Additional Information

When compared to other languages, JavaScript is a very **abstracted** language. Abstraction is a concept in computer science that allows developers to simplify complex systems by hiding unnecessary details. This means that programmers only need to focus on the essential functions or features of a language. For example, when you make a variable in JavaScript, you don't have to worry about how that variable is being stored in the computer. Even if you wanted to change how that variable is being stored, the language prevents you from doing so. The only thing that you need to focus on is how to use the tools that JavaScript gives you. These heavily-abstracted languages are often also referred to as "high-level". An example of a less-abstracted, lower-level language would be the C programming language. In C, it is almost a **requirement** to manage the memory of the computer, and a lot of the tools within the language interact directly with the computer's processing.

So, for web development, it is not necessary to delve deep into the specifics of what is happening within the computer since these are all abstracted away from us. This means that we will be talking about computer science topics at an abstracted level during this class. However, if you would like to learn more, there will be additional (optional) lessons that describe what is happening on a lower-level. Keep in mind, these are things you absolutely don't need to know for web development, but are still worth looking into if you're curious about computers, or wanting to further your career/education in computer science.

Going off of what we learned today about variables, let's take a lower-level look at what is happening within the computer's memory when we create and modify variables.

Variables in the Background

Memory is a place in the computer that stores instructions and data that a computer needs to reach quickly. Additionally, each location in the computer's memory has an "address". You can think of it just like a house: each house has a different address, and data can live in one of those houses.

When you create a variable, the computer designates a specific 'house' in its memory to the value of that variable. When we create a variable like below, the value 42 is stored **directly** in the computer's memory address associated with 'a'.

```
let a = 42; // 'a' with the value 42 is being stored in a specific house in memory
```

If we copied 'a' into another variable called 'b', a **new** memory address is created associated with 'b', and the value '42' is copied there.

```
let b = a; // We created a new house called 'b' with a new address that has the
value 42
```

Now, there are two addresses in memory: one called 'a' and another called 'b'. Their values are both 42.

What do you think will happen to 'a' if we increase 'b' by 1?

```
b++;
```

ANSWER

- 'b' is now 43, but 'a' is still 42. When you modify 'b', you are modifying the value stored in its own memory address. Since 'b' is a separate copy at a separate address, this has no effect on 'a'.

Arrays and Objects in the Background

Memory storage for objects and arrays work a bit differently than regular primitive-type variables. Objects and Arrays are called `Reference Types`. This means that the variable associated with the object or array doesn't directly hold the object; instead it holds a reference, called a **pointer**, to the actual address of the computer's memory where it's stored.

```
const firstPerson = {
    name: "Luce"
}
```

The variable 'firstPerson' stores a reference (pointer) to the memory address of this object. Inside the actual computer, it would look something like this:

`firstPerson => 0x7ffe5367e044 = name: "Luce"`

(the arrow is the pointer, and 0x7ffe5367e044 is an example of a memory address)

```
const secondPerson = firstPerson;
```

If we were to create a second variable and set it equal to the object we created earlier, a new memory address is NOT created. Instead, they just point to the same address in memory.

`secondPerson => 0x7ffe5367e044 = name: "Luce"`

So, instead of the variable holding the actual value, the variable just holds the memory address. This means that **firstPerson and secondPerson are both pointing to the SAME address.** (the same 'house')

```
secondPerson.name = "Tight";
```

We changed the name of secondPerson to "Tight". What do you think happened to firstPerson's name?

ANSWER

- secondPerson and firstPerson become "Tight". Since they point to the same memory address, you're modifying the same resource regardless of which one you change.

Key Takeaways

Primitive Types (like Numbers, strings)

- Primitive variables store values directly in memory.
- Copying a primitive creates a new memory address.
- Changing one variable doesn't affect the other.

Reference Types (like objects, arrays)

- Objects and arrays store references (pointers) to memory locations.
- Copying a reference creates another pointer to the same memory.
- Changing the data affects all variables pointing to that memory.

Brief note about memory

While JavaScript developers don't directly interact with memory, JavaScript does have an automatic system for memory management. This system automatically allocates new memory to certain addresses when variables are created, and also cleans up any memory that isn't being used. Even though we can't interact with memory, understanding how JavaScript handles memory can help you debug issues, especially when dealing with objects or arrays. For instance, if you changed one object, you might see unexpected changes in another object because they share the same reference.

Additional Resources

In case you'd like to read more about memory management, here is an article from MDN:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_management