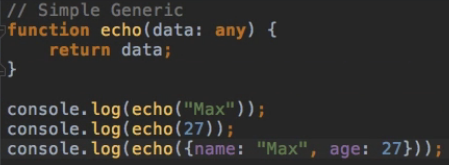
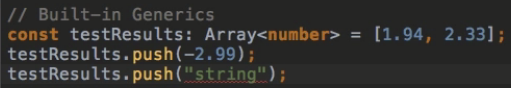
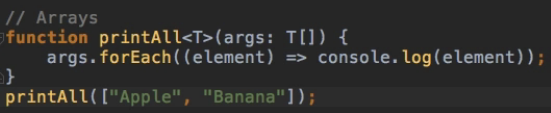
**Introduction**  
\* Generics are a powerful tool to write really dynamic code, flexible code and use both the power of TypeScript and the fact that it needs to know what type a certain property or variable for example has, while also giving you a lot of flexibility and writing certain functions which can work with a lot of types while at the same time making sure they’re getting the right type.

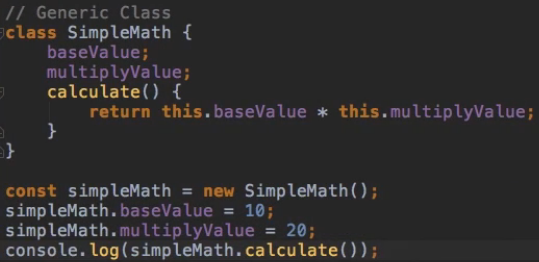
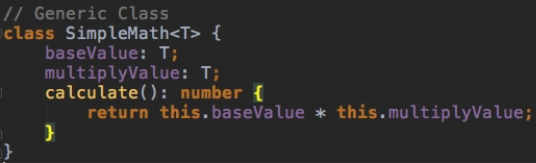
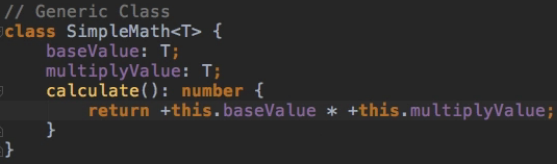
**Why and What?**  
  
\* The important thing to realize here is this function is kind of a generic function because we have this any type here but we do have a big disadvantage with this any time.  
=> I get back any data I enter.  
=> If I wanted to use .length here, my IDE wouldn’t offer it to me instantly.  
  
=> This means the IDE doesn’t know that we’re getting back a string.  
\* **Maybe the problem becomes clearer with the next example**.  
  
  
=> The issue is that a number doesn’t have a length.  
\* If I do the same on the object:  
  
  
\* The issue here clearly is that we don’t know the type here, it would be nice if we get a compilation error if we try to access length of number.  
\* But we can’t get this because the data type is any and therefore, TypeScript doesn’t know any better than we do - if the data actually has a length property for example.  
\* **So we might want a generic function like this which can handle all kinds of data but once we get back the data, it would be nice if TypeScript would be aware of the exact data type and not of type any**.

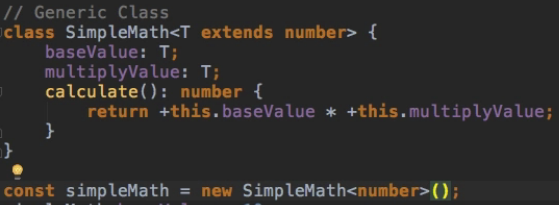
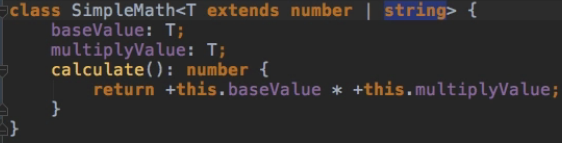
**Creating a Generic Function**  
**<T>**  
\* It can be any character.  
\* This strange construct makes this a **Generic** **Function**.  
\* With the T we’re telling TypeScript: hey, when using this function, give me the type and then I will be able to use this type.  
  
=> Now if I type . I get better IDE support and I can access length.  
  
=> **If I do the same for a number, it doesn’t offer me the length property**.  
  
\* With this setup, I have a Generic Function, I can use a number, but I can’t use the length on the number thereafter.  
\* **Since TypeScript is aware of the result type, I can’t do any operations with it that aren’t supported by that type**.  
\* You can also explicitly state which type you will use in this function.  
\* Right now TypeScript is inferring it by the type of the argument but you can also do it explicitly:  
  
=> If we passed “27”, we would get an error.  
\* **Generics are a powerful tool to improve your code, make your code both flexible and yet manageable by the TypeScript compiler**.

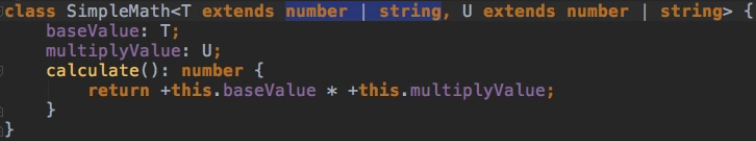
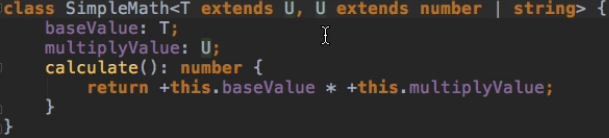
**A built-in Generic Type: Arrays**  
\* This would work and there’s nothing wrong with that:  
  
**Array<type>**  
=> **The IDE already complains that you’re trying to push a string into an Array of numbers**.  
\* **And same with TypeScript if we try to compile it**.

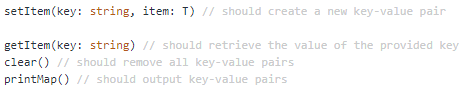
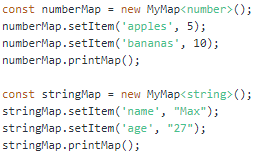
**Generic Types and Arrays**  
\* You can also assign a Generic type to be an Array for example.  
  
\* I can also be explicit:  

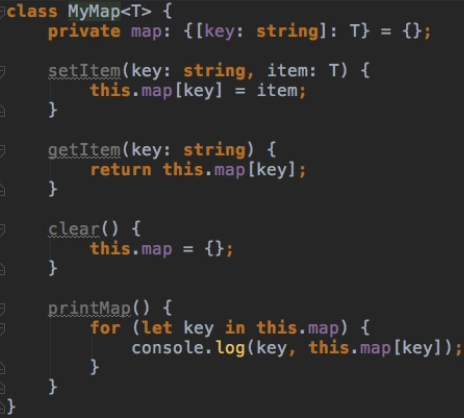
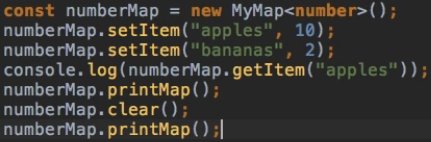
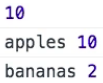
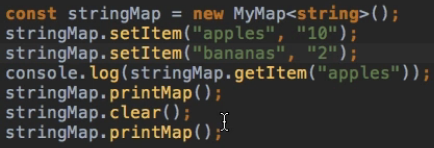

**Using Generic Types**  
  
=> We’re creating a new constant.  
=> I’m assigning a type.  
=> It’s a Generic Type.  
=> Then it’s a Function Type thereafter.  
\* **So I’m just telling TypeScript that this const will hold a function where the input is one argument of type whichever the user specifies and the function will return something of that input type**.  


**Creating a Generic Class**  
\* **If you use Generics, it will often be a Generic Class**.  
\* First let’s make it a normal class and without types.  
  
\* **The issue I got here i if I change one of the numbers to a string, I’m not getting any compilation errors**.  
  
  
=> **So we should probably make this more Generic to make sure that we actually can use it in a Generic way**.  
\* Let’s continuously improve this class.  
  
=> Now if we compile it, we do get some errors.  
  
\* The issue TypeScript has with the setup here is that we can enter any type and it recognizes this and it recognizes that this could be a type which doesn’t really match this.  
\* **TypeScript knows that T could be a value which can’t be used in this calculation**.  
**+** => Therefore I will explicitly cast these numbers by adding a + in front of them.  
  
\* **Now the errors have gone away if I use numbers again**.  
\* **But the problem which persists is if I enter a string or something, we’re not getting an error**.  
\* I’ll come back to this.  
\* For now we generated a Generic class even though it’s not that useful yet.

**Constraints**  
\* I can make it more useful by explicitly stating that T should extend something.  
  
=> **This tells which base types should be allowed**.  
=> It’s a **Generic** **Constraint**, I’m constraining which types can be used in this generic class.  
  
=> Now we’re getting an error.  
\* I could use a string because this could be cast to a number:  
  
  
  
=> Both have to be the same type so both numbers or both strings.  
\* **I wanted to show that you can create a Generic class, that you have the `extends` keyword here and that with that, you can control which values can be passed**.

**Using more than one Generic type**  
\* What if we wanted to allow this:  
  
\* **It’s a convention to use T, U, in alphabeyical order**.  
  
\* **Or we can extend the Generic we use if we want it to be the same type**:  
  
=> **Important to know - if I do it like this, then I’m NOT saying that T should have the same constraints as U, it means that T has to be the same TYPE as U. So if I pick number as a type for U, then T also has to be a number**.  
=> **So to be able to mix them, I have to add that constraint again**.  


**Module Exercise: Problem**  
\* Create a generic Map (an Object like an Array, but instead with Key-Value Pairs). The key will always be a string.  
\* Let's keep it simple and only add the following methods to the Map:  
  
\* The map should be usable like shown below:  


**Module Exercise: Solution**  
\* Map is basically a collection object where you store key-value pairs, like na object in JavaScript.  
  
   
   
\* That’s a pretty cool example of how such a Generic would make sense if you want to create a Map where you only want to store any type in theory but in one Map only 1 type.  
\* And again we get the IDE support because it knows the type.

**Module Summary**  
\* Generics are a really cool feature but you’re probably not going to use them all over the place, simply because they’re a really specific feature and oftentimes, you don’t need to use them.  
\* But you will know when to use them and then they really add extra flexibility and make your code more flexible for you as the developer.