


JavaScript's Memory Model

 medium.com/@ethannam/javascripts-memory-model-7c972cd2c239

March 20, 2019

As programmers, declaring variables, initializing them (or not), and assigning them new values later on is something we do on a daily basis.

But what *actually* happens when do this? How does JavaScript in particular handle such basic functionality internally? And more importantly, how does it benefit us as programmers to understand the underlying minutiae of JavaScript?

Below I aim to cover the following:

1. Variable declarations and assignments for JS primitives
2. JavaScript's memory model: the call stack and the heap
3. Variable declarations and assignments for JS non-primitives
4. Let vs. const

Variable declarations and assignments for JS primitives

Let's start off with a simple example. Below, we declare a variable called `myNumber` and initialize it with a value of 23.

```
let myNumber = 23
```

When this code is executed, JS will...

1. Create a unique identifier for your variable ("myNumber").
2. Allocate an address in memory (will be assigned at runtime).
3. Store a value at the address allocated (23).

Identifier

Memory

`myNumber`



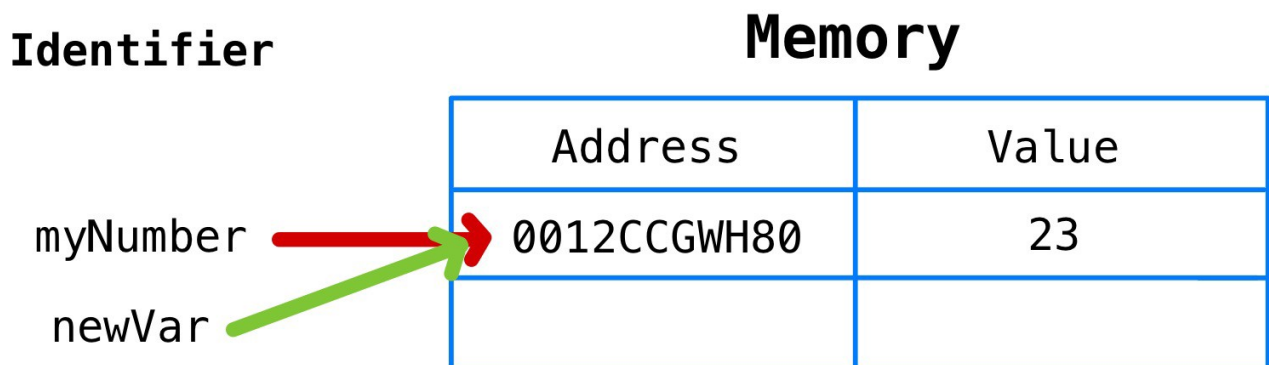
Address	Value
0012CCGWH80	23

While we would colloquially say, “myNumber equals 23”, more technically, myNumber equals the memory address that holds the value 23. This is a crucial distinction to understand.

If we were to create a new variable called “newVar” and assign it “myNumber”...

```
let newVar = myNumber
```

... since myNumber technically equals “0012CCGWH80”, newVar would also equal “0012CCGWH80”, which is the memory address that holds the value 23. Ultimately, this has the intended effect of colloquially saying, “newVar is now equal to 23.”



Since myNumber equals the memory address “0012CCGWH80”, assigning it to newVar assigns “0012CCGWH80” to newVar.

Now, what happens if I do this:

```
myNumber = myNumber + 1
```

“myNumber” will surely have the value of 24. But will newVar also have the value of 24 since they point to the same memory address?

The answer is no. Since primitive data types in JS are immutable, when “myNumber + 1” resolves to “24”, JS will allocate a new address in memory, store 24 as its value, and “myNumber” will point to the new address.

Identifier

Memory

myNumber

newVar

Address	Value
0012CCGWH80	23
0034AAAAH23	24

Here's another example:

```
let myString = 'abc'  
myString = myString + 'd'
```

While a novice JS programmer may say that the letter 'd' is simply appended to the string 'abc' wherever it exists in memory, this is technically false. When 'abc' is concatenated with 'd', since strings are also primitive data types in JS, a new memory address is allocated, 'abcd' is stored there, and "myString" points to this new memory address.

