#### **TypeScript**

------------------------------------------------------------------------------------------------------------------

#### **Introduction**

Tutorial for JavaScript Developers wanting to get started with TypeScript.

Thorough walkthrough of all the basic features.

We will go through the basic features to gain a better understanding of the fundamentals.

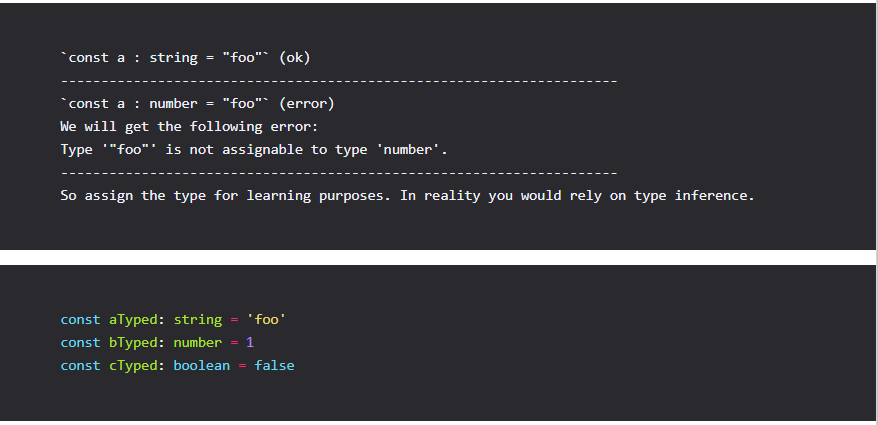
#### **Basics**

Let's begin with a couple of very basic examples

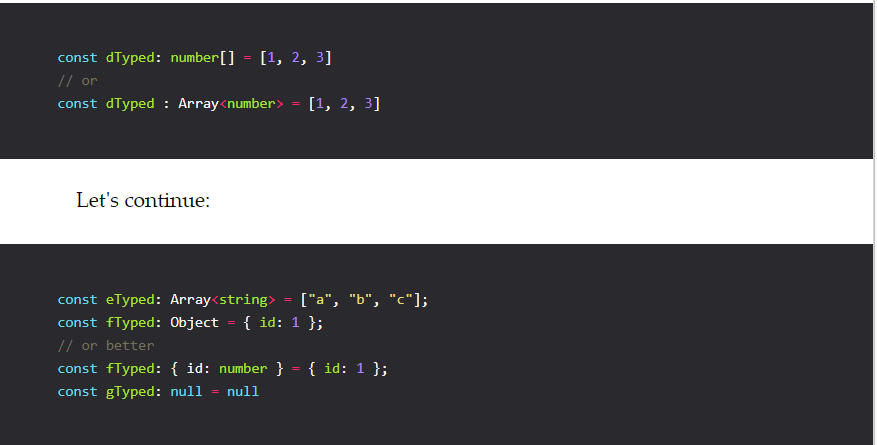


TypeScript offers a number of types.

To get a quick overview, let's be explicit and define types for the above constants.



The first three are relatively clear. But how do we type d?



What about undefined?

In TypeScript you can use the undefined type to declare a value as undefined.

We will get into more detail as we progress and cover primitives in more detail.

const hTyped : undefined = undefined;

Let's continue with better understanding the basic types in TypeScript.

For example the i and j could either be assigned a primitive type, but interestingly a literal type as well. How would that look like?

const i = 2;

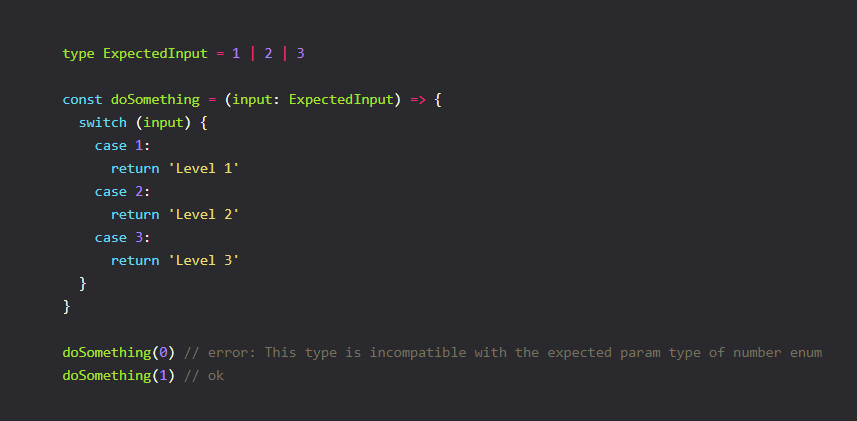
const j = "foo";

const iTyped: 2 = 2;

const jTyped: "foo" = "foo";

Now you might be wondering what value we gain from literal types?

