# Var vs Let vs Const and the temporal dead zone

Learn the new ways of declaring variables introduced in ES6.

WE'LL COVER THE FOLLOWING



With the introduction of `let` and `const` in **ES6**, we can now better define our variables depending on our needs. During our `JavaScript` primer we looked at the basic differences between these 3 keywords, now we will go into more detail.

## `Var` #

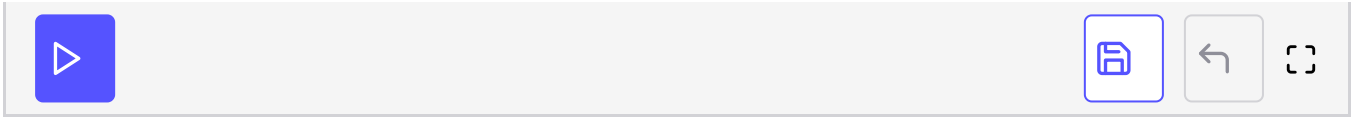
Variables declared with the `var` keyword are **function scoped**, which means that if we declare them inside a `for` loop (which is a **block** scope), they will be available even outside of it.

```
for (var i = 0; i < 10; i++) {
  var leak = "I am available outside of the loop";
}

console.log(leak);
// I am available outside of the loop
```

Let's take a look at an example with a functional scope variable:

```
function myFunc(){
  var functionScoped = "I am available inside this function";
  console.log(functionScoped);
}
myFunc();
// I am available inside this function
console.log(functionScoped);
// ReferenceError: functionScoped is not defined
```



In the first example, the value of the `var` leaked out of the block scope and could be accessed from outside of it. Whereas in the second example, `var` was confined inside a function-scope, and we could not access it from outside.

## `Let` #

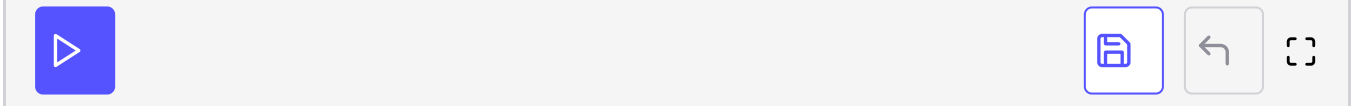
Variables declared with the `let` (and `const`) keyword are **block scoped**, meaning that they will be available only inside of the block where they are declared and its sub-blocks.

```
// using `let`
let x = "global";

if (x === "global") {
  let x = "block-scoped";

  console.log(x);
  // expected output: block-scoped
}

console.log(x);
// expected output: global
```



The same example using `var` is given below.

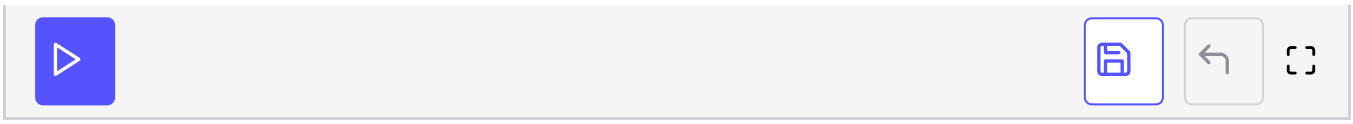
```
// using `var`
var y = "global";

if (y === "global") {
```

```
  var  y= "block-scoped";

  console.log(y);
  // expected output: block-scoped
}

console.log(y);
// expected output: block-scoped
```



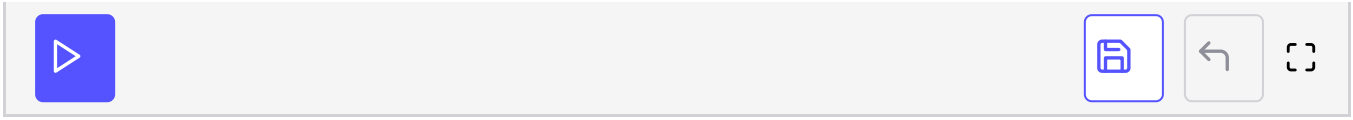
As you can see, when we assigned a new value to the variable declared with `let` inside our block scope, it **did not** change its value in the outer scope. Whereas, when we did the same with the variable declared with var, it leaked outside of the block scope and also changed it in the outer scope.

## `Const` #

Similarly to `let`, variables declared with `const` are also **block scoped**, but they differ in the fact that their value **can't change through re-assignment and can't be re-declared**.

```
const constant = 'I am a constant';
constant = " I can't be reassigned";

// Uncaught TypeError: Assignment to constant variable
```

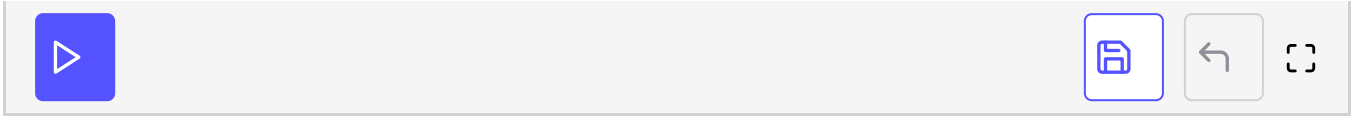


> **Important**: This **does not** mean that variables declared with `const` are immutable.

### The content of a `const` is an object #

```
const person = {
  name: 'Alberto',
  age: 25,
}
```

```
person.age = 26;
console.log(person.age);
// 26
```



In this case we are not reassigning the whole variable but just one of its properties, which works fine.

---

> Note: We can still freeze the `const` object, which will not change the contents of the object (but trying to change the values of object `JavaScript` will not throw any error).

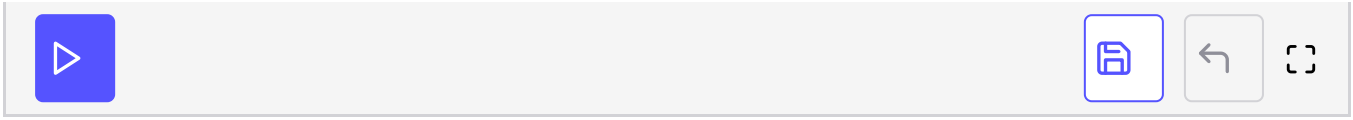
```
const person = {
  name: 'Alberto',
  age: 25,
}

person.age = 26;
console.log(person.age);
// 26

Object.freeze(person)

person.age = 30;

console.log(person.age);
// 26
```



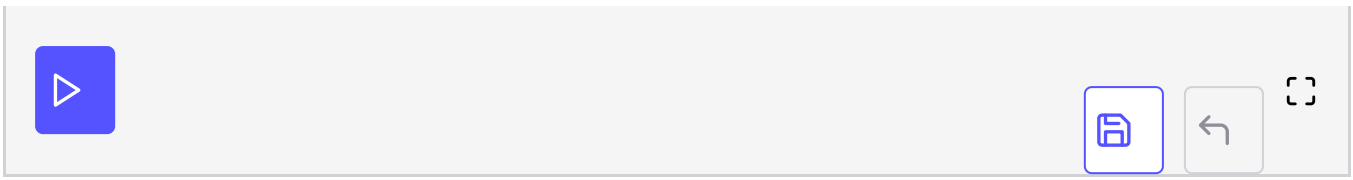
## The temporal dead zone #

Now, we'll have a look at a very important concept which may sound complicated because of its name, but I assure you it is not.

First let's have a look at a simple example:

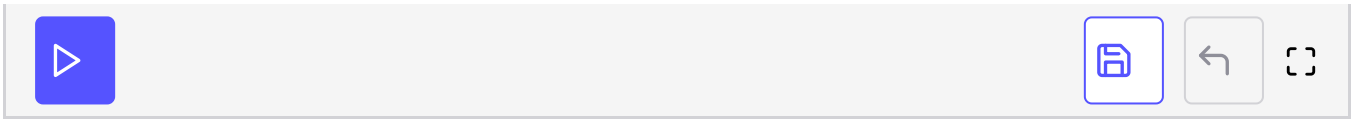
```
console.log(i);
var i = "I am a variable";

//  expected output: undefined
```



```
console.log(j);
let j = "I am a let";

// expected output: ReferenceError: can't access lexical declaration `j' before initializatic
```



`var` can be accessed **before** they are defined, but we can't access their **value**. `let` and `const` can't be accessed **before we define them**.

Despite what you may read on other sources, both `var` and `let` (and `const`) are subject to **hoisting**, which means that they are processed before any code is executed and lifted up to the top of their scope (whether it's global or block).

The main differences lie in the fact that `var` can still be accessed before they are defined. This causes the value to be `undefined`. While on the other hand, `let` lets the variables sit in a **temporal dead zone** until they are declared. And this causes an error when accessed before initialization, which makes it easier to debug code rather than having an `undefined` as the result.

---

## When to use `Var`, `Let` and `Const` #

There is no rule stating where to use each of them, and people have different opinions. Here I am going to present to you two opinions from popular developers in the JavaScript community.

The first opinion comes from Mathias Bynes:

- Use `const` by default
- Use `let` only if rebinding is needed.
- `var` should never be used in ES6.

The second opinion comes from Kyle Simpson:

- Use `var` for top-level variables that are shared across many (especially larger) scopes.
- Use `let` for localized variables in smaller scopes.
- Refactor `let` to `const` only after some code has to be written, and you're reasonably sure that you've got a case where there shouldn't be variable reassignment.

Which opinion to follow is entirely up to you. As always, do your own research and figure out which one you think is the best.

My personal opinion is to always use `const` by default and then switch to `let` if you see yourself in need of rebinding the value.

In the next lesson, we'll solve a quiz to test the concepts covered so far.