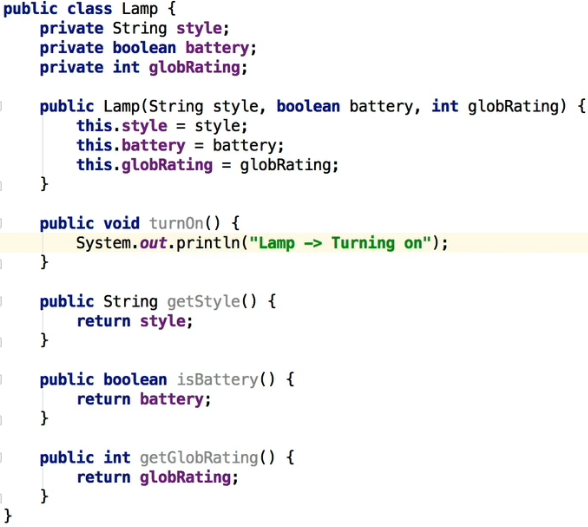
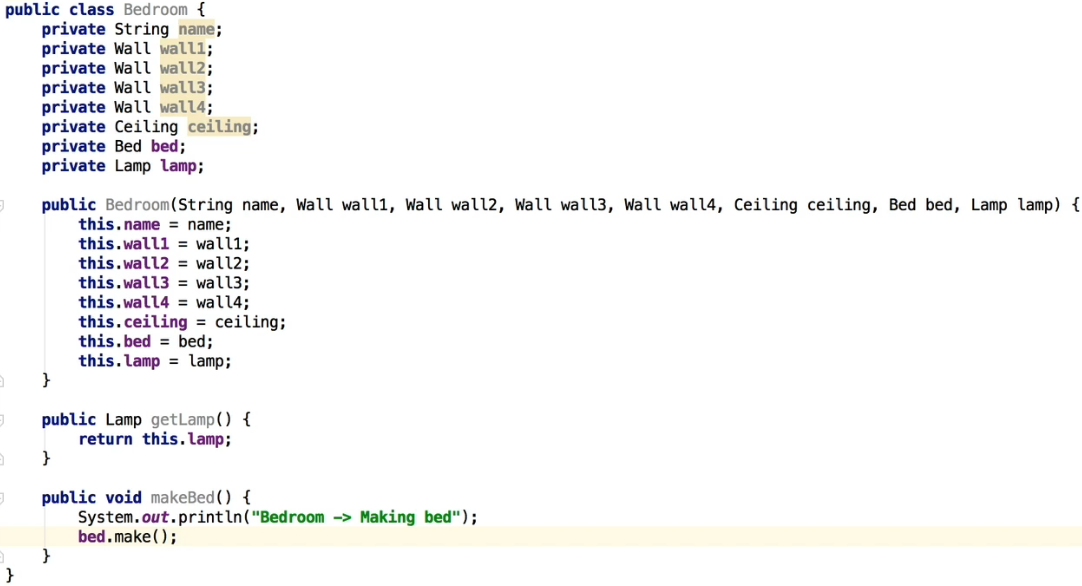
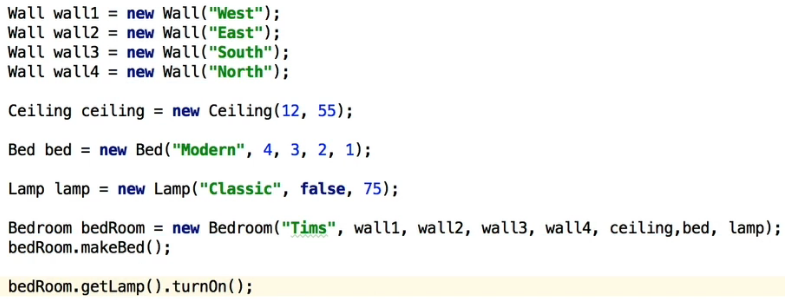
**Introduction**

