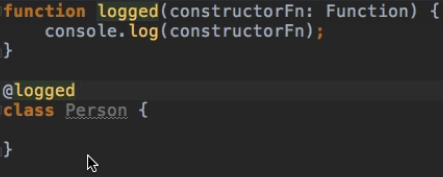
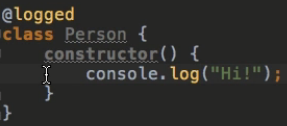
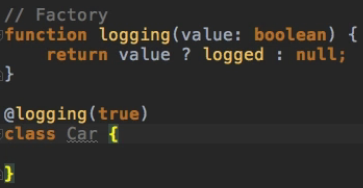
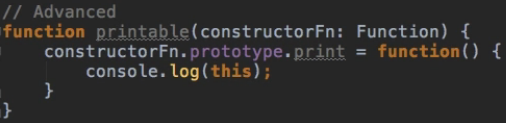
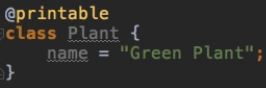
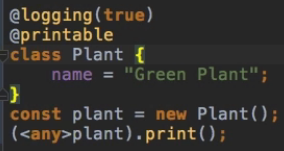
**Introduction**  
\* With Interfaces and Generics, we had a look at 2 new features introduced by TypeScript./  
\* **Decorators** are functions you can attach to classes, to methods, to properties and then work with these classes, methods, properties, or transform them.  
\* **So it’s kind of a bit of metaprogramming**.  
\* Decorators give you the chance of extending a class, of adding an extra functionality simply by adding a little thing and then all of a sudden, introducing a whole lot of new functionality to a class for example, or to a property.

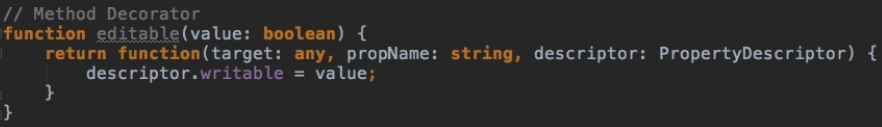
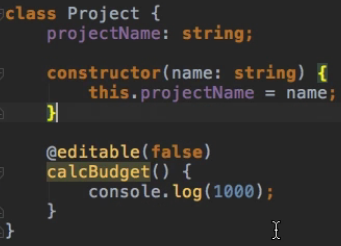
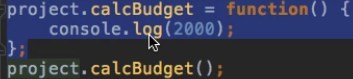
**Creating a Class Decorator**  
\* Let’s start with a Decorator which gets added to a class.  
\* **Decorators attached to a class will get 1 argument by TypeScript**.  
\* Decorators are a TypeScript feature so TypeScript manages which arguments will be passed to a Decorator depending on to what this Decorator is attached to.  
=> **You can’t change this**.  
=> **If you attach a Decorator to a class, it will get 1 argument and that is the constructor function of the class**.  
  
  
=> **This is the implicit constructor**.  
  
  
\* **Decorator is a normal function, you make it a Decorator by appending it with the @.**

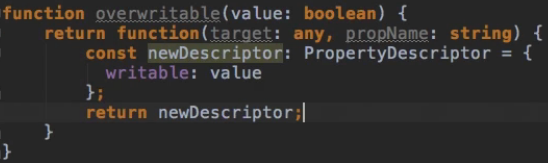
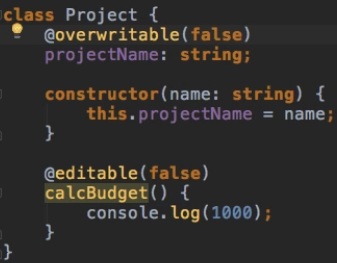
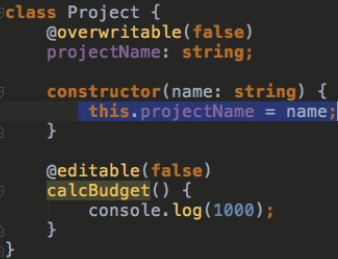
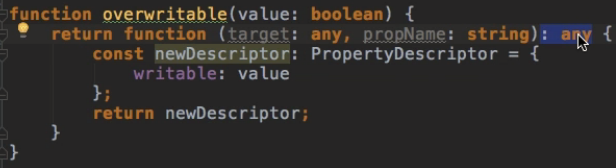
**Decorator Factories**  
\* Let’s say you want to have this logged decorator but you want to be able to decide if this should be logged or not.  
\* **Factory has to return a function which can be used as a Decorator**.  
   
