Let your JavaScript variables be constant

Jakub Janczyk Follow
Nov 27, 2017 · 10 min read

Some time ago, you could read about our top 10 ES6 features (if you haven't yet, check it out here). There is no doubt that many fantastic things were introduced in this (and all subsequent) versions of JavaScript. As you might be familiar with the language overall, do you know how it actually works inside? I feel more secure when I know what's going

on behind the scenes. Hence, I would like to guide you through a more in-depth, yet simple, explanation of ES6 variables — number one on the list from our previous article: `let` and `const`. Let's start with some basics.

Here comes `let` and `const`

There are two new ways of declaring variables that were introduced in ES6. We can still use the well-known `var` keyword (you shouldn't, however, and keep reading to learn why), but now we have two more powerful tools to use: `let` and `const`

Let

`let` is quite similar to `var`, in terms of usage. You can declare variables in exactly the same way, for example:

```
1  let myNewVariable = 2;
2  var myOldVariable = 3;
3
4  console.log(myNewVariable); // 2
5  console.log(myOldVariable); // 3
```

example1.js hosted with ♡ by GitHub view raw

But there are, in fact, multiple significant differences between them; differences that are not just cosmetic changes, but actually make your work easier, and can prevent a lot of strange mistakes and bugs. Amongst those differences are:

- `let` is block scoped (I will talk a bit more about scopes later in the article), while `var` is function scoped.
- `let` cannot be accessed before declaration (`var` can be, and is actually the source of many bugs and confusion in the JS world).
- `let` cannot be redeclared.

Before we get into details about any of those differences, let's first take a look at one more cool thing: `const` variables.

Const

`const` is mostly the same as `let` (so has the same features, compared to `var`), but with one major difference. While `let` can be reassigned, `const` cannot — so it is a variable that always has one, and only one, value that must be assigned at the time of declaration, not later in any case. Let's look at this example:

```
const myConstVariable = 2;
let myLetVariable = 3;

console.log(myConstVariable); // 2

myLetVariable = 4;  // ok

myConstVariable = 5;  //wrong - TypeError thrown
```

example1.js hosted with ♡ by GitHub view raw

Const === immutability?

But there is one gotcha with `const`. While variables cannot be reassigned, it doesn't make them truly immutable. If the `const` variable has an array or object as its value, then you may change the content as much as you like:

```
const myConstObject = {mutableProperty: 2};

// myConstObject = {}; - TypeError

myConstObject.mutableProperty = 3; //ok
console.log(myConstObject.mutableProperty); // 3

const myConstArray = [1];

// myConstArray = []; - TypeError

myConstArray.push(2) //ok
console.log(myConstArray); // [1, 2]
```

example1.js hosted with ♡ by GitHub view raw

This of course doesn't apply to primitives like string, number, boolean, etc., as they are immutable by nature.

Immutability, here we come

If you want our variables to be truly immutable, feel free to use tools like `Object.freeze()`, which would make the object immutable (or array — in fact array IS the object in JS). Unfortunately, it would only be shallow immutability, so if you have nested objects, they can still be modified:

```
const myConstNestedObject = {
  immutableObject: {
    mutableProperty: 1
  }
};

Object.freeze(myConstNestedObject);

myConstNestedObject.immutableObject = 10; // won't change
console.log(myConstNestedObject.immutableObject); // {mutableProperty: 1}

myConstNestedObject.immutableObject.mutableProperty = 10; // ok
console.log(myConstNestedObject.immutableObject.mutableProperty); // 10
```

example.js hosted with ♡ by GitHub view raw

The final option for immutability is to still use `Object.freeze()` and build some kind of utility that goes as deep as required in your object, and make every sub-object immutable. Or you could use pre-existing libraries that are doing this job for you. I like this one.

Scope of variables

Having introduced the basics, let's jump into some more advanced topics. We are going to start with the first difference between ES5 and ES6 variables I mentioned earlier — scoping.

Note: all the following examples are presented with `let` variables, but the same rules apply to `const`.