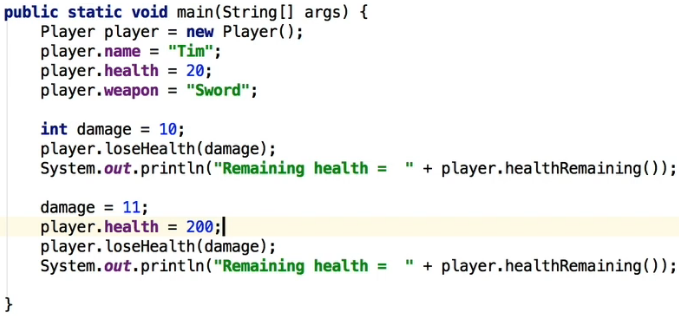
**Composition**  
\* Composition is another component of OOP and is very useful in day-to-day programming.  
\* Inheritance => **is-a** relationship, Car is-a Vehicle.  
\* Composition => deals with something different, and that sis a **has-a** relationship.  
\* Let’s explain it on an example of a computer.  
\* So a computer has-a motherboard, a case, and a monitor. Three example things that it’s got with it.  
\* So the motherboard, the case and the monitor are not computers in the same sense that a Car is-a Vehicle. But a Computer has-a motherboard, has-a case, has-a monitor. That’s what composition is, it’s actually modeling parts of the greater whole.  
  
  
  
  
  
\* Monitor has-a Resolution.  
  
\* So the Resolution class is part of a monitor, but the monitor isn’t a Resolution, it **has-a** Resolution.

\* So we’re saying that the Resolution is-a component of a Monitor. **The Monitor has-a Resolution**.  
  
  
  
\* The Case has-a Dimensions.  
\* Now we get to the fun part where we actually create a class that is comprised of the Case, the Monitor, and the Motherboard.  
  
\* PC has-a Case, has-a Monitor, has-a Motherboard.  
\* You can see the obvious advantage here - because if you’re using the `extends` option to inherit, you can only (in Java) inherit form 1 class at a time.  
\* Let’s now see how we would use it:  
  
\* We created the Resolution using a different way of creating a class on the fly - if you don’t need to create a variable, you’re not going to use it for anything else, you only want to pass it to a method   
=> you can just type it in there directly. That’s creating an instance of a class without using a variable which we don’t need in this case.  
\* Now how to access a method in one of the small parts?  
\* When you’re thinking about whether to use Inheritance or Composition, think about what you’re trying to do here - in this case, we’re trying ot build a computer and we need to access the case, the monitor, and the motherboard, PC has-a case, has-a monitor, has-a motheboard, therefore Composition is probably a great way to go. But of course a case by itself isn’t a computer, a monitor isn’t a computer and of course a motherboard isn’t a computer. They’re part of a computer but not actually the whole computer, which is why we’re using Composition instead of Inheritance for those.

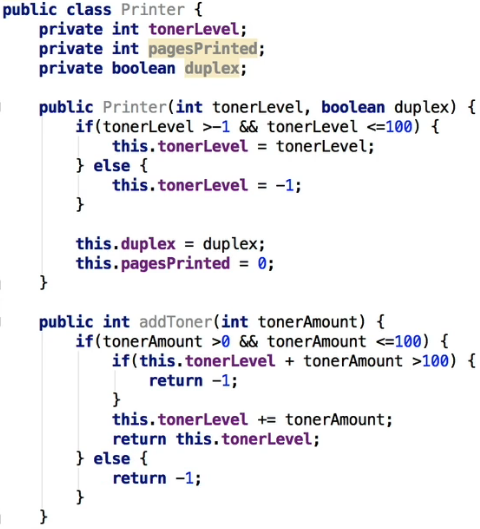
**Composition Part 2 (+ Challenge Exercise)**  
\* Let’s now look at another scenario whereby we can actually hide the functionality further so that we don’t allow the calling program to access those objects directly.  
\* In PC we’ll change the getters to `**private**`. Unless we want to do some validation or something, we can even remove those getters for now and we can access the variables directly.  
  
