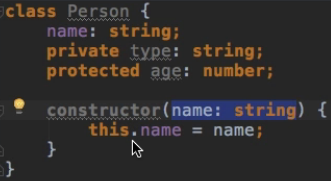
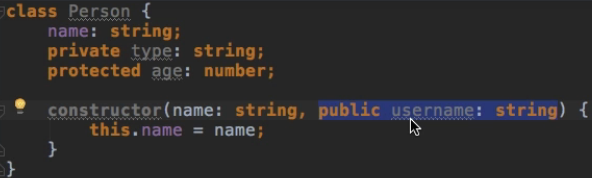
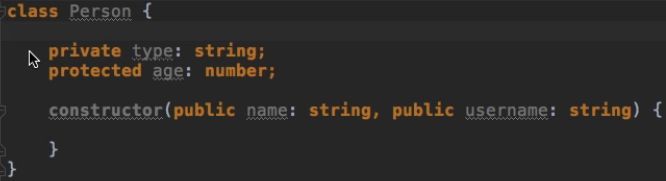
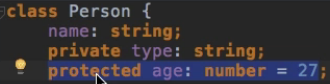
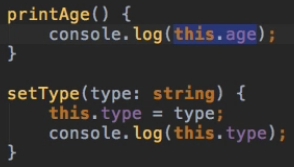
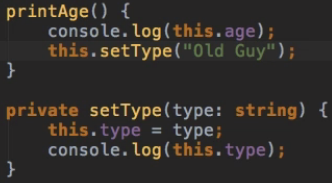
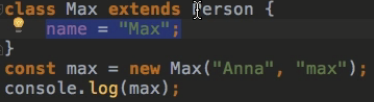
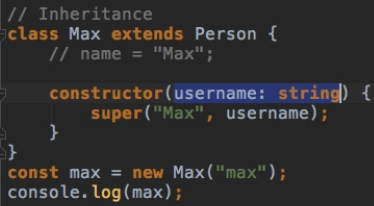
**Introduction**  
**Class** => It’s a feature ES6 introduces which makes working with objects a lot easier/cleaner.   
\* **TypeScript also offers classes and it turns out that TypeScript’s classes are even more powerful than ES6 classes**.

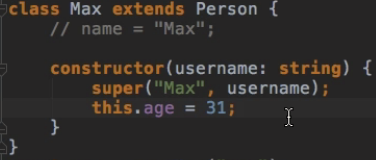
**Creating Classes and Class Properties**  
\* Classes basically allow you to prepare some kind of blueprints for objects.  
\* **So you can later on create objects based on a class to then make sure that this object already has some properties or methods which you define in that class**.  
\* You don’t have to set the properties with `this` keyword in the constructor of a class, you assign them directly in the class body, as you might know it from other languages like C# or Java.  
**public => makes the property available to whoever wants to access it, it’s public by default.**  
**private => makes the property available only within the class itself or whichever object you later create based on this class.**  
**protected** => **like private but additionally, they’re also accessible from any objects or classes which inherit from this class**.  
\* You can, however, provide a constructor() function which may take some arguments and which then allows you to assign this.property to property.  
  
\* **There’s also a quick shortcut TypeScript offers you for setting up a property which you also want to set in the constructor**.  
  
=> With the `public/private/procted` I’m telling TypeScript to not only expect to get a userName argument in the constructor function, but also automatically create a public/private/protected property in this class and assign the argument you get passed into the constructor, to this property.  
\* **It’s basically a shortcut for cre ating a property up there like the `name` and then getting the argument and assigning the argument**.  
\* This would work the same:  
  
\* **We use a class by instantiating it with the new keyword, which means I create an object based on this class**.  
  
  
\* It’s of **type Person**.  
\* We can also access the properties like this:  
  
\* **You can also initialize properties right when declaring them**:  


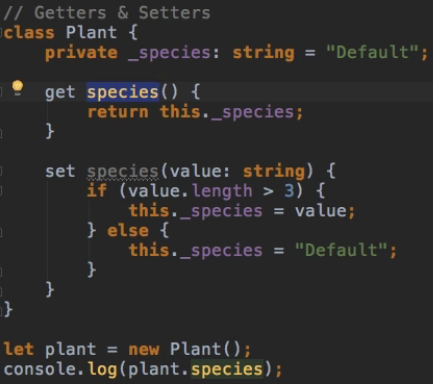
**Class Methods and Access Modifiers**  
\* **We set up a method like a function without the function keyword**.  
  
\* **Methods can also be public / private / protected**.  
  
\* **You have to use `this` if you want to access properties or methods of an object when you’re inside of it**.  
\* **Your classes can also inherit from each other, so you may extend classes and implement inheritance**.

**Inheritance**  
**extends**  
\* This means take the class you’re extending, use all of the code inside of it **-except for private-** and now whatever I specify in the new class, gets either appended if it isn’t included in the parent class, or overwritten if it was included.  
  