Global and function scoped variables

But what does scope actually mean in JS? This article doesn't offer a complete explanation of scopes, so to keep things simple, the scope of the variable determines

where your variables are available. Taken from a different perspective, one can say that scope defines which variables (or functions) you can use in the specific area of code. Scope can be global (so things declared within a global scope are accessible everywhere in your code) or local. Obviously, locally scoped variables are accessible only within some boundaries. Prior to ES6, there was only one thing that allowed us to define local scope — function. Let's look at the following example:

```
// global scope
var globalVariable = 10;

function functionWithVariable() {
  // local scope
  var localVariable = 5;
  console.log(globalVariable);  // 10
  console.log(localVariable);   // 5
}

functionWithVariable();

//global scope again
console.log(globalVariable);  // 10
console.log(localVariable);   // undefined
```

example.js hosted with ♡ by GitHub view raw

Here, the variable `globalVariable` has global scope, so can be easily accessed within our function or in any other area of our code. But the variable `localVariable`, that is defined within a function, is only visible inside it.

So, whatever is created inside a function can be accessed there and within all nested functions (it might even be nested multiple times). This is possible thanks to closures, but this is a topic I won't be covering in this article. Stay tuned, however, as we'll certainly cover it in more detail in a future blog post!

Hoisting

Now, there are a few drawbacks related to having function scope. Before I get into details, let me introduce one more definition: hoisting. Again, simply speaking, this is the JavaScript mechanism that "moves" all variables (and functions) declarations to the top of our scope. Look at the following code:

```
function func() {
  console.log(localVariable);   // undefined

  var localVariable = 5;

  console.log(localVariable);   // 5
}

func();
```

example.js hosted with ♡ by GitHub view raw

Why does it even work? We haven't declared our variable yet, but still console.log prints undefined? Why not an error that our variable was not defined yet? Let's take a closer look.

Compiled variables

The JavaScript interpreter is going through the code twice to be able to run it. The first stage is called compilation and this is where hoisting is performed. After it, our code becomes something like the following (I've made some simplifications just to show what is relevant):

```
function func() {
  var localVariable = undefined;

  console.log(localVariable); // undefined

  localVariable = 5;

  console.log(localVariable); // 5
}

func();
```

example.js hosted with ♡ by GitHub view raw

What you can see here is that our variable `localVariable` was moved to the top of our function scope defined by function `func`. Strictly speaking, the declaration of our variable was moved, but not its definition. That's why we can use the variable and print

it. It is undefined because we haven't defined it yet, but it is already available for us to use.

Hoisting example — what can go wrong

Now we move to a nasty example where our function scope can do us more harm than good. It's not that the function scope is bad. We just have to be aware of some pitfalls that it introduces (along with hoisting), and here is one of them. Take a look at the following code:

```
var callbacks = [];
for (var i = 0; i < 4; i++) {
  callbacks.push(() => console.log(i));
}

callbacks[0]();
callbacks[1]();
callbacks[2]();
callbacks[3]();
```

example.js hosted with ♡ by GitHub view raw

What do you think would be printed out to the console? Well, your first guess would probably be 0 1 2 3, right? If so, I have a little surprise for you. The actual result is: 4 4 4 4. Wait, what? How did that happen? Let's "compile" this code and we will see. Our code now looks like this:

```
var callbacks;
var i;

callbacks = [];
for (i = 0; i < 4; i++) {
  callbacks.push(() => console.log(i));
}

callbacks[0]();
callbacks[1]();
callbacks[2]();
callbacks[3]();
```

example.js hosted with ♡ by GitHub view raw

Do you see the problem? There is only one variable `i` in the whole scope! And it wouldn't be re-declared, just its value would change in each loop iteration. Then, when we want to print it out by calling the function later, there would be just one value — the one that was assigned in the last loop iteration.

Are we doomed then? No.

