How JS Works

What is JAVASCRIPT?

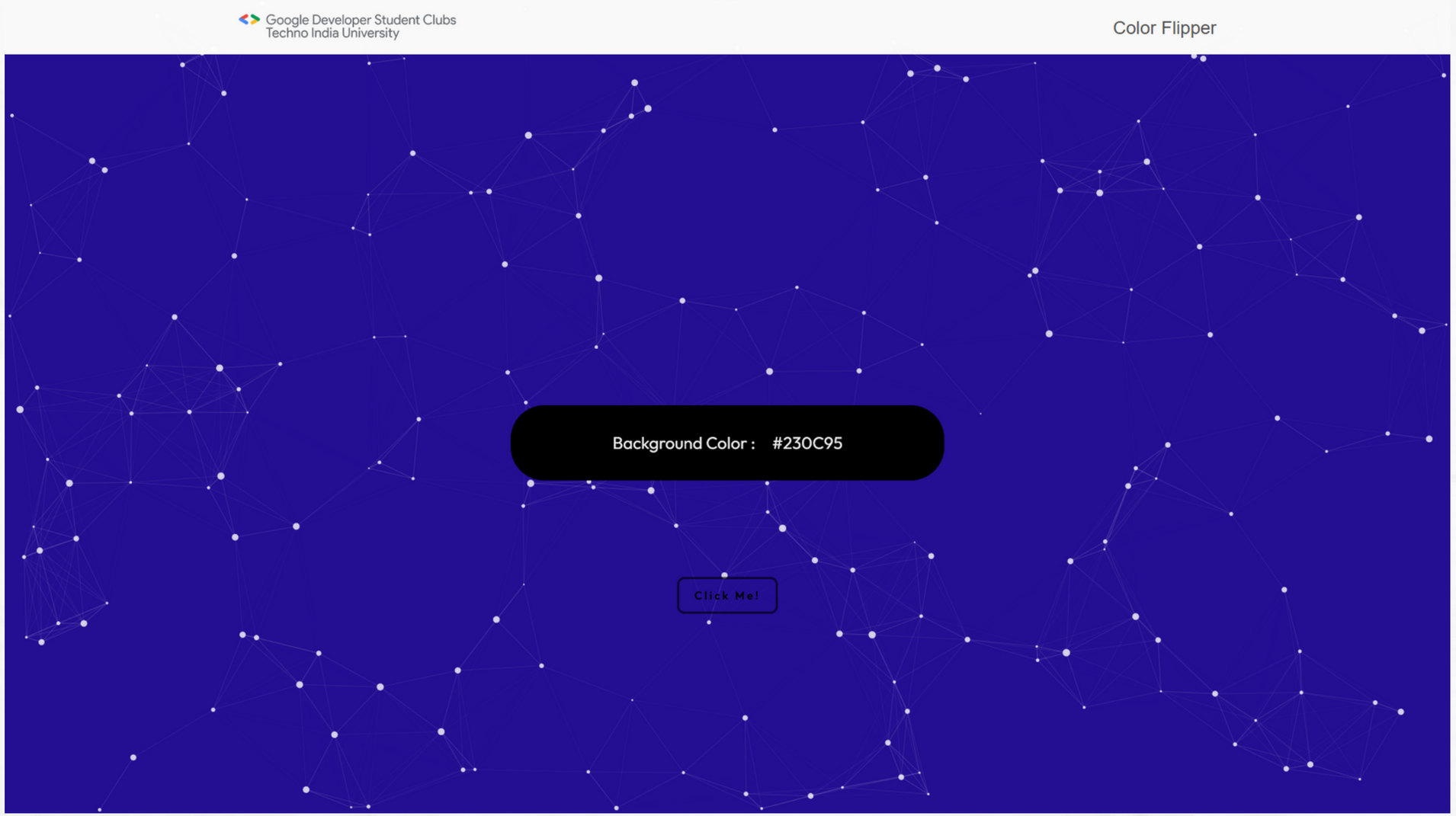
JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.

JavaScript was first known as *LiveScript*, but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name LiveScript. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers.

