JavaScript Under the Hood: Mastering the Inner Workings

Ori Baram · Follow
19 min read · Mar 17, 2023

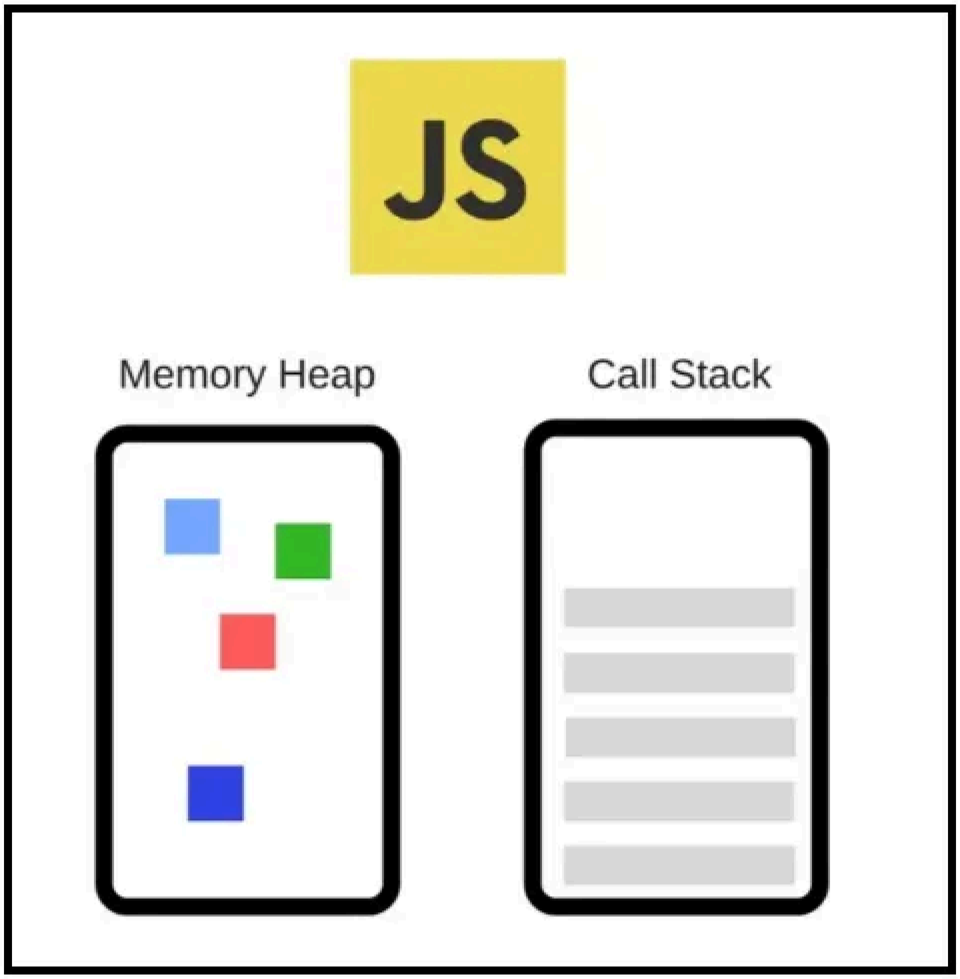
270 3

This article is heavily inspired, among other things, by lectures and presentations from the amazing Complete JavaScript Course by Jonas Schmedtmann.

In this article we will focus on JavaScript as it runs in the browser.

Every browser has a JavaScript engine. The most popular engine is Google's V8 engine. This engine is the engine of Google Chrome and also of Node.js. Of course, all other browsers have their own JavaScript engines.

A JavaScript engine will always consist of a *Call Stack* and a *Memory Heap* and will run on *one thread* (The JavaScript engine itself runs in several threads (processes) that do different things: compressor for example, garbage collection, etc., but that is not our interest in this article). We will elaborate on them later.