\* I can only attach the logged function because it gets the constructor, logging is not really a decorator attached, I’m attaching the logging result since I’m executing the logging function and whatever the function returns, is the logged function which is a valid function to be used as a class decorator because it gets this constructor argument.  
  
**Creating a useful Decorator**  
\* I will take the prototype of the constructor function and keep in mind that’s just the prototype of any object created based on this class or based on this constructor function in the end, that’s just how JavaScript works.  
  
=> This should do the following: each object created or instantiated based on a class which has the printable Decorator will automatically receive the print() method because I’m adding it to the prototype.  
  
\* Unfortunately, I have to explicitly cast this to type any, otherwise TypeScript doesn’t really recognize that this print function does exist even though it does - just a little bug it seems.  
  
  
\* The constructor function basically is the blueprint here:  
  
=> That’s just how JavaScript works, the class is just syntactic sugar - we have a constructor function in the background.  
=> And then on the prototype of this constructor function - so kind of the parent class of all objects, I add the print method.

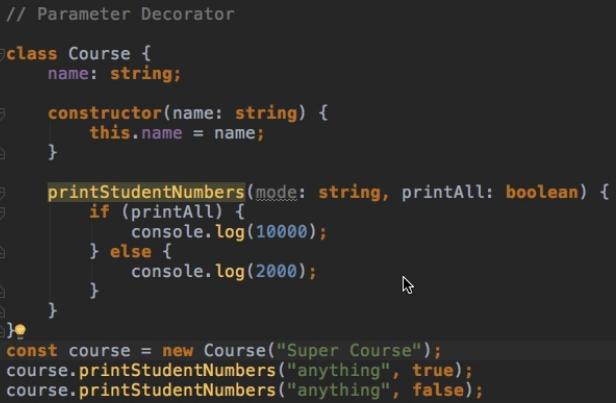
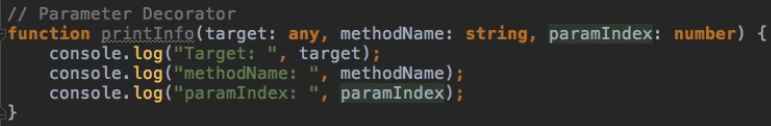
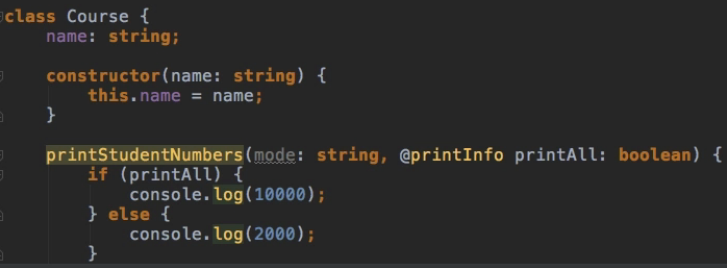
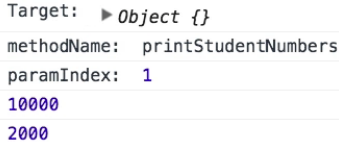
**Using Multiple Decorators**  
  


**A first Summary**  
\* **Which arguments can be used by the Decorator function depends on where you want to attach that function as a decorator**.  
\* You use the @.  
\* You can either directly attach the Decorator you want to use by leaving out the parentheses ().  
\* In this case you have to make sure that the parameters passed to the functions match the parameters that TypeScript will give you for this type of decorators. So a class decorator in this case.  
\* The alternative way is to use a factory which you have to execute by adding parentheses and any potential arguments you specified. You’re totally free about which arguents you expect here. The Factory has to return a function which can be used as a Decorator - which means a function with the right parameters.

\* How do we know which parameters a function should get for the different types of Decorators?  
<http://www.typescriptlang.org/docs/handbook/decorators.html>

**Method Decorators**  
\* I want to write a decorator which allows me to make this calcBudget() method editable or not editable because right now I can change the function:  
  
\* **I want to add some metadata to the method to make it not editable**.  
\* The function we attach to a class takes 1 argument - constructor function.  
\* The function we attach to a method takes   
1st argument - **target**: **any** because that varies on the question if we have a static class or a static method or instantiated one - so if we’re calling a static method, then this would resolve to the constructor function, if we’re using an instantiated one, target would be the prototype of that object.  
2nd argument - **propName**, name of the property  
3rd argument - **descriptor**: **PropertyDescriptor**  
=> Descriptor has nothing to do with TypeScript or metadata, it has to do with plain old JavaScript and it comes from this Object.defineProperty() method which allows us to create new properties with vanilla JavaScript.  
=> Descriptor allows you to configure a property, make it reconfigurable so that you can change the configuration afterwards, can make it a new variable, assign a value, make it writable and so on.  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty>  
\* **With our metadata we get access to the descriptor and we can change it**.  
  
   
=> **Now we’re blocking the editting of this method**.  
  
=> **Now we print 1000 twice because we’re preventing the method from being edited**.  
\* So this overwriting attempt here doesn’t work:  
  
\* **It’s not throwing an error but it doesn’t change the function either**.

**Property Decorators**  
\* **There is a difference between Property and Method decorators because due to some technical reasons, TypeScript can’t access properties in the same way it can access methods**.  
\* **Therefore, if we wanted to make this projectName for example uneditable, we might get unexpected results**.  
1st argument - **target**: **any** because that varies on the question if we have a static class or a static method or instantiated one - so if we’re calling a static method, then this would resolve to the constructor function, if we’re using an instantiated one, target would be the prototype of that object.  
2nd argument - **propName**, name of the property  
=> We don’t have property descriptor because TypeScript can’t access it.  
\* **BUT this function can RETURN a PropertyDescriptor**.  
=> **So we can SET the Descriptor but we can’t access the existing one**.  
  
  
\* Let’s print the whole project so that we have a look at what this does:  
  
  
=> The reason for the empty object is that with our overwritable metadata here, I’m setting writable to false in this case and that prevents every writing access to this property, including the one here in the constructor:  
  
=> **So no value gets set at any point and what’s why we get this undesired result of not seeing anything**.  
\* It’s hard to use that PropertyDescriptor because of the fact that for example changing the writable state will pretty much lock up this property for your whole code.  
\* However, you could use some other logic which might be more suitable or you might want to change the enumerability or the configurability of that property, then this might be fine.  
\* Otherwise, this Property Decorator should mostly be used to read values and react based on that - so as a listener kind of which then does things when you attach it to a property.  
\* **If you want to return something, you should add any or void if you don’t want to return anything**:  


**Parameter Decorators**  
\* Decorators attached to parameters passed to methods.  
   
1st argument - **target**: **any** because that varies on the question if we have a static class or a static method or instantiated one - so if we’re calling a static method, then this would resolve to the constructor function, if we’re using an instantiated one, target would be the prototype of that object.  
2nd argument - **methodName**, name of the method where this paramater is used.  
3rd argument - **paramIndex**, so the index of the paramater - which position this parameter has in the list of arguments passed to this method, starts at 0.  
\* **You can obviously execute any logic you want in the decorator function**.  
  
  


**Module Summary**  
\* Decorators are really an interesting feature and they’re in development, so we may see some additions or changes to them in the future, something of course I will inform you about in this course.  
\* **Basically decorators give you the ability to transform certain classes or add new data to it or add new information to it**.  
\* Decorators may aslo be used in conjunction with metadata, metadata is something not really related to TypeScript, not introduced by TypeScript, which is why we don’t cover it specifically here, but if you want to learn more about metadata and decorators and how you may use them together, definitely have a look at the official TypeScript webpage > Documentation > Handbook > Decorators. At the very bottom, you will learn how to add metadata with a decorator which might be interesting to you in your project too.

**Resources**  
Decorators  
<http://www.typescriptlang.org/docs/handbook/decorators.html>  
Object.defineProperty  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty>