

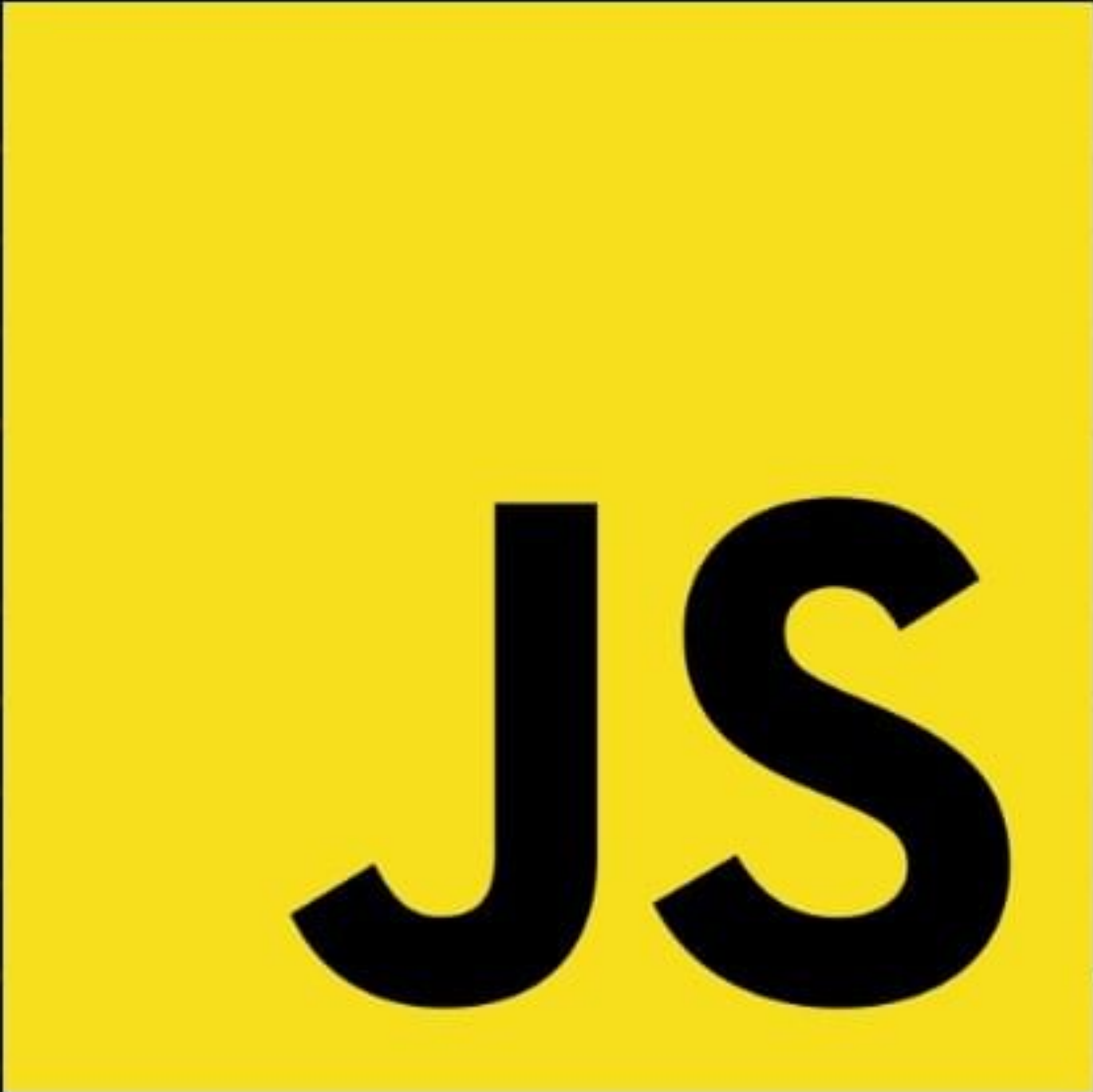
2022

# How Does JavaScript Work?

[ You Don't Want to Miss this ]



Gulraiz.



JS

🔖 Save it for later

Almost everyone has already heard of the **V8 Engine as a concept**, and most people know that JavaScript is **single-threaded** or that it is using a **callback queue**.

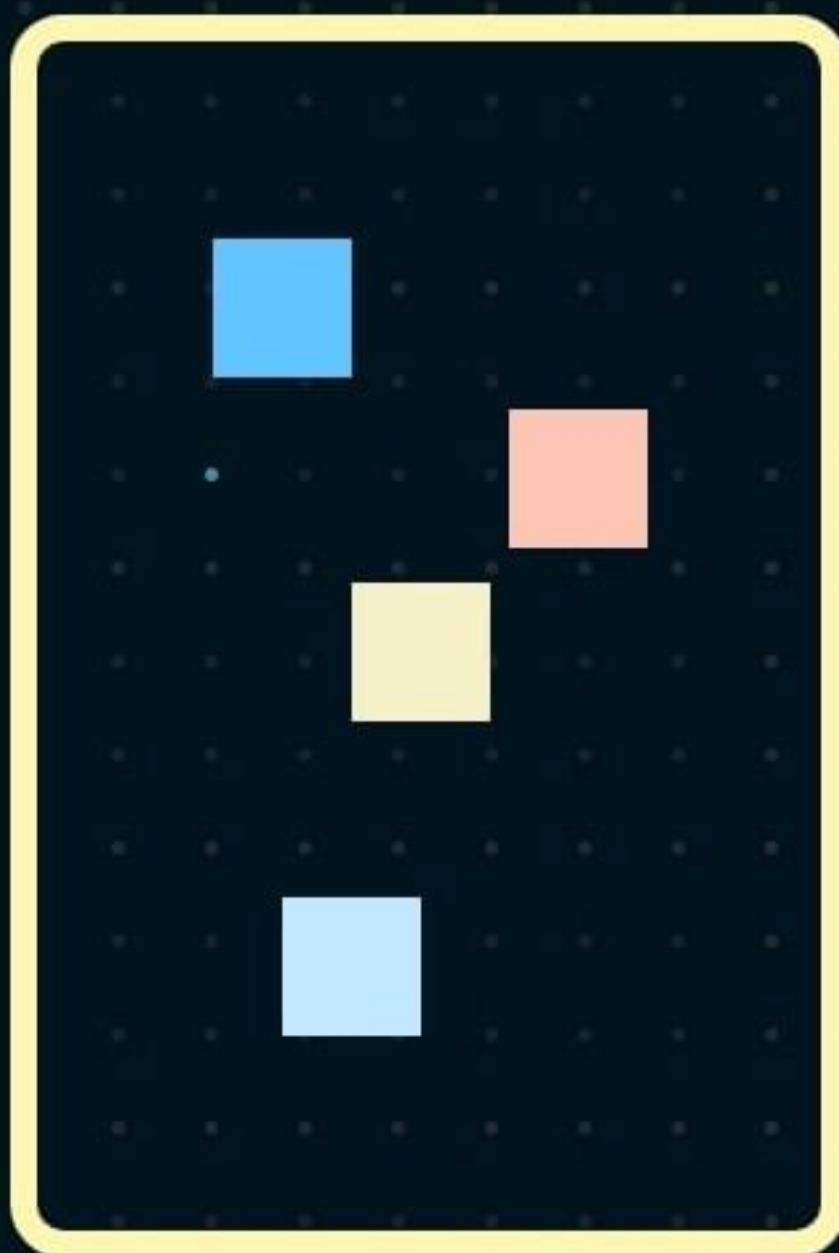
In this post, we'll learn **how JS actually runs**. You'll be able to write **better, non-blocking apps**.



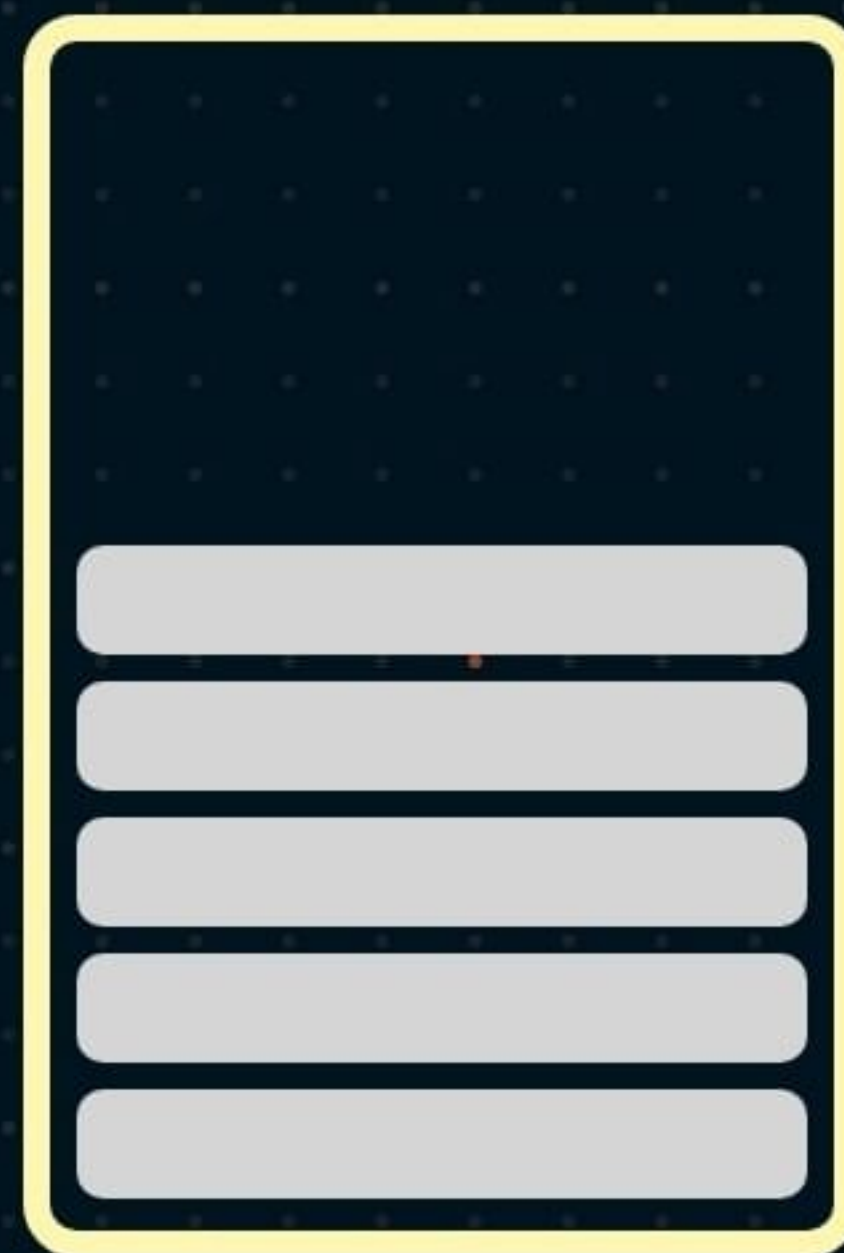
# The JavaScript Engine

The V8 Engine is used inside **Chrome** and **Node.js**  
Here's the **visual representation** of how it looks

Memory Heap



Call Stack



The Engine consists of two components:

⚡ **Memory Heap** — this is where the memory allocation happens

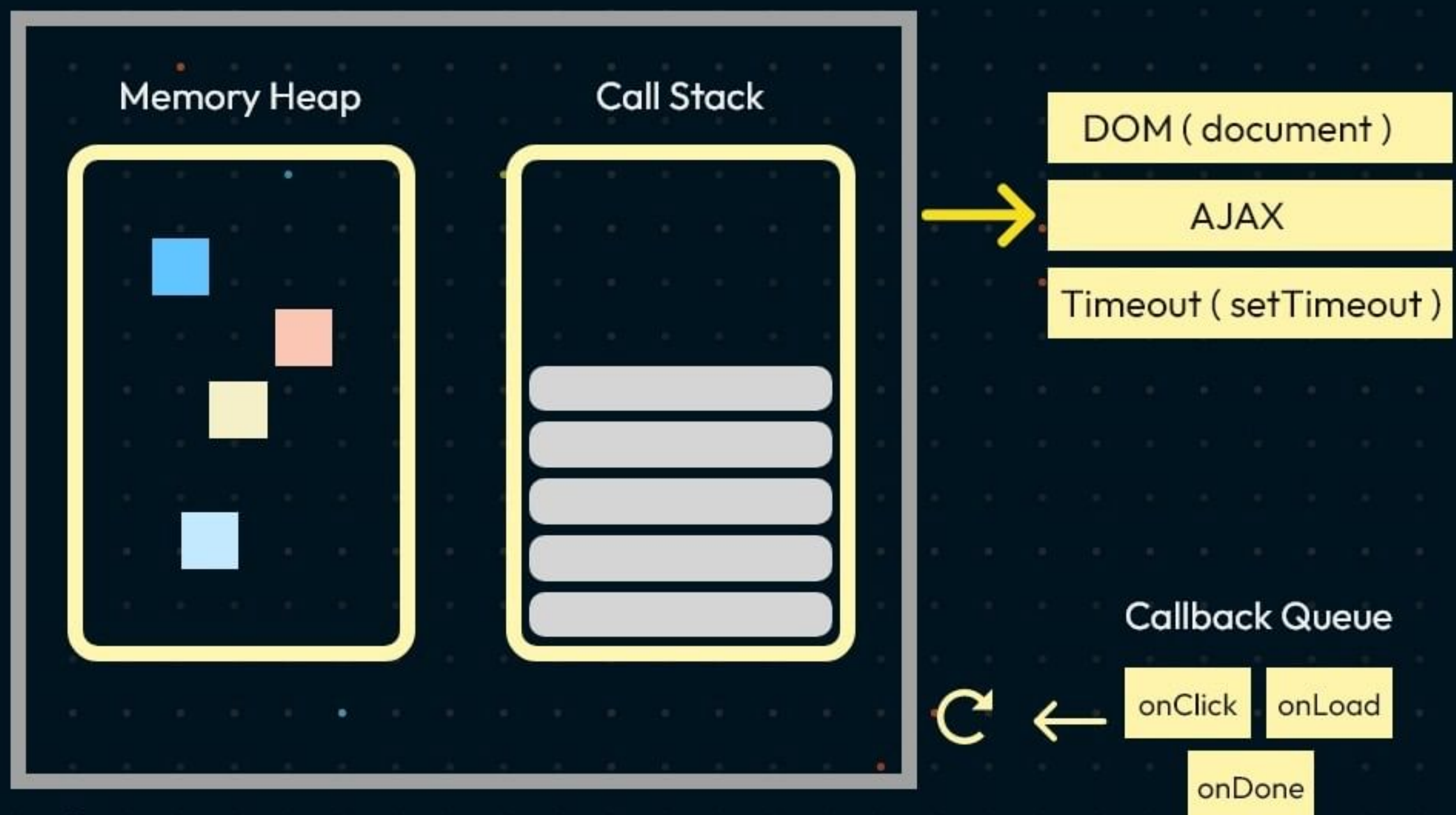
⚡ **Call Stack** — this is where your stack frames are as your code executes



# The Runtime

There are **APIs in the browser** that has been used by almost any JavaScript developer out there (e.g. **“setTimeout”**). Those APIs, however, **are not provided** by the Engine.

So, where are they coming from?

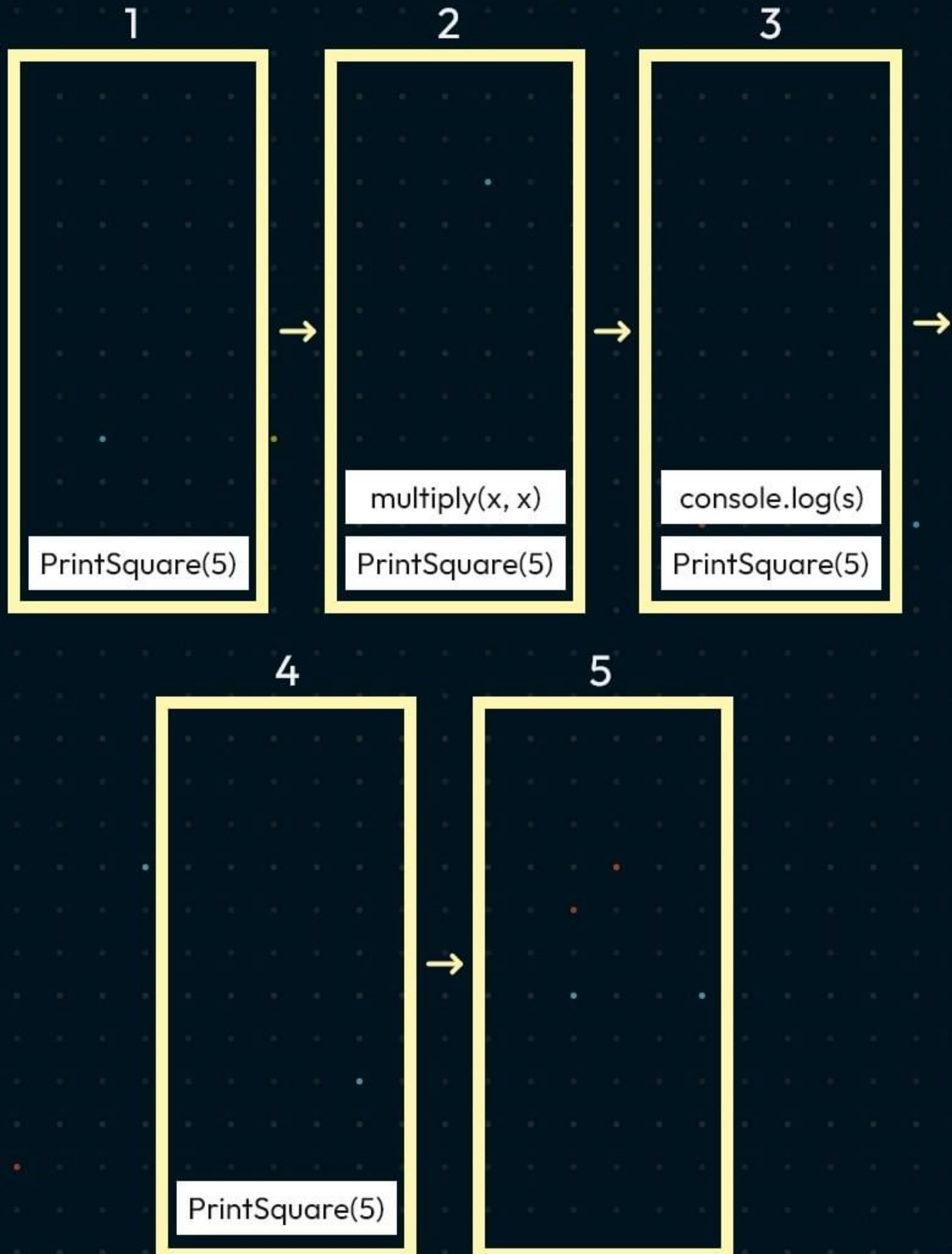


We have Engine but there is actually a lot more. We have **DOM, AJAX, SetTimeout**, and much more.



# The Runtime

JS is a **single-threaded** Language which means It has a **single call stack**. Means It can do **one thing at a time**.





# The Runtime

JS is a **single-threaded** Language which means It has a **single call stack**. Means It can do **one thing at a time**.

Remember,

The **Call stack is a Data Structure** that records basically wherein the program we are.

For Example,

If **we step into a function**, we put it on the **top of the stack**.

If **we return from a function**, we **pop off the top of the stack**.

```
function multiply(x, y) {  
    return x * y;  
}  
function printSquare(x) {  
    var s = multiply(x, x);  
    console.log(s);  
}  
printSquare(5);
```

When the **engine starts executing** this code, the Call Stack will be empty. Afterward, the steps will be the following:



Each entry in the Call Stack is called **Stack Frame**.

this is exactly how stack traces are being **constructed** when **an exception** is being thrown — it is basically **the state of the Call Stack** when the exception happened.

# Checkout my Older Posts