JavaScript Engine

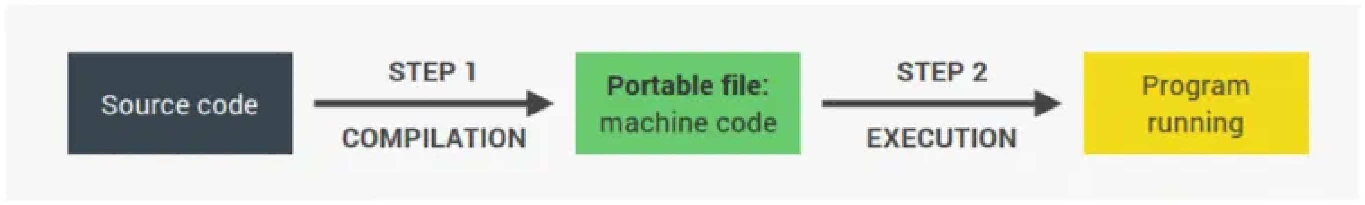
Compilation, Interpretation and Just-In-Time Compilation

Computers fundamentally understand machine code, which can be represented as assembly language. Therefore, all software must eventually be converted into a format the computer can execute directly. There are three common approaches to this conversion process:

1. **compilation**
2. **Interpretation**
3. **Just-in-time compilation**

Compilation (Ahead of time Compilation)

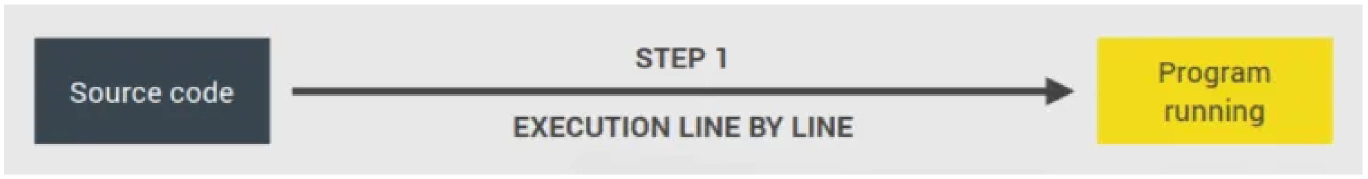
In this method, all the code is converted to machine language at once, and then written to a file in assembly, so that the computer can run the software, which can happen even a long time after the file was created.



Compilation

Interpretation

In this method, the Interpreter goes through the code in an initial pass and then executes it line by line. During the runtime, while running line by line, the code is also compiled into machine language.



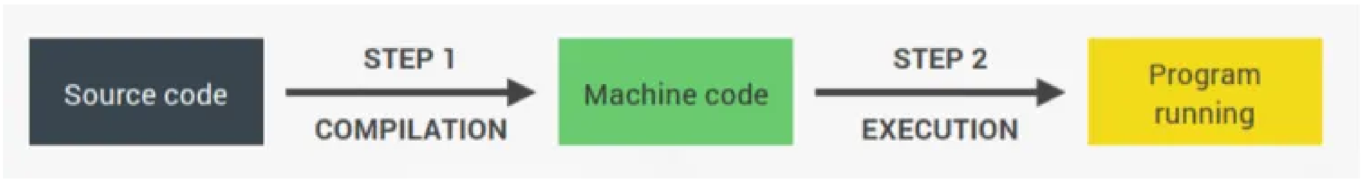
Interpretation

In the past, JavaScript was primarily an interpreted language, which led to certain performance limitations. As web developers discovered, interpreting JavaScript line by line made it challenging to implement optimizations effectively. This is partly due to JavaScript's dynamic nature, where data types can change at runtime. For example, if a variable is consistently used as a Boolean, an interpreter may still allocate more memory than necessary because it cannot make assumptions about the variable's type.

Over time, advances in JavaScript engines, such as *just-in-time compilation (JIT)*, have been introduced to address these inefficiencies, resulting in significant performance improvements.