We can constraint what values we expect.

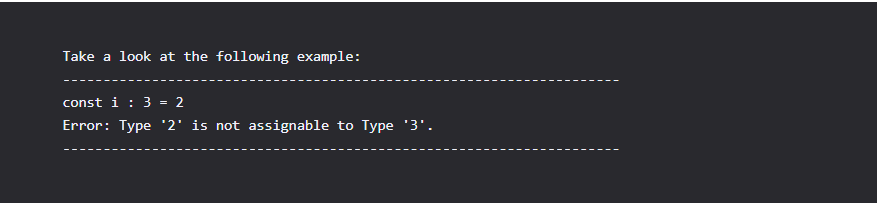


We were dealing with const variables up until now. What about let or var? While const variables can't be reassigned, so TypeScript can inter the type and know for sure it will never change.

This is not the case when working with let or var.



As we can see in the above example, once you assign a type to a let or var variable any re-assignment has to be of that same type otherwise TypeScript will complain.



### Any Vs. Unknown

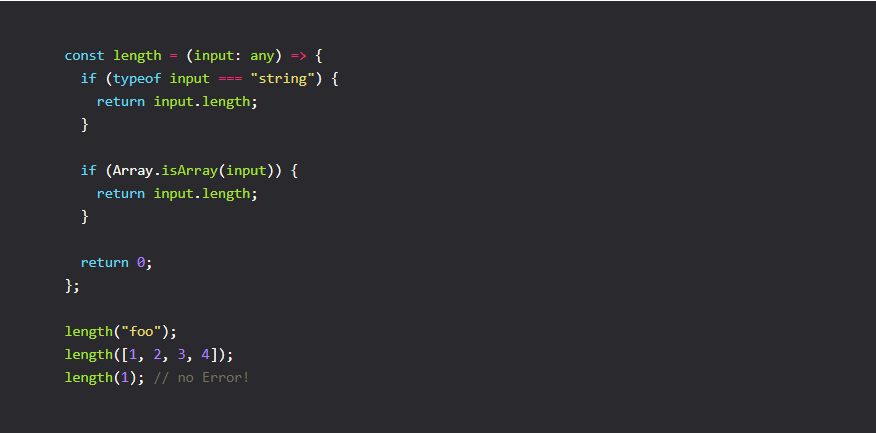
Sometimes you can't tell what the exact type is or you are currently converting from an existing non-typed code base gradually. Here is where any and unknown are helpful. It's important to note that they fulfill different purposes. any should be used as a last resort, as it skips type checking.

In contrast unknown is useful when you can't be sure what the input type, as the name already implies, is. Check the following example:



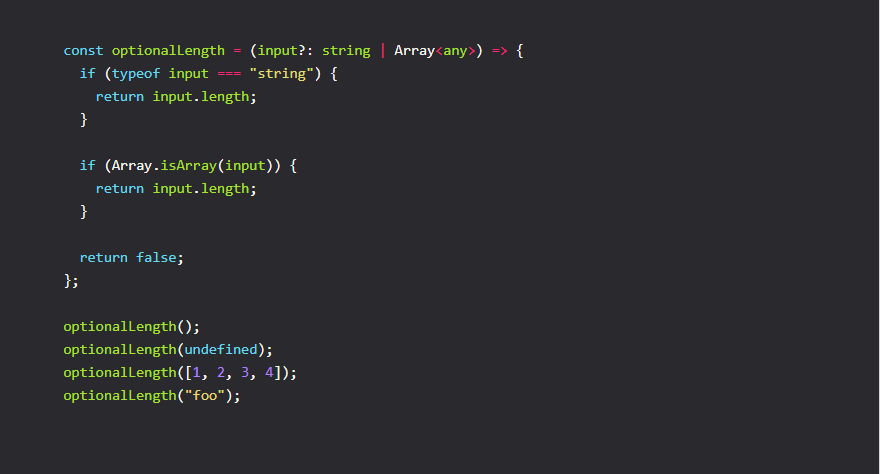
We need to refine the input by checking the type and then returning an appropriate value else the compiler will complain.

With any we completely bypass the type checker. We can pass in any value to length and will never receive an error. As already mentioned use any as a last resort if possible!



#### **Optional Values**

Sometimes we want a certain value to be optional. For example take a look at the following function:



As we can see, we can call optionalLength with undefined, an array or a string.

But as you would expect, passing in a number would cause an error.

optionalLength(1) // Error!

Also passing in null will result in an error:

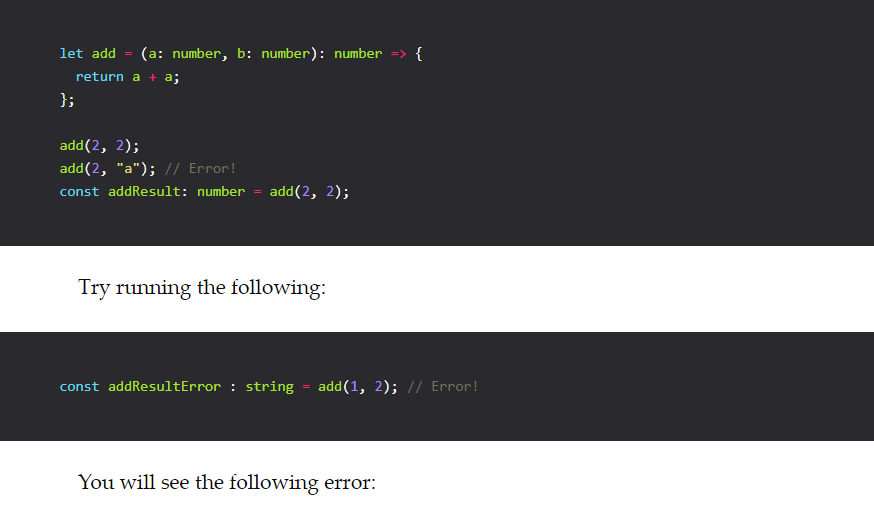
optionalLength(null); // error! We need to be explicit about null

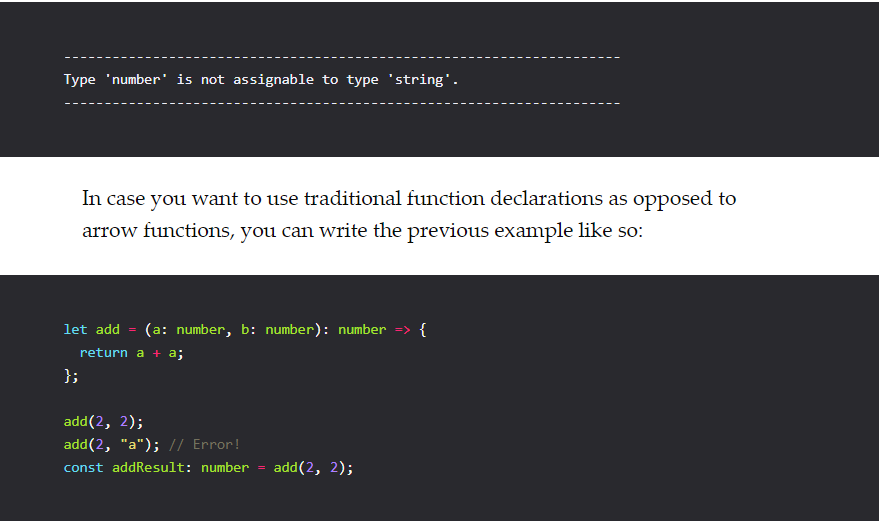
To fix this we need to change the type signature to expect null:

const optionalLength = (input?: string | Array<any> | null) => {};

#### **Functions**

Now that we have covered the very basics, it's time to get more advanced. We have already seen a couple of functions in the previous section, but let's take a more detailed look at Function types. First off all, we would like to type the input and output of a function, so let's see how this is done.



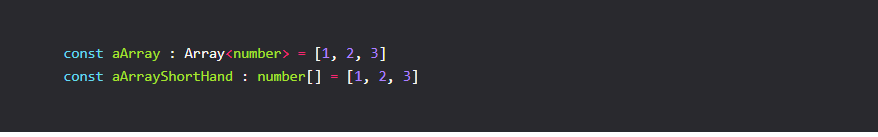


#### **Arrays**

Let's continue with arrays.

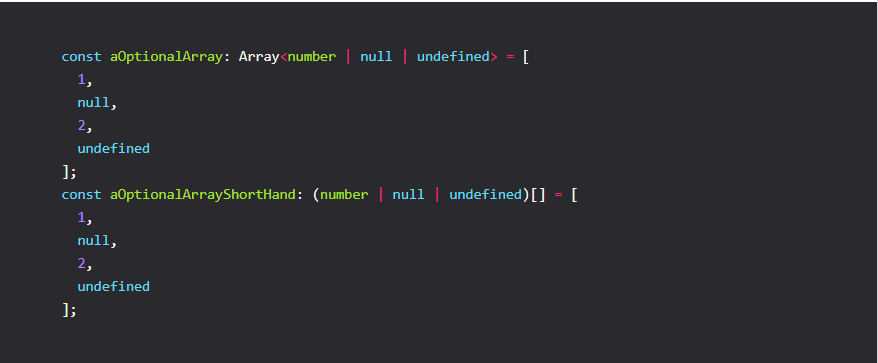
If you recall at the very beginning, we typed a simple array. There are two ways to type an array: Array<Type> or Type[].

So f.e. these two are equivalent:



What if we might have a null value inside our array. The answer is very similar to

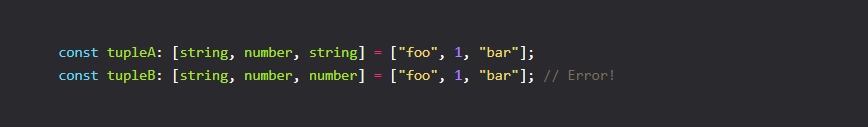
what we have seen in the Optional section. We need to be explicit about null or undefined in this specific case.



What if we want to be more specific with our array definition?

Take the following example:

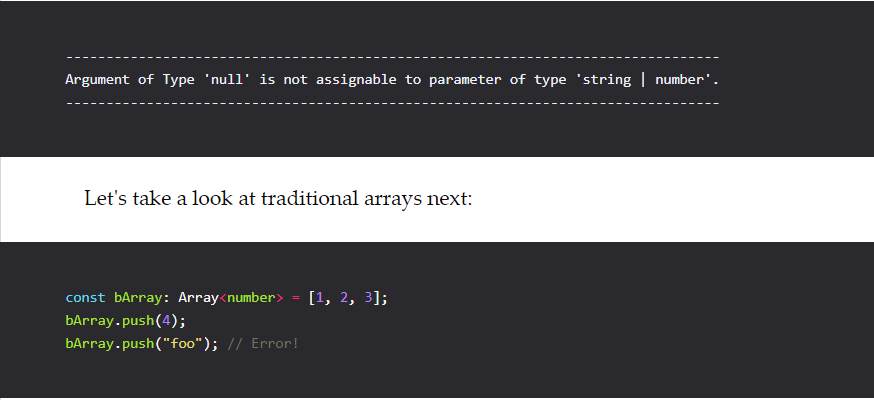
We have an array consisting of exactly three items, in short a tuple containing a string, a number and another string: ['foo', 1, 'bar'].



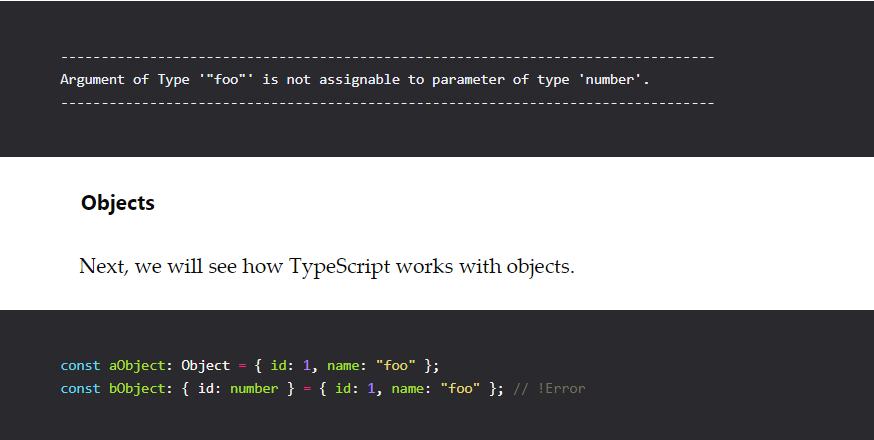
Another important aspect is that once you have a tuple defined, you can still use any of the existing Array methods which mutate the array. The compiler will not complain as opposed to Flow, where the compiler would complain.