Let and Const to the rescue

Along with new ways of declaring variables, a new scope was introduced: block scope. Block is just a set of curly braces and everything within it. So it can be an `if`, `while` or `for` statement with braces, standalone braces or even a function (Yes, the function scope is the block scope). `let` and `const` are block scoped. It means that whenever you define any variable inside the block, it would not be available outside of it. Let's take a look at this example:

```
function func() {
  // function scope
  let localVariable = 5;
  var oldLocalVariable = 5;

  if (true) {
    // block scope
    let nestedLocalVariable = 6;
    var oldNestedLocalVariable = 6;

    console.log(nestedLocalVariable); // 6
    console.log(oldNestedLocalVariable); // 6
  }

  // those are stil valid
  console.log(localVariable); // 5
  console.log(oldLocalVariable); // 5

  // and this one as well
  console.log(oldNestedLocalVariable); // 6

  // but this on isn't
  console.log(nestedLocalVariable); // ReferenceError: nestedLocalVariable is not defined
}
```

example.js hosted with ♡ by GitHub view raw

Can you spot the difference? Can you see how `let` can solve the problem I presented a few paragraphs earlier? So, our `for` loop is containing a set of curly braces, therefore it defines the block scope! Now if we were to use `let` or `const` instead of `var` when declaring our loop variable, that code would roughly translate to something like the following. Note: I've made quite a big simplification here, but I'm sure you get the point :)

```
let callbacks = [];
for (; i < 4; i++) {
  let i = 0 //, 1, 2, 3
  callbacks.push(() => console.log(i));
}

callbacks[0]();
callbacks[1]();
callbacks[2]();
callbacks[3]();
```

example.js hosted with ♡ by GitHub view raw

Each loop iteration would now have its own variable declared, so there would be no overwriting, and we are sure that this code does whatever we intended it to do.

That was quite an example to conclude this part, but let's take a look at one more. I'm sure you know why the given values are printed and what is responsible for this behavior:

```
function func() {
  var functionScopedVariable = 10;
  let blockScopedVariable = 10;

  console.log(functionScopedVariable);  // 10
  console.log(blockScopedVariable);  // 10

  if (true) {
    var functionScopedVariable = 5;
    let blockScopedVariable = 5;

    console.log(functionScopedVariable);  // 5
```

```
    console.log(blockScopedVariable);  // 5
  }

  console.log(functionScopedVariable);  // 5
  console.log(blockScopedVariable);  // 10
}

func();
```

example.js hosted with ♡ by GitHub view raw

Using variables before declaration? Not anymore!

Another thing that changed with the new way of declaring variables is that now they cannot be used before declaration. We showed that this code works:

```
function func() {
  console.log(localVariable);   // undefined

  var localVariable = 5;

  console.log(localVariable);   // 5
}

func();
```

example.js hosted with ♡ by GitHub view raw

But this one is not:

```
function func() {
  console.log(localVariable); // ReferenceError: localVariable is not defined

  let localVariable = 10;

  console.log(localVariable); // 10
}

func();
```

example.js hosted with ♡ by GitHub view raw

Wait, let's take a few steps back. A few paragraphs ago, I said that there is a mysterious thing called hoisting, that moves all variables declarations to the top of the given scope. Does it mean that if I cannot use my variable before its actual declaration in code, then there is no more hoisting? The answer is no. Hoisting is still relevant, and applies to all types of variables, even new ones. But one thing actually changed with `const` and `let`.

First, let's take a look at how variables with `var` keywords work in ES5 (and still do if you are using them in ES6+!). This is what the specification says about it:

> *A* `var` *statement declares variables that are scoped to the running execution context's VariableEnvironment. Var variables are created when their containing Lexical Environment is instantiated and are initialized to* `undefined` *when created. […] A variable defined by a VariableDeclaration with an Initializer is assigned the value of its Initializer's AssignmentExpression when the VariableDeclaration is executed, not when the variable is created.*

That's a lot of formality, so let me simplify this a bit:

1. When you enter a given scope, all variables defined within it are created.
2. All those variables exist, are accessible and have `undefined` assigned as a value.
3. When initialization is reached in code (its execution), actual values are assigned.

And here is how new variables are behaving. Again, from the specification:

> `let` *and* `const` *declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable's LexicalBinding is evaluated. A variable defined by a LexicalBinding with an Initializer is assigned the value of its Initializer's AssignmentExpression when the LexicalBinding is evaluated, not when the variable is created. If a LexicalBinding in a* `let` *declaration does not have an Initializer the variable is assigned the value undefined when the LexicalBinding is evaluated.*

Again, to simplify:

1. When you enter a given scope, all variables defined within it are created — similarly to `var`.
2. Here is the difference: all those variables exist, just like `var`, but they are NOT accessible yet (and have no value, even `undefined`).
3. Only when initialization is reached in code (its execution), can they be defined, therefore become accessible.
4. If the `let` variable is declared and initialized in the same place, the proper value is assigned, otherwise the variable is `undefined`. The `const` variable must always be initialized in the same place as it is declared.