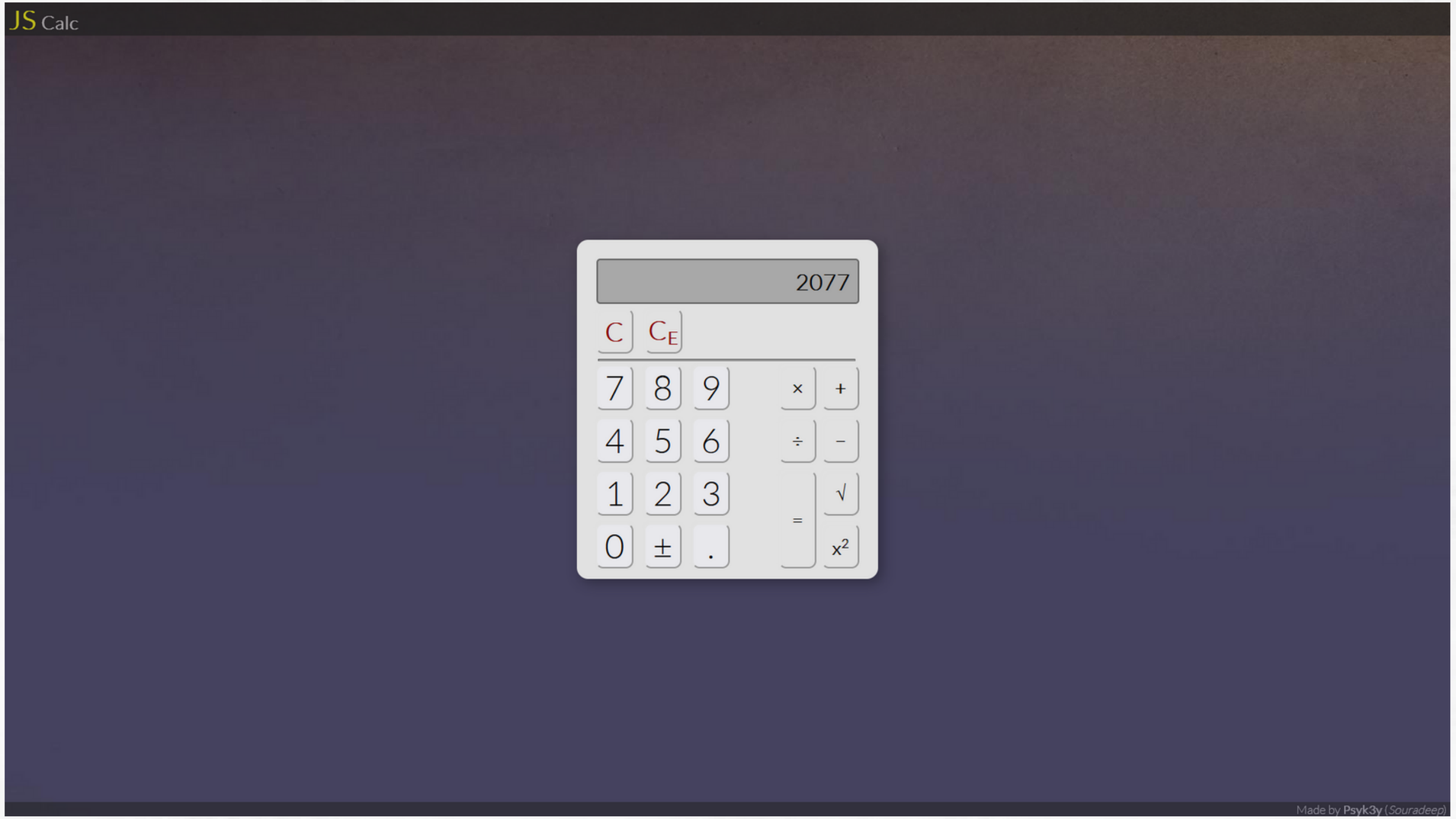
OR

It helps you make cooler* websites.

