Declaring a Variable

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

WE'LL COVER THE FOLLOWING

Declaring with `var`

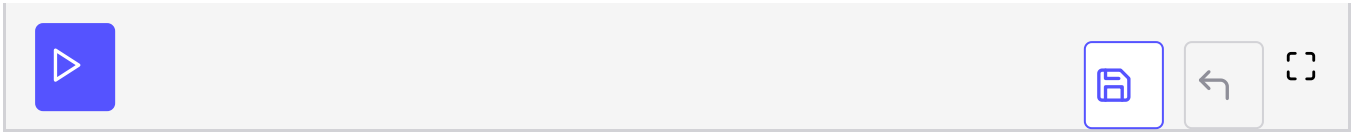
Let's get started with `var`. This has been the way to define variables since the inception of JavaScript. However, the release of ECMAScript2015 brought the declarations `let` and `const` which fixed many of the drawbacks perpetuated in previous versions of ECMAScript. In many modern languages, declaring 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` issue 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 you from redefining the same variable, which overrides the initial declaration or initialization.

```
function varFunction(){
    var x = "111";
    if(true){
        var x = "999"; // Variable x redefined
    }
    console.log(x);
}
```

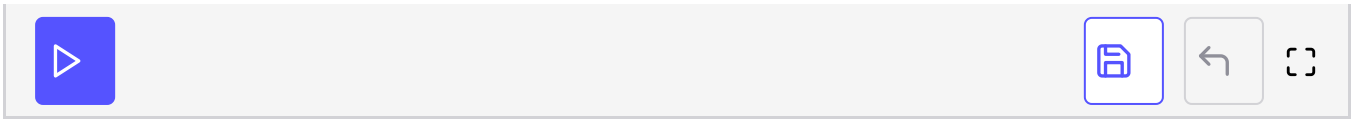
```
varFunction()
```



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).

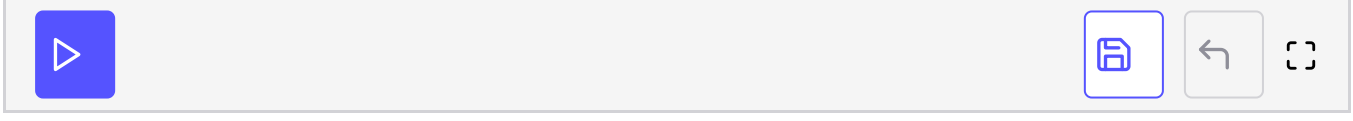
However, the new definition is contained within the scope of the function. For example, trying to output `x` from outside the function will return `undefined`.

```
var x;
function varFunction(){
    var x = "111";
    if(true){
        var x = "999";
    }
    console.log(x);
}
varFunction()
console.log(x)
```



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

```
var x;
function varFunction(){
    var x = "111";
}
console.log(x)
```



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

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 determine a block.

For example, if you 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 you are defining a `for` loop and you define the iterative `i`, this one should use `let` which allows modifying its values while being only available for the loop.

```
function letFunction() {
    let x = "111";
    if (true) {
        let x = 999;
    }
    console.log(x);
}
letFunction();
```

There are two big to note when `let` is compared with `var` in this example.

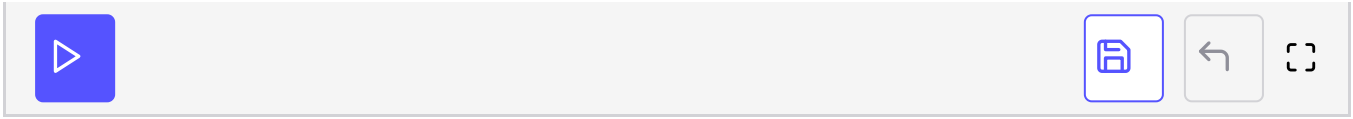
1. Though the output is expected to be the one defined in the scope. It means the value *999* remains in the scope of the condition, and the value “111” is available only in the scope of the function.

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 out that the former `x` is a `string` while the latter is a `number`.

> 📜**Note**: The below code is expected to throw an error as explained ❌

```
function varLetFunction(){
```

```
    var myStringValue = "varStringValue";
    let myStringValueLet = "letStringValue";
    if(true){

        var myStringValue = 123;
        let myStringValueLet = 12345;
    }
}
```



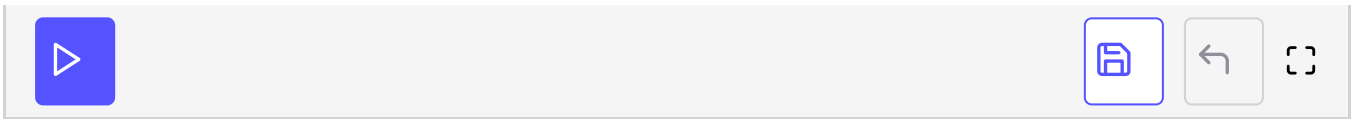
The code above does not compile because of the `var` variable. The error that you should see is:

```
error TS2403: Subsequent variable declarations must have the same type.  V
ariable 'myStringValue' must be of type 'string', but here has type 'numbe
r'.
```

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` which are two alternatives with a cleaner scope definition.

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

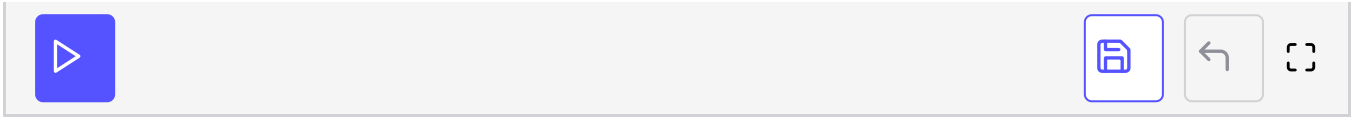
```
var myVarVariableOutSide = "I am outside";
let myLetVariableOutSide = "I am outside too";
function letOutsideFunction() {
    console.log(myVarVariableOutSide);
    console.log(myLetVariableOutSide);
}
letOutsideFunction();
```



The same is true for the condition and looping statement.

```
var myVarVariableOutSide = "I am outside";
let myLetVariableOutSide = "I am outside too";
function letOutsideFunction2() {
    myVarVariableOutSide += " and inside";
    myLetVariableOutSide += " and inside as well";
    console.log(myVarVariableOutSide);
    console.log(myLetVariableOutSide);
    if(true){
```

```
        console.log(myVarVariableOutSide);
        console.log(myLetVariableOutSide);
    }

}
letOutsideFunction2();
```

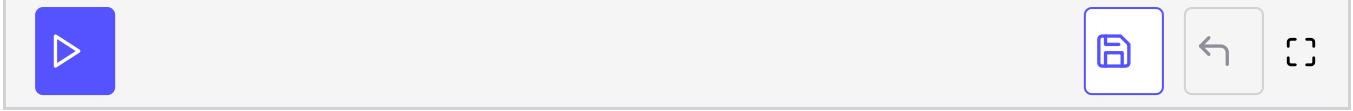


Declaring with `const`

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 you have a **constant object**, the value of that object can change.

> 📜**Note**: The below code is expected to throw an error as explained ❌

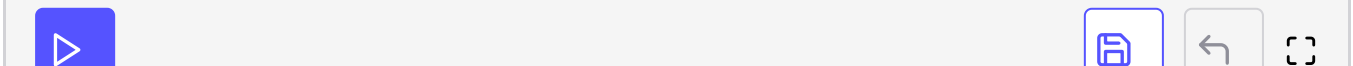
```
const x = "111";
x = "this won't compile";
```



The assignation to members of the constant variable is authorized. For example, you declare and initialize a variable that holds the current `user` of your application to a constant. You won't be able to assign the current `user` to any other user. The following code demonstrates that `user2` cannot be assigned to `user1`.

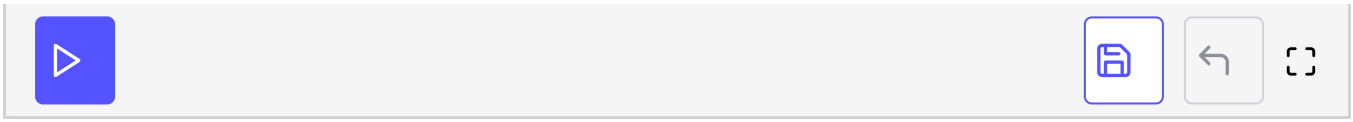
> 📜**Note**: The below code is expected to throw an error as explained ❌

```
const user1 = { id: 1, name: "MyName1" };
const user2 = { id: 2, name: "MyName2" };

user1 = user2; // Cannot assign a constant!
```



However, you can set its name if a public member is available.

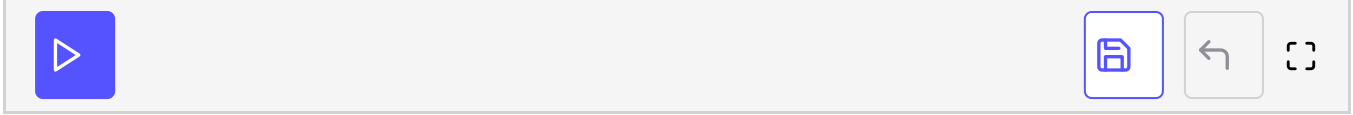
```
const user1 = { id: 1, name: "MyName1" };
const user2 = { id: 2, name: "MyName2" };

user1.name = user2.name; // Legit!
user1.id = 1000; // Legit !
```



While developing, you will be very surprised that most variable can be constant. It gives more protection to direct assignation.

Let's move on to array and constant.

```
function constFunction() {
    const myList = [1, 2, 3];
    myList.push(4);
    console.log(myList);
}
constFunction();
```



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

```
error TS2588: Cannot assign to 'myList' because it is a constant.
```

> 📜**Note**: The below code is expected to throw an error as explained ✖

```
function constBlockedFunction() {
    const myList = [1, 2, 3];
    myList = [1, 2, 3, 4]
    console.log(myList);
}
constBlockedFunction();
```

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, remove **line 6** and the code will compile.

> 📜**Note**: The below code is expected to throw an error as explained ❌

```
function constChangeObject() {
    let obj1 = { p1: "p1value" };
    obj1 = { p1: "p1value changed" };

    const obj2 = { p2: "p2value" };
    obj2 = { p2: "Does not compile" };
    obj2.p2 = "Work!";
}
```

A small note about TypeScript and variables. TypeScript is excellent to find typos. For example, if by mistake you write `mylist` instead of `myList` the code will not compile. That is true for all declaration (var, let, const).

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

```
error TS2552: Cannot find name 'mylist'. Did you mean 'myList'?
```

> 📜**Note**: The below code is expected to throw an error as explained ❌

```
function constBlockedFunction() {
    const myList = [1, 2, 3];
    mylist = [1, 2, 3, 4]
    console.log(myList);
}
constBlockedFunction();
```

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

What about type?

So far, you 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 used automatically 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 semi-colon.

```
var varName1: number;
let varName2: string;
const varName3: boolean;
```

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.