Let's now look at some examples of how it works.

Temporal Dead Zone

In fact, this description leads us to one more definition. That one's rather scary, as it is called: **Temporal Dead Zone (TDZ)**. Don't worry, nobody is killing anyone! This term lets us specify the area of code that doesn't have access to our variables. Let's take a look at the following code and associated comments to explain simply what the TDZ is:

```
function func() {
  // Start of TDZ for deadVariable

  // we can still do something here, just our deadVariable is not available yet
  const exampleVariable = 5;
  console.log(exampleVariable); // 5

  // End of TDZ for deadVariable
  let deadVariable = 10;

  console.log(deadVariable);  // 10
}

func();
```

example.js hosted with ♡ by GitHub view raw

See, dead simple — pun intended :) One thing is worth mentioning though. As the name suggests, this is a TEMPORAL dead zone, meaning that this zone is defined by time, not by placement. So you cannot access the variable before its declaration is processed by the JS compiler when running your code. It actually doesn't matter where you put your variable's usage, as long as it is accessed AFTER the declaration is executed. That's why the following code works just fine, even though we theoretically used our variable `deadOrAlive` before declaring it:

```
function func() {
   return deadOrAlive;
}

let deadOrAlive = 'alive!'

console.log(func());  // alive!
```

example.js hosted with ♡ by GitHub view raw

Here are the steps of executing this code:

1. Function is being declared.
2. Our variable is being declared and initialized with value `alive!`.
3. We are now calling our function.
4. Our variable is accessed and prints correct value `alive!`.

And here is a similar example that doesn't work. Can you see why? :)

```
function func() {
   return deadOrAlive;
}

console.log(func());  // ReferenceError: deadOrAlive is not defined

let deadOrAlive = 'dead!'
```

example.js hosted with ♡ by GitHub view raw

So TDZ is a great new mechanism that might prevent a lot of bugs from happening. And we don't really need to do a thing. Just remember to never use variables before their declaration. Even if we do, we would get a nice error explaining the mistake. Only one condition — you must use `const` or `let` instead of `var`!

Double declaration

One last thing that changed along with the new variables — they can be declared only once in a given scope. With `var` you could do the following and everything would work fine:

```
var doubledVariable = 5;
var doubledVariable = 6;

console.log(doubledVariable); // 6
```

example.js hosted with ♡ by GitHub view raw

But now, when you try this with `const` or `let`, you get an error:

```
let doubledVariable = 5;
let doubledVariable = 6;  // SyntaxError: Identifier 'doubledVariable' has already been declared
```

example.js hosted with ♡ by GitHub view raw

On the other hand, variables with the same name can still be declared in nested block scopes:

```
let doubledVariable = 5;

if (true) {
  let doubledVariable = 6;
  console.log(doubledVariable); // 6
}

console.log(doubledVariable); // 5
```

example.js hosted with ♡ by GitHub view raw

This is another great thing that can prevent a lot of bugs. Did it ever happen to you that multiple `var`s with the same name were declared and the value from one overwrote the other? If so, there is a simple, out-of-the-box solution.

Summary

To summarize all this, there are two new ways of declaring variables in ES6: via `const` or `let` keywords. Amongst other things, both are block scoped and cannot be accessed before the declaration is reached. They are major upgrades compared to previous `var` keywords, and can save you a lot of trouble. I presented just a few examples that might save you some hard debugging time, but there are many more. Simply search the web if you're interested :) I personally stopped using `var` keywords long ago, and now my code is filled with `let` and `const` keywords. I recommend the same for you. As far as possible, try using `const` and `let` when you need to change the value of the variables later in your code. **No more `var`!**

Happy coding, and stay tuned for more awesome stuff on our blog!

Top 10 ES6 features by example

In this article, I will try to introduce the most useful features of ES6 in a succinct way.

blog.pragmatists.com

JavaScript Frontend Front End Development Programming