**A JavaScript Crash Course - Data Types, var, let, and const -** [James Bryce](https://medium.com/@jybryce04?source=post_page-----a8f4322284b2--------------------------------) Jan 6, 2022

Part 2: Learn about the different data types and how to initialize variables

In the first lesson, we didn’t do too much with variables. Actually, we did not do anything particularly interesting. Be patient, friends. In due time you will be writing fully-functional full-stack applications! Today, however, we’re going to start diving deep into data types and learning about the keywords let and const.

For your convenience, here is our course outline:

* [Introduction, Hello World, and Variables](https://medium.com/@jybryce04/a-javascript-crash-course-part-1-introduction-and-hello-world-33fd4b0ecfeb)
* A Brief History of JavaScript (optional, but it provides context)
* High-Level Explanation of How JavaScript Works (stuff you may find interesting)
* Data Types, var, let, and const (This article)
* More About Arrays, Strings, and Objects
* Logical/Arithmetic Operators; Conditionals and Loops
* Functions, Scope, and variable hoisting; reference vs. value
* Exception Handling
* IIFE, Closures, and Callbacks
* ES6 Classes and OOP

We’ll begin today by getting familiar with JavaScript’s data types. Next, we’ll examine the issues with var, we’ll introduce let and const, then we’ll conclude with some remarks on arrays. Let’s get started!

**Primitive Data Types**

Below is a list of all types (aside from [Symbols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol) and [Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects)) in JavaScript.

* Number
* String
* Boolean
* BigInt
* Undefined
* Null

**Number**

Numbers can be either integers or floating-point. More specifically, a *Number* in JavaScript is a [64-bit double-precision value](https://en.wikipedia.org/wiki/Double-precision_floating-point_format). This means that a Number value occupies 8 bytes of memory and can store huge numbers (though the precision goes down after 2⁵³- 1, also known as Number.MAX\_SAFE\_INTEGER, or below -2⁵³ - 1, Number.MIN\_SAFE\_INTEGER). Additionally, a Number can take on the value of -Infinity, +Infinity, or NaN (not a number). Examples of Numbers are:

* 10
* 5.37 and -2.1
* 1023.782
* 5
* 0 and -34

**String**

A String is any text value enclosed in single-quotes or double-quotes. Some examples of Strings are

* “Hello, World!”
* ‘This is a string.’
* “10”
* ‘abcdefg’

We’ll learn a lot about what you can do with strings in the next article.

**Boolean**

The Boolean type can have one of two values: true or false. It represents the result of a yes or no question such as *Is the stack empty?*or *Is x greater than y?*More formally, it is the result of the logical evaluation of a statement as being true or false. We’ll explore this concept more when we talk about conditionals.

**BigInt**

A BigInt can be any integer value bigger than Number.MAX\_SAFE\_INTEGER. A BigInt is its own data type distinct from Number representing arbitrarily precise integer values. For our purposes, though, a major discussion of BigInt would detract from the current course objectives — it is a topic I plan to discuss in future articles.

**Undefined**

A type of undefined literally means the variable has not been initialized. In other words, it is, by definition not defined. You’ll often encounter frustrating errors related to a variable being undefined. Good debugging practices, using let and const, and using [TypeScript](https://www.typescriptlang.org/docs/) can all aid in overcoming errors with undefined values.

**Null**

It may seem like this is the same as undefined, but a null value is an intentional lack of value. Null is usually a placeholder or used in boolean expressions, statements that evaluate to true or false, to determine whether an object exists. In short, null !== undefined (read as null is strictly not equal to undefined). In fact, if you run typeof null you’ll get object back; contrarily, typeof undefined returns undefined.

There are also Symbols, introduced in ES6, but I’ll discuss those later as they are a very *unique*beast and will go well with a future discussion on object properties — we’ll be learning about objects soon! Now let’s take a look at how to initialize variables.

**The Keyword *var***

var is typically not used in contemporary JavaScript code anymore. That being said, you may need to work on code that was written before ES6 at some point in your career. It is worth mentioning for that reason and for completeness in my opinion.

**The Issues with var**

Below are the main problems with varthat sparked the need for let and const to be introduced:

* The scope of a variable declared with var is global when declared outside of functions
* The scope of a variable when declared with var is local to the function it is declared in when inside the aforementioned function.
* A variable declared with var can be re-declared or re-initialized inside the same enclosing scope without errors
* Because of variable hoisting, a variable initialized with var *after*it is used is implicitly declared above its use at runtime, but initialized to undefined.

Text

Description automatically generated

An example of initializing a variable with the same name globally and inside a function

I encourage you to copy the above example and test it out in your environment, but you don’t have to memorize it as it is not useful outside of this context really. The example demonstrates the first two bullet points.

I admit that I have introduced a lot of new syntax since the first lesson, but there is no need for you to worry about that just yet. Just make note of the keywords (in purple), and how I inserted the value ten into the output. You may be coming from a C-like language regardless so the curly-brace syntax with the function might be intuitive to you. You might also notice the syntax in the output that looks similar to f-strings in Python.

Let’s now take a look at the third bullet point.

Text

Description automatically generated

Notice how we have reinitialized twelve to 13 in the if statement

We haven’t covered conditionals yet, but if you have experience with a previous language I am confident you can make sense of the if-statement. However, you might be annoyed with the result of the last console.log.

This example demonstrates the biggest problem var has in my opinion. The problem is stated by the third bullet point, but it is worth reiterating here: *a variable declared with*var*can be re-declared or re-initialized inside the same enclosing scope without errors.*

That’s exactly what is happening in this example (try it out and see for yourself if you don’t believe me!). The if-statement’s scope is enclosed in the global scope. twelve was first initialized with var globally. Then, it was re-initialized with var in the if-statement.

This did not cause any errors, but it allows for the introduction of a bug that can be hard to find in a large codebase. Can you see why I do not like var yet?

We’ll come back to the last bullet point when we get to *Functions, Scope, and variable hoisting; reference vs. value*\*, but at present, I hope I have made it clear as to why we generally avoid var these days.

**Introducing Keywords let and const**

Since var leads to issues with our code, what should we do? Luckily for us, in 2015 [a major revision to the standards and specifications for implementations of JavaScript](https://www.w3schools.com/js/js_es6.asp) was released. Because of this update to the language, known as ES6, which stands for EcmaScript version 6, we can now use the keywords let and const to declare and initialize variables (among many other features).

**Why and When to use**let

With let, you cannot accidentally redeclare the variable as you can with var. Additionally, let has the advantage of having *block scope*. Prior to ES6, JavaScript only had either global or function scope support with var.

As we saw earlier, the variable twelve could be redeclared inside the if block, which would change the value globally since if is just a block, not a function. Both the global and block declarations used the same name for the variable, and both were declared using var. With let, however, you cannot redeclare a variable in the enclosing scope accidentally!

And, if you use let, the engine will spit a ReferenceError at you if you try to use a variable declared with let outside of the block in which you declared it.

I have given you a wall of text, but now let’s take a look at an example!

A screenshot of a computer

Description automatically generated with medium confidence

Screenshot of sample code that demonstrates block scope with let

On line 1, we initialize the variable x with let to the value 20. On line 4, since we’re using let, we actually create a new variable within the if's scope and set it to 31. We then log the value of the x inside the if to the console on line 5. On line 7, we log the value of the global variable x to the console and it is the value it was set to on line 1, unlike what happened with our var example.

Next, we make another if statement on line 10 and initialize a variable y in it with let. Finally, on line 14 we try to output the value of y.

This causes the JS console to get very angry and tell us that we cannot reference something that doesn’t exist. It doesn’t actually get angry. It doesn’t have emotions or sentience. However, it is doing exactly what it was designed to do in this case: it is preventing coding mischief that introduces bugs all thanks to our use of let.

In case you don’t believe me, you can try running the above yourself or take a look at my output when I ran it with Node.js on the command line (image below).

Text

Description automatically generated

Output of the let example in my terminal

You should use let if you plan on updating the value of the variable — otherwise use const.

**Why and When to use const**

As implied above, you should declare a variable const when you don’t plan on changing its value. In fact, a general rule of thumb is to always declare a variable const unless you know for certain the value needs to be updated or reassigned.

Contrary to what you might think, const does not mean the *value*is constant, but rather that *the reference to the value is constant.*Due to this fact, you can’t reassign a const value, a const array, or an object declared const. You can, however, update the elements of a const array or update the properties of a const object.

For me, I always use const when I am defining an array, an object, or a function. In fact, let’s take a look at const in action with arrays before we wrap up this lesson.

**Arrays**

We’ll be learning a lot more about arrays in the next lesson, but I think now is a good time to introduce them.

An array in JavaScript is a variable that can store multiple values. Each value is accessed by an index representing its position in the array. However, JavaScript, like C++, uses zero-based indexing. This means that element 1 of the array is referenced with 0 and therefore the second element is referenced with 1 and so on. Defining a new array uses this syntax:

const arr\_name = [*element0, element1, element2...element(n-1)];*

To access *element0*we would simply type arr\_name[0]. Let’s try a more concrete example!

Text

Description automatically generated

Array of animals

On line 1 we define our array. Remember: we use const for declaring arrays! This is an array of strings, but you can put any type(s) you want in a JavaScript array. We then log the whole array to the console, three of the elements (note their positions vs. index), and then we try to log an element that doesn’t exist.

Below is the output in my terminal window.

Graphical user interface, text

Description automatically generated

Output of the array example

You might be surprised that we don’t get an error trying to reference an element that is not there. This can be a source of bugs and confusion, but also a good thing. For example, a search function that iterates through an array for an element could use a boolean expression like val !== undefined as the condition for continuing to search the array or say that the value it was looking for does not exist.

Before you go, let’s review what we’ve covered today:

* The various fundamental data types: Number, Boolean, String, undefined, and null. Others include BigInt, Symbol, and objects — which we will cover later.
* We discovered the issues with using var
* We met let and const as the solution to the problems var can cause
* We briefly looked ahead at arrays

And that’s the end of part 2. Next time, we’ll look more closely at arrays and strings and introduce objects. I hope you’ll join me again in that lesson as well. Until then, thank you for your time, and may your code be ever clean and free of bugs!