*Subjectively

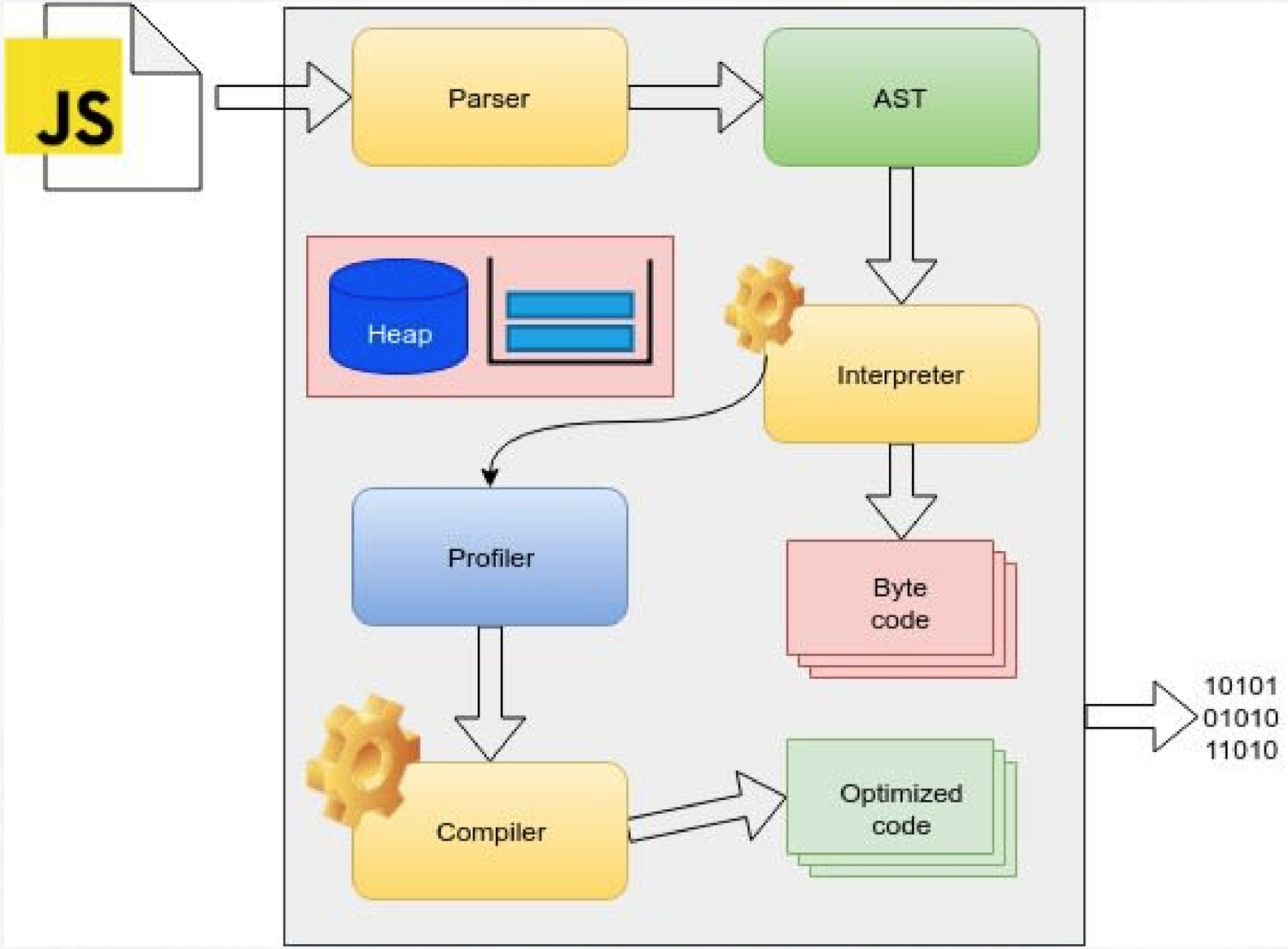


A Color Flipper



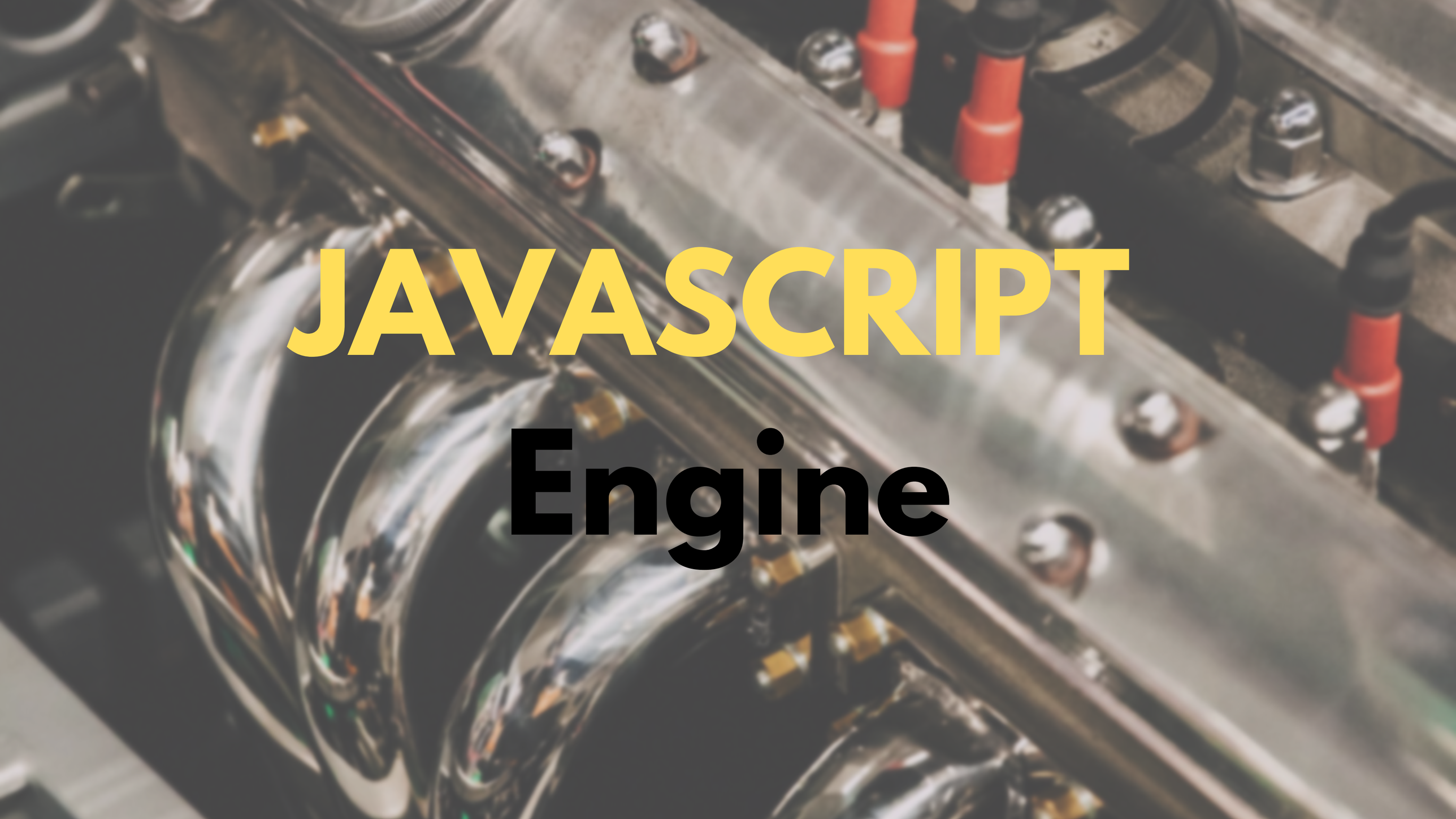
A Calculator

But, How Does it Work?

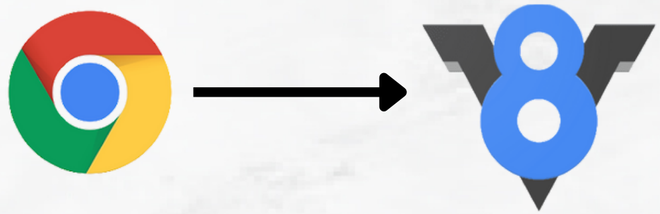

JS
Parser
AST
Heap
Interpreter
Profiler
Byte
code
Compiler
Optimized
code
10101
01010
11010

All JavaScript code needs to be hosted and run in some kind of environment. In most cases, that environment would be a web browser.

