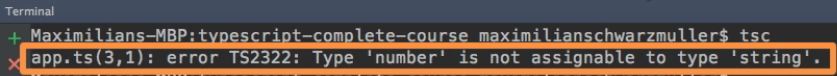
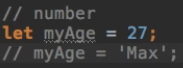
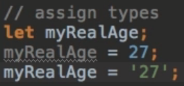
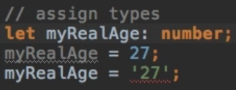
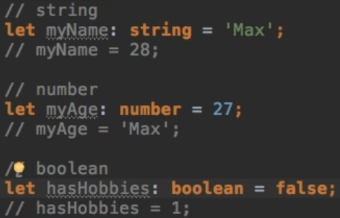
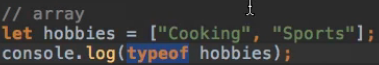
**Introduction**  
\* Why to use types.  
\* Why they are awesome.  
\* How to be flexible about types and keep the best of both worlds - Dynamic Types of JavaScript and Strong Typings of TypeScript.  
\* Which types exist.  
\* How you create your own types.

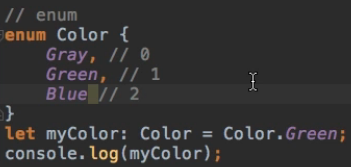
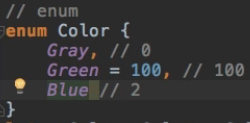
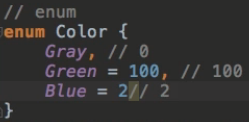
**Type Basics**  
\* Types also exist in JavaScript - we do have strings, numbers for example.  
**string**  
\* The issue with JavaScript simply is that we’re not explicit about them.  
=> When we create a variable, we’re not explicit about the type that this variable could/should/may have.   
\* In TypeScript, we can be explicit.  
  
=> **TypeScript recognizes that I assign a string here and therefore automatically saves this information**.  
=> It keeps in mind that this variable here should be of type string.  
  
  
=> **TypeScript recognized that this should be a string because we assigned one initially and therefore we’re not able to overwrite this with a number**.  
\* **That’s the big difference to JavaScript, in JavaScript, we would be able to do this because JavaScript has Dynamic Types and here we do have kind of Static Types**.  
\* **We define them once - either explicitly or implicitly like here, letting TypeScript infer it**.

**Numbers & Booleans**  
  
\* Important to know regarding numbers - these can also be floating point numbers. That’s also a valid number.  
=> **TypeScript doesn’t care about the type of the number - interger / float**.  
**number** => both interegers and floats.  
  
**boolean**  
\* **Even though 1 could be cast to a boolean you could say, it still gives me an error because 1 is a number**.

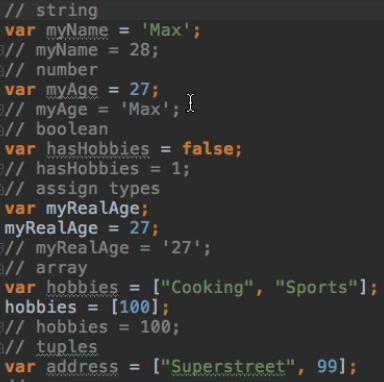
**Assigning Types Explicitly**  
 **any**=> **Since I don’t assign a type in the declaration, TypeScript treats this as type any**.  
\* **When creating a variable, we can assign a type by writing : type**.  
  


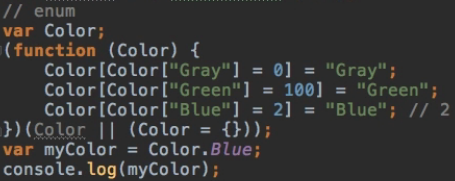
**Arrays and Types**  
**typeof** => checks the type.  
  
  
=> This gives us Object, that’s not very surprising because obviously an array is an object.  
  
  
**array of strings / array of numbers**  
**[]**\* If I do this:  
  
\* **That would take it as an any[].**

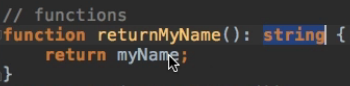
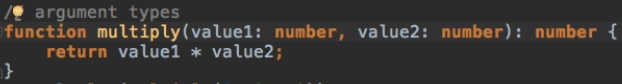
**Tuples**  
**tuple** => **[type, type] array with mixed types and a limited number of items**.  
  
=> **The order is important**.  
\* You can use them if you’ve got some data object or some array which has 1 single way it may look like - if you get an **array which has a fixed value format**.

