# **Variables**

Declaring a variable

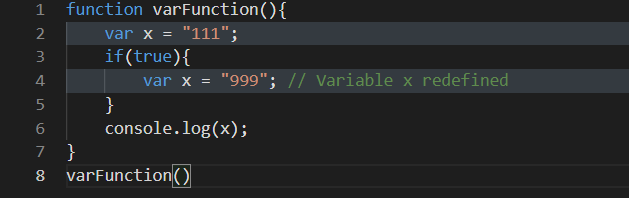
This lesson explains the archaic "var" that is supported in TS and describes the cost of using this old-fashioned way to declare a variable. We will see how we can maximize the declaration with "let" and "const" as alternatives.

Declaring with var

Let’s get started with var. This has been the way to define variables since the inception of JavaScript. However, the release of ECMAScript2015 brought the declarations let and const, which fixed many of the drawbacks perpetuated in previous versions of ECMAScript. In many modern languages, declaring a variable is as simple as being alive in the scope where this one was defined.

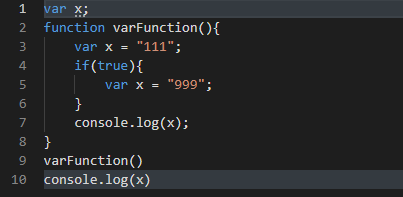
For example, a variable declared in a conditional statement is only available inside the conditional statement – not before or after.

One issue with var is that the location of a variable makes it unpredictable. A variable declared using var is function-scoped when declared inside a function, but global-scoped when declared outside of a function. Also, var does not stop us from redefining the same variable, which overrides the initial declaration or initialization.



As seen in the example, because the variable x is outside the closure of the if statement, the new declaration redefines x (line 4) and overrides its previous value (line 2).

However, the new definition is contained within the scope of the function. For example, trying to output x from outside the function will return undefined –



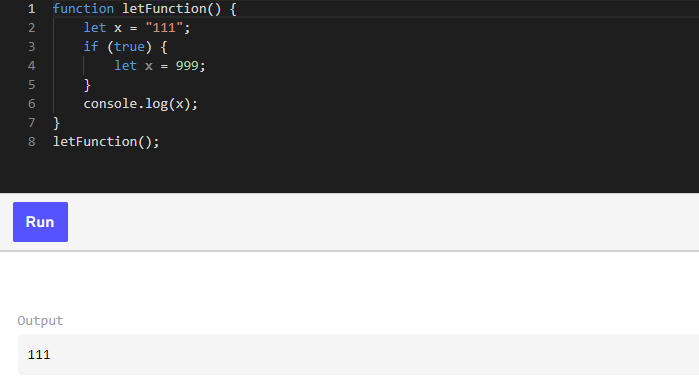
It is important to note that not declaring the variable x would result in TypeScript signaling that it Cannot find name x. However, if TypeScript is not set as strict: true, the code will signal an error but will generate the JavaScript, run, and display undefined.

TypeScript allows stopping creating JavaScript files if an error is found. Usually, it is ideal to not produce a JavaScript file if TypeScript found an issue. In order to enable this feature, open the TypeScript’s configuration file tsconfig.json and set "noEmitOnError": true. In the last example, the code will not produce JavaScript.

Declaring with let

The keyword let comes to the rescue by setting the lifespan of the variable at the block where it was declared. A scoped variable lifespan is the normal behavior of the declaration mentioned earlier in many languages. Curly braces here determine a block.

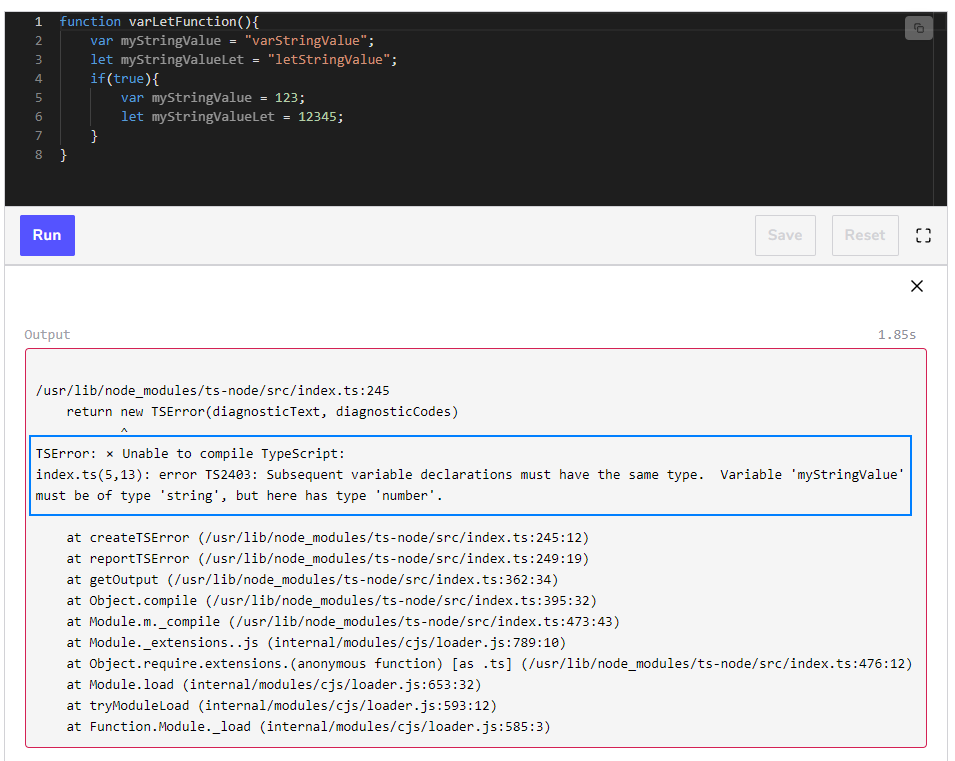
For example, if we declare a variable with let within an if statement, the variable will not be accessible as soon as the execution leaves the if. This rule is true for a function, a loop, and a class. In the case of loops, if we are defining a for loop and we define the iterative i, this one should use let which allows modifying its values while being only available for the loop.



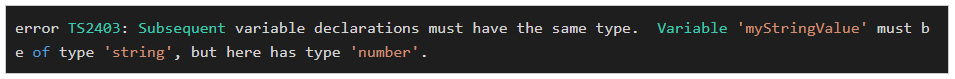
There are two things to note when let is compared with var in this example –

1. Though the output is expected to be the one defined in the scope, it means the value *999* remains in the scope of the condition, and the value “111” is available only in the scope of the function.
2. Second, since they are two separate values, both x’s can be of different types. At the moment, we are not explicitly mentioning the type. However, TypeScript will determine that the former x is a string while the latter is a number.

The below code throws an error –

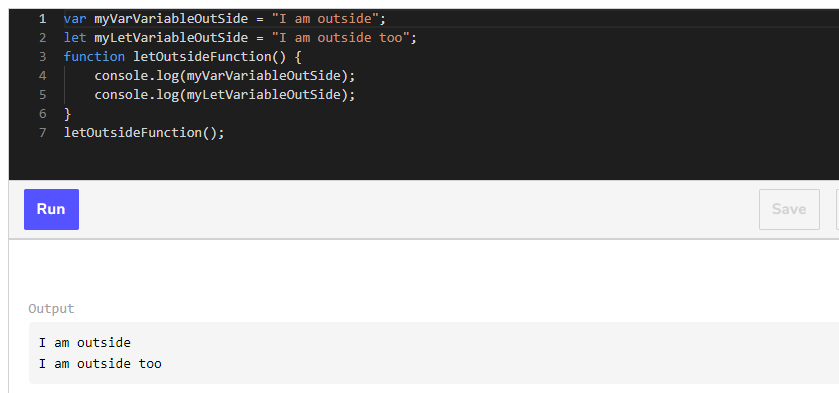


The code above does not compile because of the var variable. The error that we should see is –

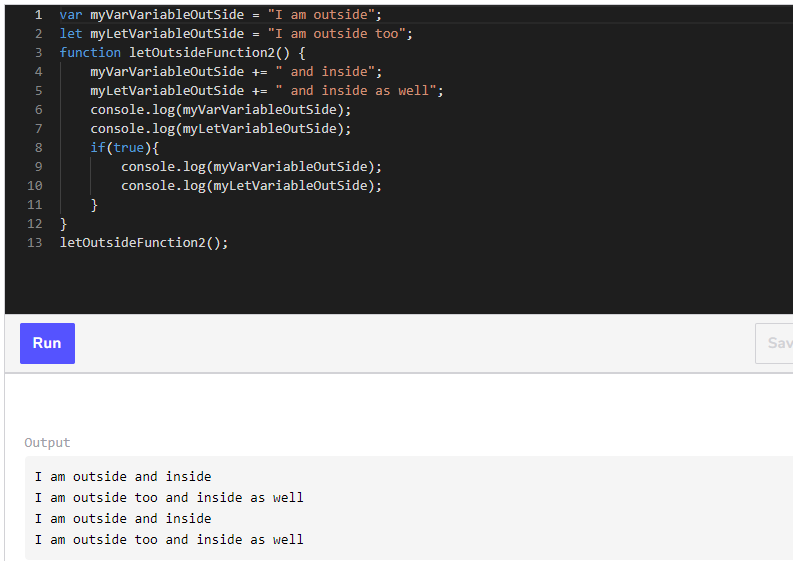


As you can see, var brings confusion that TypeScript can handle by providing guidance. As a rule of thumb, var is never used since the adoption of let and const provides a cleaner scope definition.

Concerning scope, let or var or the upcoming const allows the inner scope to have access to the variable. For example, if we define a variable in a module, or in a namespace, or globally, we can access them inside our function –



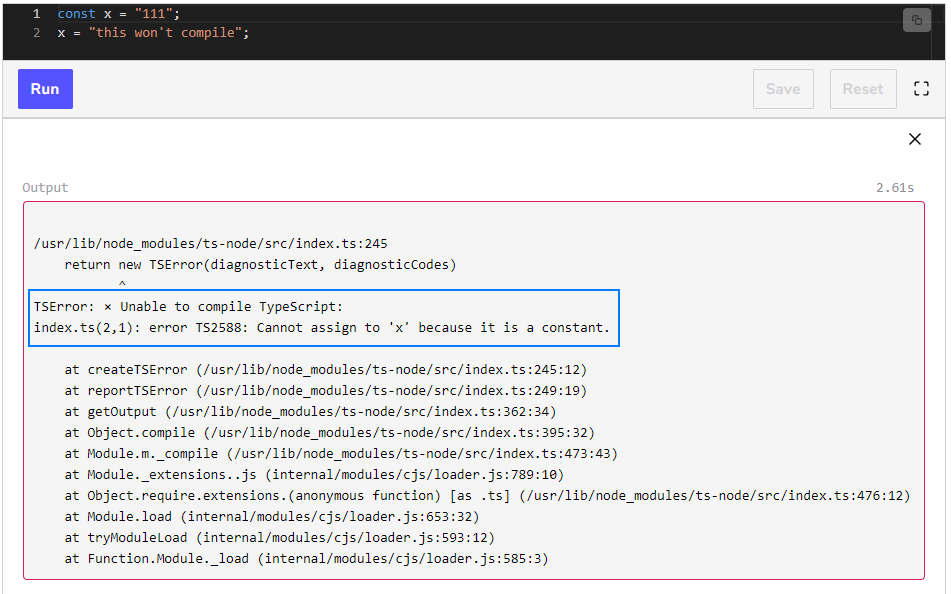
The same is true for condition and looping statement.



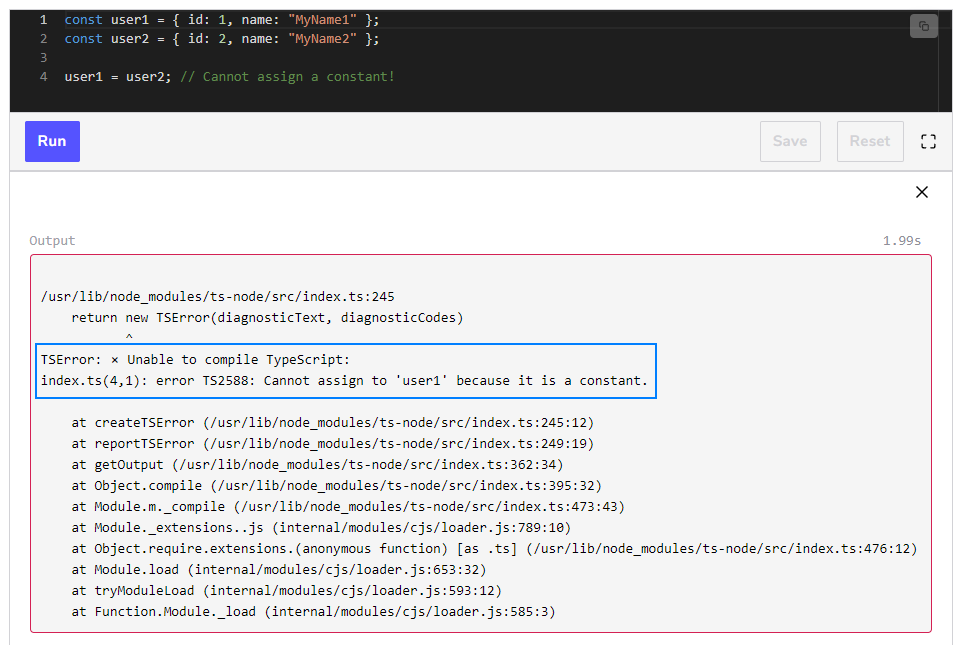
Declaring with const

The keyword const (short for constant) is similar to let in terms of the scope of its lifespan. However, it can only be initialized once: in its **declaration**. The restriction is powerful because it not only syntactically indicates that the purpose is not to change its value, but TypeScript will also ensure that no value can be set. It’s important to understand that if we have a **constant object**, the value of that object cannot change.

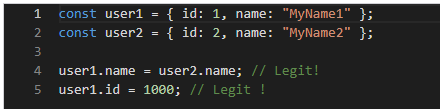
This will throw an error –



The assignation to members of the constant variable is authorized. For example, say we declare and initialize a variable that holds the current user of our application to a constant. We won’t be able to assign the current user to any other user. The following code demonstrates that user2 cannot be assigned to user1 –

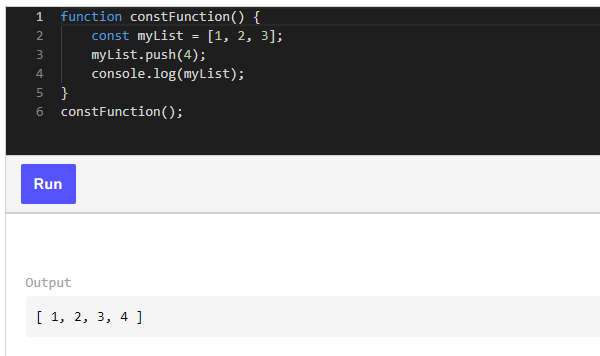


However, we can set its name if a public member is available –



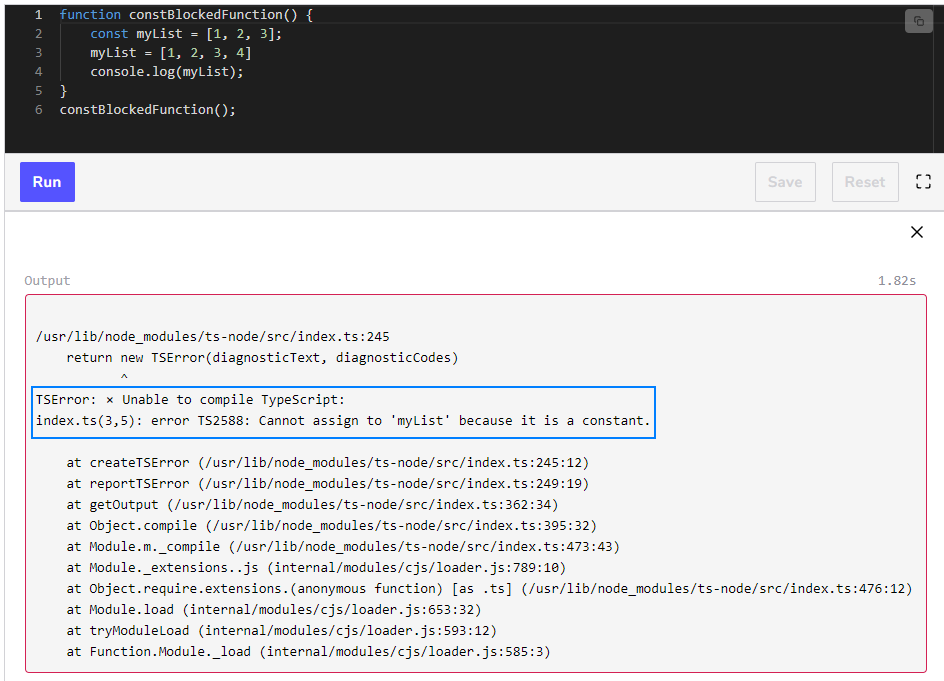
While developing, we will be very surprised that most variables can be constant. It gives more protection to direct assignment.

Let’s move on to array and constant.

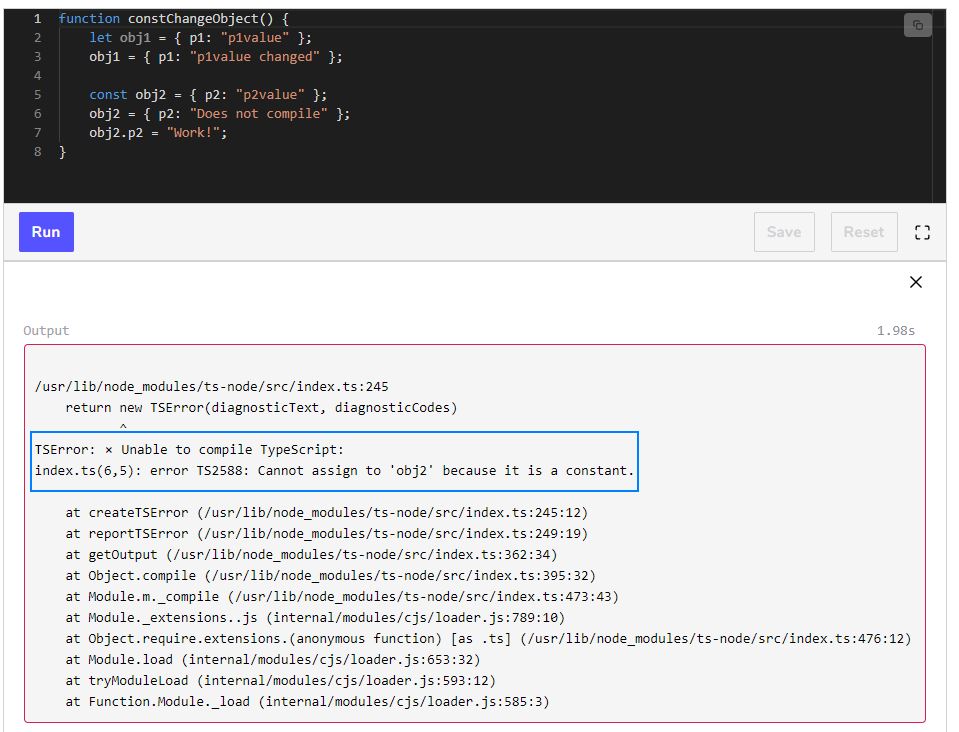


The previous code works because even though the list values change, the reference to the **list** does not. The same holds true with an **object**. We can change values inside an object, but not assign a new object to the variable. Otherwise, TypeScript returns an error, as seen below –





Here is a similar example with two objects. The obj1 is declared with let and can be assigned using =. However, obj2 cannot be re-assigned. Nevertheless, even though it was declared as a const and that the reference cannot change, the value inside the object can. In the snippet below, we can remove **line 6** and the code will compile –



A small thing to note about TypeScript and variables is that TypeScript is excellent at finding typos. For example, if by mistake, we write mylist instead of myList, the code will not compile. That is true for all declarations (var, let, const).

The error will provide a suggestion for a correction. The following code gives –



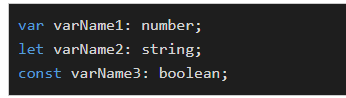
const is the preferred way to declare a variable. It is the strictest and ensures that the variable will not change in the future.

What about type?

So far, we have declared variables without specifying their type. However, a feature of TypeScript is that it automatically assigns types to all variables for us. A feature of TypeScript is to be able to infer the type of a variable.

In the upcoming lessons, we will see different primitive and custom types that are automatically used by TypeScript as well as how to explicitly declare a variable with a type.

In a nutshell, specifying the type happens after naming the variable with the help of colons –



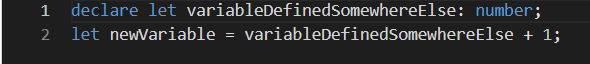
In either case, implicit or explicit, once a variable type is assigned, TypeScript will respect and enforce it.

Avoiding type changes allows consistency in the code.

Declaring Types in Untyped code

This lesson delves into how to declare types on a JavaScript code that does not have a type.

The keyword declare can be used before one of the previous three declaration types (var, let, const). As the name suggests, it declares to TypeScript that the variable is somewhere but not saying where. This is not used frequently but can be useful if we need to tell the transpiler that the variable is present, just not in the current project (or loaded module), and may not be visible.

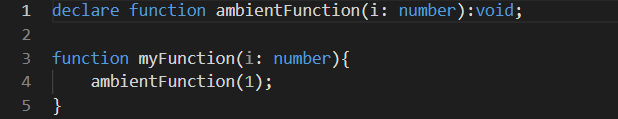


The official jargon for this is ambient declaration. If we are working with JavaScript and there is no definition file, we may want to use declare since we know that we will be importing the code with the variable present. Unfortunately, we will lose all Intellisense since declare doesn’t have a type provided.

Ambient declarations can be defined anywhere. However, a good practice is to have a file with the extension (.d.ts). The file does not produce any JavaScript but indicates that the variables or function exists. An example for people who use jQuery is to declare an ambient declaration, for jQuery. jQuery will be added along with the code. Thus, we should assume it’s present –



It is possible to declare more than just variables. We can define a function as well. In fact, we can declare anything that exists in JavaScript –



A common use case is a library that already exists in JavaScript (but has not migrated to TypeScript), which can be used in TypeScript with a good definition by having a *.d.ts*. It is common to migrate the definition file along with the JavaScript file. If this is not the case, it is possible to download the definition type after we install the JavaScript library –

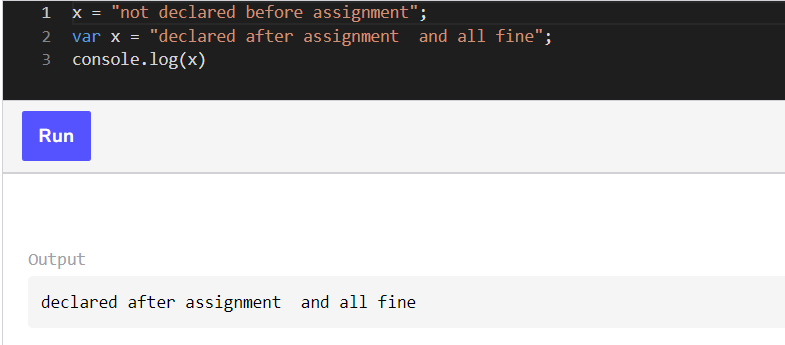


Reference – [DefinitelyTyped/DefinitelyTyped: The repository for high quality TypeScript type definitions. (github.com)](https://github.com/DefinitelyTyped/DefinitelyTyped)

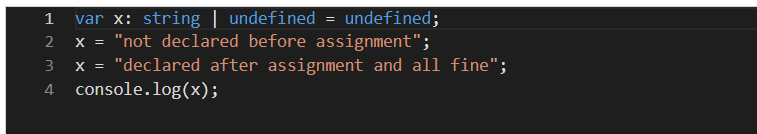
Hoisting Variables

This lesson goes over the JavaScript principle of hoisting in TypeScript.

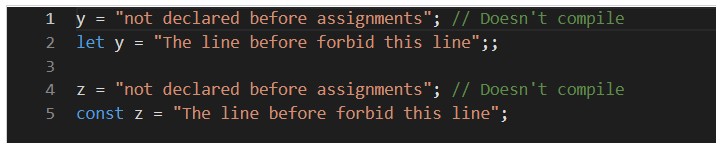
Before moving on, let’s talk about the concept of *hoisting*. It is a quirk of JavaScript that brings all declarations made with var to the top of the function (or into the global scope if declared outside a function).



The code above compiles because var x goes above the two assignments, and becomes something like this –



This peculiarity does not affect let or const. This means that if we are using var, we can use the variable and declare it later and the code will still work. This is, however, a bad practice that makes the code hard to follow. This ambiguity is solved by let and const if we use a variable that has not been declared first. The following code snippet does not compile because the variable declarations with let and const are after the assignments –



The need to use var is now rarer since the inception of stricter let and const. Nevertheless, TypeScript can catch many errors like declaration and assignment on a codebase that still uses var.

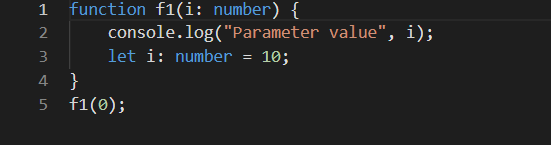
TypeScript Scope is JavaScript Scope

This lesson discusses the principle of shadowing, capturing, and declaring a variable in JavaScript and naturally porting to TypeScript.

Shadowing Scope

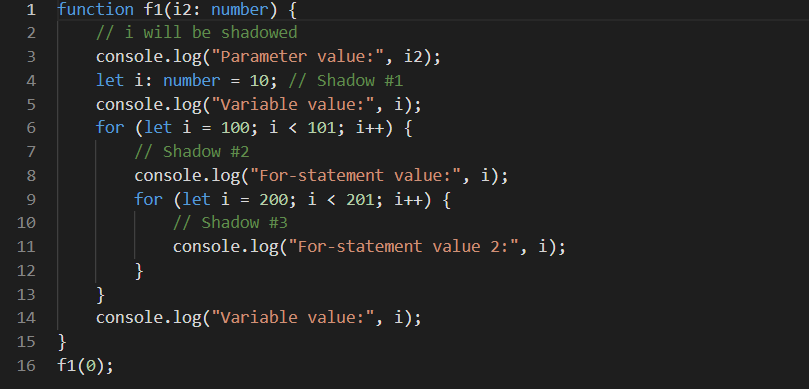
We briefly encountered the concept of scope when exploring the three declaration types discussed in the previous lessons. You saw that a variable declared using var has a broader scope than let and const. However, there are some other cases involving the scope with let and const.

The first case is shadowing. This occurs when one variable is declared twice, in an outer scope, and an inner scope. For example, if you have two loops and both of them are using the variable i, TypeScript is smart enough to understand that both declarations are for two different variables. However, it is confusing and susceptible to error, hence it is not recommended even if the code will transpile without a problem.

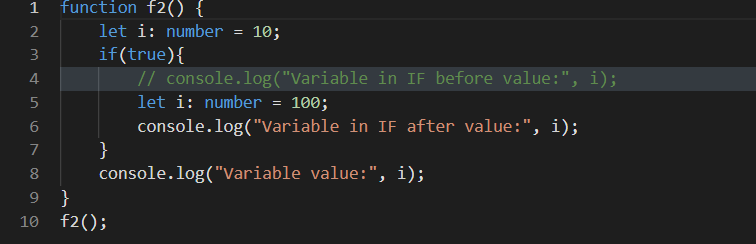


The code above will not allow the parameter and a let variable define the same variable name. TypeScript will find and throw an error that the variable has been duplicated. In JavaScript, this error would not have been caught.

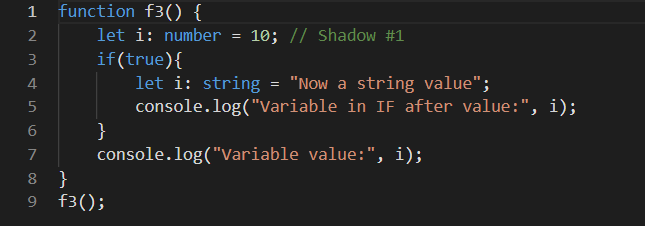
The following code demonstrates that it is possible to define the same variable name when the variable is in a different scope. In the example below, each iterative loop has its own scope, hence TypeScript uses the closest i depending on the context of execution. This behavior is the same in JavaScript.



The following example demonstrates that we can define the same variable name. In the following example, try uncommenting line 4. Suddenly, TypeScript finds a duplicate of declaration in the same scope and throws an error –

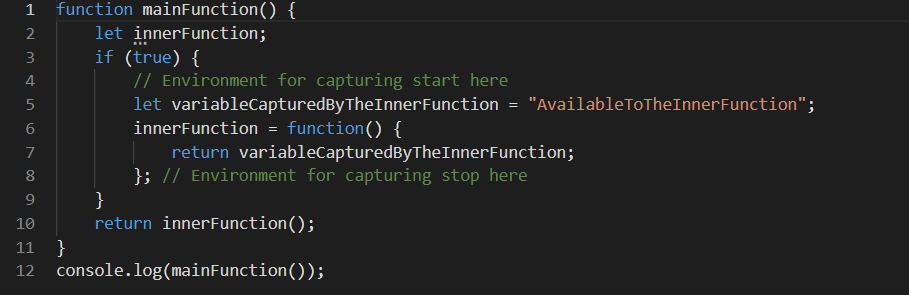


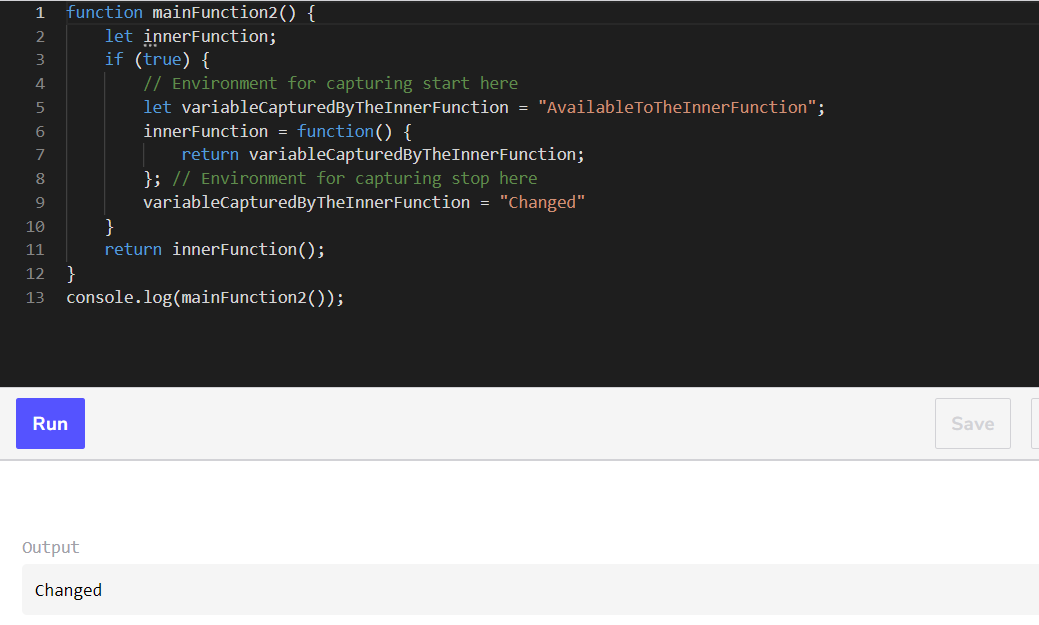
The definition of a variable before it’s used is confusing. In the example, the variable i is also of type number but could have been something else. In the following code, we can see that each scope has its own set of variables. In the previous example, TypeScript finds a variable already defined for the scope.



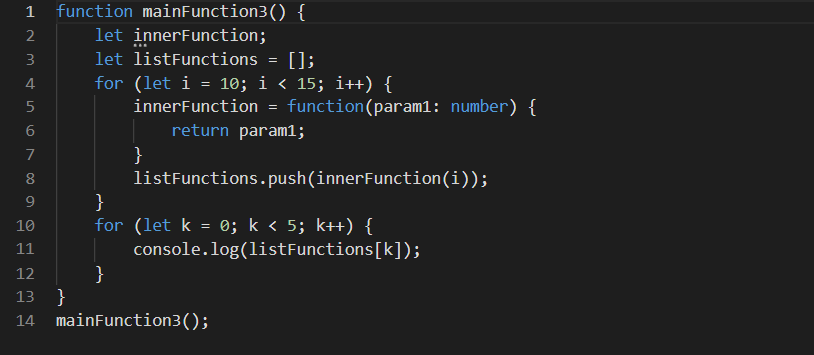
Capturing Scope

The second case of scoping is called capturing. This occurs when we have a variable that we define in an inner scope and then use inside a function that we assign to another scope. During the assignment of the function, the variable defined in the inner scope will be captured, like a snapshot, and when leaving the scope, the function declared before the inner scope will still have the value of the variable –

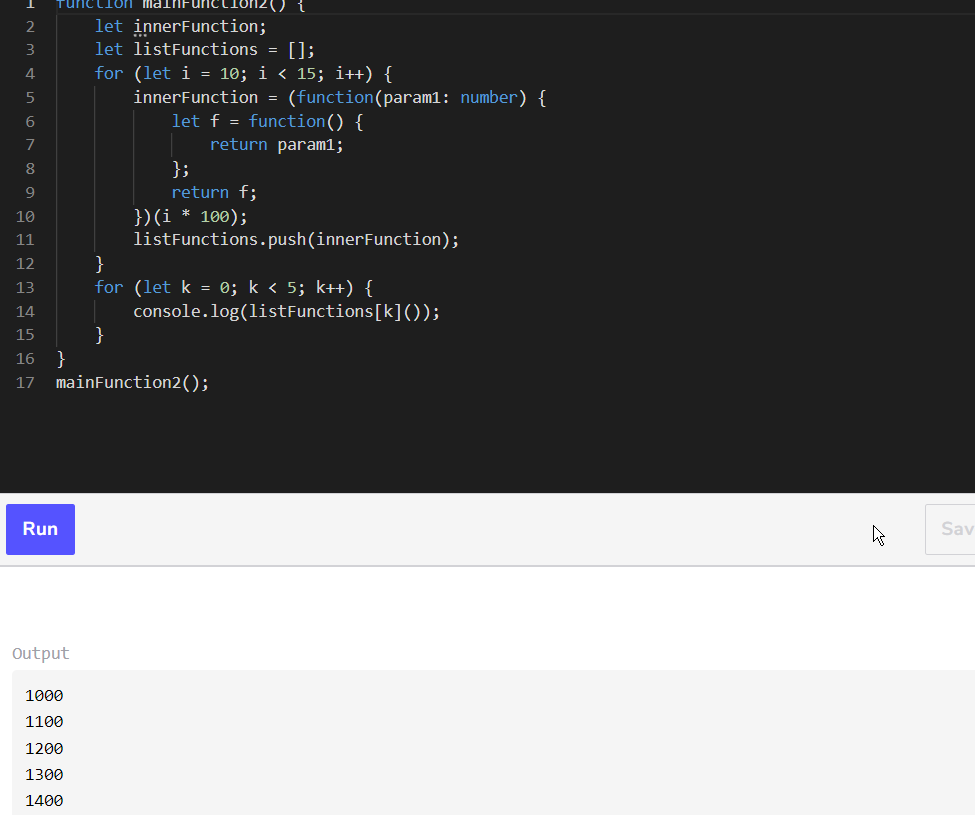




The following code does not leverage capturing. The function loops five times from lines 10 to 14 and adds the function to a list. Then, every function is invoked. The result below should be expected.



Within a particular scope, we can *freeze* a variable. Lines 5 and 10 create a scope by generating an anonymous function. This function then passes a value that will be available inside the function f at any time in the future.

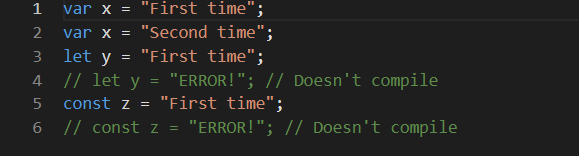


The result is different from the un-scoped function. Often, with an asynchronous function, we need to ensure that the data passed to the function remains the same, regardless of the future execution time. Leveraging *capturing* is the JavaScript way to do this. TypeScript allows it by adding type as you will see in a few lessons.

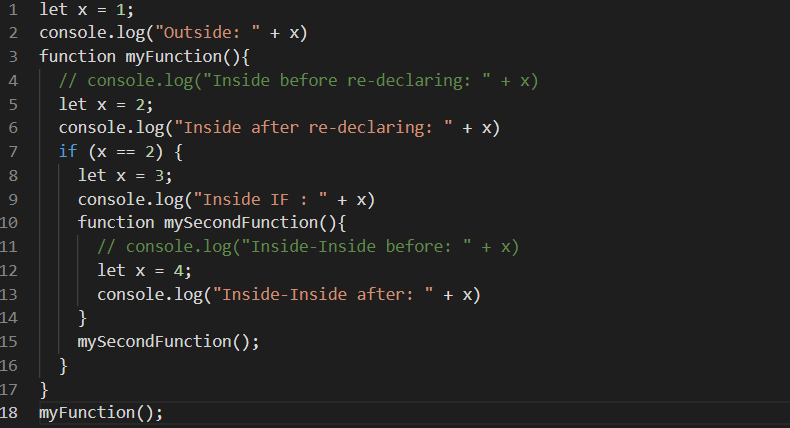
Declaration Scope

Now that we have discussed var, let and const, the difference should be more obvious. One detail that we have not yet explored is the number of declarations that each variable can have under the same scope. While the declaration of var is unlimited, declarations with let and const must be unique per scope.

If we uncomment lines 4 and 6 of the following code, TypeScript notifies us of the duplicated declaration –



As mentioned, let and const can be declared once per *scope*. This means we can have several variables with the same name, but only the one in the scope of invocation is visible.

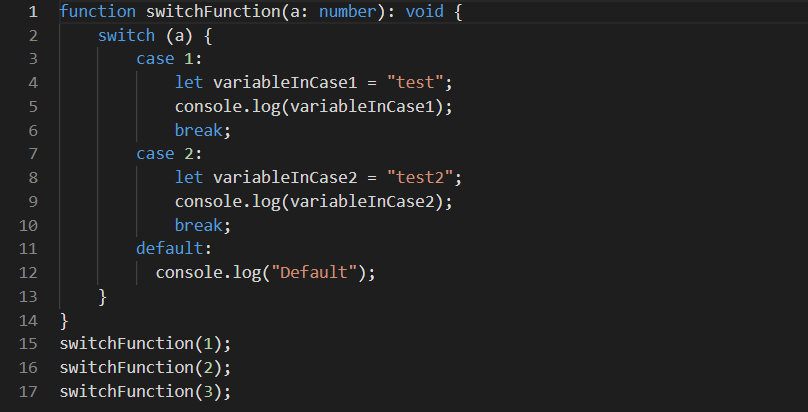


The code above has two commented lines of code. Uncommenting the code will result in two errors. TypeScript will figure out that the statement is accessing a variable not defined in the scope.

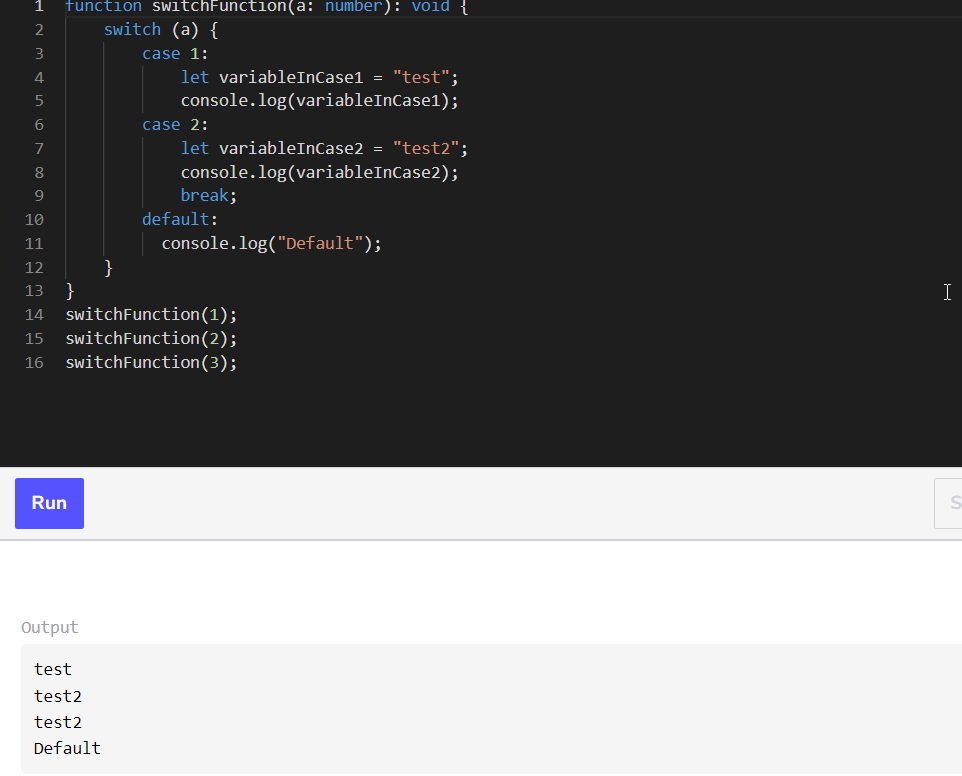
Switch Scope

In this lesson, you will see two ways to use the switch statement and how those ways can affect the scope of variable declaration.

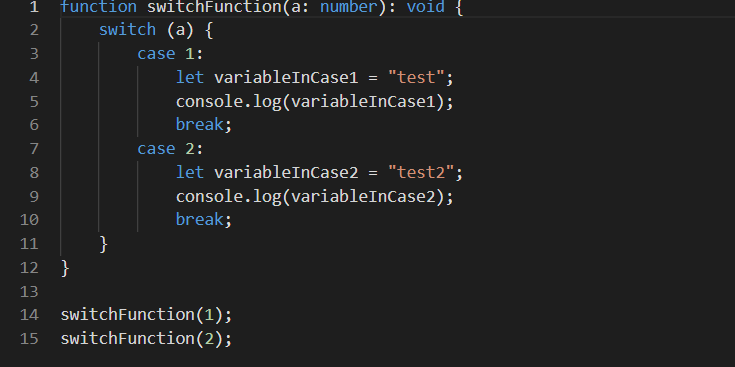
A switch statement is similar to an if statement. The condition set between the two parentheses must be met to reach one of the cases. If not, the default case will be entered.



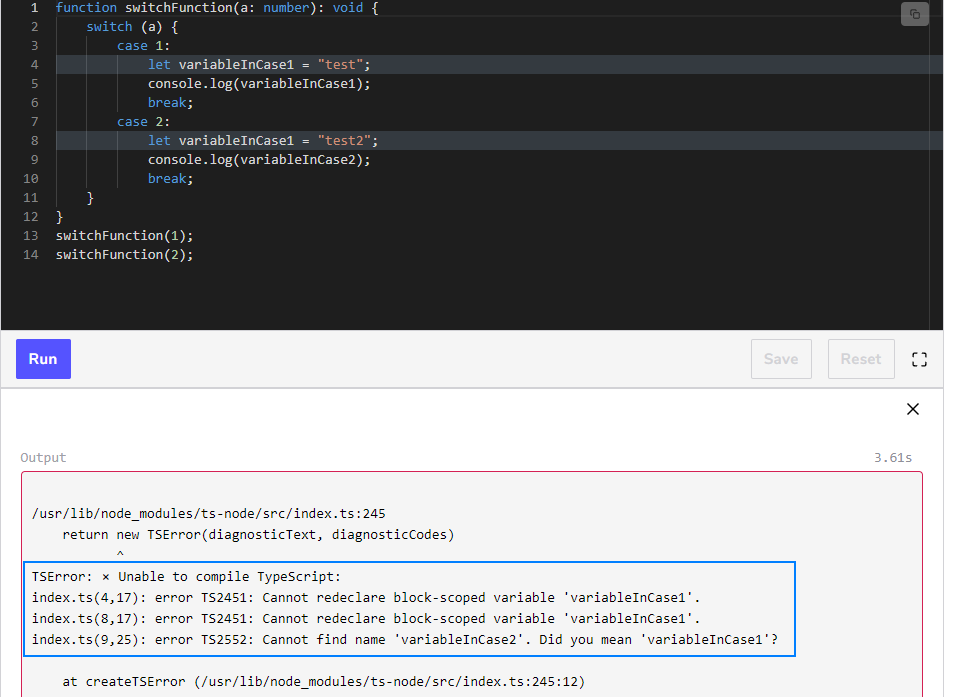
Not adding the keyword break will fall through the case underneath. A quick modification of the previous example shows that the value 1 will now print test2 twice because the value 1 gets into the case 2 –



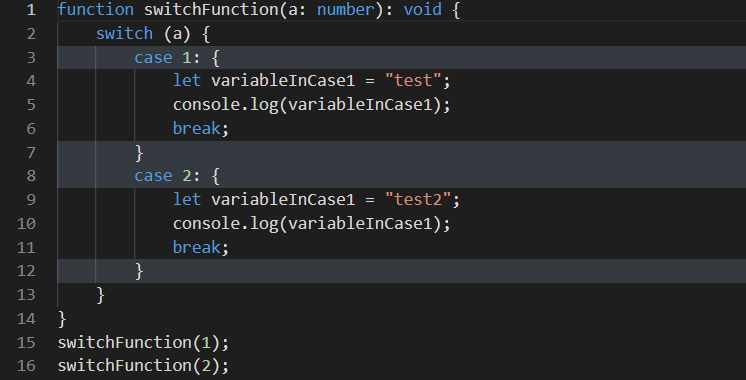
The switch statement requires curly brackets after the colon and after the break statement. Otherwise, variables defined within the parent scope are shared. This is not a constraint with TypeScript, but it is with ECMAScript.



The above code can be problematic. For example, let’s change the variable to use the same name. The scope is expanded to the whole switch case. The following code does not compile in TypeScript.



To avoid stumbling into a situation where variables are shared across cases, it is suggested to use curly braces for each case.



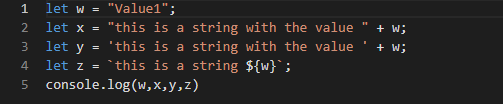
It’s a good practice to always use curly brackets around each case. Adding brackets makes the scope explicit and ensures that code from one case does not impact other cases.

The multiple methods of declaring a string

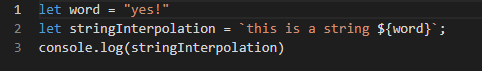
In this lesson, we will look at how to initialize a string in a single line, as well as how to write a string on multiple lines.

Strings on a single line

The first primitive is the string. A string is made of characters. It can be assigned a single quote or a double quote. A string’s content can be a number but will behave as characters if between quotes. Both single and double quotes are accepted. However, the guideline of the TypeScript project uses a double quote. By the way, TypeScript is written in TypeScript! It means the language itself is written with the language, hence following its guidance is a good idea.

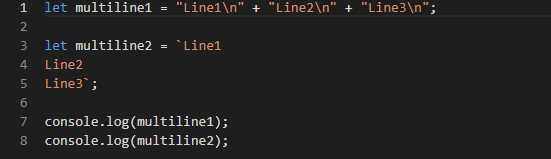


There is also the possibility of using the backquote for string interpolation. String interpolation allows templating a string to avoid concatenating many strings with the plus sign. It helps to have a cleaner string built. The syntax starts with a backquote and id followed by all desired strings as usual with a single or double-quotes. The value of a variable is embedded in the string by using the dollar sign followed by a curly brace. The insertion of the name of the variable must follow. The use of an ending curly brace is required, and from there, we can write any other characters and close the string by using the second backquote. This is demonstrated in line 2 of the code playground below by way of ${word}. The placeholder is often a variable, but we can insert any TypeScript expression –



Strings on multiple lines

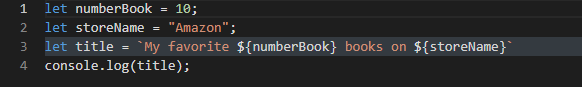
TS can write strings on multiple lines without the need to use of the backslash n \n that is required by JS.



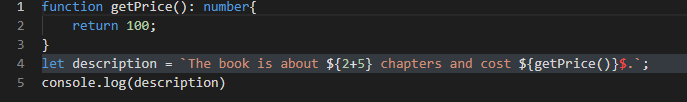
This shorthand removes the boilerplate of having to bring many characters by allowing us to hit enter to switch lines. Similar to string interpolation, it requires the use of backquotes. Inside the backquote, every changed line will be considered as if we were explicitly using the \n. The result of the string interpolation produces multiple strings with backslash n.

String interpolation for formatting

String interpolation has the advantage of providing a placeholder to execute the TypeScript code. The main use case is when we have a string with dynamic areas where values can change.



But as mentioned, it executes code. Albeit mostly used for variables, you can have TypeScript code executed like below –



So far, we have used the implicit way to define a string. However, defining a string explicitly is a matter of using the semicolon and the string keyword –

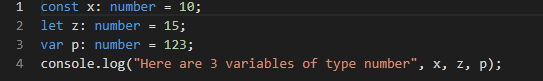


Number in TypeScript

This lesson delves into how to define integers, floats, and doubles in TypeScript.

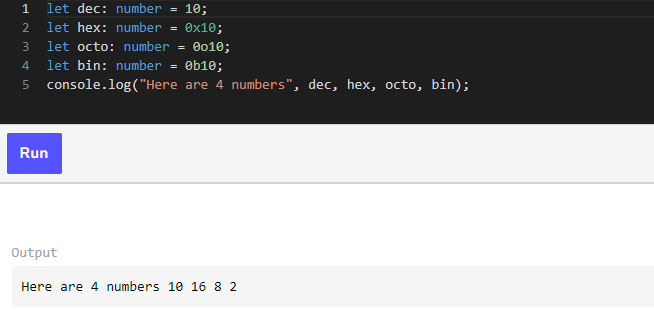
Another common primitive is the number. Since TypeScript is a superset of JavaScript, numbers work the same way in both languages. The openness of JavaScript allows for a broad set of numbers. Integers, signed floats, or unsigned floats are permitted. By default, a number will be base 10.

When a type is explicitly assigned to a variable, the type will be removed once the JavaScript is generated. The reason is that typing does not exist in JavaScript. That explains why TypeScript only has number. The following code will produce three variables without an explicit type in JavaScript but if typeof is used, it will return the dynamic type: number –

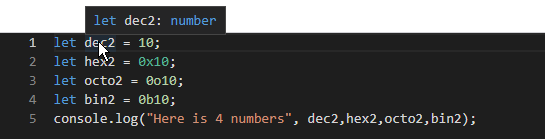


Number base

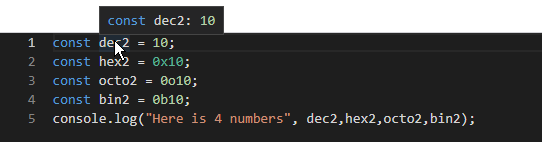
We can also assign base 16 (hexadecimal), base 8 (octal) or base 2 (binary) with the prefix 0x, 0o and 0b, though they’re rarely used –

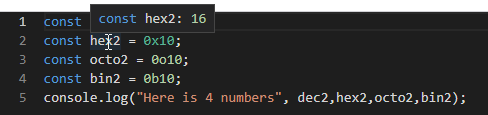


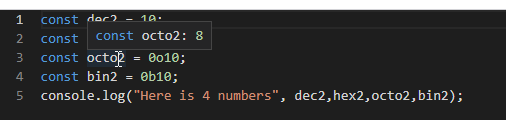
Like most variables in TypeScript, there is no need to explicitly mark the variable type at the time of declaration. TypeScript can infer the type. The following code is the same as the code above. If we move our cursor on top of each variable, we will see number –

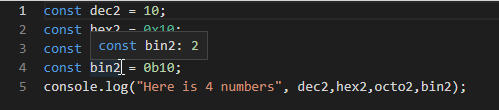


However, the following code does not define the four variables as number –









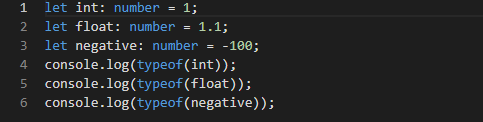
The code compiles, but if we move our cursor above each variable, we will see the type 10, 16, 8, and 2. The type is actually the value meaning that only those values are acceptable. The difference between this snippet and the one before is let and const. With let, the variable may be reassigned at any time during the life of the variable, hence the narrowest type that TypeScript can infer is number.

However, in the last example with const, TypeScript knows that the value will not change, hence it can narrow the type down to the only value possible.

The number: integer, decimal, and signed

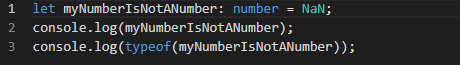
The number type is the same as in JavaScript: it defines the type for integer, float, double, etc. Hence integer, float, and positive or negative numbers will all be referred to as the single type – number. A type declared as number (implicitly or explicitly) can be checked at runtime as well with typeof which will return number.

Numbers are also not signed, which means they can be positive or negative.



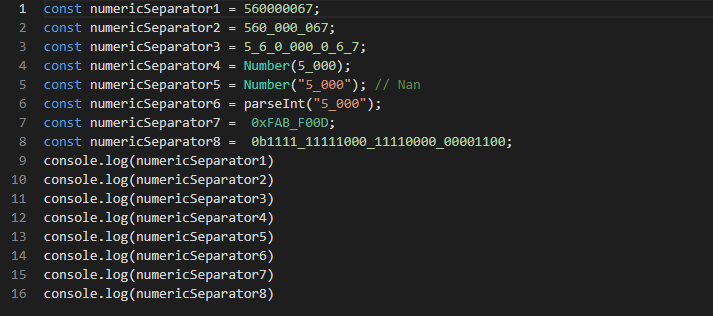
Not a Number (NaN)

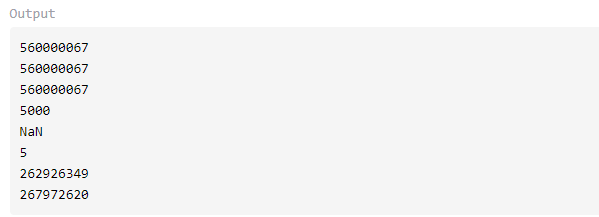
In TypeScript, as in JavaScript, the type number can be assigned with NaN meaning that it is not a number –



Separator

A numeric separator is a feature that simplifies how we write numbers. A long number can be hard to read and adding a separator can reduce confusion. When writing a number, we can use the underscore symbol to mark every thousand, for example. There is no rule on where to place a group separator other than it must be between two numbers –





When using NaN, JavaScript cannot transform a string with a separator as a proper number. TypeScript does not warn or give an error at transpilation time because it only checks that the type is string which is legit but does not evaluate every operation.

Numeric separators work with decimal, octal, binary, and hexadecimal bases. It is available from TypeScript 2.7 and is now available in the last version of ECMAScript. It is possible to use this feature in older versions of ECMAScript because TypeScript transforms the separator out during transpilation and we can target older versions of ECMAScript and still use the feature.

Booleans, Functions and Objects

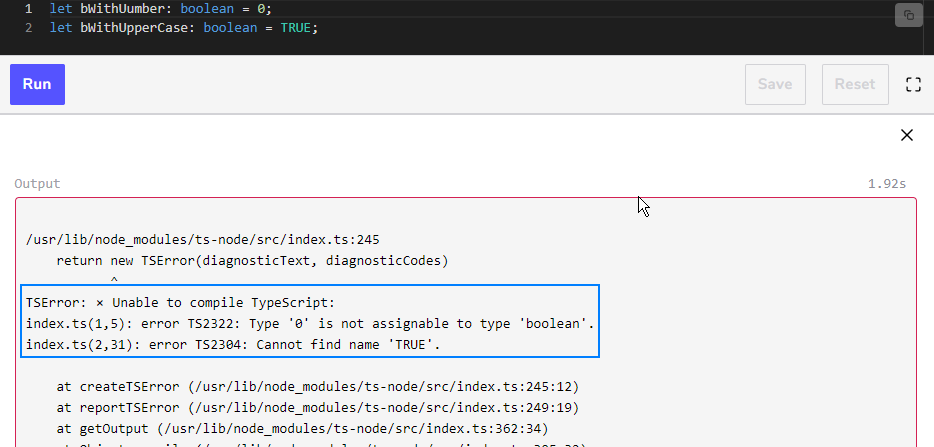
In this lesson, we will learn how to declare a variable that can hold true or false.

Boolean primitive type

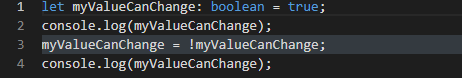
A boolean value is the most basic primitive in JavaScript and it remains the same with TypeScript. Boolean values restrict the assignment to two values: true and false. These terms are case sensitive – only the lowercase format is accepted.



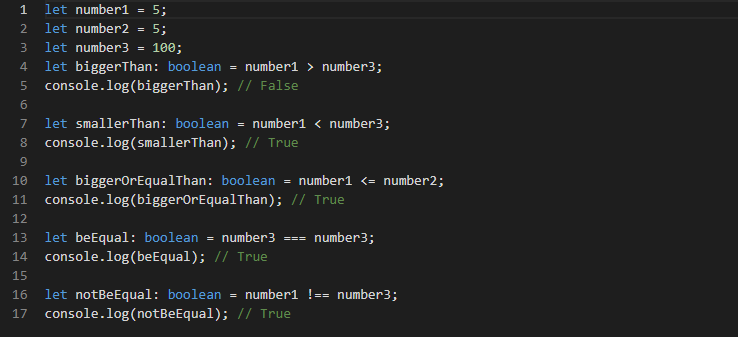
You cannot assign the value 0 or 1, or the true or false values using any upper case letters –



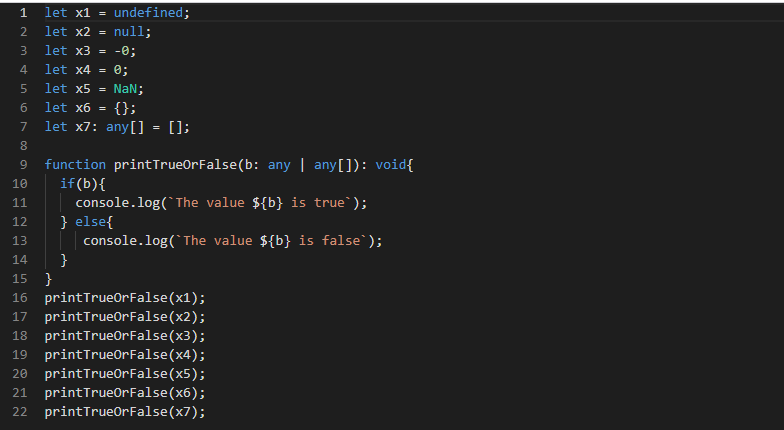
We can reverse a boolean value by assigning the same value with an exclamation point before it –

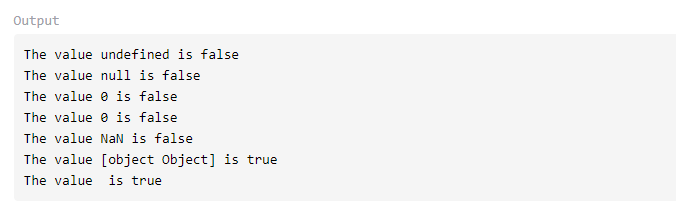


Boolean values are often the result of an operation like === or <, >, !==, <=, >=



TypeScript allows other value types to act like a boolean. For example, undefined and null, -0, 0 and NaN will return false. This is because JavaScript includes many values to be “falsy” –





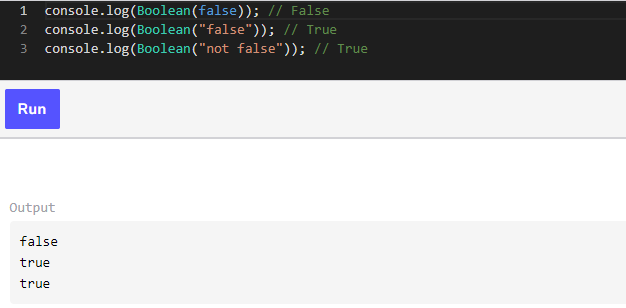
The previous example prints “is false” for each function call, except for the last two.

A detail to note is that the parameter of the function is of type any. If we change the type to boolean, TypeScript will not compile. The last two elements are true because an empty object is still an object and an array, even when empty, is still an array.

The Boolean function

TypeScript, like JavaScript, lets us invoke a Boolean function. This function proves handy to convert different types into a boolean type.

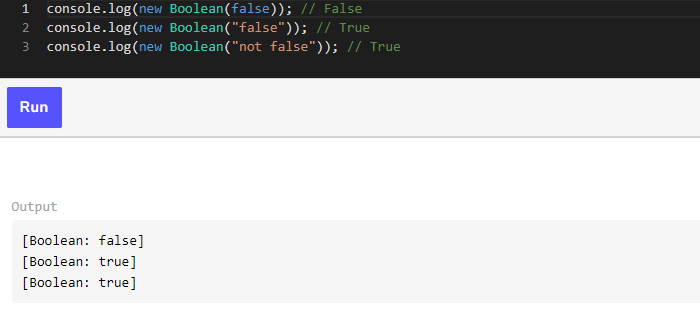
Typescript follows JavaScript rules; hence few values can return surprising results. For example, Boolean("false") will return true and Boolean("not false") will also return true –



Let’s keep in mind that Boolean is taking an unknown type that we will see in a few lessons.

Boolean Objects

The boolean object behaves like the boolean function. One difference is that the result is not a boolean but an object that wraps the boolean value. The following outputs are objects like [Boolean: false] instead of the primitive variable value of false –

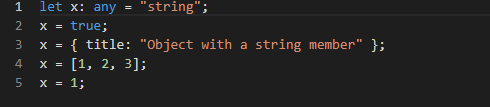


Avoiding ‘any’ at any time possible

In this lesson, we will learn about a variable type that we should only use in particular situations.

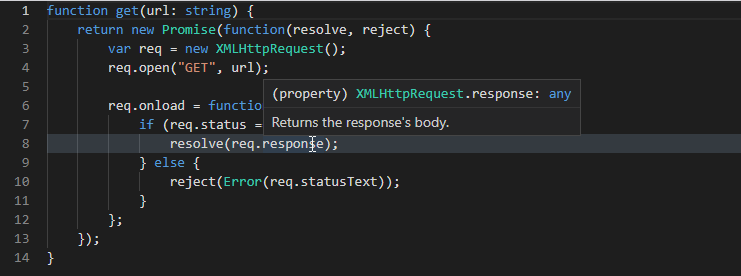
Dangerous World

We must avoid (as much as possible) the type any, principally because it can hold any value and therefore doesn’t enforce any protections. If we are integrating an existing JavaScript project with TypeScript, every variable will be, by default, set to any until they are defined –



This is also the case with a value coming from an Ajax response in JSON format. Every any variable will let us assign any value but could also invoke any function.

The following code has a response of type any on line 8. Unfortunately, the response can be a string or a JSON object of any form. It is possible to mitigate this issue with casting, which we will see later to not propagate the any further in the code.

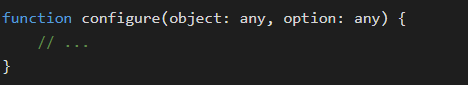


The danger is that the function may not be available. For example, say we set a variable with a number value that calls for an array of the function .length. This will transpile, but raise a runtime exception because a number doesn’t have a length function in the browser, and return undefined when running under Node js –

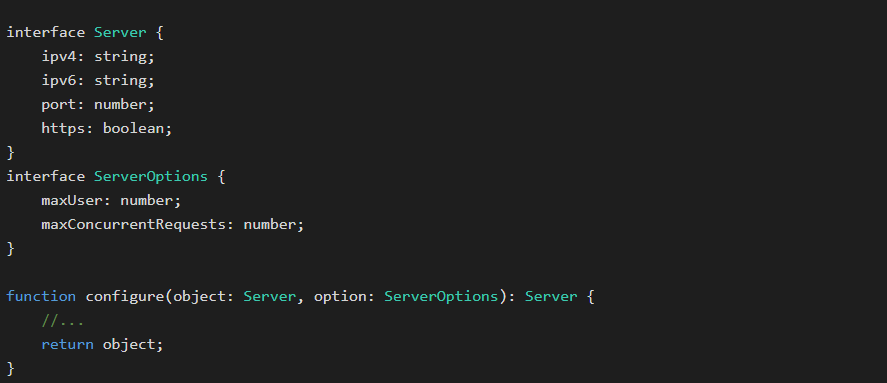


Readability

A piece of code that uses any is harder to maintain because it is harder to understand. The way code is typed is a live documentation of what is expected. For example –



On the other hand, the below code is much clearer –

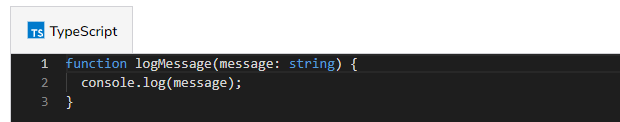


While both pieces of code in execution will perform in the same way, the second one is clearer about what inputs are needed and what the output will be. Similarly, the readability inside functions is improved when a local variable is well defined.

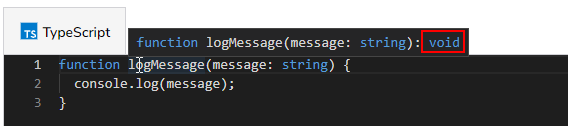
Understanding and using the "void" type

An example

The code below outputs a string to the console –

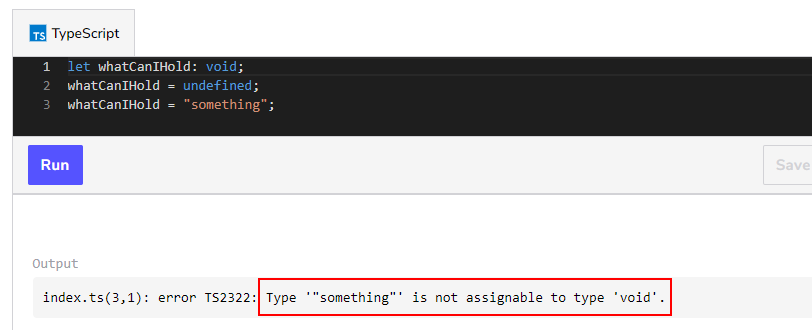


The function doesn’t return anything, so what is the return type of the function? It is "void". If we don’t define any return type, implicitly "void" return type is considered –



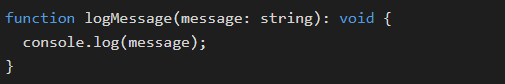
The void type can’t hold any data; it can only be *undefined* or *null* if the *strictNullChecks* compiler option is off.

The code below shows the TypeScript compiler not happy with a variable of type void set to a string value –



When to use the void type?

"void" is only really useful for function return types and it can be explicitly defined on functions after the parameter parentheses like in the example below –



Wrap up

So, the void type is to define that a function doesn’t return anything. TypeScript will correctly infer this type if a function doesn’t return anything, so we don’t need to define it explicitly.

More information can be found about the void type in the TypeScript handbook – <https://www.typescriptlang.org/docs/handbook/basic-types.html#void>

In the next lesson, we will learn about the never type, which is often confused with the void type.

Mutable & Immutable arrays

In this lesson, we will learn about two different syntaxes to define an array.

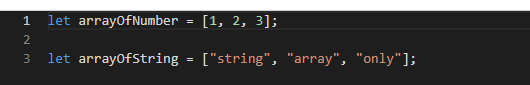
Mutable Arrays

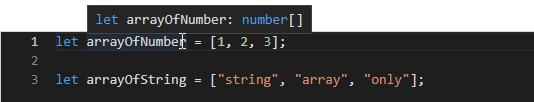
Arrays in TypeScript are exactly like the ones in JavaScript in terms of features. The difference is that TypeScript assigns a type to the list.

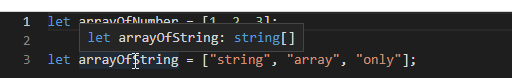
The syntax, as shown below, utilizes square brackets [] with the actual type before the brackets and after the colon : like so –



It is also possible to initialize values during the declaration. In the following example, the two arrays are typed. TypeScript infers that the first one is a number and the second is a string –



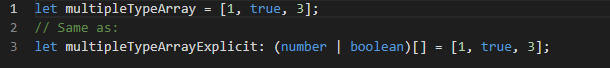




Using multiple types will require us to evaluate what the type of each value is before using an individual item of the array. This is because the variable’s operations are type-dependent. There is an equivalent syntax that uses the generic Array<T>. Both are the same –

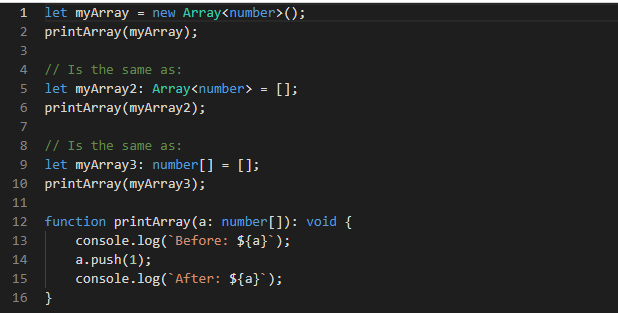


We can initialize an empty list by setting the variable equal to empty square brackets. An array can use one or many unions to allow multiple types. Unions will be discussed later. At that stage, remember that with multiple types, we must wrap the union with parentheses –



The preference to explicitly declare type or not depends on our style of coding. For maintainability, it is good practice to be explicit. If we take a peek at the code above, we may notice that the array takes two types without having to read further. It also forces future additions to be constrained by the expected type. Without explicitly typing the variable, someone could add a string and suddenly, the variable allows a number, boolean, and string.

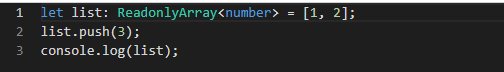
Before moving on, it is important to note that we can also instantiate a strongly-typed object array. This is equivalent to creating a new array without assigning any values –

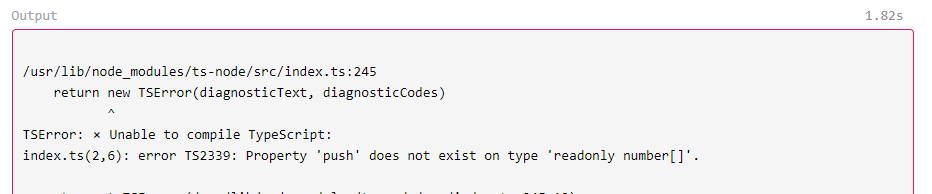


Immutable Arrays

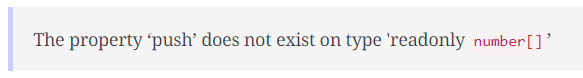
While the two syntaxes presented above refer to mutable collections, there is also the possibility of creating a list that is immutable. The ReadonlyArray is a generic array that only allows us to read from the array once it’s constructed. As with the mutable array, there are two ways to write a read-only collection –

The first approach is to use ReadonlyArray<T> instead of Array<T> –

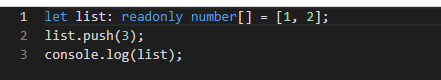


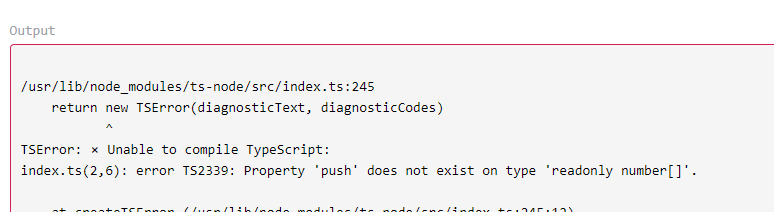


The code above does not compile because we cannot mutate the array with push. This error is interesting and introduces the second way to write a read-only collection –

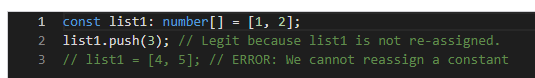


The second way is to use the keyword readonly in front of the type and square brackets –

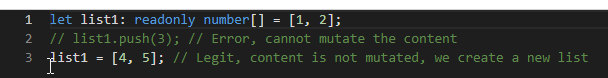




We now might ask ourselves, "What is the difference between a constant array and a read-only array?" The answer is that a constant array won’t let us assign values to a list while a read-only array blocks us from changing values.



Whereas if we use readonly –



Control flow analysis for array construction

Control flow analysis is how TypeScript dynamically figures out what type it should infer. Arrays can be tricky since they accept many values which can be of many types. For example, the following code accepts a list of numbers or strings –

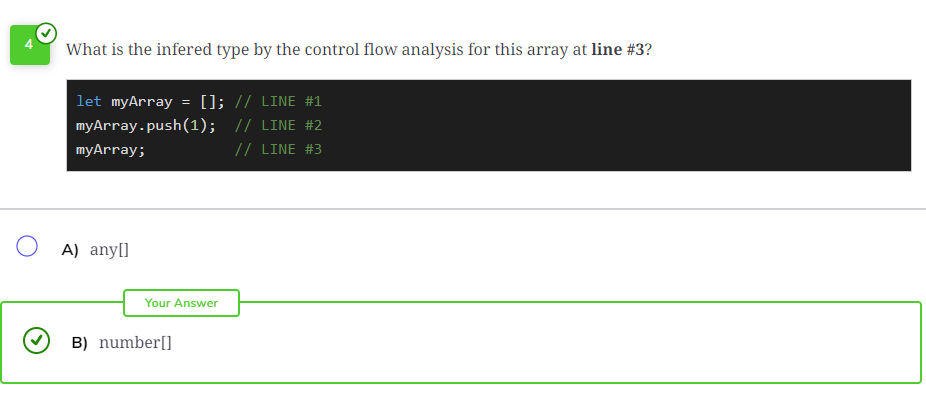


If we let TypeScript figure it out, we will have –



Which means anything can go into this array.

The below scenario might surprise us –



The empty array is an evolving type that will be analyzed during the “flow” of the code, meaning depending on what happens with the following operations. Functions like push, shift, unshift, or setting directly to an index a value myArray[index] = value will transform the type. The type is finally attributed once it stops changing, hence the above question gets to its real type at the end of the code (line# 2), not before (line# 1 – which would be any[]).

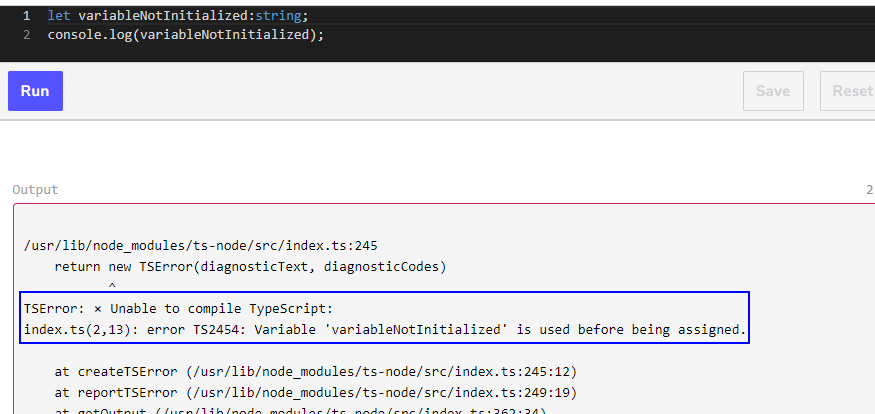
In this lesson, we saw two different syntaxes used to write an array in TypeScript. We also illustrated that it is possible to have the content of an array to be immutable with readonly. We described the difference between a constant and a read-only list as well.

Undefined vs Null

In this lesson, you will see the ubiquitous type undefined, which is used to define something that does not exist.

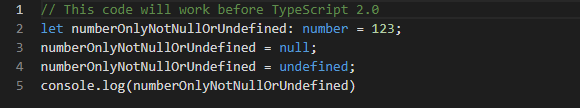
undefined

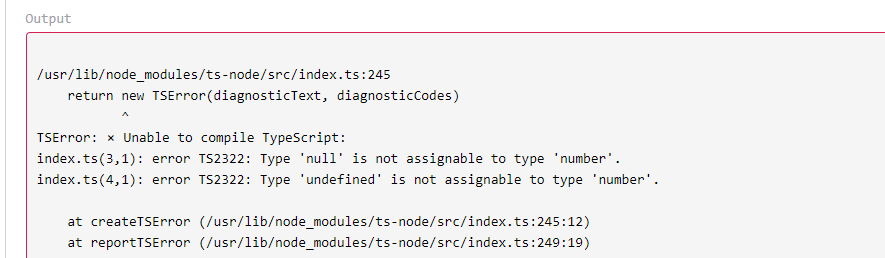
A variable declared but not initialized is undefined. Undefined is not quite the same as the type null. In both cases, an assignment can set undefined or null to a variable explicitly. The following code does not compile because the variable is consumed before initialization and TypeScript when configured to be strict, does not allow for interaction with an unassigned variable –



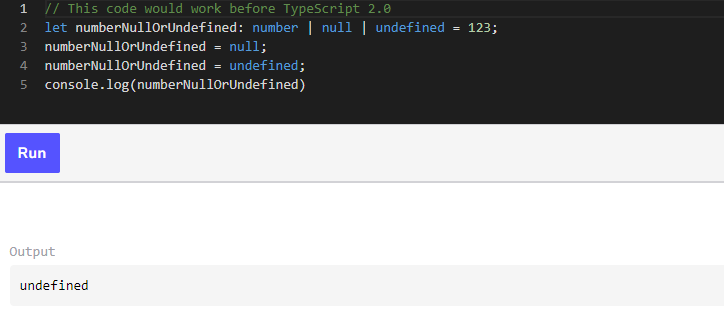
TypeScript must be made stricter in order to prevent it from assigning null or undefined to every type.

We must set TypeScript’s strictNullChecks option to true to block the possibility implicitly assigning null and undefined to our variables (available since TypeScript 2.0).

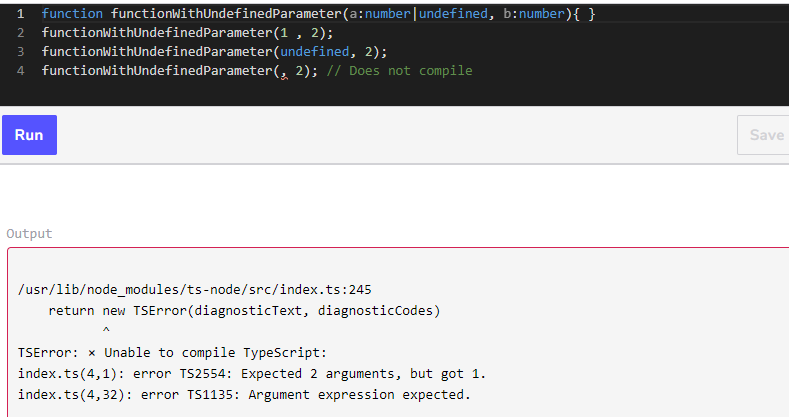




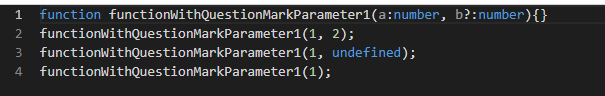
Being forced to be explicit about all possible values forces developers to use union or the question mark to optionally define the variable, which allows undefined. A nullable number is considered to be two types. Dual type (or more) is possible with a union. The union uses the pipe character between the main type (for example, a number) and null –



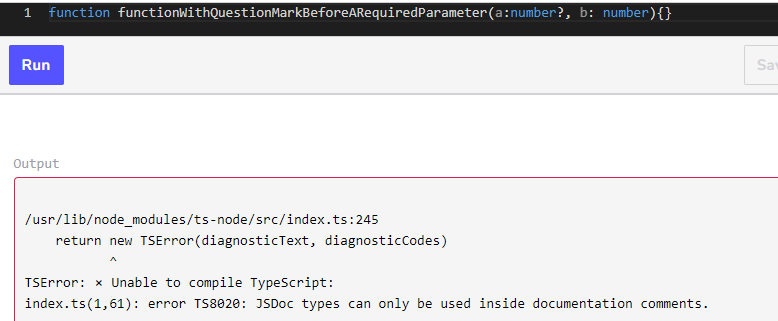
The union of any other type and undefined makes the type optional. Using the question mark syntax or union with undefined produces a similar result with only minor differences –



The difference is that with | undefined, the parameter must be passed with the value or undefined. However, with ?, you can pass undefined or nothing at all –



The question mark is more succinct but also doesn’t allow a non-undefined parameter to follow in a function signature. The following code does not compile –

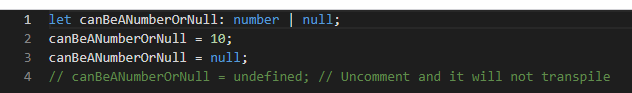


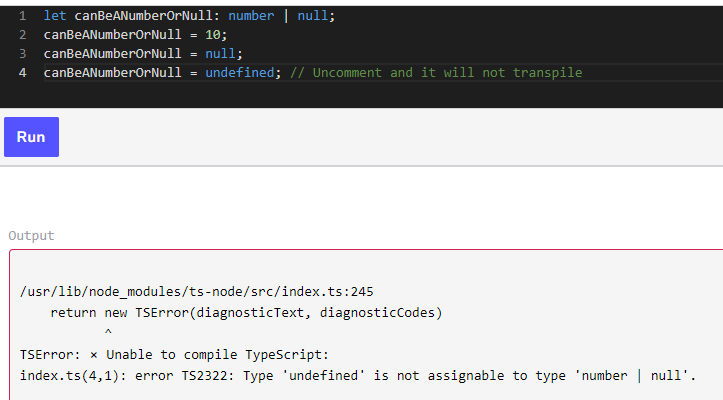
This lesson was a glimpse of the values used for variables that are not defined or optionally defined. Anytime a value does not hold a valid value, undefined is the best option (and an easier option for developers to manage).

null

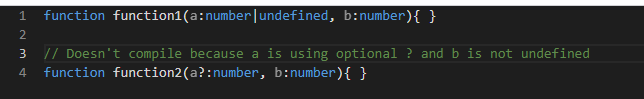
As discussed in the last section, increasing the strictness of TypeScript forces developers to use a question mark to define the variable optionally define their variables, which allows undefined. This distinction is used to ensure consistency in our code. With a clean-cut assertion by type of what can be undefined (optional), null, or with a specific value, the code becomes more comprehensible.

For example, a nullable number is two types, not a single one. Dual type (or more than one) is possible with the concept of *union*. The *union* uses the pipe character between the main type (for example, a number) and null –

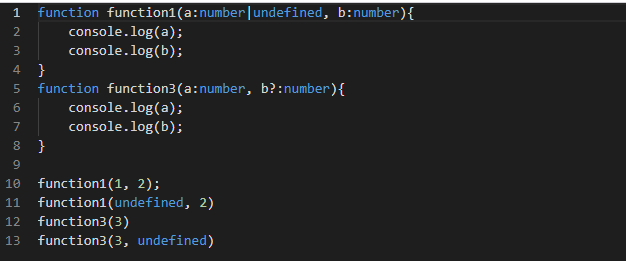




The union of any other type and undefined makes the type optional. Using question mark syntax or union with undefined produces the same result with only minor differences. The question mark is more succinct but also does not allow a non-undefined parameter to follow in a function signature –



Optional parameters should always come after non-optional parameters. A union with undefined means that the user must specify undefined which is not really optional since it must be explicit –



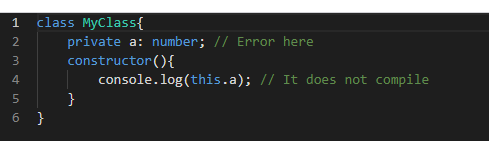
It is good practice to avoid using null as much as possible and rely instead on undefined. This helps to avoid having to handle undefined **and** null as well as the actual type.

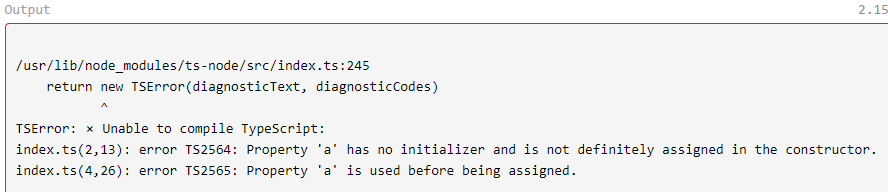
We chose undefined instead of null because of the natural tendency of JavaScript to lean toward undefined. The following code does not compile. However, the same code in JavaScript (without the type specified) would print undefined –



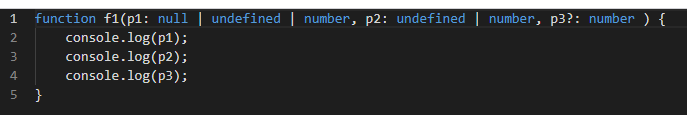
Members in a class set to a single type (without a union) cannot be defined as null or undefined explicitly but will be undefined until initialized.

The time window in between creates a state where the variable is undefined. A variable can be undefined regardless of its type. The following code would have return undefined normally, but with strictness turned up, TypeScript will not compile instead, it will return an exception indicating that the variable must be initialized –



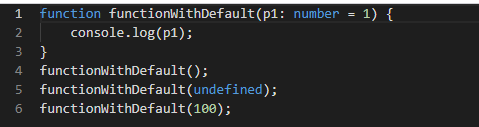


The following code demonstrates that null adds an additional level of complexity that most of the time can be avoided by using a type or undefined –

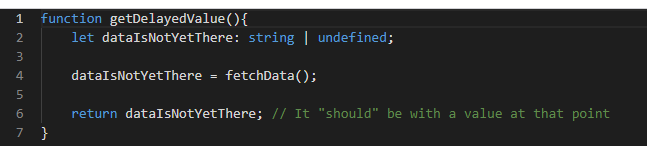


There are several use cases where undefined can be handy. One case is when a function or class does not require the variable; for example, an optional parameter or defined members that are not used all the time.

Another case is for optional data. Data can be optional and handled with a default behaviour or value when code needs to access the value of the variable. Often, third-party libraries provide default values but let users customize function calls. The third-party code checks to see if the option is defined (not-undefined) and if it is, uses it. Otherwise, if undefined, the library will use the default value –



When data is pulled from external sources, undefined may be used. The variable starts undefined until the data arrives from the external sources –



Undefined and optional values transpile into the same code as if nothing were assigned to the variables. This is because JavaScript is not aware of the concept. For example, a numeric variable with no value will simply be undefined (not null).

A good use case for null is to differentiate between –

1. An uninitialized variable: should be undefined
2. A variable clearly representing the absence of something: should be null
3. A variable representing a value: should neither be undefined, nor null

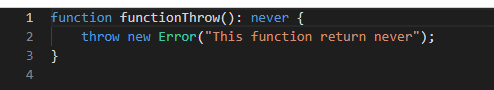
For example, consider the situation where we fetch data. The function can return an actual value, null if the data is actually present but has no value, or undefined if the data has not yet been computed.

The Primitive Type never

In this lesson, we will see the type never, which is used to indicate that something must never happen.

The type never means that nothing occurs. It is used when a type guard cannot occur or in a situation where an exception is always thrown.

There is a difference between void and never. A function that has the explicit return type of never won’t allow your code to return undefined, which is different from a void function that allows code to return undefined–

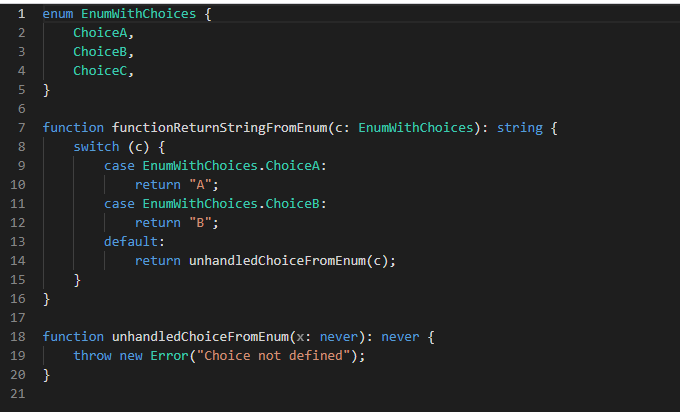


The never type is a subtype for every type. Hence, we can return never (for example, throwing an exception) when a return type is specified to be void or string, but cannot return a string when explicitly marked as never.

TypeScript can benefit from the never type by performing an exhaustive check. An exhaustive check verifies that every possibility (for all types in the union or all choices in an enum) is handled. The idea is that TypeScript can find an unhandled scenario as early as design-time and also at compilation time. It works by having a potential path that falls under the else condition, which returns never.

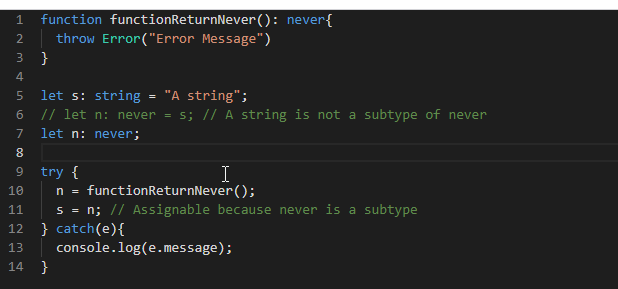
However, when all types of a union or enum cause the code to return something other than never, the compiler won’t complain. Using never is helpful when code around multiple type values evolve. When an option is added, for example, to a union or enum, TypeScript will compute that the function can return never and not compile. Since version 2.0, TypeScript can find out if the code was entered in the default case (or with else case if you are not using the switch statement).

For example, in the code below, there is an enum with two items. TypeScript knows that only two cases are possible and the default (else) case cannot occur. This insight of TypeScript is perfect since the function return type only accepts a string and does not accept never. If in the future you add a new item from enum, (for example, a ChoiceC, without adding a new case in the switch statement), then, the code can call the unhandledChoice function which returns never.

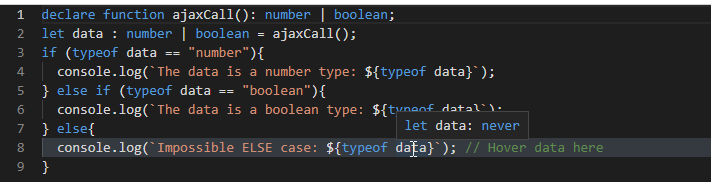


The type never is also used in the mapped type that we will see in later lessons. In every situation where never is used, it is to mark that the code should not be in a specific state, else it will not compile.

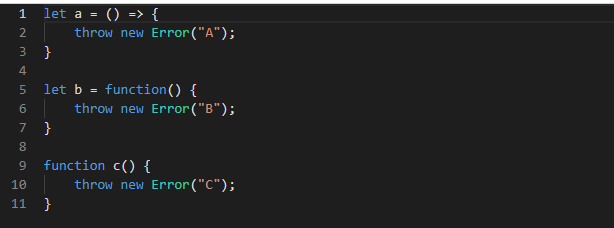
The primitive type never has been around since TypeScript 2.0. Its usage is limited, but its unique characteristics make it powerful. For example, never is a subtype of every type but it cannot be a subtype of any type other than itself.

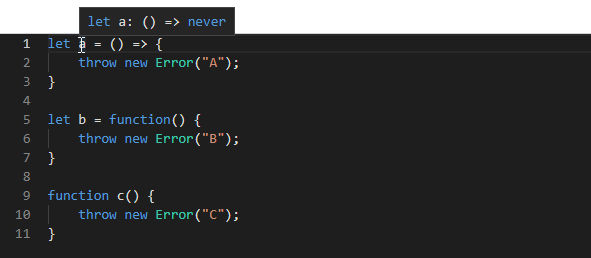


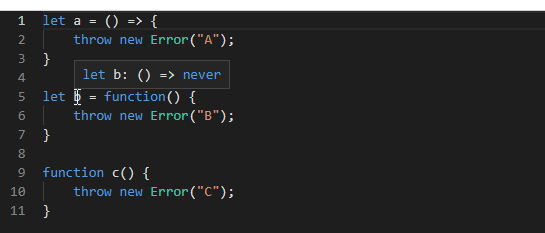
In cases where TypeScript is unable to logically identify a variable as a specific type, it will set the value to never. In the following example, the else case is theoretically impossible because the data variable can only be number or boolean, however, the else is coded anyway. The value of the variable data is, in that case, never. You can hover on the variable and see this for yourself.

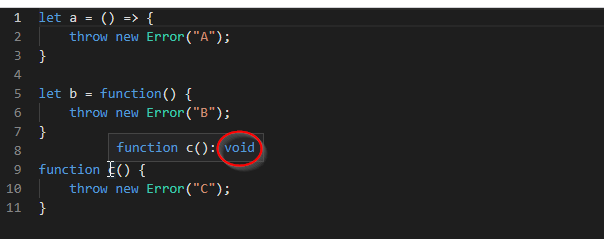


In a few lessons, we will discuss the different types of functions. But while we are still explaining the type never, let’s take a glimpse at how we define three functions and how they act differently with their inferred type. If you hover your cursor on the variables a, b, and on the function c, you might be surprised to see that the types are never, never, and void. There is a historical reason for this which serves a purpose on how JavaScript is used. Further details will be seen in the function lesson.









In the end, never indicates a state that is not meant to be. An exception is not expected behaviour. An infinite loop in a function is not meant to be sustainable in a system, a condition that is never visited should not exist.

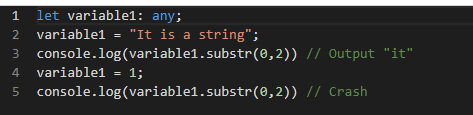
Reference video: <https://www.youtube.com/watch?v=aldIFYWu6xc>

Unknown: A better any

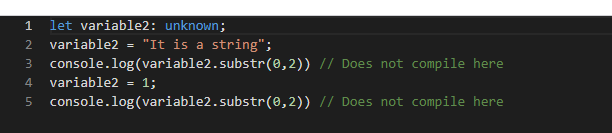
In this lesson, we will see the type, `unknown`.

The unknown type is half a specific explicit type and half the type any which allows everything. Declaring a variable as unknown allows us to set a wide variety of types without allowing unwanted access to properties or the value of a type. The following code below demonstrates that a variable with type any can be assigned a string and then used with a function of the string type.

Later, the variable is assigned to a number that does not have substr function. However, TypeScript does not catch an attempt to invoke a function that does not exist –

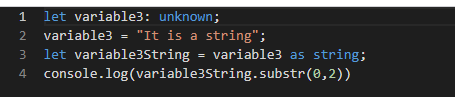


Changing the type from any to unknown indicates to TypeScript that the type can receive any value but should be used cautiously. It does not allow the function to be invoked.

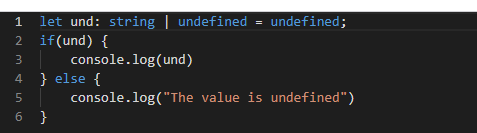


The only way to access hidden properties or values is to explicitly tell TypeScript a variable’s type. This can be done by casting or using a type assertion. Here is an example that lets an unknown variable use the string function, substr. variable3 is of unknown type but is explicitly cast by asserting its type as string.

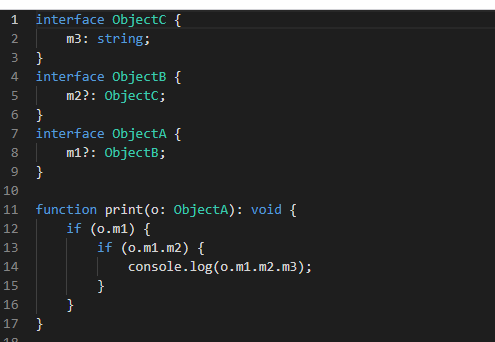
Forcing a type is not recommended because it can lead to specifying the wrong one. For example, variable3 may be a number asserted to be a string. Asserting an unknown type is dangerous and should be used with caution.

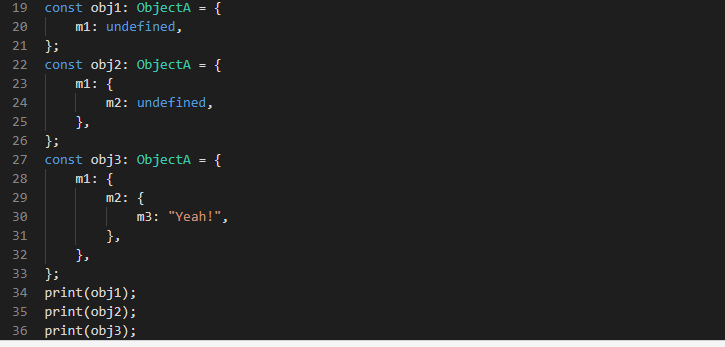


unknown and null can both be validated without using == or === because of JavaScript. Both are falsy.