  
\* Composition is essentially creating objects within objects.  
\* As a general rule, when you’re designing your programs in Java, you probably wanna look at Composition first. Most of the experts will tell you that as a rule, to look at using Composition before Inheritance, and it does give you the added advantages and more flexibility. But of course it depends on what you’re trying to achieve.   
\* Now a practice exercise:  
  
  
  


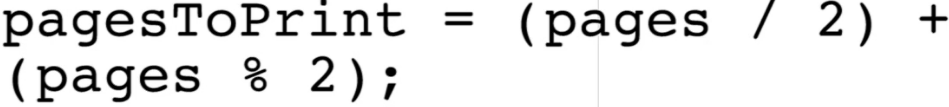
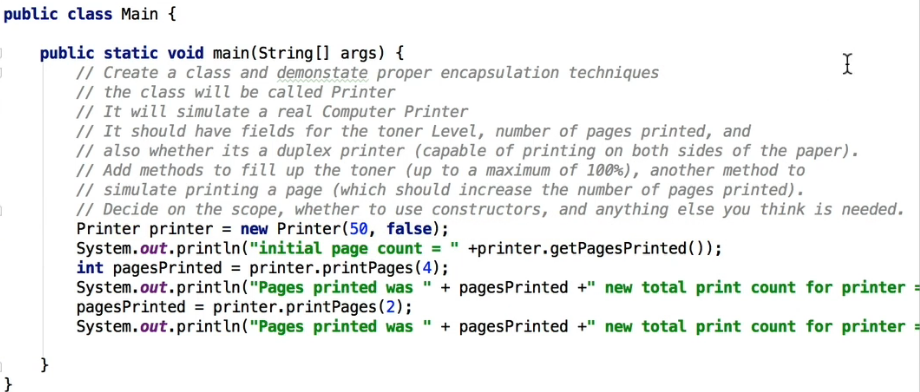
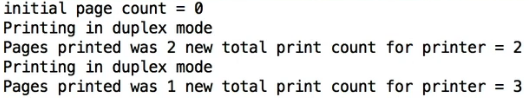


\* Now the master class.  
  