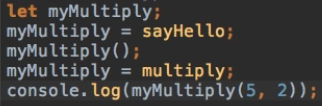
**Enums**  
**enum** => **make numbers more expressive, encode numbers into more user-friendly values which you can use during coding**.  
\* Let’s say you have a couple of colors and you want to check in a switch statement which one was chosen.   
\* You could have a switch statement where you check for strings being “green” or “blue”, so you have a string representing the color.  
\* That would work.  
\* But it’s just occupying more memory for information your computer doesn’t really care about.  
=> **You could also just have 0, 1, 2, 3 each number standing for one color**.  
=> **That would be how you could normalize this, which also has some other advantages, saves you the danger of making typos and so on, so that would be a good thing**.  
=> But it is of course a bit harder to remember which value stands for what color.  
=> To make this eaasier, you can use enum.  
  
  
=> **This will basically create a new type we can use later on**.  
=> Behind the scenes, numbers are assigned to that automatically.  
\* **You might also have some applications where you want to overwrite the default values taken**.  
  
=> **It always continues counting based on the last item that was defined, or from 0**.  
  
\* **Now another color would be 3 and if we didn’t assign Blue to 2, then Blue would be 101**.

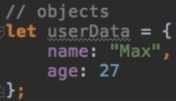
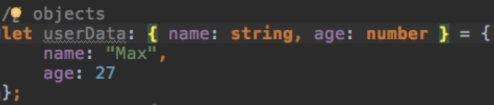
**The “Any” Type**  
**any**  
\* **It should really be an exception that you use any, try to avoid it, try to be explicit, try to really find out which type you want to use and only if you really can’t tell and need all the flexibility, then use any**.

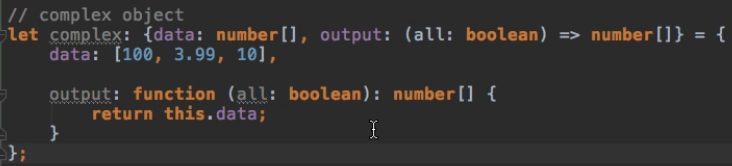
**Understanding the created JavaScript Code**  
\* What does all this code we saw compile to?  
\* How does it look in JavaScript code?  
\* We can simply look at the app.js file.  
  
\* The types are simply stripped out.  
\* The role of TypeSscript is to be aware of the types and warn you at compilation time.  
\* Once the code has made it to the JavaScript code, you’re done.  
\* The only thing we’re getting then are runtime errors again, so types are no longer available there.  
\* But it was the job of TypeScript to make sure you wouldn’t assign anything wrong before we get to that step, before we actually compile the code.

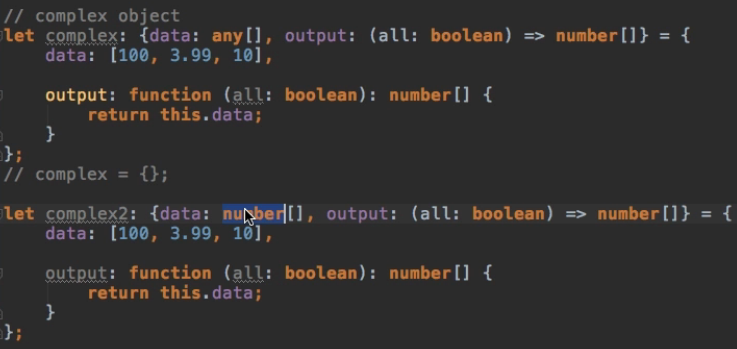
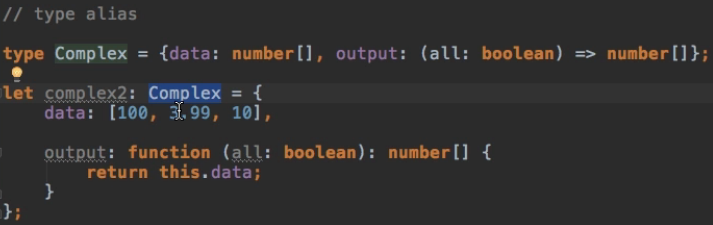
\* The only thing you do see is the enum which is kind of constructed with this self-executed function and the color object. So the enum is created in code too, so that we have this mapping then.  


**Using Types in Functions (Arguments & Return Values)**  
\* **Type of the return value**:  
  
**void => a special type, when a function doesn’t return anything.**  
 **=> It gives you an error if you try to return something.**  
\* **NaN** => **a special type of JavaScript to indicate that we were expecting a number but it isn’t**.  
  
**argument types**  


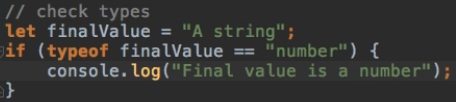
**Functions as Types**  
\* It feels kind of wrong to be able to assign it to both of these functions. **(arg1: type, arg2: type) => type**  
=> **It indicates which functions this variable will be able to hold**.  
=> **The names of the arguments are ignored, you can name them anything you want**.

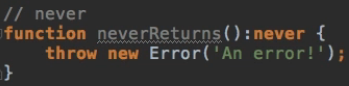
**Objects and Types**  
\* **TypeScript infers this to be an object with `name` field which is of type string and `age` of type number**.  
  
