**Asynchronous JavaScript - Part 2… -** [Marc Kirk](https://medium.com/@byteslovesbits?source=post_page-----f300a31c24c2--------------------------------) Feb 22, 2022

**JavaScript is Synchronous**

In [part1](https://medium.com/@byteslovesbits/asynchronous-javascript-explained-a6419a391b8b), I briefly mentioned that JavaScript is a synchronous programming language. Generally speaking, JavaScript will execute your code in a line by line fashion, but only after the environment hoists declarations to the top of their current scope. Figure 1 illustrates this concept.

Graphical user interface

Description automatically generated

Figure 1 — Hoisting, moves declared functions to the top of the function’s current scope so that the function can be used before it is declared

***Hoisting Definition:****moving declared functions, variables and classes to the top of their current scope before execution.*

Hoisting is a relatively simple concept. However, to become better JavaScript programmers, we need to understand the inner workings of the environment that executes JavaScript. This is where the magic happens! This is how we ***tame the asynchronous beast!***

**The JavaScript Engine**

***A JavaScript Engine****is software that executes JavaScript code.*

The JavaScript Language is officially defined in the [ecma-262 document](https://www.ecma-international.org/publications-and-standards/standards/ecma-262/). This document or definition, serves to standardize the language. Standardizing the language is a fancy way of saying — we want users with different web-browsers to have the same experience when visiting the same page.

For the user experience to be the same, web-browsers must include a JavaScript engine that complies with the ecma-262 standard. Remember, JavaScript is executed through this engine.

JavaScript engines are commonly baked into web-browsers. For example, Chrome uses the V8 engine and Firefox uses the SpiderMonkey engine. JavaScript engines are not limited to web-browsers. **Node.js** is a back-end JavaScript runtime environment that runs on the V8 engine. We download node.js as an executable which contains everything we need to run JavaScript outside of the browser.

**Chrome V8 and the Call Stack**

Before diving into asynchronous concepts, let’s look at a visualization of how the browser executes code using the V8 JavaScript engine. V8 uses a data structure known as a call stack.

The call stack is simple. As a function is called, the function is added to the top of the stack, as a function returns, the function is removed from the top of the stack. We push onto a stack and pop off a stack. This is how synchronous JavaScript works.

Text

Description automatically generated

Figure 2 — the JavaScript Call Stack

We can observer this behaviour through the browser’s developer tools. Figure 3 shows this behaviour within Chrome’s developer tools. As we debug the code, we can see where in the call stack we are. Once we return from function a, a will be removed from the call stack — leaving just c and anonymous.

A screenshot of a computer

Description automatically generated with medium confidence

Now here is where it gets tricky! JavaScript as a language is single threaded. This means you cannot tell JavaScript to create multiple threads that run in parallel. In JavaScript land, functions can only be run one after the other. Just like in the call stack animation above. Having said that, JavaScript implementations such as V8 use multiple threads to do many things simultaneously.

V8 is multithreaded. This means that you can do other stuff in the background. For example, as you scroll down your favourite article, the page could be fetching data from an end-point that connects to a database, which then updates the page. If you have multiple CPU cores, each asynchronous thread of execution may be assigned to one of those cores. So, whilst the JavaScript language per se is not multi-threaded, the environment which executes JavaScript code is.

In [part 3](https://medium.com/@byteslovesbits/asynchronous-javascript-part-3-864df36177f3), we will begin to explore our first method of asynchronous JavaScript through the use of **callbacks**. Callbacks will lay the foundations as we progress to better, more modern methods of asynchronous programming.