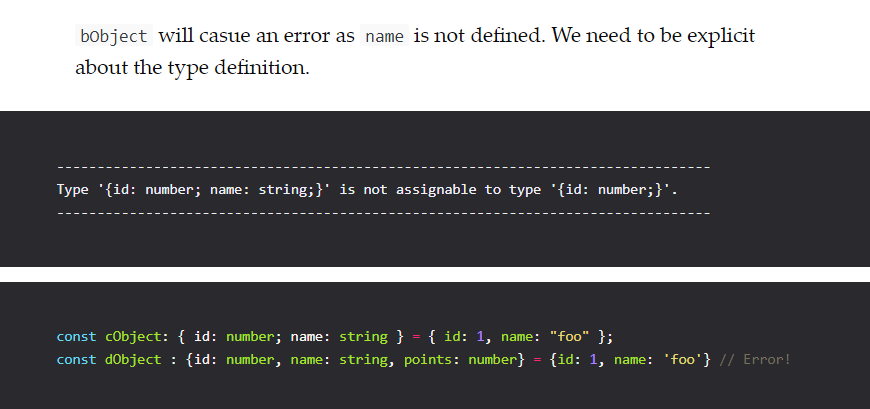


So once you define a tuple you can push any type that exists inside the tuple. For example pushing a null into a tuple that expects either string or number results in an error:

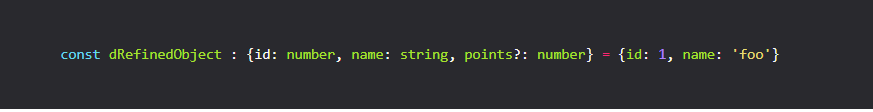


We can see similar results as in the previous tuple example. It's possible to push into an existing array, as long as the types match. For example pushing a string "foo" into an array of numbers would result in the following error:

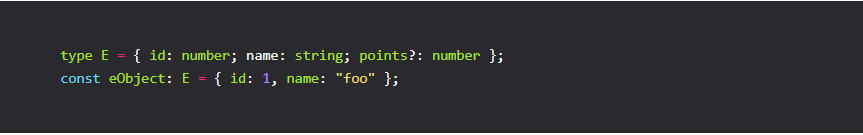




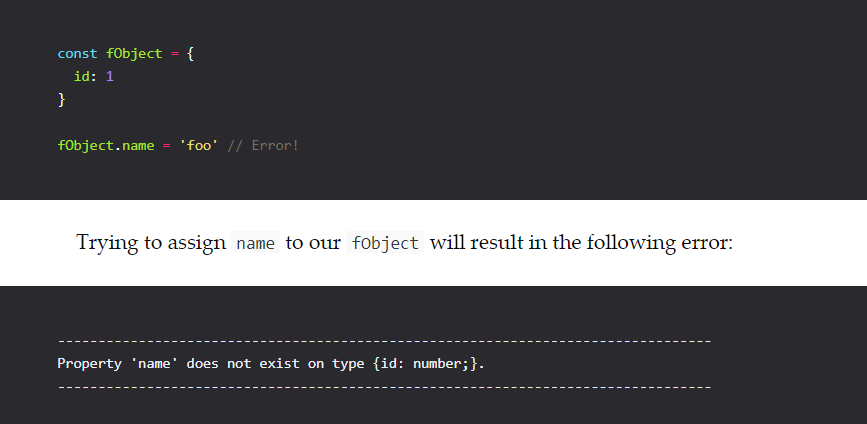
dObject will cause an error as points is not defined. We want to make points optional. We've already seen how to make a value optional, so let's see how to achieve the same for an object property.