\* **Here the names of the properties are important**.  
=> **Objects** - **what matters is the names, the order isn’t clear**.  
=> **Functions** - **what matters is the order, the names don’t matter**.  
  
**{ field1: type, field2: type}**

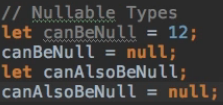
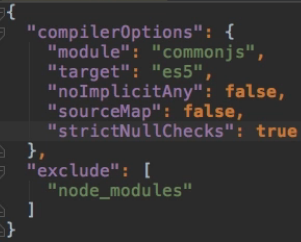
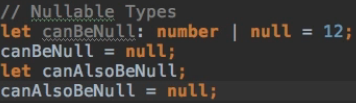
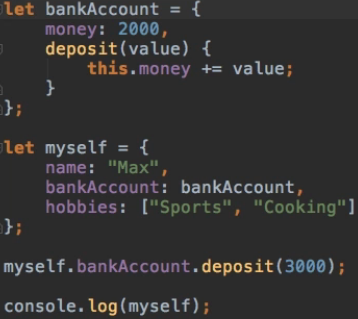
**Example: Putting it all together in a Complex Object**  


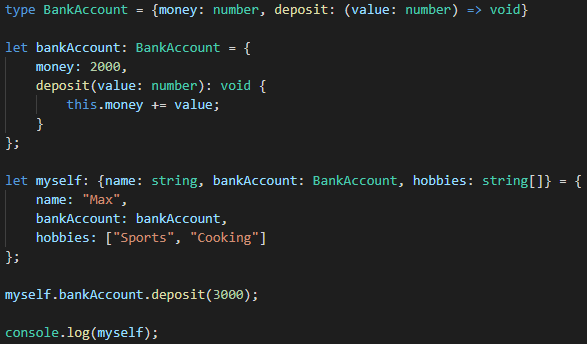
**Creating custom Types with Type Aliases**  
\* Complex types like these lead us to an issue though.  
\* Imagine we would have another variable here.  
\* If I want to re-use the type, I have to copy it and if I want to change it, I have to change it in both places.  
  
\* So a better approach would be to have a way to store this type.  
\* Later on in this course, we’ll learn about classes which allow us to create blueprints of which we can create objects.  
\* But we can also use a **Type Alias, if we just want to store a type**.  
**type => this tells TypeScript to create a new type.**  


**Allowing multiple Types with Union Types**  
**Union |**   


**Checking for Types during Runtime**  
\* **We can use the typeof command**.  
  
\* You would use this for example in a function or code where you’re not really sure what values you are getting and you just want to do something upon certain types.  
\* For example for a calculation - you only want to perform it when you’re getting numbers.

**The “never” Type (added with TypeScript 2.0)**  
\* Why is this never?  
**never**  
=> **Because this function never finishes, it doesn’t return void, this function returns an error, so it’s not returning nothing, it’s never returning anything.**  
  
\* **This function throws an error and therefore never returns**.  
\* It is something you probably won’t use all over the place but you might use it in functions or in parts of your code where you know it should never be reached because it kind of then is stuck and never returns. And as here when you throw an error, that probably is not the place where you want to get with your code in the first place. Nonetheless, if you want to define such places in your application where you know that if you go there, then you never return.  
\* In such cases thata’s the type you’re going to use - or TypScript will automatically infer it anyways.

**Nullable Types (added with TypeScript 2.0)**  
\* Non-Nullable types means you shouldn’t be able to assign null to types except for if you explicitly state that you want to do that.  
\* **All uninitialized variables in JavaScript and TypeScript are undefined**.  
  
\* You might already be using this a lot if you want to reset a value, if you want to clear a value.  
\* **The issue is that this can lead to problems becausae you might have null in a place where you want to use this variable**, do a calculation or imagine it’s not a simple number but a more complex object and you try to access a property on this object, and if it’s null, you can’t access the property.  
=> **So you always have to check first if it’s null or not or find some other workaround**.  
\* **With TypeScript 2.0, you can be explicit about which may be null and which should never be null**.  
\* You can force this behavior by going into the **tsconfig.json** and adding a new field to the compiler option, which is called **strictNullChecks**.  
  
\* **There’s 1 exception - if your variable is UNDEFINED, it may also be null since null and undefined are these two special values in JavaScript**.  
\* **You can use UNION | if you want to allow a variable to be null as well**.  
  
\* **If we initialize it as null, it’s like its own type and if we set it to null at the start, it can only be null**.  
  
=> This would give us an error.  
**Module Exercise: Problem**  
  
\* Your exercise is to be as explicit as possible regarding the types.

**Module Exercise: Problem Code**  


**Module Summary**  
\* Types really are the meat of TypeScript.  
\* It offers a lot of other cool features but types are what give TypeScript the name and they make your programs much more safe or better.  
\* You will prevent a lot of nasty bugs or errors happening where you don’t get an error message to begin with, just by making sure that you are using the right types and not messing something up during development.