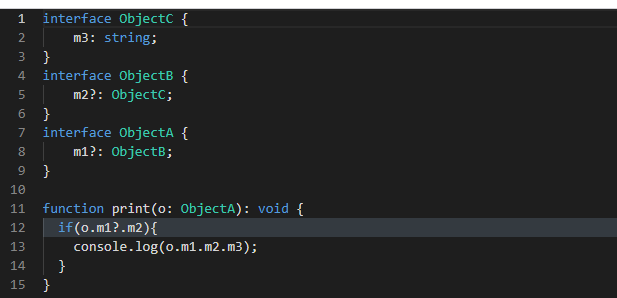


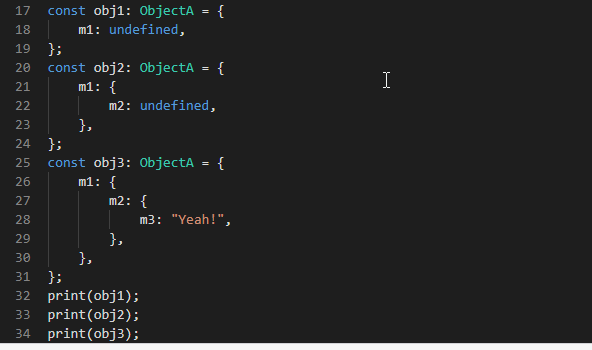
In case we need to display a value in an object that has many undefined/null (or optional) fields, several checks are required. The following example shows that only the last object displays the string because the others are nested with undefined values –



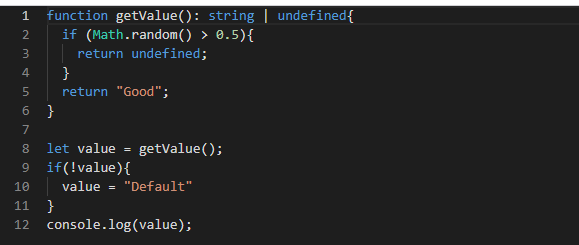


TypeScript versions 3.7 and up allow us to shortcut the conditions of null and undefined by using optional chaining. Optional chaining uses ?. and returns undefined if the chain of ?. contains a property that is null or undefined. Otherwise, it returns the value. If you change the previous example to use optional chaining, the code is reduced to –

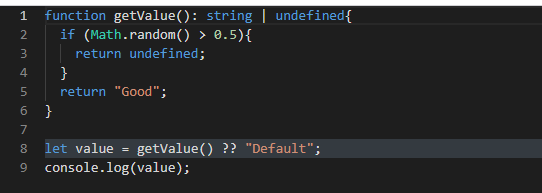




In the same vein, TypeScript has nullish coalescing that allows the code to be reduced before invoking something that can be null or undefined. If we run the following code several times, once in a while, we will get the value from the function and sometimes the one from the default value –



With TypeScript, since version 3.7, it is possible to use ?? to avoid the if statement –



A few lessons ago, we learned that the constructor of the Boolean object uses unknown. The constructor could take any, but with unknown, the type is sure to remain the same inside the boolean’s constructor and keep the code inside the constructor to access a limited range of properties. This wraps up our discussion of the unknown type.

Literal Type to narrow Primitive Type

In this lesson, you will see how string and number literals are different from conventional strings and numbers.

Literal Type

A literal type means that the value is an **exact one**. For example, a string literal of “test” would mean that the value of the variable can only be “test”.

A literal type can be made up of multiple types or values from primitive JavaScript types.

String Literals

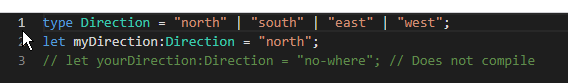
A string literal is a way to define a string that limits the potential values to be used.

It’s used mostly with a union, which allows specifying more than one string value. Imagine that we allow several strings’ values but want to limit the choice to specific ones. We could use an enumeration, but a string may be more clear or compatible with existing libraries. For example, we may want to limit the value to “north”, “south”, “east”, or “west” –



To create a string literal, we can define each value separated by the pipe symbol | (union of values). TypeScript will not compile if it goes outside the defined range.

The code below does not compile because yourDirection is declared to be of the Direction string literal type. Changing it to a string allows us to assign any string or change the value to one of the four defined types, and will fix the transpilation –



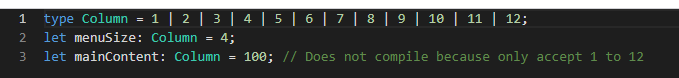
A string literal can be assigned without type by assigning a string value to with the const declaration or as const –



Number Literals

Similarly, it is possible to use numbers as a set of values. Using multiple defined numbers is convenient if we have a set of values that we accept but are not all numbers.

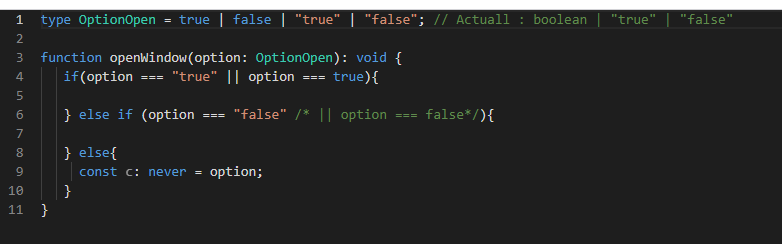
For example, if we create a framework where we want to create a view grid system that works on a grid of twelve columns, we may restrict the choices from one to twelve –



Literal Mixed Type

It is possible to also create a mixed type that causes the literal to be of multiple types with the union. In the following code below, we have a line that does not compile because one of the literals is not covered, causing the never type to be assigned even though it cannot be compiled.

Uncommenting the false option would result in compliable code since all values of the type are handled –



We will see later that with object-oriented overload functions, string literals can become handy to distinguish between overload when only the value of the string changes and that literal type can be used to discriminate between objects.

In this lesson, we saw that we can circumscribe the potential string values of a variable to a smaller subset of strings. We also saw that it is possible to extend this concept to number with the number literal type.

Literal types are the perfect mix between good design and runtime validation since the condition is executed at runtime. They can be used to narrow type, verify that all possibilities are covered, select the proper overloaded methods, and handle cases where a type evolves (versioning).

Symbol & Unique Symbol

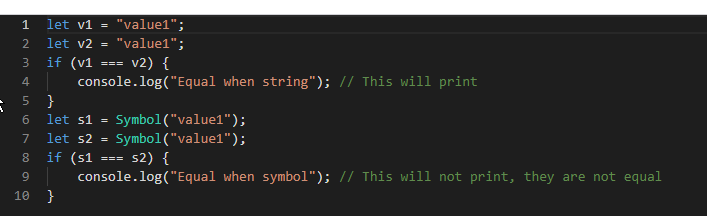
In this lesson, we see how TypeScript strongly types the primitive type, symbol, and its subtype, unique symbol.

Symbol

Symbol is a primitive type in ECMAScript 2015 and beyond. TypeScript supports the standard. The equal sign assigns a value to a symbol **without** the keyword new but must have parentheses, like an object. A symbol’s goal is to provide a unique and immutable variable.

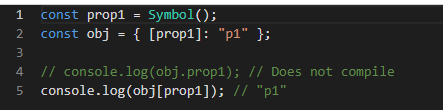
A symbol can take a parameter with a string value. Defining two symbols with the same parameter will produce a different symbol. In fact, the parameter is just there to help developers when printing the symbol to the console. It’s a way to differentiate them visually.

The main difference between a constant and a symbol is that the symbol is unique. With a string constant, someone could pass a string with the same value as the constant and it would be accepted. However, using a constant symbol, only the same symbol constant would equal that value. Nothing can coerce a symbol into a string. This means that we cannot add a string to it and expect to become a string –



An object property can be a symbol. Its assignment uses the symbol between brackets. Do keep in mind that a property defined with a symbol won’t appear when we invoke Object.defineProperty or Object.getOwnPropertyNames.

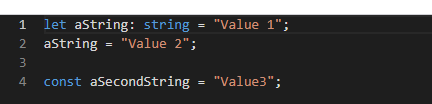
To get all properties defined by symbols, we must use getOwnPropertySymbols. If all properties defined are required, we must use Reflect.ownKeys(). In the end, the goal is to provide a unique way to define a specific member of the object and avoid a potential collision that a string cannot prevent.

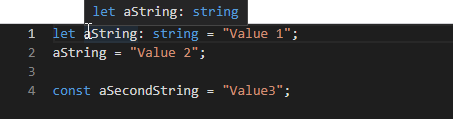


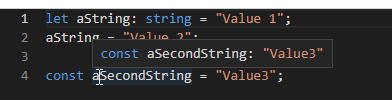
Unique Symbol

A unique symbol can only be defined with the use of const or readonly static. A unique symbol is used to create a literal type that cannot be of another symbol. Hence, the type is not symbol but a symbol with a unique identity.

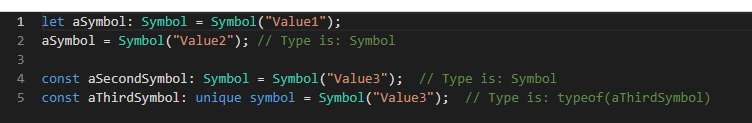
A way to look at this is the way a string can be a string or a string literal. Hovering over the first variable of the following code shows that it is type string, while the type of the second variable is Value3 –

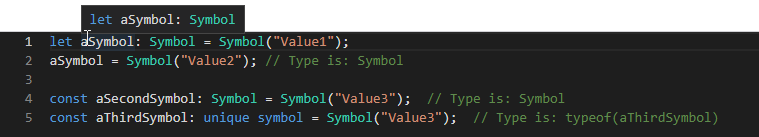


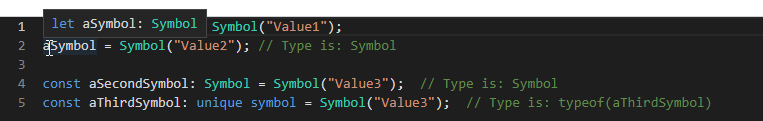


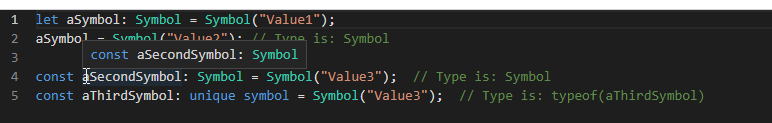


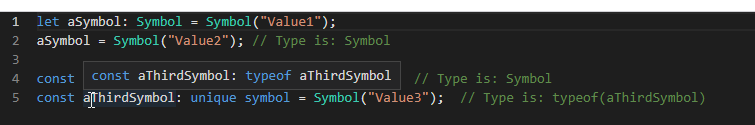
Back to the unique symbol, it is similar. In the following code, both the first and second variables are of type Symbol. However, the last symbol is not of type Symbol, but of type typeof(aThirdSymbol) –



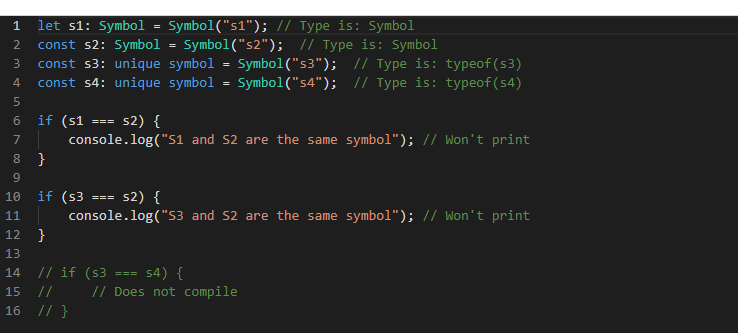


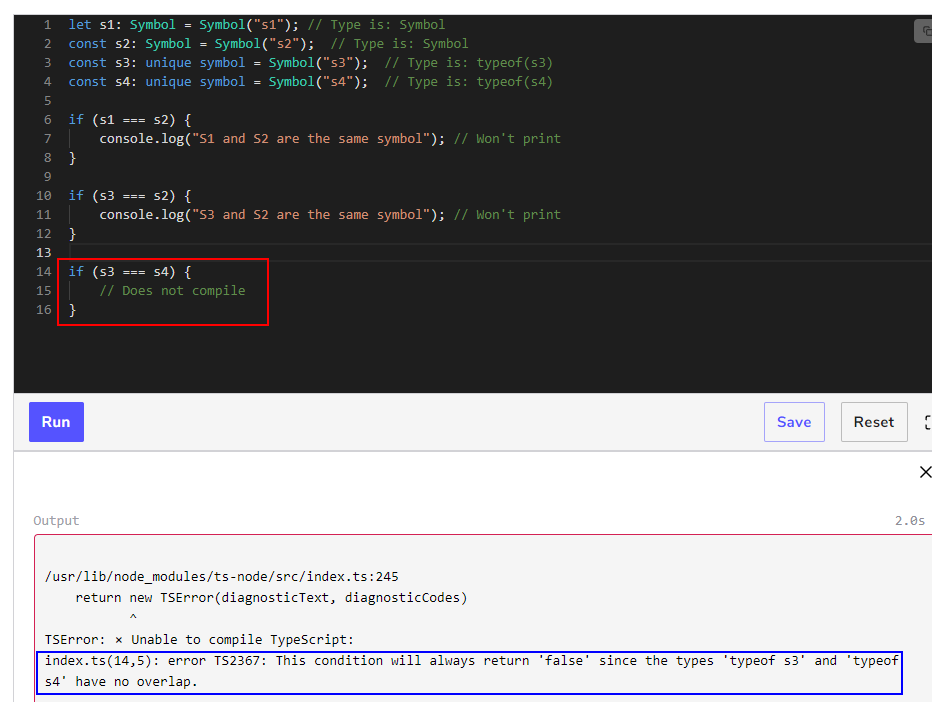






A unique symbol can only be declared with const. They are also unique therefore, if compared, will always return false. The next example compares a Symbol with another Symbol as well as to a unique symbol –





Casting to Change Type

In this lesson, we will see how to move from one type to another.

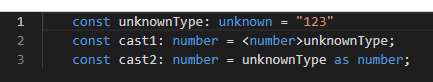
How can we cast?

TypeScript can cast using two different forms: <> or as. The former is not recommended because it conflicts with the JSX/TSX format which is now becoming popular because of React. The latter is just as good, and it works in all situations.

The first way is to use the symbols < and > with the type desired in between. This syntax requires the cast before the variable that we want to coerce.

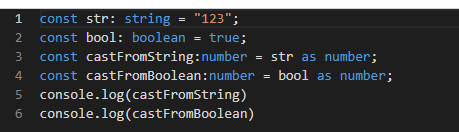
The second way is to use the keyword as. as is placed after the variable we want to cast and followed by the type we want to cast.

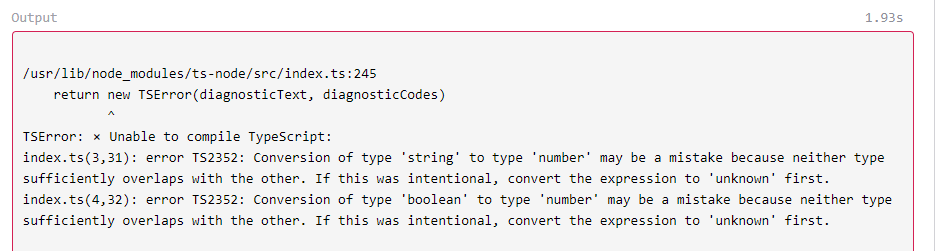
The following code demonstrates an unknown type cast to a number –



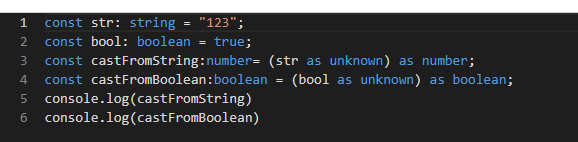
Casting Constraints

If we try to cast to a string directly without using an unknown type, TypeScript will warn that there are not sufficient overlaps. The TypeScript transpiler gives the solution: casting to unknown –



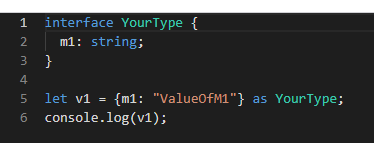


In that case, a double cast is needed. First, we cast to unknown and then to the desired type –

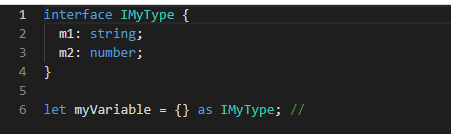


Type Assertion

Casting is a delicate subject since we can cast any variable into anything else without properly respecting its original type. A particular case is “type assertion”. **Type assertion** is when we tell TypeScript what type an object is –



Where it can be delicate is, for example, if we have an interface that requires many fields and we cast an empty object to it, it will compile even if we do not have the members –



The fallacy of the cast when the underlying object doesn’t respect the type schema is one reason why it’s better to assign a type to the variable and not cast.