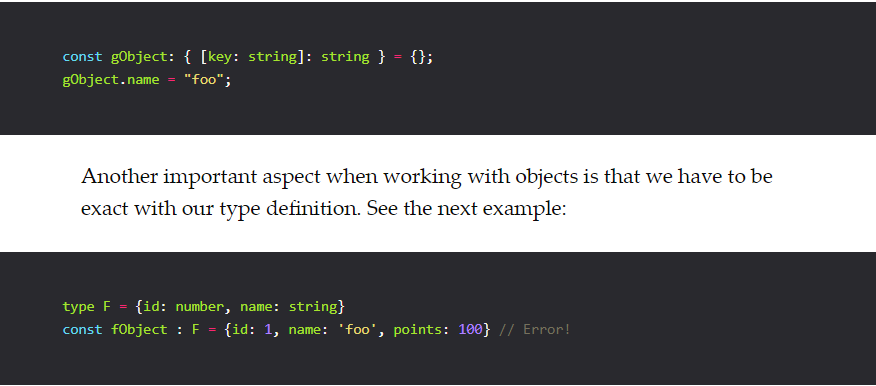
By declaring points?: number, we are saying that points might not be defined. To make things more readable, you will probably resort back to defining a type alias for the object declaration. This is especially helpful if you also plan to reuse a type definition.

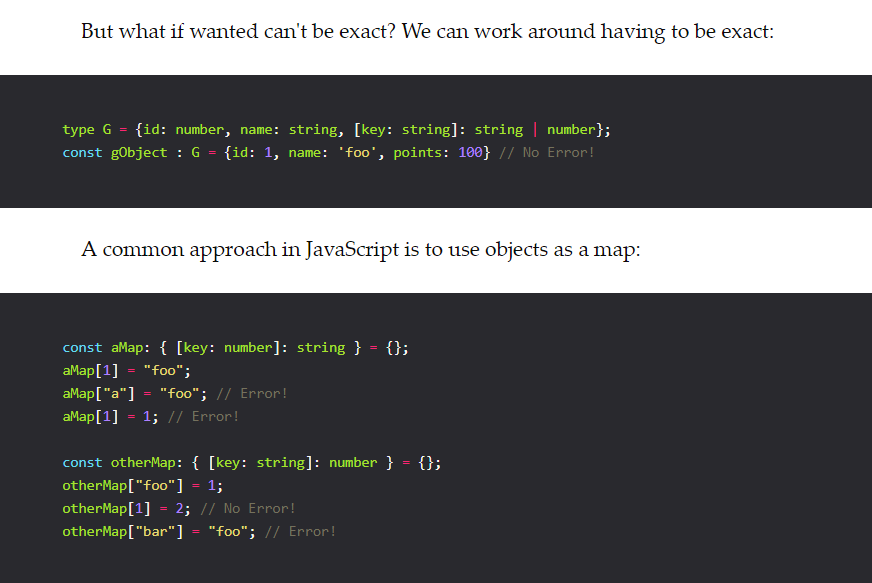


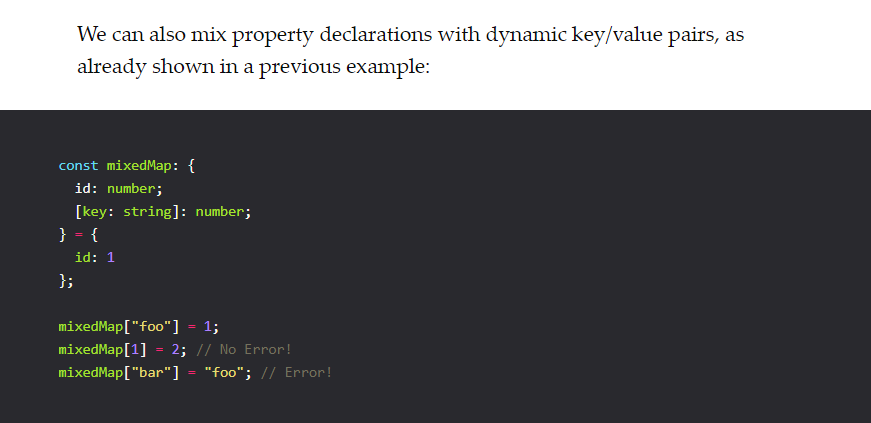
Another important thing to note when working with objects, is that we can not add non existent properties to a defined object. Take a look at the following code snippet:

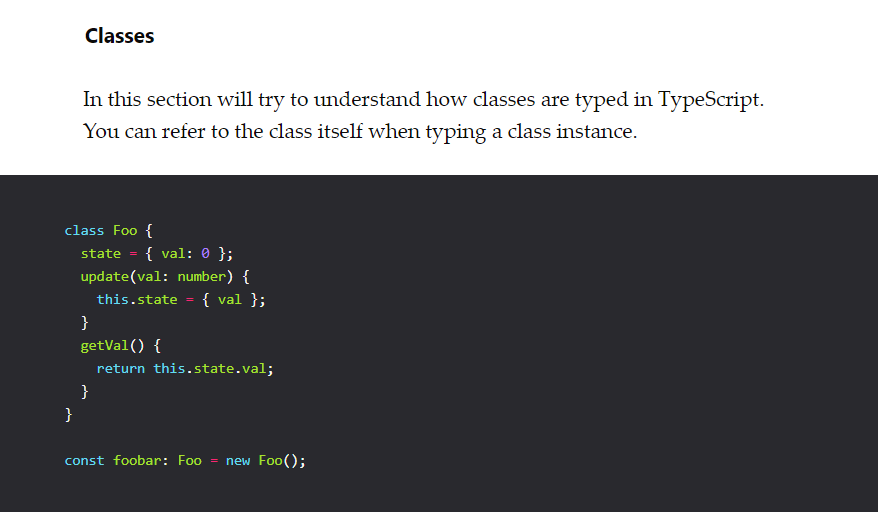


So the above doesn't work. Per definition we can add new properties to an object with defined properties. We need to be more explicit about the types, to be able to make an object expandable. We can define a type that expect a key of type string and string property types f.e.





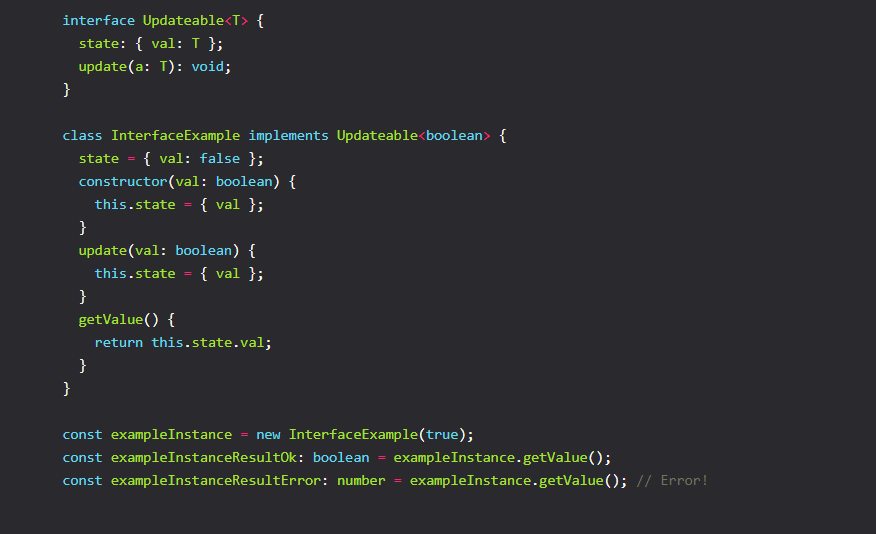






#### **Interfaces**

To round things off, let's also take a look at interfaces. What if we had a class Bar, that also had a state property and an update function?



#### **Generics**

Now we're getting into more advanced territory here. Up until now we should have covered all the necessary basics. Let's continue with our previous example and add generics. For example our Example class might also accept a string instead of a number. We want to abstract the type definition in this case.



If you uncommented the above example you will notice that everything works. Interestingly you don't even have to explicitly define a type for const exampleGenericString = new Example('foo').

TypeScript will know that our return value is a string, as can be seen in the following line. We can do a lot more with generics, like f.e. define type aliases or functions. Let's see some examples to get a better idea of the possibilities.



More detail typescript function knowledge:

<https://www.typescriptlang.org/docs/handbook/functions.html>