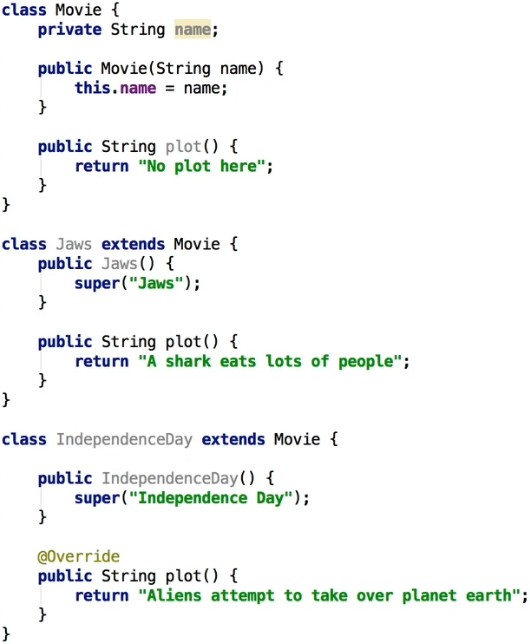
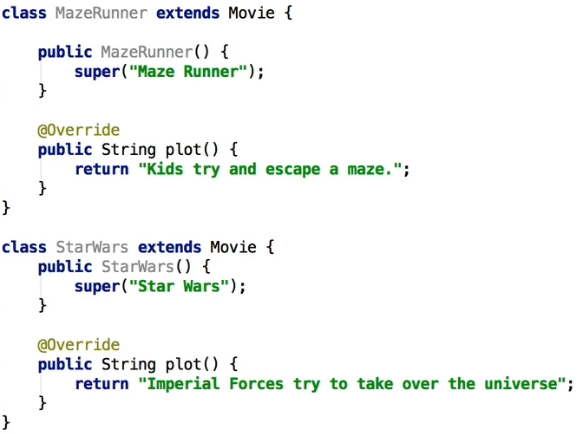
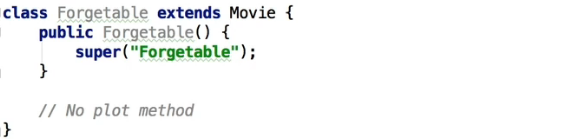
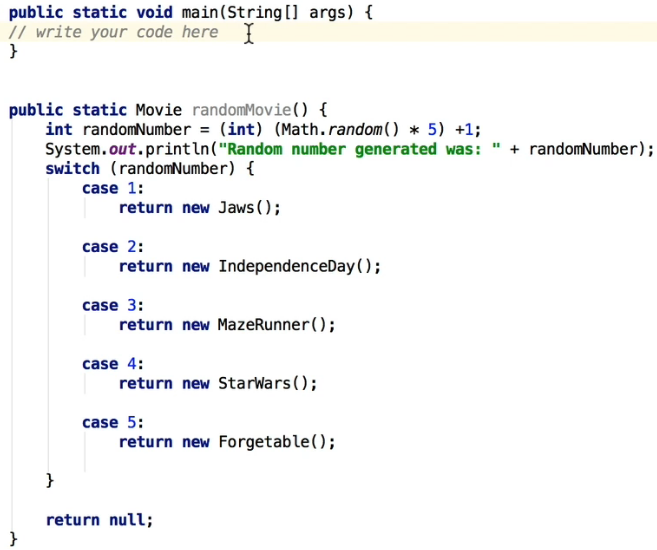
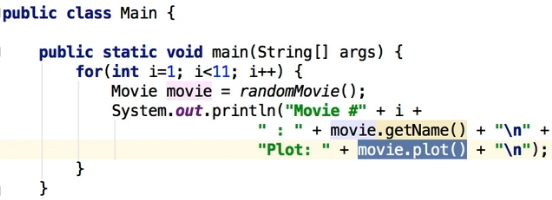
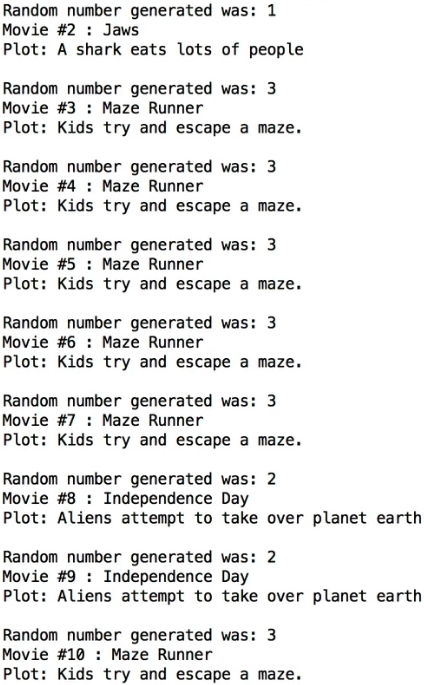
**Encapsulation**  
\* Encapsulation is the mechanism that allow you to restrict access to certain components in the objects that you are creating. So you’re able to protect the members of a class from external access in order to really guard against unauthorized access.  
\* We’re not talking security here, but we’re stopping outsiders, in other words classes or code outside of the class that you’re working on, from accessing the inner workings of a class.   
\* So it can be very useful to hide that inner working from another class to sort of give you more control and to be able to enable you to change things without breaking code elsewhere.  
  
\* Since we didn’t create a contructor, a default constructor with no parameters has been created for us.  
  