Just-in-time compilation

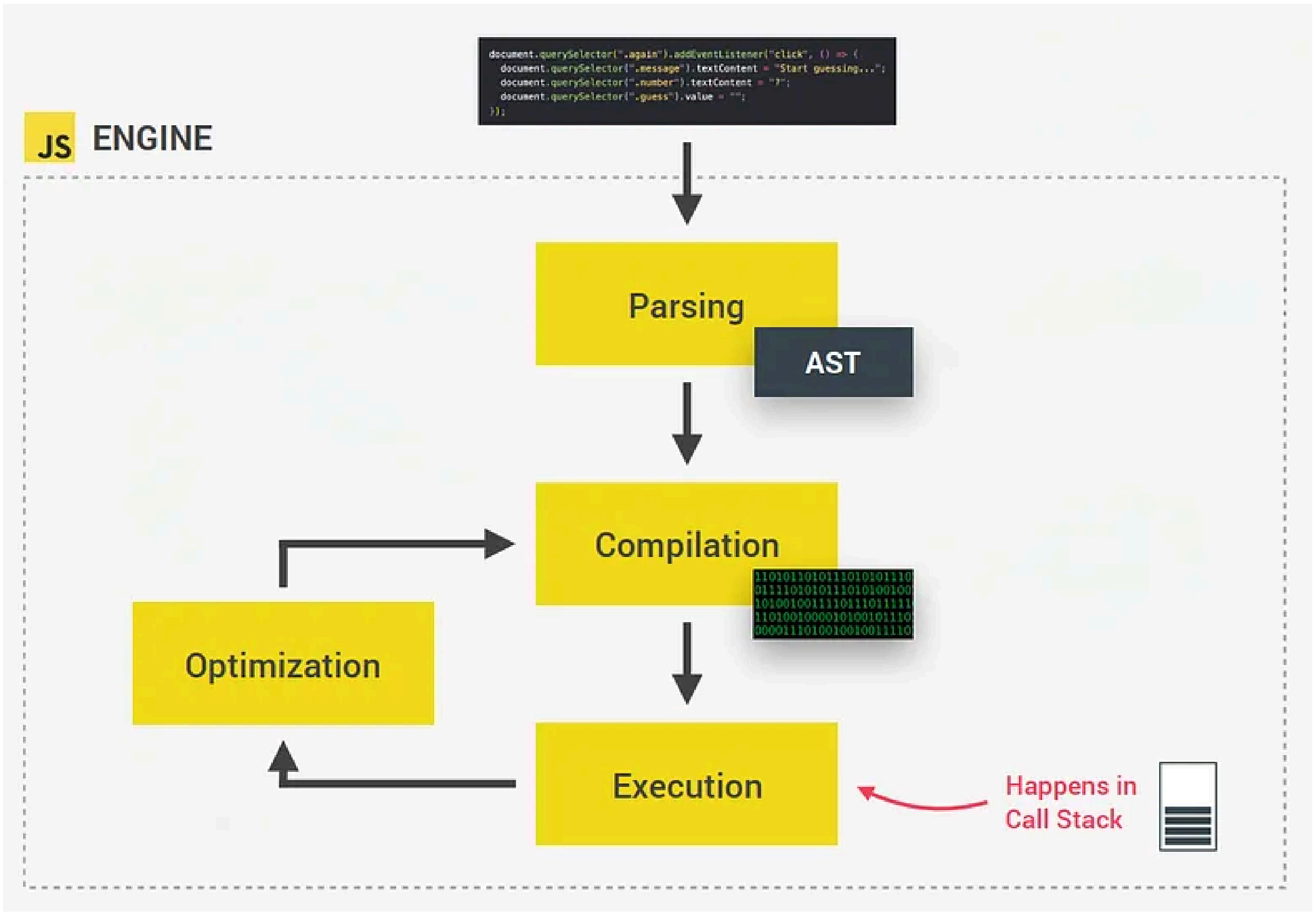
In this approach, the entire code is translated into machine language in a single step and is executed immediately afterward. During the conversion process, no intermediate files are created; instead, the code is directly compiled into machine language and executed without delay. This method streamlines the execution process by combining the compilation and execution steps, thereby enhancing the overall efficiency of code processing and execution.



Today, JavaScript employs *just-in-time (JIT) compilation*, which combines the benefits of both interpretation and traditional compilation. This real-time compilation process is faster than interpretation because it eliminates the overhead associated with interpreting code line by line. JIT compilation also offers advantages over ahead-of-time compilation, as it can perform optimizations based on runtime information that would not be available during a traditional compilation process. As a result, code executed using JIT compilation is often faster and more efficient than code executed through either interpretation or ahead-of-time compilation alone.

Just-in-time compilation in JavaScript

Now we will talk about how just-in-time compilation is performed in real time in JavaScript.