=> Here the name would still be “Max”, because the content defined in the main class always overwrites the content of your superclass, of your parent class.

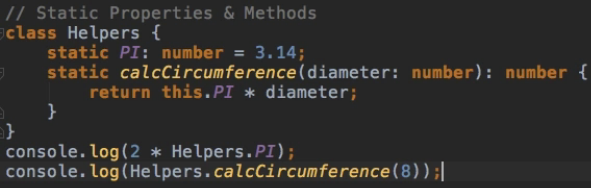
**Inheritance and Constructors**  
\* I can add a constructor to this child class to make sure I don’t have to pass a name like in the parent because the name is set anyway.  
\* When extending a class, you always have to call super() first, which calls the constructor if the parent class.  
\* I didn’t have to do that before because I didn’t have a constructor and in such a case, the parent’s constructor will always be called.  
**super()**  
=> **Refers to the parent class**.

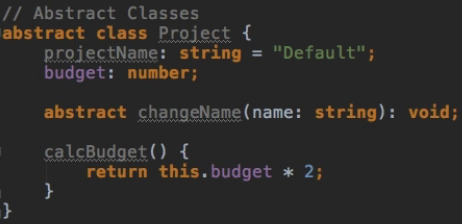
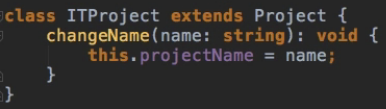
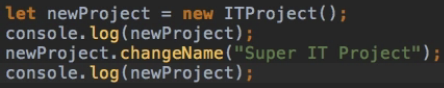
\* I’m not using the access modifier for username because that’s already defined in the parent.  


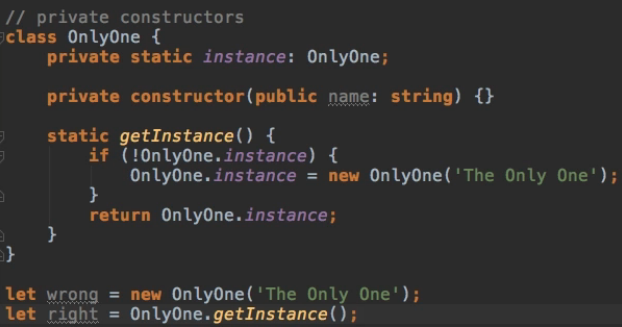
**Inheritance Wrap Up**  
\* What if I wanted to change the age?  
\* **Notice that age is protected, which means it’s available in child classes too**.  
\* **I wouldn’t have access to `type` because it’s private**.  


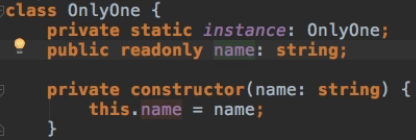
**Getters and Setters**  
\* Vanilla JavaScript also has some ways to create getters and setters but TypeScript makes it really easy.  
\* **Generally, you shouldn’t prefix your private properties with \_ underscore, t hat’s the style guide for TypeScript, but in this case I’ll make an exception - because I don’t want to be able to access this \_species property from outside but I want to offer a controlled access, which means through a getter ot setter which allow me to execute some code before actually returning or setting the value**.  
\* The `species` will be the property name accessible from outside later on.  
\* I do add parentheses () but I won’t call it like a method from outside, that’s important to understand.  
=> I’ll call it like a property but I set it up like a method because I can execute some code here, like a method which gets called.  
\* You can think of this value being passed to this kind of only theoretically existent method here, automatically.  
\* That’s why I have the \_ underscore - because I want to use this `species` name and therefore I have to make sure to have a different namy for my private property.  
**get  
set**  
  

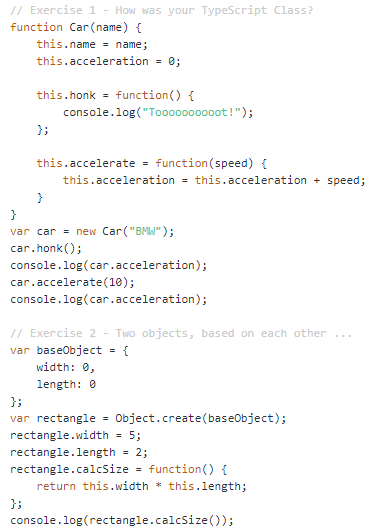
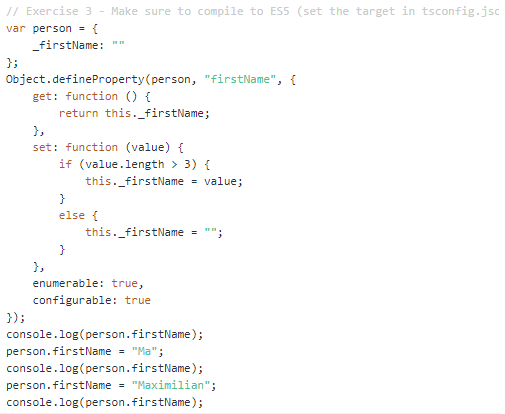
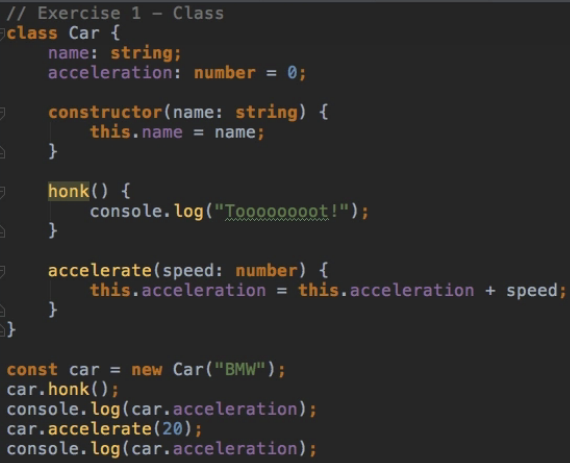
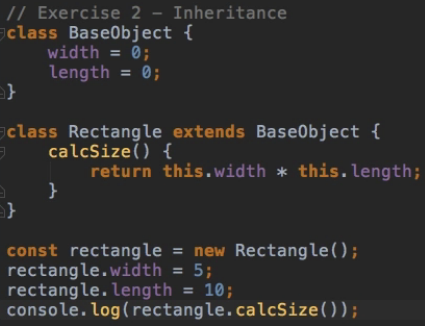
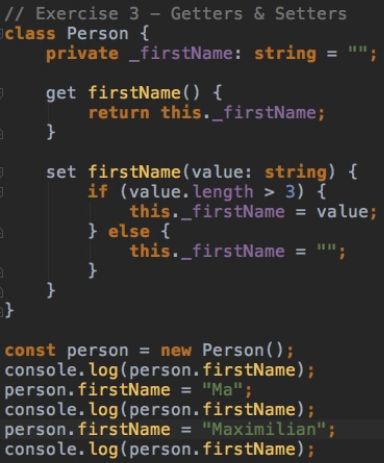

  
  
  
  
\* So that’s a convenient way to use getters and setters to control access to your properties and make sure that certain criteria are met before assigning a value or returning a value.

**Static Properties and Methods**  
\* This class should bundle some helper properties or methods I can use throughout my application for example.  
\* We don’t always want to instantiate classes.  
**static**  
=> **You can use static properties/methods without instantiation the class**.  


**Abstract Classes**  
**abstract**  
**Abstract Class => Abstract classes cannot be instantiated directly**.  
=> **You have to inherit from them, always, they’re basically just there to be inherited from**.  
=> **The class might provide some basic setup that other - more specialized - classes which then will be instantiated, need**.  
**Abstract Method => you don’t have the function body, we only define what the function should look like, we’re not implementing any logic.** **=> Once we extend the abstract class, we need to implement all abstract methods.**  
\* If we don’t implement the abstract method, we get:  
  
  
  
\* We don’t see the budget because we didn’t initialize it or set it.

**Private Constructors & Singletons (added with TypeScript 2.0)**  
  
\* How can we use a class if we can’t construct it, if we can’t create an instance of it?  
**Singleton Class**  
=> **Allows you to set up a class where you only want to have 1 insteance during runtime**.  
\* **This pattern is something you see in a lot of languages**.  
\* **It might be useful if you know that you want to have an instance of this class but you only need 1 instance** **because let’s say we want to have an array of data in this class and this array has to be the same in my whole application**.  
\* **By making the constructor private, you can force this class to only be used as a Singleton**.

**“readonly” Properties (added with TypeScript 2.0)**  
  
\* The `name` is created with the shortcut, it’s still accessible outside.  
  
\* **What if we wanted to make sure that we can only set it in the constructor but not at any other place?**  
\* **I want to make it readable though**.  
\* **There are 2 ways to achieve this:**  
1) **Using a getter** - if you don’t specify a setter but only a getter, then you would make that property read-only.  
\* If that’s our goal, we can take a shorter route and simply do:  
**readonly**  
2) **readonly** - in front of the property.  
  
\* To avoid the IDE error, you can take the long way:  


**Module Exercise: Problem**  
\* Re-write all this code using TypeScript and Class Features.  
   
\* In Vanilla JavaScript, there are a couple of ways to implement inheritance because there’s this rather strange concept of inheritance using prototypes - not really strange but less intuitive than the extends keyword in TypeScript.  
=> Have the base object, then create a new object with Object.create() which will set baseObject that you pass, as the prototype of the new object, therefore I have access to width and length of this object as it is based on this object.  
  
  
\* With Object.defineProperty() in Vanilla JavaScript, we could define the property name and then configure the property in the JavaScript object passed as the 3rd argument.  
  
**tsc solutions.ts -t ES5** => that’s how to set what you want to compile it to, for 1 file.

**Module Summary**  
\* Don’t mistake TypeScript classes with ES6 classes.  
\* TypeScript classes are really more powerful than that.