\* **One possible problem** here - we’ve created a method loseHealth() which is our code that is going to manage the whole process of the player losing health. We’d want all our functionality to lose health to be called from that component in there.  
\* But the problem then becomes, first of all, we’ve got the ability to change the player health, but we could actually change something here:   
  
\* So we’re taking the control out of the player class.   
\* So by **being able to access those fields directly**, you’re really potentially opening up your application to be accessed in ways that you didn’t want it to. Because you may not ever want the player to be able to change the health like that. You might want to control that process and a good way of controlling the amount of health that we assign is with a constructor, for example.  
\* **Second problem is**: if we **change the name of a field** - such as the `name` to `fullName`, this is an internal change, and in theory it shouldn’t really affect any other class because it’s just internal field name that we’ve defined. But now in the main class we get an error because player.name no longer works.  
\* **Third** **problem**: we might actually forget to assign the field values, because we’ve actually put these names in manually, we **might forget to actually initialize this altogether**. And consequently when the class is called for the firs time, it doesn’t have any health if we don’t initialize the player.health. So in other words, we’re not guaranteeing and ensuring that to access the player class you can only access it when the data is valid. Now we can do that with a constructor, we can actually make sure that the data is valid and that you’ve actually got a valid class. But when you’re giving control and getting people to manually get access to fields, there’s no real way of us guaranteeing that the player health is set.  
\* And the other thing is, of course, we might want to have a method that is in our Player class to define the maximum amount of health for a player. So we might want to have some validation, we might want to go through a method, for example, to make sure that they haven’t selected a health that’s greater than 100, or some other form of validation. And by allowing access at this level for people to actually just type in that directly, we can’t actually call that validation.  
\* So this is what encapsulation actually does for us, this is why we don’t want to code like this. We want the ability to be able to override these things so that we can make sure that the class that we’re giving for access, that we’re using for access, is actually valid and that we’ve got more control over these things.  
\* Now let’s see the right way of doing it:  
  
\* We’re guaranteeing now that the name, the health, and the weapon are initialized when the class is created. We also have some validation for the health, we could do more validation of course.  
\* By making the fields private, we’re making sure that the fields aren’t accessible to any classes that are outside.  
\* IntelliJ renaming trick => select what you want to rename => right-click => Refactor => Rename…  
\* Another cool feature of encapsulation: we can make all these changes to this EnhancedPlayer class - we can create private fields and private functions that we don’t want to be exposed to any other class, and we can actually change those names at any time in this code without affecting any other code.  

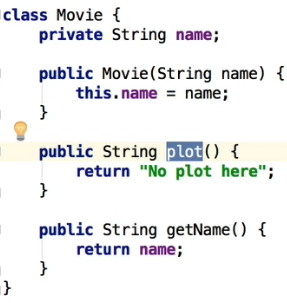

  
\* We made sure the only way to reduce the health is through the loseHealth function, no other way.  
\* So this is why we want to use encapsulation, we really want to keep things confined and sort of protect the members (fields) of the class, and some other methods, from external access. So we’re doing that to prevent unauthorized setup or access. In this case, we’re using it to make sure that the EnhancedPlayer - when we create a new object from the class - that is is valid, it’s got the right amount of parameters and they are initialized. But likewise, we’re also making sure that there’s no unauthorized access to the fields directly.  
\* So that’s why you want to always use encapsulation. It’s something that you should really get used to. You definitely don’t want to be having public fields - it’s very rare that you would need to do that. There are situations where you will want to, but generally speaking you don’t want to do that. You don’t want to expose that access to other classes because, as you saw, it’s more work to make those changes when you change a field name or something of that nature, there’s just more work for you to go back to all the other places, all the other code that’s using it to make sure that’s changed too.

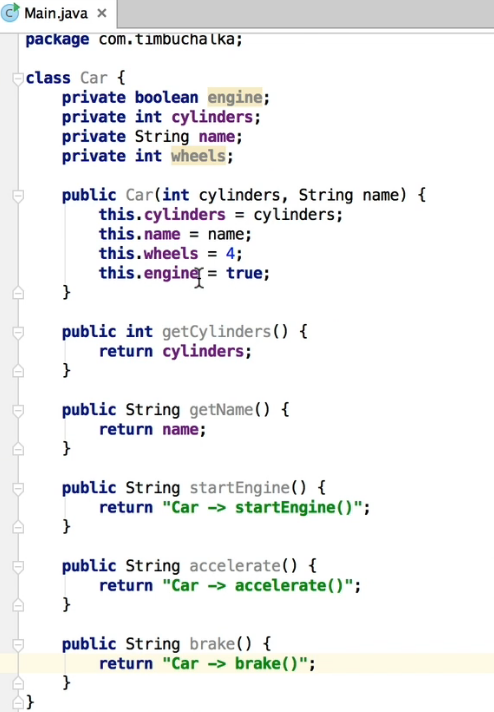
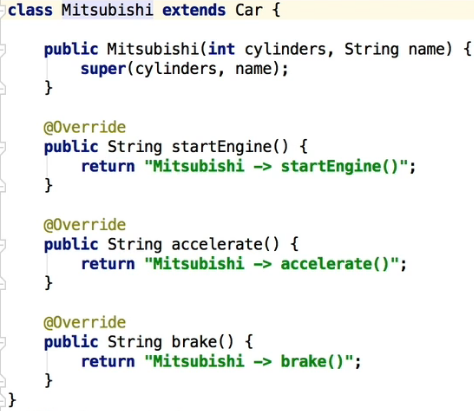
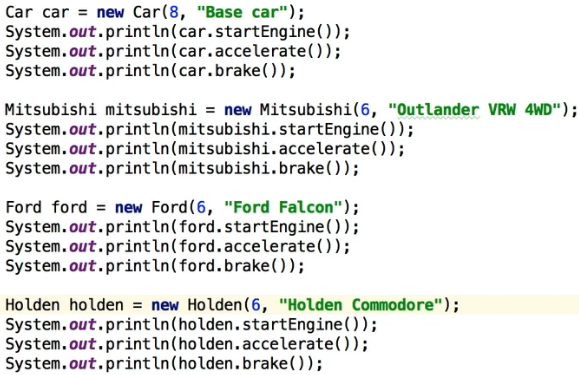
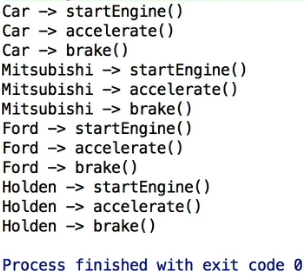
**Encapsulation (+ Challenge Exercise)**  
\* **It’s a good way to indicate that a variable hasn’t been set correctly by assigning it to -1 if it doesn’t fulfill some validation**.  


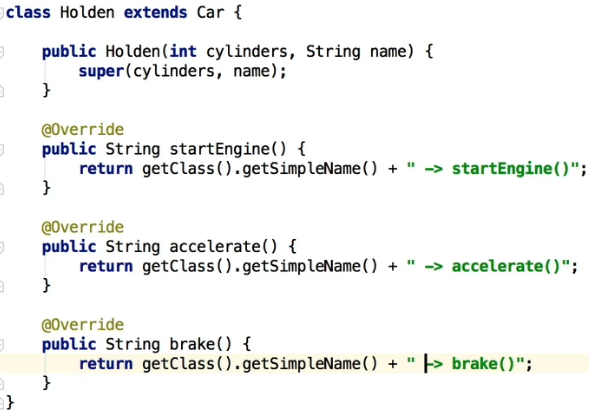
  
\* **There’s a logical error, the code for printPages duplex mode should be**:  
  
\* Let’s now try using this class.  
  
  
\* So that’s the Printer class using Encapsulation.   
\* It’s a very very good OOP sound programming practice to use Encapsulation to hide that functionality from your other classes. In this case, you can see that we’re not accessing any fields directly, and we’ve got proper methods that are called that are actually doing the appropriate calculations or adding and subtracting numbers, hidden away from the user.  
\* So if you’re someone who’s using this Printer class, you’ve got no idea of the internal workings of that class and in fact you don’t need to know it because all you’re doing is you’re using any of these methods that have been set to public (scope of public) and you’re using those in a correct manner and everything’s working perfectly.