Identifier

Memory

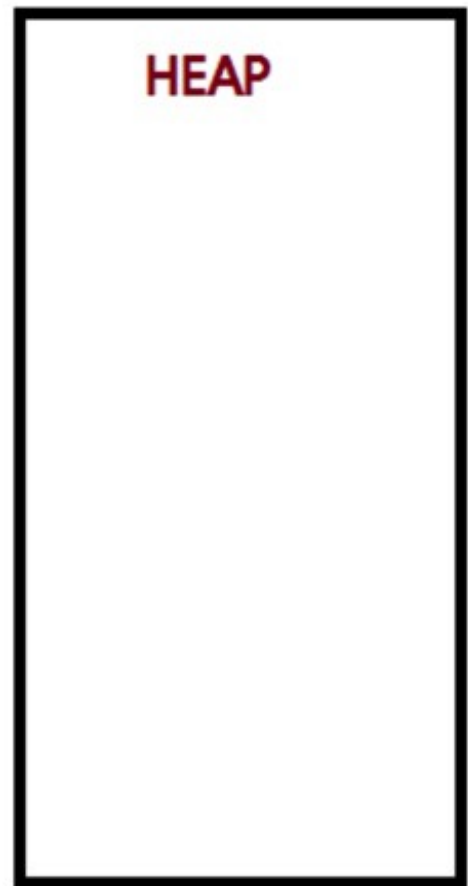
myString

Address	Value
0045FFCBI89	abc
0276GGHBC00	abcd

The next step is to understand where this memory allocation is happening for primitives.

JavaScript's memory model: the call stack and the heap

For the purposes of this blog, the JS memory model can be understood as having two distinct areas: the call stack and the heap.



The call stack is where primitives are stored (in addition to function calls). A rough representation of the call stack after declaring the variables in the previous section is below.



In the illustrations that follow, I've abstracted away the memory addresses to show the values of each variable. However, don't forget that in actuality the variable points to a memory address, which then holds a value. This will be key in understanding the section on `let` vs. `const`.

Now, the heap.

The heap is where non-primitives are stored. The key difference is that the heap can store unordered data that can grow dynamically—perfect for arrays and objects.

Variable declarations and assignments for JS non-primitives

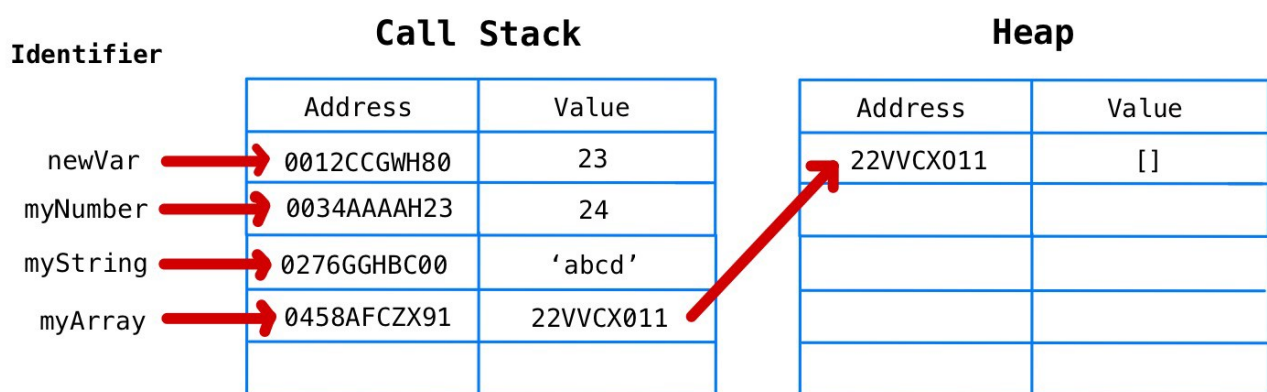
Non-primitive JS data types behave differently compared to primitive JS data types.

