Chapter 9 of AR’s book Exploring ES6

let and const behave more strictly and throw more exceptions (e.g. when you access their variables inside their scope before they are declared). Block-scoping helps with keeping the effects of code fragments more local (see the next section for a demonstration). And it’s more mainstream than function-scoping, which eases moving between JavaScript and other programming languages.

Because behavior differs between var and let, you can’t blindly replace var with let or const in existing code; you have to be careful during refactoring

Standard seems to be to avoid var, preferably use const and, when needed, let

let works similarly to var, but the variable it declares is block-scoped

const works like let, but the variable you declare must be immediately initialized and the value cannot be changed afterwards – values are read-only

it has to do with the binding of consts in memory

in for-of loops, each iteration creates a new storage space for a const, you can const declare a loop variable

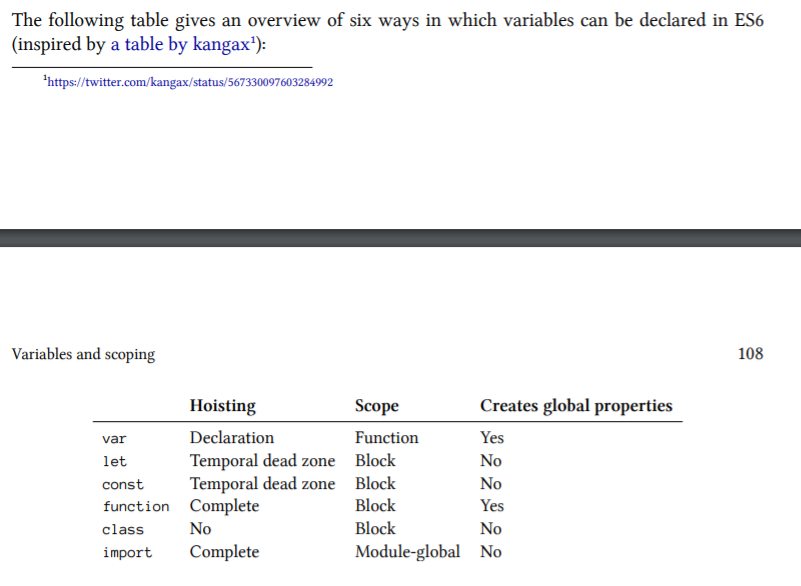
for (const x of ['a', 'b']) {

console.log(x);

}

The lack of hoisting for let and const causes temporal dead zone

In ECMAScript 6, accessing a let or const variable before its declaration (within its scope) causes a ReferenceError . The time span when that happens, between the creation of a variable's binding and its declaration, is called the temporal dead zone.



Lets allow for mutation

Consts are immutable to an extent

You cannot overwrite the very base of the const

You can’t do something like const someConst = {}; someConst=’abc’;

That will throw a TypeError because of the strict binding that is produced for const-declared variables

However, you can do const someConst = {}; someConst.prop=123

Const only means that a variable always has the same value – it doesn’t mean that the value itself is or becomes immutable

Temporal Dead Zones

When entering the scope of a let or a const, it can’t be accessed (got or set) until execution reaches the declaration

For vars, as soon as the scope for a var variable is entered, storage space is created for it, and the variable is immediately initialized, by setting it to undefined

Only when the execution within the scope reaches the declaration, the variable is set to the value specified by the initializer

For lets, when the scope of a let (its surrounding block) is entered, storage space is created for it

However, it remains uninitialized – not even undefined is set as its value

this difference generates the temporal dead zone

when the execution within the scope reaches the declaration, the variable is set to the value specified by the initializer – if there is one – if there isn’t then the value of the variable is set to undefined

so the difference comes down to when the variable is initialized and the scoping (block versus function)

Why is there a temporal dead zone

There are several reasons why const and let have temporal dead zones

Catch programming errors - being able to access a variable before its declaration is strange

Making const work properly is difficult – TDZs provide a rational semantics for const.

Future proofing for guards – a mechanism for enforcing at runtime that a variable has the correct value, hoisting generating a value of undefined may be in conflict with the guarantee given by its guard

Var vs. let and const in loops

If you var-declare a variable in the head of a for loop, it creates a single binding for that variable

Every i in the bodies of the three arrow functions refers to the same binding, which is why they all return the same value

If you let-declare a variable in the head of a for loop, it creates a binding per iteration

Let/const declaration also provide good error catching

For instance, when var would do nothing when you name a variable the same name as a parameter

Parameter default values and the temporal dead zone

Parameters with default values are treated like a sequence of let statements and are subject to temporal dead zones

function(x=1, y=x) {}//this is okay because x has been initialized and declared by the time that y tries to access it

function(x=y, y=2){}//this isn’t okay because y hasn’t been declared yet

parameter default values don’t see the scope of the body

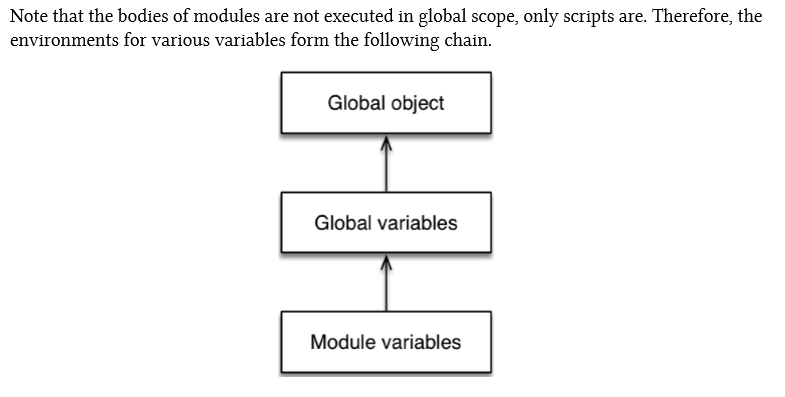
the scope of parameter default values is separate from the scope of the body

scope of parameter default values wraps the scope of a function body

global object is more of a bug than a feature

var and function declarations create global variables that are part of the global object

let, const and class declarations create global variables that are not properties of the global object



Functions

are block-scoped, like let

Create properties in the global object (while in global scope) like var

Are hoisted – created at the beginning of the scope

Classes

Are block-scoped

Don’t create properties on the global object

Are not hoisted – this is because the values of their extends clauses are defined via expressions and those expressions have to be executed at the appropriate times

Don’t be lazy with variable declaration, if you want something to be a part of the window object, specify that explicitly so that your code is easily understood