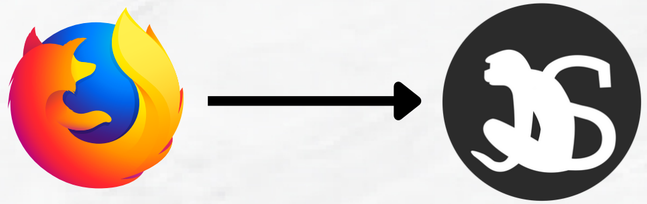
For any piece of JavaScript code to be executed in a web browser, a lot of processes take place behind the scenes.


JAVASCRIPT
Engine

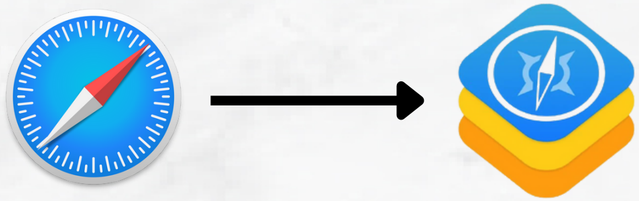
A JavaScript engine is simply a computer program that receives JavaScript source code and compiles it to the binary instructions (machine code) that a CPU can understand. JavaScript engines are typically developed by web browser vendors, and each major browser has one.



V8 Engine



Spider Monkey



JavaScriptCore

Parser

A Parser or Syntax Parser is a program that reads your code line-by-line. It understands how the code fits the syntax defined by the Programming Language and what it (the code) is expected to do.

While reading through HTML, if the browser encounters JavaScript code to run via a *<script>* tag or an attribute that contains JavaScript code like *onClick*, it sends it to its JavaScript engine.

The browser's JavaScript engine then creates a special environment to handle the transformation and execution of this JavaScript code. This environment is known as the ***Execution Context.***

EXECUTION CONTEXT

The Execution Context contains the code that's currently running, and everything that aids in its execution.

During the Execution Context run-time, the specific code gets parsed by a parser, the variables and functions are stored in memory, executable byte-code gets generated, and the code gets executed.

There are two kinds of Execution Context in JavaScript:

Global Execution Context

Function Execution Context

Global Execution Context

Whenever the JavaScript engine receives a script file, it first creates a default Execution Context known as the Global Execution Context.
The GEC is the base/default Execution Context where all JavaScript code that **is not inside of a function** gets executed

In any JavaScript project, no matter how large it is, there's ***only one global execution context***.

Function Execution Context

Whenever a function is called, the JavaScript engine creates a different type of Execution Context known as a Function Execution Context (FEC) within the GEC to evaluate and execute the code within that function.

Since every function call gets its own FEC, there can be more than one FEC in the run-time of a script.

WHAT'S INSIDE EXECUTION CONTEXT?

1. Variable Environment
   - `let`, `const` and `var` declarations
   - Functions
   - ~~`arguments`~~ object
2. Scope chain
3. ~~this~~ keyword

NOT in arrow functions!

Generated during "creation phase", right before execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

Global

```
name = 'Jonas'
first = <function>
second = <function>
x = <unknown>
```

Literally the function code

Need to run `first()` first

first()

```
a = 1
b = <unknown>
```

Need to run `second()` first

second()

```
c = 2
arguments = [7, 9]
```

Array of passed arguments. Available in all "regular" functions (not arrow)

(Technically, values only become known during execution)

How are Execution Contexts Created?

Now that we are aware of what Execution Contexts are, and the different types available, let's look at how the are created.

The creation of an Execution Context (GEC or FEC) happens in two phases:

1. Creation Phase
2. Execution Phase


Creation Phase

In the creation phase, the Execution Context is first associated with an Execution Context Object (ECO). The Execution Context Object stores a lot of important data which the code in the Execution Context uses during its run-time.

The creation phase occurs in 3 stages, during which the properties of the Execution Context Object are defined and set. These stages are:

1. Creation of the Variable Object (VO)
2. Creation of the Scope Chain
3. Setting the value of the *this* keyword

The Variable Object (VO)

The Variable Object (VO) is an object-like container created within an Execution Context. It stores the variables and function declarations defined within that Execution Context.
In the GEC, for each variable declared with the var keyword, a property is added to VO that points to that variable and is set to 'undefined'.
Also, for every function declaration, a property is added to the VO, pointing to that function, and that property is stored in memory. This means that all the function declarations will be stored and made accessible inside the VO, even before the code starts running.

The FEC, on the other hand, does not construct a VO. Rather, it generates an array-like object called the 'argument' object, which includes all of the arguments supplied to the function. Learn more about the argument object here.

This process of storing variables and function declaration in memory prior to the execution of the code is known as Hoisting.


Hoisting

Function and variable declarations are hoisted in JavaScript. This means that they are stored in memory of the current Execution Context's VO and made available within the Execution Context even before the execution of the code begins.

Function Hoisting

In most scenarios when building an application, developers can choose to define functions at the top of a script, and only later call them down the code

Variable Hoisting

Variables initialized with the var keyword are stored in the memory of the current Execution Context's VO as a property, and initialized with the value undefined. This means, unlike functions, trying to access the value of the variable before it is defined will result in undefined.


Execution Phase

Finally, right after the creation phase of an Execution Context comes the execution phase. This is the stage where the actual code execution begins.

Up until this point, the VO contained variables with the values of undefined. If the code is run at this point it is bound to return errors, as we can't work with undefined values.

At this stage, the JavaScript engine reads the code in the current Execution Context once more, then updates the VO with the actual values of these variables. Then the code is parsed by a parser, gets transpired to executable byte code, and finally gets executed.


Global Execution Context
Creation Phase (Web Browser)
Global Object: window
this: window
x: undefined
timesTen: function(){...}
y: undefined
Global Execution Context
Execution Phase (Web Browser)
Global Object: window
this: window
x: 10
timesTen: function(){...}
y: timesTen(x)

Execution Stack

The Execution Stack, also known as the Call Stack, keeps track of all the Execution Contexts created during the life cycle of a script.

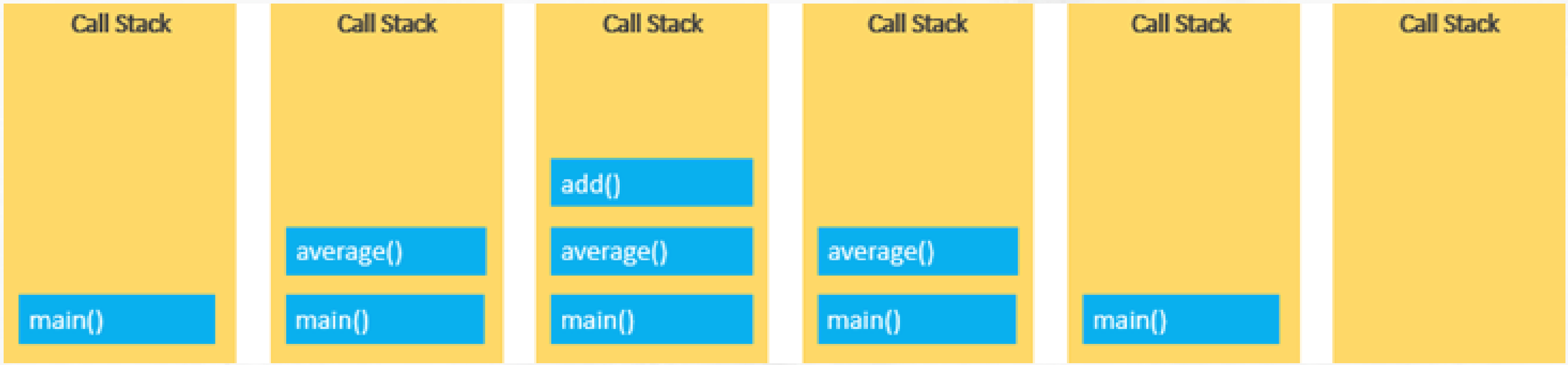
JavaScript is a single-threaded language, which means that it is capable of only executing a single task at a time. Thus, when other actions, functions, and events occur, an Execution Context is created for each of these events. Due to the single-threaded nature of JavaScript, a stack of piled-up execution contexts to be executed is created, known as the Execution Stack.

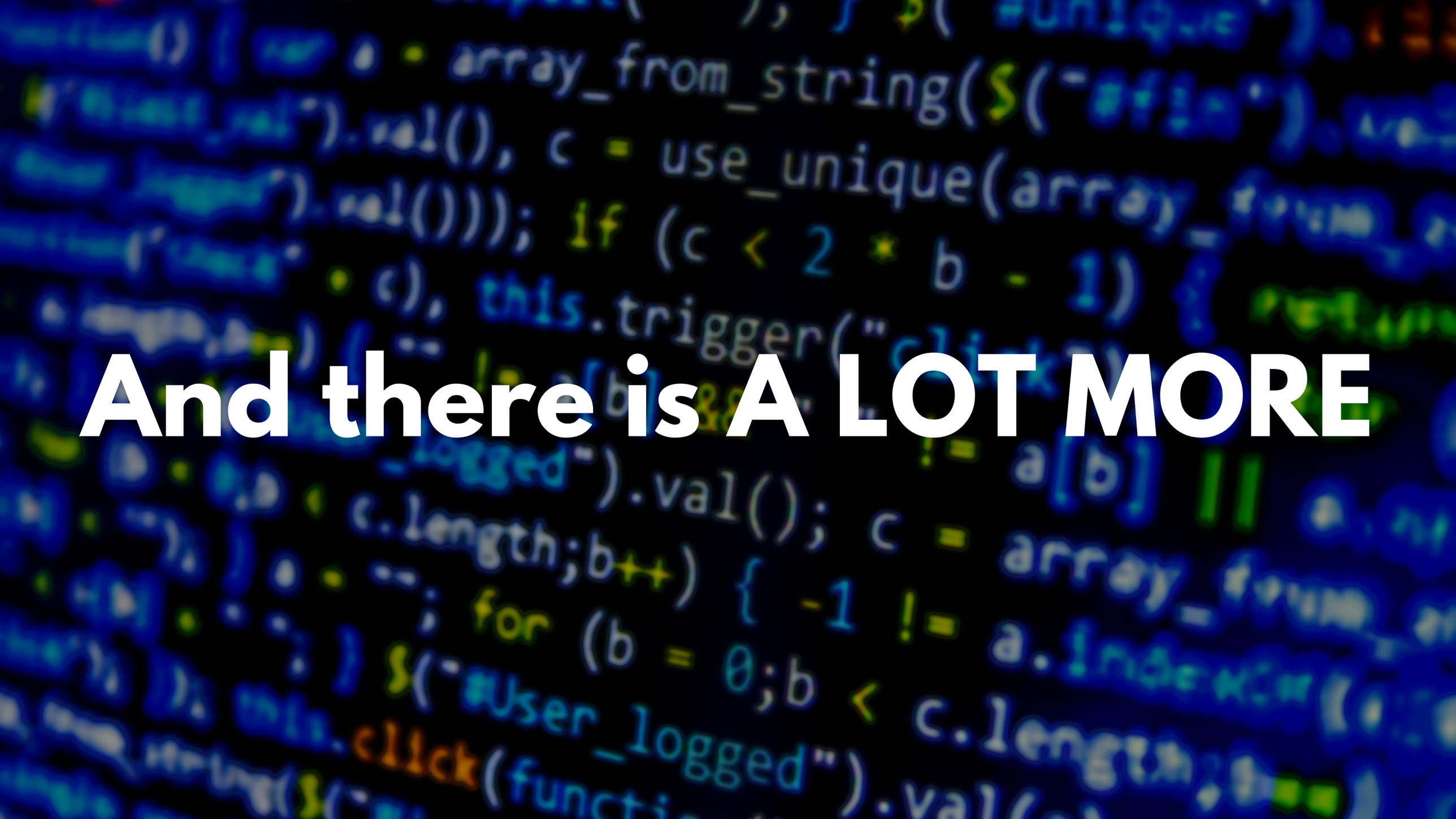
Example

```
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

let x = average(10, 20);
```


Call Stack
main()
Call Stack
average()
main()
Call Stack
add()
average()
main()
Call Stack
average()
main()
Call Stack
main()
Call Stack


And there is A LOT MORE