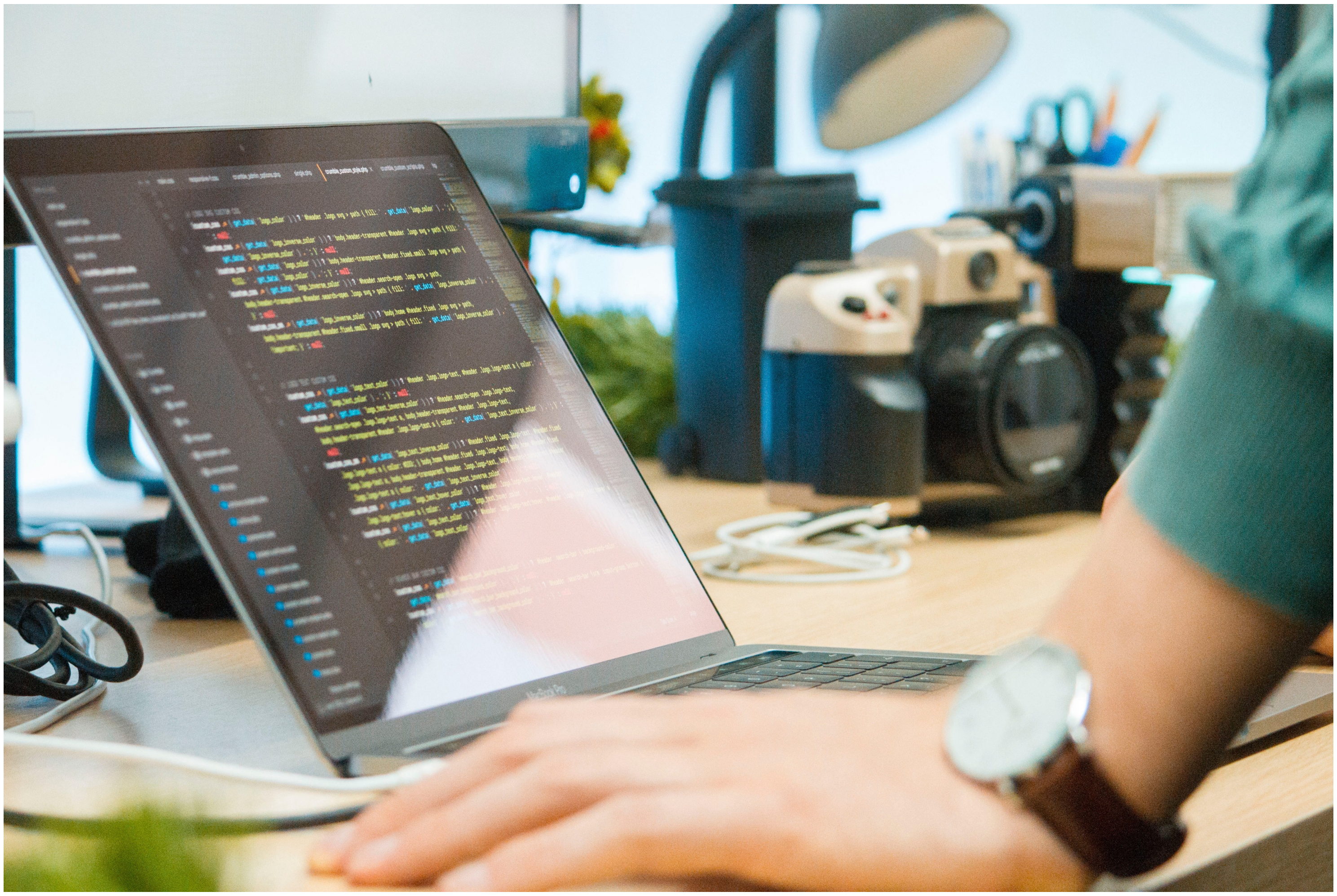
Difference between var, let and const in Javascript.

Understanding Function scope vs. Block scope in Javascript

Prashant Ram Follow
May 22, 2019 · 5 min read ★



The TL;DR version

In Javascript one can define variables using the keywords *`var, let`* or *`const.`*

```
var a=10;
let b=20;

const PI=3.14;
```

`var`: The scope of a variable defined with the keyword "var" is limited to the "function" within which it is defined. If it is defined outside any function, the scope of the variable is global.
var is "function scoped".

`let`: The scope of a variable defined with the keyword "let" or "const" is limited to the "block" defined by curly braces i.e. `{}` .
"let" and "const" are"block scoped".

`const`: The scope of a variable defined with the keyword "const" is limited to the block defined by curly braces. However if a variable is defined with keyword const, it cannot be reassigned.
"const" cannot be re-assigned to a new value. However it CAN be mutated.

Function scoped vs. Block scoped

Let us understand this by some code examples,

Block Scope

In Javascript you can define a code block using curly braces i.e `{}` .
Consider the following code that has 2 code blocks each delimited by `{}` .

```
{
  var a=10;
  console.log(a);
} //block 1

{
  a++;
  console.log(a);
} //block 2

/* Since we are using "var a=10", scope of "a" is limited to the
function within which it is defined. In this case it is within the
global function scope */
```

In the above example, since we are using the keyword `var` to define the variable `a`, the scope of `a` is limited to the function within which it is defined. *Since `a` is not defined within any function, the scope of the variable `a` is global, which means that `a` is recognized within block 2.*

In effect if a variable is defined with keyword `var`, Javascript does not recognize the `{}` as the scope delimiter. Instead the variable must be enclosed within a "function" to limit it's scope to that function.

Let us re-write the code above using the keyword `let`.
The `let` keyword was introduced as part of ES6 syntax, as an alternative to `var` to define variables in Javascript.

```
{
 let a=10;
 console.log(a);
} //block 1

{
  a++;
  console.log(a);
} //block 2

/* Since we are using "let a=10", scope of "a" is limited to block 1
and "a" is not recognized in block 2 */
```

Note that now when you run the code above you will get an error, variable `a` not recognized in block2. This is because we have defined the variable `a` using the keyword `let`, which limits the scope of variable a to the code block within which it was defined.

Function Scope

In Javascript you limit the scope of a variable by defining it within a function. This is known as **function scope**.

The keyword `var` is function scoped i.e. it does not recognize curly brackets i.e. `{}` , as delimiters. Instead it recognizes the function body as the delimiter.

If we want to define a variable using `var` , and prevent it from being defined in the global namespace you can re-write it by enclosing the code blocks within functions.

```
function block1() {
var a=10;
 console.log(a);
} //function scope of block 1

function block2() {
  a++;
  console.log(a);
} //function scope of block 2

/* Since we have enclosed block1 and block2, within separate
functions, the scope of "var a=10", is limited to block 1 and "a" is
not recognized in block 2 */
```

The above code is in effect the same as if we were using `let a=10` instead of `var a=10` . The scope of the variable `a` is limited to the function within which it is defined, and `a` is no longer in the global namespace.

Why would you chose "let" over "var"?

While programming in Javascript it is a good practice **not** to define variables as global variables. This is because it is possible to inadvertently modify the global variable from anywhere within the Javascript code. To prevent this one needs to ensure that the scope of the variables are limited to the code block within which they need to be executed.

In the past before keyword `let` was introduced as part of ES6, to circumvent the issue of variable scoping using `var` , programmers used the IIFE pattern to prevent the pollution of the global name space. However since the introduction of `let` , the IIFE pattern is no longer required, and the scope of the variable defined using `let` is limited to the code block within which it is defined.

For more information: ***What is an IIFE in Javascript?***

The "const" keyword

If a variable is defined using the `const` keyword, its scope is limited to the **block scope**. In addition the variable cannot be reassigned to a different value.

```
{
 const PI=3.14;
 console.log(PI);
} //block 1

{
  console.log(PI);
} //block 2

/* Since we are using "const PI=3.14", scope of "PI" is limited to
block 1 and "PI" is not recognized in block 2 */
```

Note that it is important to understand that `const` does NOT mean that the value is fixed and immutable. This is a common misunderstanding amongst many Javascript developers, and they incorrectly mentioned that a value defined by the `const` keyword is immutable (i.e. it cannot be changed).

In the following example we can show that the value of the variable defined within the `const` keyword is mutable, i.e. it can be changed.

```
{
 const a = [1,2,3];
 const b = {name: "hello"};

 a.push(4,5);     //mutating the value of constant "a"
 b.name="World";  //mutating the value of constant "b"

 console.log(a); //this will show [1,2,3,4,5]
 console.log(b); //this will show {name: "World"}

}

/* This code will run without any errors, and shows that we CAN
mutate the values that are defined by "const" */
```

However note that these variables defined by `const` cannot be re-assigned.

```
{
 const name = "Mike";
 const PI = 3.14;

 const a = [1,2,3];
 const b = {name: "hello"};


 name="Joe";
//this will throw an error, since we are attempting to re-assign
"name" to a different value.

 PI = PI + 1;
//this will throw an error, since we are attempting to re-assign PI
to a different value.

 a = [1,2,3,4,5];
//this will throw an error, since we are attempting to re-assign
"a" to a different value.

 b = {name: "hello"};
//this will throw an error, since we are attempting to re-assign
"b" to a different value.

}
```

Summary

> var is functional scope
>
> let, const are both BLOCK scope
>
> const, cannot be reassigned, however it is mutable

. . .

Found this useful? Hit the 👏 button to show how much you liked it! □

Follow me on Medium for the latest updates and posts!

Read Next: Promises in Javascript Explained!

Other Articles:

- Javascript — Closures Simplified!

JavaScript Javascript Tips Programming Codingbootcamp Coding