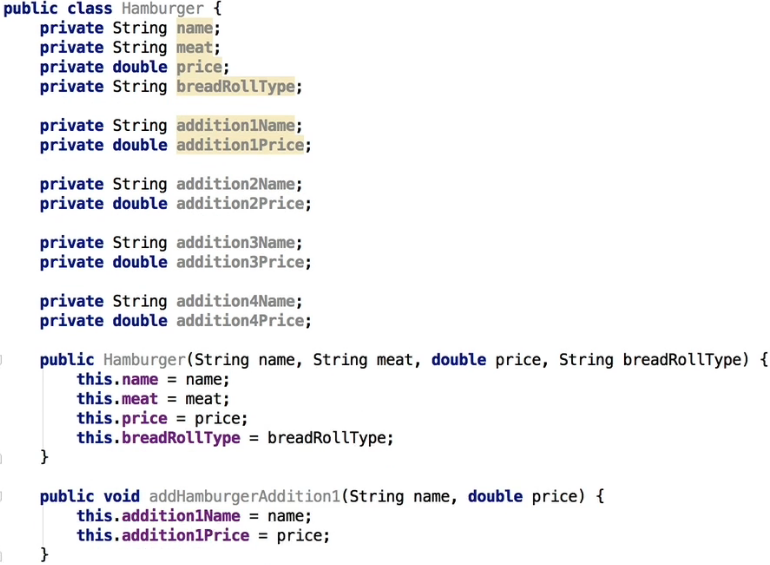
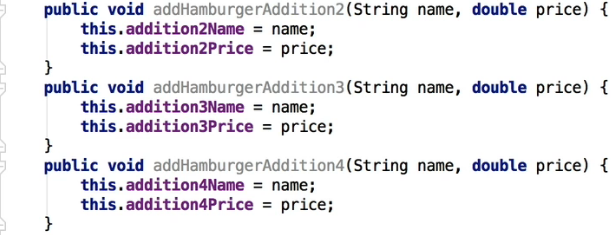
**Polymorphism**  
\* Polymorphism is really the method or the mechanics in OOP that allows actions to act differently based on the actual object that the action is being performed on.  
\* I’m going to create a few classes all within this main.java source file, which is another way of creating classes that aren’t going to be reused. It’s really only useful to create classes in the same java source file if they’re quite small and compact. Also bear in mind that when you’re creating these classes, depending on the modifier, may or may not be accessible outside of the package.

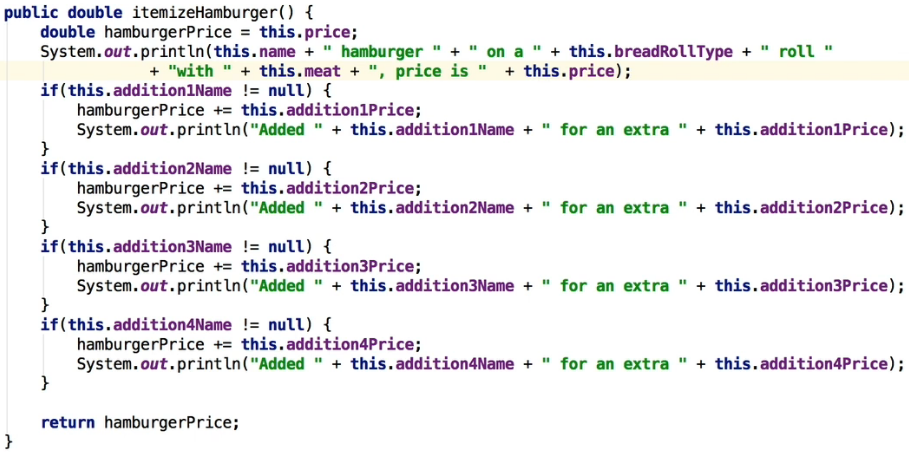
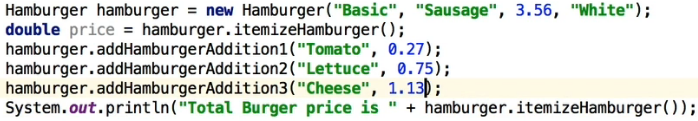
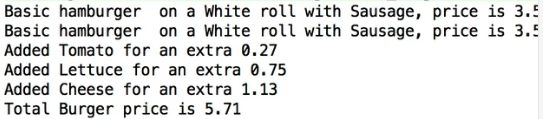
  
  
  
\* Now we want to create a STATIC method because it’s part of this main.java and not a separate class and the purpose of this method will be to return a random movie, so it’s gonna return one of those 5 classes (movies). We need to create it inside the MAIN method.  
   