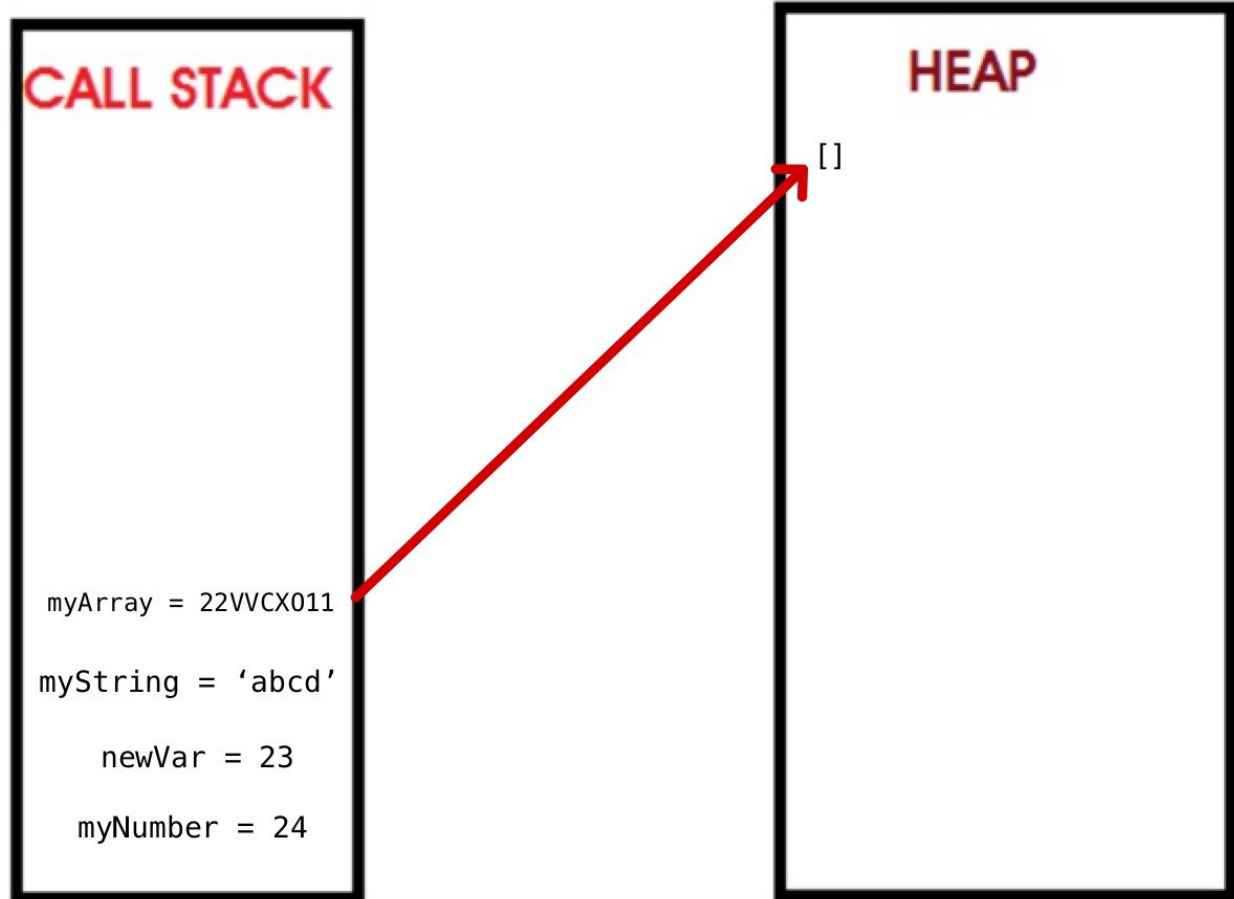
Let's start off with a simple example. Below, we declare a variable called "myArray" and initialize it with an empty array.

```
let myArray = []
```

When you declare a variable "myArray" and assign it a non-primitive data type like "[]", this is what happens in memory:

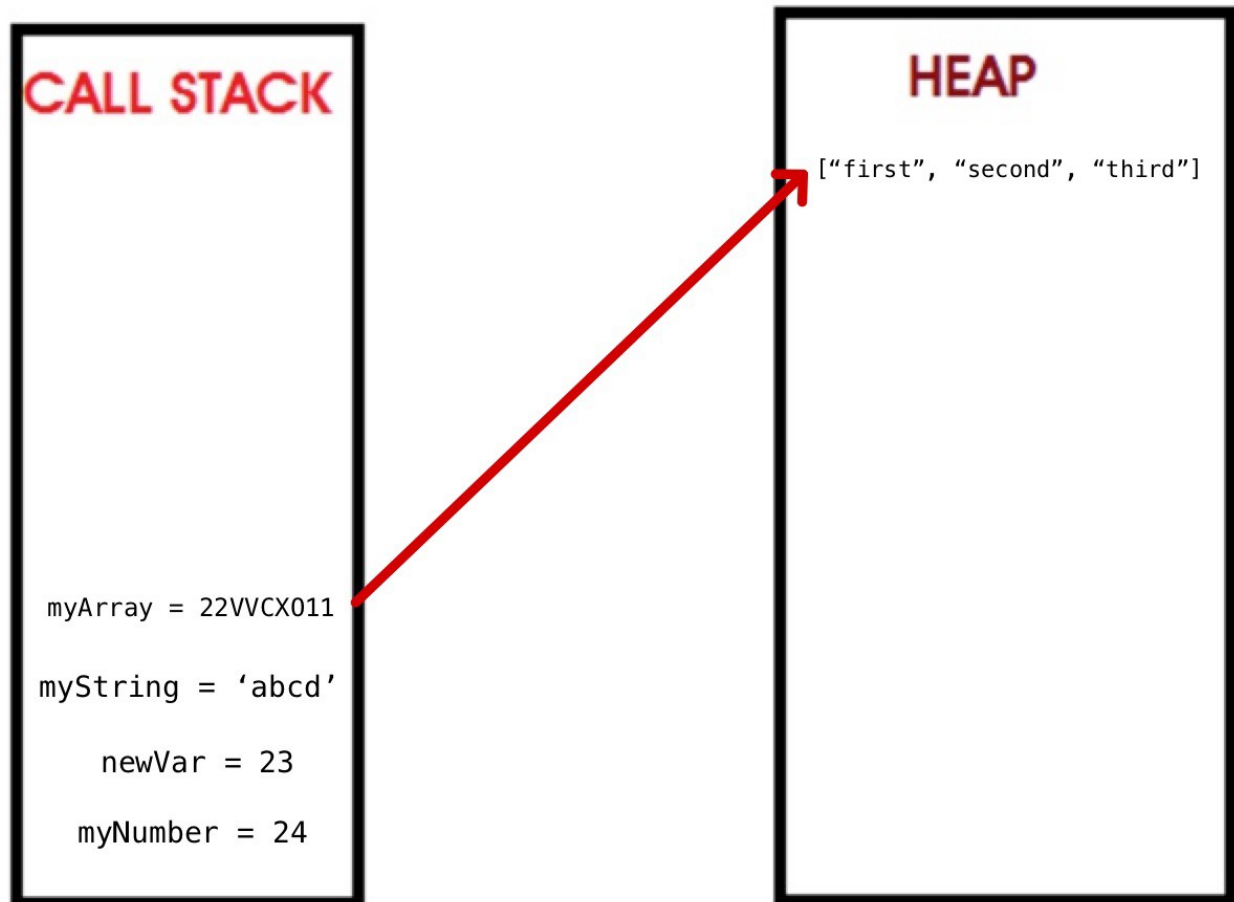
1. Create a unique identifier for your variable ("myArray").
2. Allocate an address in memory (will be assigned at runtime).
3. Store a value of a memory address allocated on the heap (will be assigned at runtime).
4. The memory address on the heap stores the value assigned (the empty array []).





From here, we could push, pop, or do whatever we wanted to to our array.

```
myArray.push("first")  
myArray.push("second")  
myArray.push("third")  
myArray.push("fourth")  
myArray.pop()
```



Let vs. const

In general, we should be using ***const*** as much as possible and only using ***let*** when we know a variable will change.

Let's be really clear about what we mean by "change".

A mistake is to interpret "change" as a change in value. A JS programmer who interprets "change" this way will do something like this:

```
let sum = 0
sum = 1 + 2 + 3 + 4 + 5
```

```
let numbers = []
numbers.push(1)
numbers.push(2)
numbers.push(3)
numbers.push(4)
numbers.push(5)
```

This programmer correctly declared "sum" using ***let***, because they knew

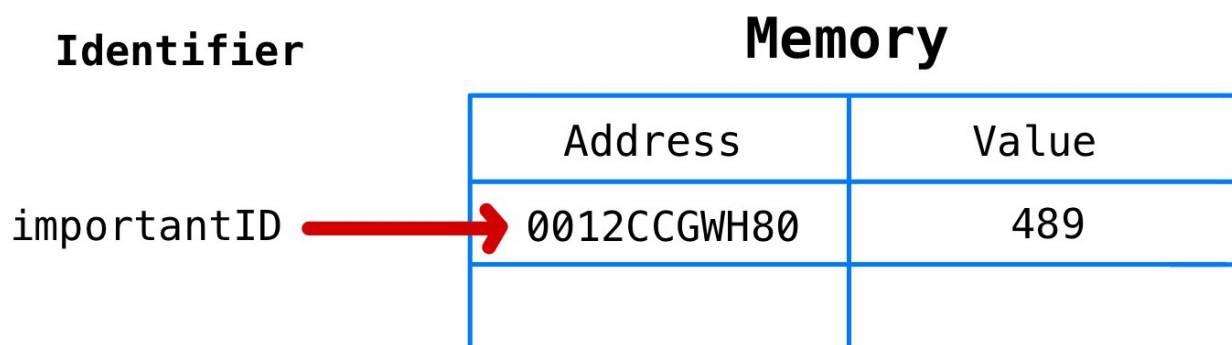
that the value would change. However, this programmer incorrectly declared “numbers” using **let**, because they interpret pushing things onto the array as changing its value.

The correct way to interpret “change” is a change in memory address. *Let* allows you to change memory addresses. *Const* does not allow you to change memory addresses.

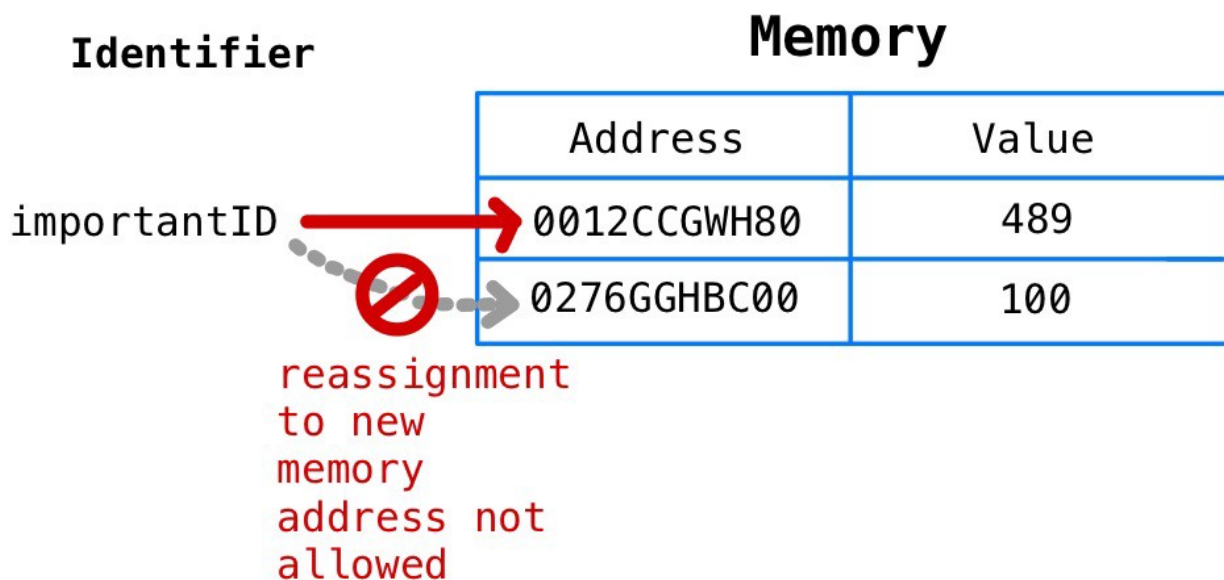
```
const importantID = 489
importantID = 100 // TypeError: Assignment to constant variable
```

Let’s visualize what’s happening here.

When “importantID” is declared, a memory address is allocated, and the value of 489 is stored. Remember to think of the variable “importantID” as equalling the memory address.



When 100 is assigned to “importantID”, since 100 is a primitive, a new memory address is allocated, and the value of 100 is stored there. Then JS tries to assign the new memory address to “importantID”, and this is where the error is thrown. This is the behavior we want, since we don’t want to change the ID of this very important ID...

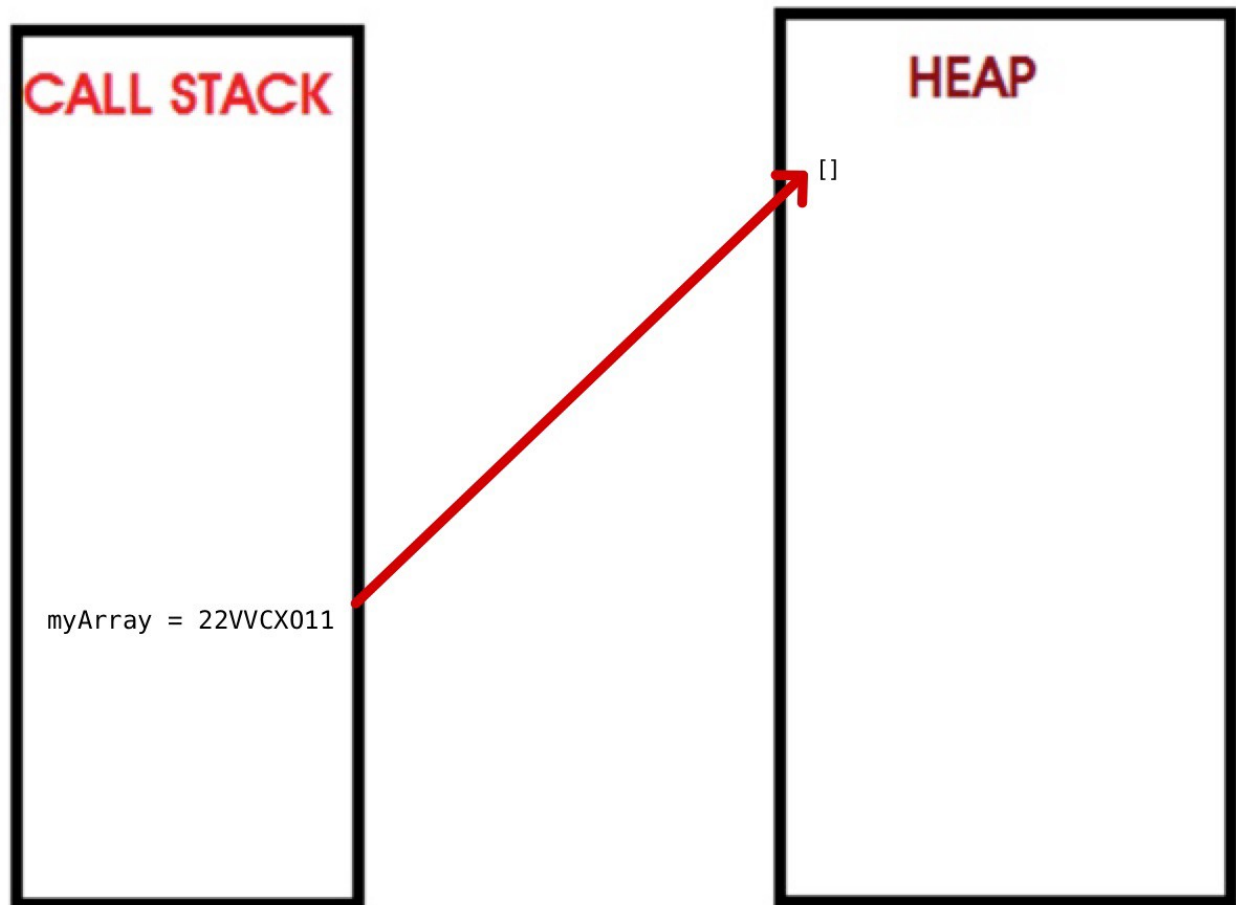
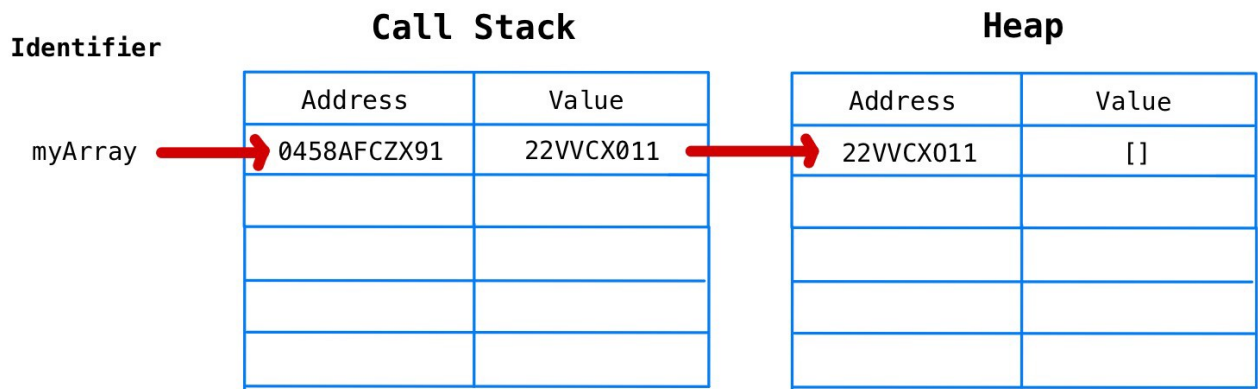


When you assign 100 to `importantID`, you're actually trying to assign the new memory address where 100 is stored. This is not allowed since `importantID` was declared with `const`.

As mentioned above, the hypothetical novice JS programmer incorrectly declared their array using ***let***. Instead, they should have declared it with ***const***. This may seem confusing at the outset. It's not at all intuitive, I admit. A beginner would think that the array is only useful to us if we can change it, and ***const*** makes the array unchangeable, so why use it? However, remember: "change" is defined by the memory address. Let's take a deeper dive on why it's totally okay and preferred to declare the array using `const`.

```
const myArray = []
```

When `myArray` is declared, a memory address is allocated on the call stack, and the value is a memory address that is allocated on the heap. The value stored on the heap is the actual empty array. Visualized, it looks like this:

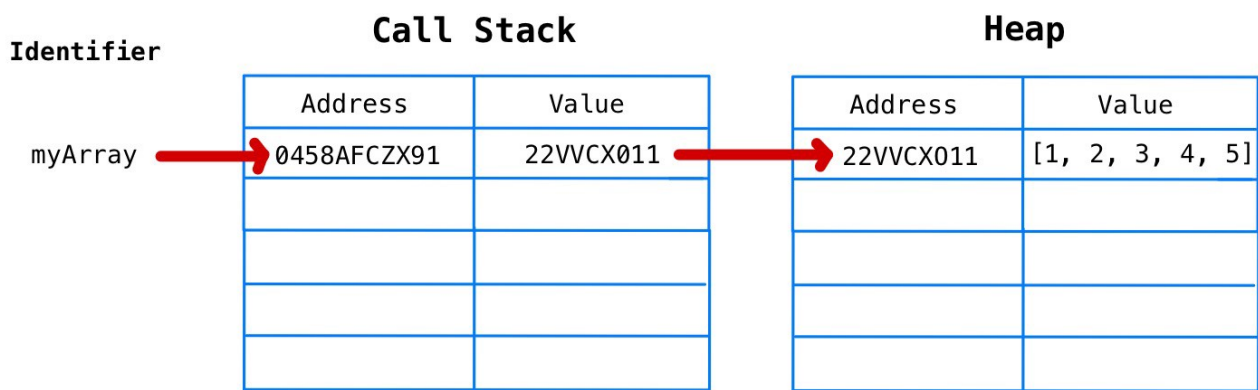


If we were to do this...

```

myArray.push(1)
myArray.push(2)
myArray.push(3)
myArray.push(4)
myArray.push(5)

```

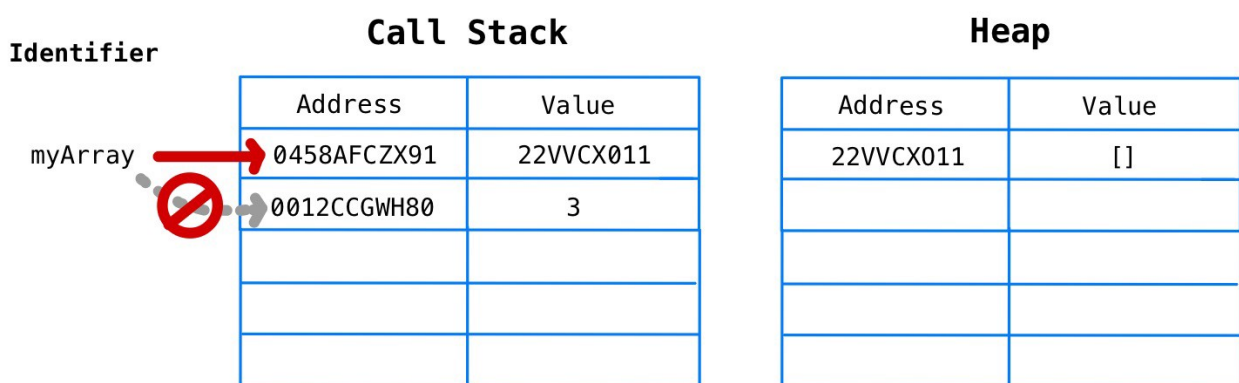


... this pushes numbers onto the array that exists in the heap. However, **the memory address of "myArray" has not changed.** This is why although "myArray" was declared with `const`, no error is thrown. "myArray" still equals "0458AFCZX91", which has a value of another memory address "22VVCX011", which has a value of the array on the heap.

An error would be thrown if we did something like this:

```
myArray = 3
```

Since 3 is a primitive, a memory address on the call stack would be allocated, and a value of 3 would be stored, then we would try to assign the new memory address to `myArray`. But since `myArray` was declared with `const`, this is not allowed.

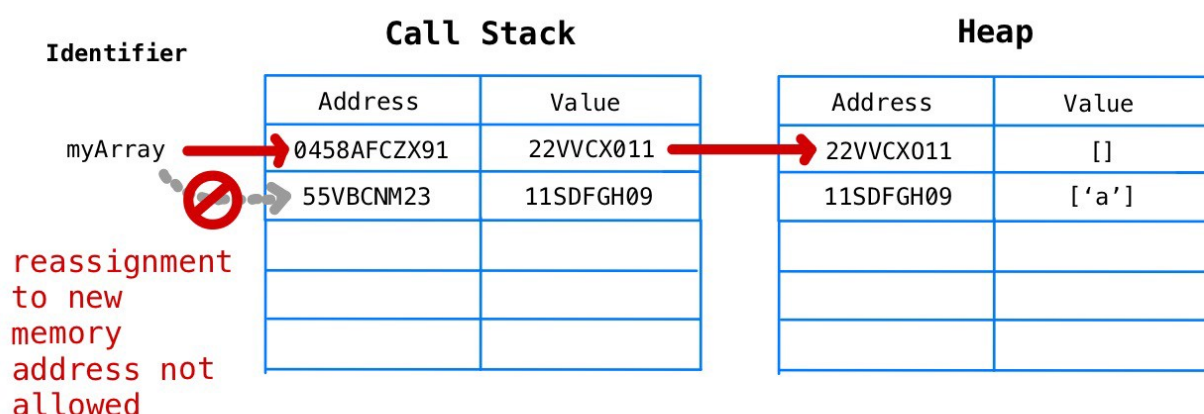


Another example that would throw an error:

```
myArray = ['a']
```

Since `['a']` is a new non-primitive array, a new memory address on the call stack would be allocated, and a value of a memory address on the heap

would be stored, the value stored at the heap memory address would be ['a']. Then we would try to assign the call stack memory address to myArray, and this would throw an error.



For objects declared with **const**, like arrays, since objects are non-primitive, you can add keys, update values, so on and so forth.

```
const myObj = {}  
myObj['newKey'] = 'someValue' // this will not throw an error
```

Why Is This Useful for Us to Know

JavaScript is the #1 programming language in the world (according to [GitHub](#) and [Stack Overflow's Annual Developer Survey](#)). Developing a mastery and becoming a "JS Ninja" is what we all aspire to be. Any decent JS course or book advocates for **const** and **let** over **var**, but they don't necessarily say why. It is unintuitive for beginners why certain **const** variables throw an error upon "changing" its value while others do not. It makes sense to me why these programmers then default to using **let** everywhere to avoid the hassle.

However, this is not recommended. Google, who has some of the best coders in the world, says in their JavaScript style guide, "Declare all local variables with either **const** or **let**. Use **const** by default, unless a variable needs to be reassigned. The **var** keyword must not be used" ([source](#)).

Though they don't explicitly state why, as far as I can tell, there are a few reasons:

1. Preemptively limit future bugs.

2. Variables declared with `const` must be initialized upon declaration, which forces coders to often place them more thoughtfully in terms of scope. This ultimately leads to better memory management and performance.
3. To communicate through your code, to anybody who may come across it, what variables are immutable (as far as JS is concerned) and what variables can be reassigned.

I hope the above explanations help you to start seeing why or when you should ***const*** or ***let*** in your code.

References: