# **Variables and Values - Reading Material**

# Understanding Var

*Lesson 1 of 4*

****Creating Variables****

There are ****two ways to create variables**** in JavaScript. You can either****use the var keyword or the let keyword**** which we’ll see in the next topic.

We’ve been learning about var for quite some time now. However, we’ve never really gone into its workings and specifics.

To understand var, it is important to understand a key concept called - scope.

****Scope****

Simply put, the ****scope is****another name for****context or the limits within which variables and values are visible and accessible****.

When you write your application, it contains thousands of lines of code and this includes variables and values. But not every variable is visible to every part of the code.

That would not be optimal as it gravely increases the chances of arbitrary code overwriting values. Instead, languages offer the concept of scope, a defined context within which your variables are visible and accessible.

This defined level of access ensures that you know how and what has access to variables and their mutation.

Now when JavaScript executes in the browser, a topmost context known as the global or window scope is provided by the browser. This is an object, just like the objects we saw in the last module. But this special object contains a lot of stuff that can be used to control and manipulate the page in the browser.

****Variables created using var are either globally scoped**** such as the ones created outside of functions, ****or they’re function scoped****when created inside functions in which case they’re only visible and available inside the function and any inner functions.

This is a very important point to understand because it affects the availability of variables created using var. We’ll see how this differs from the let keyword which I’ll introduce in the next topic.

And now that you know about the scope, let’s introduce a very important concept - JavaScript is a lexically scoped language.

****Lexical Scoping****

****Lexical scope refers to scope or context that is defined by the position of variable declaration****. We saw how variables defined using var inside a function are only visible inside the function. They’re lexically scoped to the inside of a function.

****Variable Hoisting****

Another key concept here is variable hoisting. When a script is executed, before anything else, all variables in the script are declared and set to undefined. Once this is done, they’re set to their values.

Technically, variables are hoisted to the top and set to undefined, and then later assigned their individual values. This is why if you try to access a variable before it is declared in your code, you get undefined because the variable is there as it was hoisted however the statement which sets the value hasn’t been executed yet.

Note that JavaScript does allow you to create variables without using var or let. These end up in the global namespace NO MATTER where they’re created. So even if you do this inside a function, you’re still writing variables in the global scope.

This is global namespace pollution at its worst. In fact, this is one of the reasons for implementing the use of strict directive because it throws an error and prohibits the creation of such variables.

# Block Scoped Variables With Let

*Lesson 2 of 4*

****Using Var Keyword****

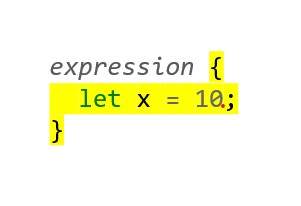
In our previous topics, we used ****var****to declare and instantiate variables. Such variables are either globally scoped if outside of any function or scoped to the function if declared inside. And they’re hoisted to the top of their scope before any other code runs.

****Let Keyword****

JavaScript also offers a more restricted alternative to var. Variables created using ****let**** are neither created in the global namespace nor limited to function scope.

****Block Scope****

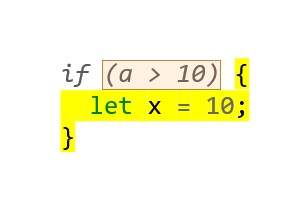
Instead, such variables are limited to the block they’re declared or initialized in. So, let enables you to declare block-scoped variables.



As you can see here, a block is the region within an open and closed pair of curly braces. Many expressions and constructs in JavaScript use such blocks to define a set of statements that must be executed together.

For instance, an if statement as we saw earlier allows you to define a block that will be executed only if the condition inside the parenthesis evaluates to true.

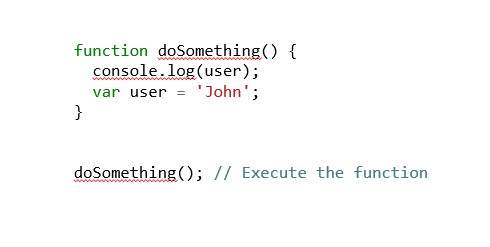
Any variable created inside this block using the let keyword is visible only inside the block and not outside. There is no such restriction in the case of var, which only adheres to the bounds of a function for its scope.



****Let Variables****

Variables created using let cannot be created again in the same scope. So I cannot have two let statements initializing the same variable name. This helps you avoid name collisions in the same scope which might inadvertently occur with var since it has no such limitation.

And by the way, an interesting thing about let is that if you create variables outside of any block, it does not end up in the global scope which is unlike var where top-level variables are part of the global scope.



Can you guess what would be the result of the console.log here?

If you answered ‘undefined’, you’re spot on. Because variables with var are hoisted, they’re first declared and set to undefined and values assigned later. So, when this function runs, the var is created and set to undefined until it is set. This is why the console.log is able to access this variable albeit with the undefined even before it is declared and set.

In the case of let, this doesn’t work. variables created using let are not available until they’re not fully initialized. In technical words, the region that you see marked in red here is known as the temporal dead zone or the area within the block where the let variable isn’t available because unlike var it is not hoisted and initialized later.

Overall, let, is restrictive however is much safer to use and reason about. Its narrow block or expression-based scope ensures that you know precisely where it's visible and where it is not.

From this point onwards in the course, we’ll strive to use let or const wherever possible instead of var. Not because it is better, but because it gives us more control. Having said that, we can, should, and will use var where the scope needs to be as wide as the enclosing function.

Both var and let have their own uses. It has become a common perception that var should be avoided and let is better. However, I would urge you to identify the scope within which your variable needs to work and choose accordingly rather than prejudicial belief.

# Creating Constants With Const

*Lesson 3 of 4*

****Constants****

What does the word ‘constant’ mean to you?

The value of Pi is a constant i.e. it is a known value that isn’t variable. Likewise, the mass of an electron is known and it is a constant. In our applications, we often have to work with data that will not change over time.

For instance, the URL to a server from where your application fetches data will remain a constant.

In such circumstances, you may store such data in a variable, however, variables are susceptible to change and mutation.

Wouldn’t it be great if we could declare a value and expect it to remain stable and consistent without the fear of being mutated?

Well, that is what constants allow you to do in JavaScript.



You create constants much like variables, only that you use the ****const****keyword as seen here.

Constants work exactly like let. They’re block scoped or expression scoped and cannot be re-declared but in addition they cannot be reassigned.

# Type Conversion

*Lesson 4 of 4*

When we work with different data types, JavaScript performs automatic type conversion at times, but there are other instances where type conversion has to be performed.

****Type Conversion****

When we try to add a number to a string using the plus operator, strings take precedence, and we get concatenation.

However, if you try to subtract a number and a string, you’ll notice the opposite happens, the string is treated or converted as a number.

This is an example of automatic type conversion at work.