Shadowing

What is Shadowing:

when a variable is declared in a certain scope having the same name defined on its outer scope and when we call the variable from the inner scope, the value assigned to the variable in the inner scope is the value that will be stored in the variable in the memory space. This is known as **Shadowing or Variable Shadowing**. In JavaScript, the introduction of *let*and*const* in ECMAScript 6 along with block scoping allows variable shadowing. Diagram

Description automatically generated

**Scope:**

In JS there are two types of scope:

* Local scope
* Global scope

Graphical user interface, text, application

Description automatically generated

**Variables which are declared inside a function can’t be used outside of the function. Sometimes system may return some garbage, but the exact defined value can’t be obtained.**

-Variables defined **outside** a function are called **global variables**.  
-Variables defined **within** a function are **local variables**.

Example (Shadowing):

Text

Description automatically generated

Shadowing Using “ Var”:

Text

Description automatically generated

When greetings is referred inside the first print statement, due to \*shadowing\* the JavaScript interpreter used the inner declaration of the greetings variable, not the global one. But since the inner declaration of greetings was not initialized yet, it printed its default value undefined.

Issue of let shadowing:

As we can see variables declared with var are confusing and prone to introduce bugs in the program, if not written with at most care. And that is the reason you should never use var to declare variables in your program. Always use let or const to declare variables

Shadowing Using “ let”:

Text

Description automatically generated

As we see the output is more predictable with the let keyword. The first time the greetings is referred in the print statement, it took the global value because it was not yet declared locally. The JavaScript interpreter declared the second instance of greeting in the local scope inside the if block. It retains the local value Hello Julia inside the if block. But outside the if block in where the variable was declared, the same greetings variable is not visible to the interpreter. Hence it took the global declaration in the third print statement.

Shadowing Using “ Const”:

Graphical user interface, text

Description automatically generated

If we run this code, the JavaScript interpreter will raise an Exception where we try to change the value of const variable greetings.

There is no other difference between let and const

**Deadzone (Hoisting):**

The **Temporal Dead Zone** is a behavior in JavaScript that occurs when declaring a variable with the let and const keywords, but not with var. A shorthand you’ll often hear to describe it is that “Let’s don’t hoist,” but this is not technically true. Read on for a brief description of what’s really occurring.

In JavaScript, variable declarations (but not assignments) are [hoisted to the top of the scope](https://wsvincent.com/javascript-hoisting/). The code below works as expected:

Text

Description automatically generated Text

Description automatically generated

The answer is undefined because the variable declaration is [hoisted](https://wsvincent.com/javascript-hoisting/) to the top of the scope. So in the eyes of the JavaScript interpreter

The JavaScript interpreter works in a two-step process:

* **compile time**: run through all code looking for variable/function declarations
* **runtime**: execute the code including assignments and function invocations

But if you use either the let or const keywords to declare a variable, this same code will throw an error:

Let =>

Text

Description automatically generated

Const =>

Text

Description automatically generated

This is the **Temporal Dead Zone** where we’re trying to access a variable that has not yet been initialized (it has been declared and therefore exists, but has no value, not even undefined). It’s common to hear the phrase, “let/const don’t hoist” as a shorthand to remember this behavior. But technically something else is going on.

The let and const keywords do step 1–so technically they do hoist–but not step 2, the assignment to undefined. Therefore it’s more accurate to say that let/const **do hoist but don’t get initialized.**

The end result is the same. Just remember that when using let/const you should always move variable declarations to the top of your scope to avoid the **temporal dead zone**.