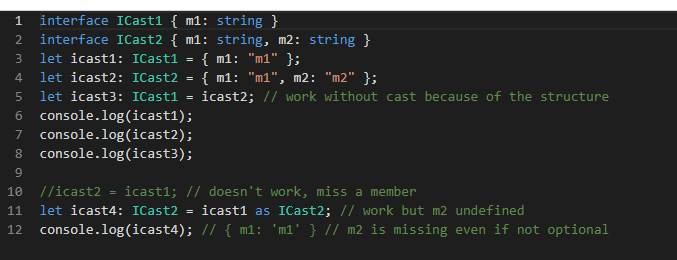
However, there is a situation where we must cast. For example, if we receive a JSON payload from an Ajax call, this will be of any type by nature since the response of an Ajax is undetermined until the consumer does the call. In that case, we must cast to manipulate the data in a typed fashion in the rest of our application.

The constraint here is that we must be sure that we are receiving the data in a format that provides all expected members. Otherwise, it would be wiser to define these members to be optional (undefined as well as the expected type).

Casting Restrictions

Casting has some restrictions. For instance, we cannot cast a typed object into something that is not a subtype of the original type. If we have a TypeC that inherits TypeB which inherits TypeA, we can cast a TypeC to TypeA or TypeB without problem or TypeB to TypeA without casting.

However, going the other way around requires a cast. Nevertheless, there are issues with both cases. When going from a subtype to a type, without casting, the problem is that TypeScript will only validate access to the public type from the desired interface. However, under the hood, the object still contains all the members. For example (see below), TypeB has two members; when casting, it only exposes (at design time) the first member, which is in TypeA. However, printing the object reveals that both members are still there. The lack of cohesion between the type’s schema and the actual object structure is an important detail. For example, sending an object to an API without manually grooming the object may pass more information than anticipated –



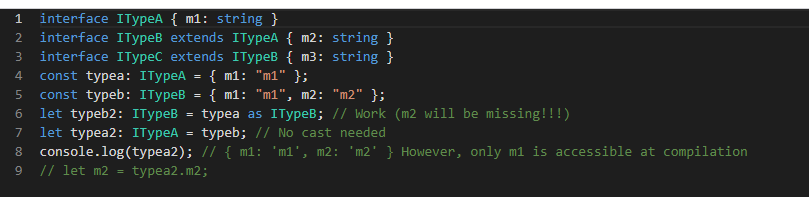
The first example above shows an example of two different types (ICast2 to ICast1) without casting (line 5). This is possible because they are the same structure.

The second example, line 11, cast but is not fine. It coerces the type which causes m2 to be undefined because it does not exist in ICast1. Line 10 is commented.

Uncommenting shows that without casting, when the structure is not similar, TypeScript blocks the transpilation.

The second issue is with casting. Since casting tells TypeScript that we know what we’re doing, it won’t complain. However, non-optional members that aren’t present will be undefined even if the contract specifies that the type must have the member. We can see an example below of when an object of TypeA (base interface) is cast down to TypeB. The cast coerces the change of type, but m2 is still not present.

While it is good enough for TypeScript that we manually override the validation, it can be problematic if later in the code, we try to access m2 and believe that it cannot be undefined. In fact, this can cause a runtime error if we try to access a function of the member. The example below has line 9 commented. It demonstrates that the member is not available at design time. However, when commented, we can see that it is present at runtime (JavaScript).



The best practice with casting is to do it as little as possible. To cast at a strategic place in our code (like when getting an untyped object) is smart. When we need to create a new type, it’s better to assign a type (using explicit declaration) than to cast it. Doing so will provide IntelliSense support and transpiler protection, which keep the code stable with the expected type.

# **Enum (Enumeration)**

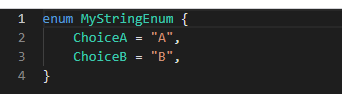
Enum with and without values

In this lesson, we will discover how to use an enum with explicit and implicit values.

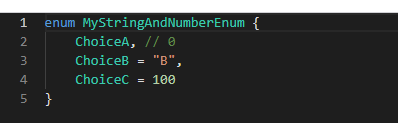
The role of enum

An enum is a structure that proposes several allowed values for a variable. It is a way to constrain variable values by defining specific possible entries.

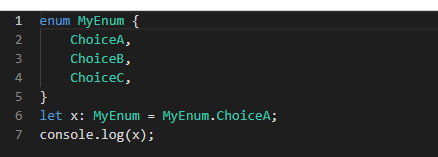
enum with values: enum can be of string type. In that case, every member requires a value without exception –



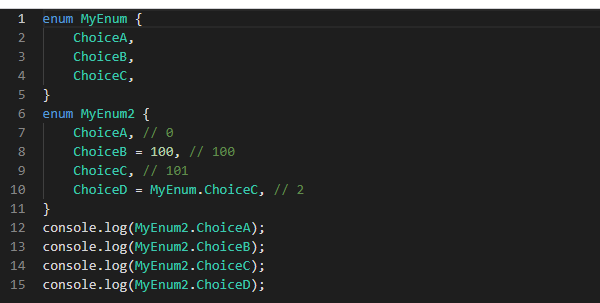
A mixed enum value type is acceptable if every member is defined. For example, we can have one item be an integer and another be a string type. It is however recommended not to mix types since it might be more confusing than pragmatic –

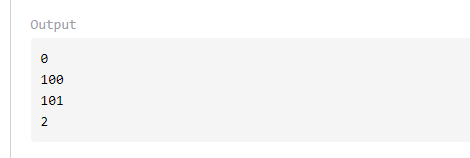


enum without values: enum is a type that enforces a limited and defined group of constants. enum must have a name and accepted values. Later, we can use the enum as a type. The consumer must use the enum with its name followed by a dot and a potential value from the defined list –



The values are all constants starting from 0 for the first item and increasing by one until the end. This type of enum has **implicit** value. Developers can specify a specific value by equating to an integer. In that case, the enum is **explicit** –



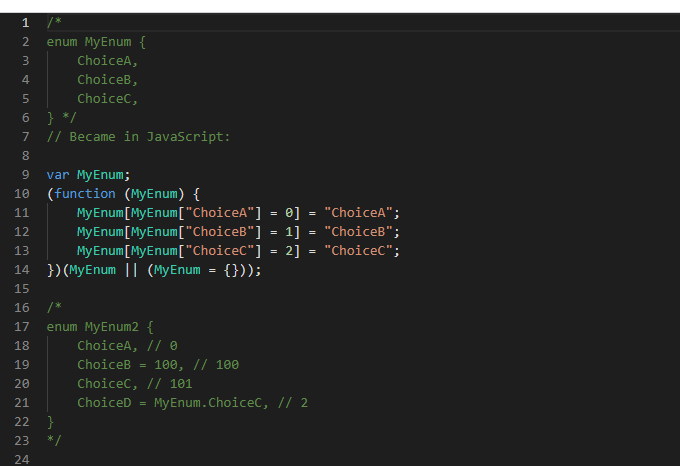


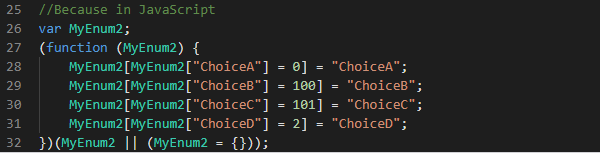
enum members’ values can be set directly or by using computation. There are two types of computation –

1. a constant one
2. a purely computed one

A computed constant is a value provided by another enum or a value computed by addition, subtraction, bitwise, modulo, multiplication, division, “or,” “and,” “xor” operator, or complement operator (~). Purely computed values come from a **function**.

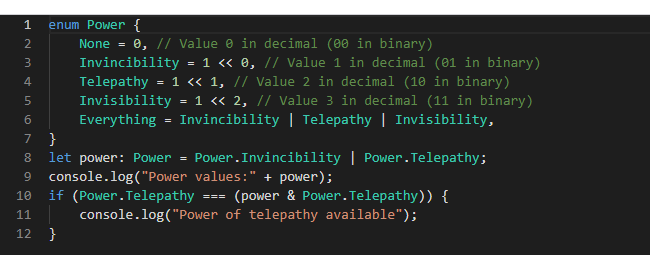
enum generates a function in JavaScript with a set that allows us to specify the number or name used to access the value. Here is the output of the two previously studied enum –





enum with bitwise values: enum is a good candidate for bitwise operations since the value can be explicitly set (value set during the definition of the enum) and you can use the bit shift operator. Once defined, you can use it as any variable to determine if it contains the one you need or use the ampersand (&) to check if the one you want is present. The pipe symbol (|) lets you add many enum choices to a variable.

The following code not only initializes the value with the | but also checks the value. With bitwise, we cannot directly use an equal sign. The reason is that bitwise operation returns a number, not a boolean. Hence, we need to compare the number to the desired comparison value. Line 10 demonstrates how to check the value of an enum –

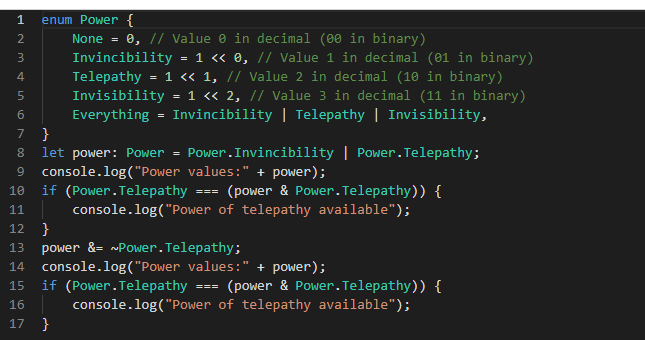


The value of the previous example is 3 because the Invincibility value is 1<<2, which is binary 10.

The Telepathy value is 1<<1 which gives the binary 01 and the or operation provided by the pipe symbol gives binary 11 which is 3.

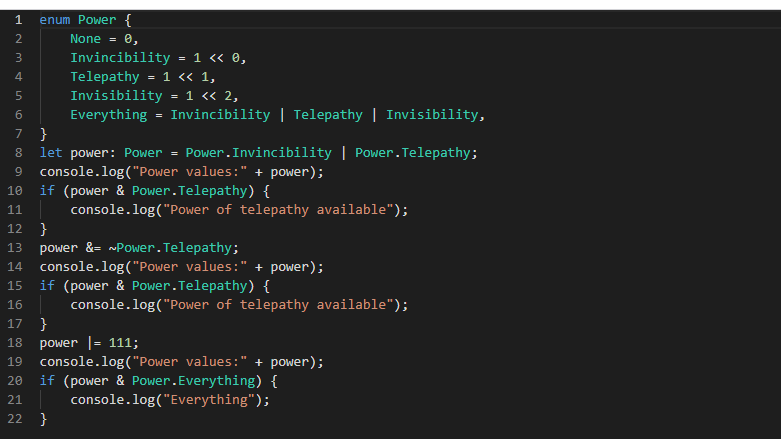
It is possible to remove a value from a bitwise enum on the fly by using &= ~ which performs an and operation on the inverse of the value.

For example, the following code supplements the previous example by removing the Telepathy power. Line 13 has the remove operation –



The value is 1 because from the 3, (which is in binary 11) we use and of the inverse of 10 which is 01. 11 and 01 = 01 which is 1.

Adding a value on the fly uses the pipe as when we initialized the value. Line 18 shows that not only can we use Power.Everything to set all the values of the enum, but we can also directly use a number that represents the binary of the values. In that case, 111 sets the first three powers to true –



Now let’s see how to access enum values next.

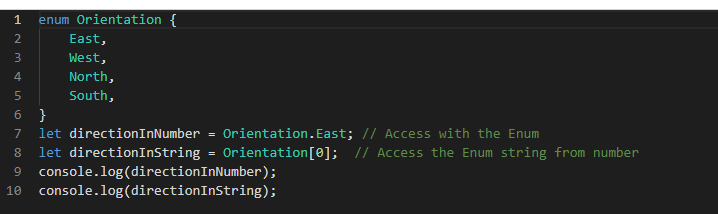
Accessing Enum Values

In this lesson, we will see how to access information from an enum.

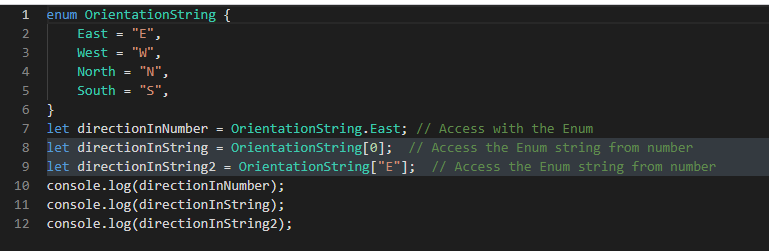
TS map objects to allow access

A variable set with an enum that has a number lets us access the enum name from the integer. However, an enum with string values does not have this capability. This means we can use the enum name followed by the name of the constant to get the value. Also, with a number, we can also use the value to return the name.

For example, an enum called Orientation with East, West, North, South could use Orientation.East to get the value zero or use Orientation[0] to get East. This works because TypeScript generates a map object which gives us access using the name of the entry or the value. Here is the generated code of the orientation enum –

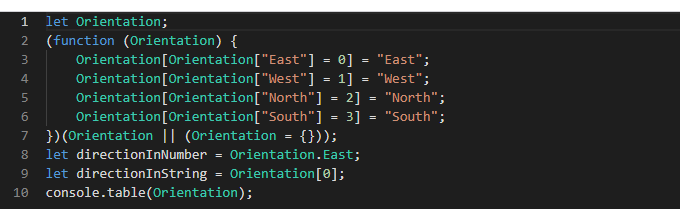


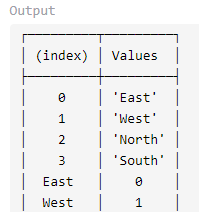
As mentioned, it is not possible with an enum that has strings for value. The following code does not compile because lines 8 and 9 wrongly accessed the enum –



The JavaScript output

The JavaScript output looks like the following for the first valid example –







The JavaScript code generated by TypeScript creates a closure that assigns to a variable (Orientation) the four possible values by number as well as with string. The Orientation variable is an array with eight elements. The code on line 10 added an output that demonstrates how the values are accessible either way.

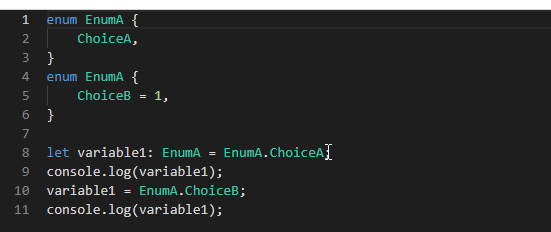
Merging and Adding Functionality to Enum

In this lesson, you will see two advanced features: merging and adding functions to enum.

Merging Values

Like interfaces, an enum can be defined in more than one place. We can start defining the enum and later define it again. In the end, all values merge into a single enum.

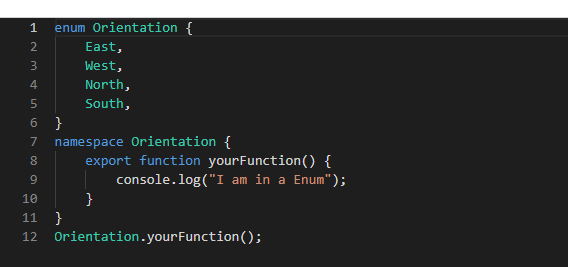
There is one constraint with multiple definitions of a single enum: the first value of every enum must have an explicit value. If an explicit value is defined twice, only the last value will be associated with the enum when using the reverse value to find an enum. Listing the same value twice is not a feature of multiple definitions; a single enumeration definition can have several entries with the same values as well.



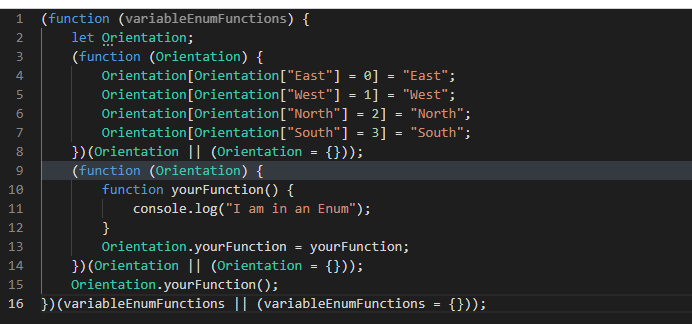
Adding Functions

Another feature of enum is that we can attach functions that will be accessible statically by the enum.

Using an enum with a function means that we can use Orientation.East as well as Orientation.yourFunction. Defining a function inside an enum requires the use of a namespace with an exported function.



The generated JavaScript hooks the function to the enum’s function –



As we can see, the final product is that an enum is a function that wraps other functions. Hence, it is possible to add functions to an enum.

# **Mapped Type**