  
\* We could also do default: return null.  
\* We’re returning a Movie type from the randomMovie() method, it’s a superclass of the individual movies, we can do that thanks to Inheritance => is-a Movie.  
  
\* We’re also gonna add a GETTER for the base Movie class - getName().  
\* This **movie.plot()** is really the definition of how **POLYMORPHISM** works.  
  
\* This is the beauty about POLYMORPHISM - the movie.plot() is actually doing quite a bit of work. What it’s doing is looking at the Object called `movie` that was returned from the randomMovie() method and then it’s going down and looking into that Object and saying: has it got a plot() method?

\* So you can see that with that one line of code movie.plot(), this is what POLYMORPHISM is. **We’re actually assigning different functionality depending on the type of Object that we’ve created**. Bearing in mind that all these Objects are Inherited from the base Movie class, but we’ve assigned a unique bit of functionality in each of those plot() methods, for each of those classes.  
\* Now in this case it’s a very simple method, it only outputs the plot of the movie but you can imagine that we could have far more complex functionality built into that if we wanted to.   
\* **Forgetable didn’t have its own plot() method so it used the Inherited one from the base Movie class (“No plot here”)**.  
\* So Java looked at that and said: okay, there’s no plot() method here that has been overridden in this Forgetable class, so as a result it looks and says: okay, I can see that we’re extending from the Movie class, and it finds the method called plot().  
  
\* So you can see how POLYMORPHISM works now. If you’re inheriting from another class and you’ve got a method and you override that method, that’s what POLYMORPHISM is, it’s giving unique functionality for the class that has inherited from a base class.  
\* You’ll find this incredibly useful once you start developing more Java applications because it really enables you to write quite generic code.  
\* Imagine if you didn’t have the ability to do polymorphism - you’d have to figure out a way to decide what the type of class was, and then to somehow call the appropriate class, so you’d probably have a large switch statement with if class equals this name, then call this particular method. So it would be a lot more work. Java really makes it easy for you by adding this polymorphism capability automatically.

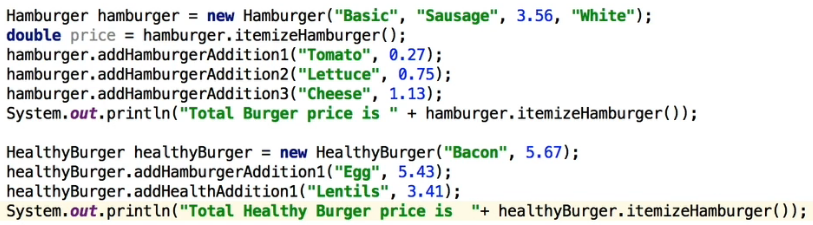
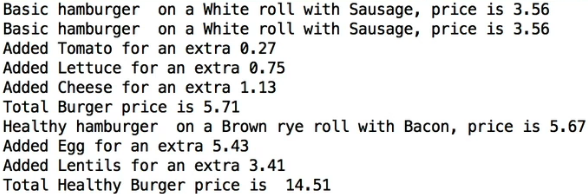
**Polymorphism (+ Challenge Exercise)**  
  
  
  


\* Last thing I want to show you is that there’s actually a way within Java instead of having to remember the class if you want to output the class, you can actually retrieve the class name.  
\* **getClass().getSimpleName()**.  


**OOP Master Challenge Exercise**  
  


**OOP Challenge - Solution**  


  
  
  
\* Now to disable the option to add additions to the delux burger, we can override those methods.  