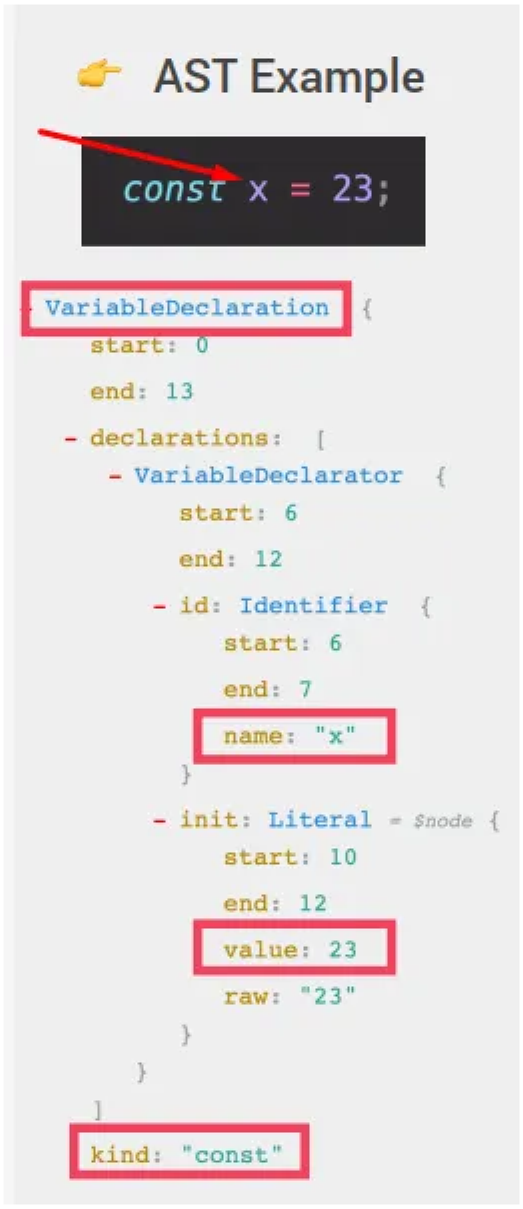
```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  document.querySelector(".guess").value = "";
});
```

Steps involved in JIT compilation for JavaScript

Parsing

In the first step of processing JavaScript code, parsing is performed, which involves reading the code and creating an Abstract Syntax Tree (AST). During this process, the code is broken down into meaningful components according to JavaScript's syntax, such as significant keywords like const, function, or new. These components are then organized into a structured tree. This stage also includes checking the code for any syntax errors. The resulting AST serves as the basis for converting the JavaScript source code into machine code, which can be executed by the JavaScript engine in subsequent stages.

Say we have a simple primitive variable like this x.



```
const x = 23;
```

```
VariableDeclaration {
   start: 0
   end: 13
 - declarations: [
    - VariableDeclarator {
         start: 6
         end: 12
       - id: Identifier {
            start: 6
            end: 7
            name: "x"
         }
       - init: Literal = $node {
            start: 10
            end: 12
            value: 23
            raw: "23"
         }
      }
   ]
   kind: "const"
```

So this is what the AST looks like for this single line of code. Besides the simple things, such as the name, the value and the type, there is a lot more data, which is not our concern. Understanding the exact structure of the AST is not crucial, and there's no need to delve into its details. This information is provided merely for general knowledge purposes.

Important note: *Despite both being referred to as 'trees,' the AST and the DOM Tree are entirely separate concepts and should not be confused due to the shared terminology. The AST Is a representation of all the code inside the JavaScript engine.*

Compilation and Execution

The subsequent stage is the compilation phase, where the generated AST is converted into machine code. Following this, the machine code is executed immediately, as modern JavaScript engines employ JIT Compilation, as previously mentioned. We will soon discuss the execution stage occurring within the call stack. However, the story doesn't end there. Modern JavaScript engines, such as the V8 engine, employ sophisticated optimization strategies. Initially, they generate an inefficient version of the machine code to enable prompt execution, and then, in the background, the code undergoes optimization processes. After each optimization, the code is recompiled in the optimized version, with the previous code being replaced without disrupting or halting the execution. This process is what makes modern engines so fast.

These processes of compression, compilation, and optimization occur within specialized threads (processes) inside the engine, which are inaccessible to us. This thread operates independently from the main thread where the call stack runs and our code executes.

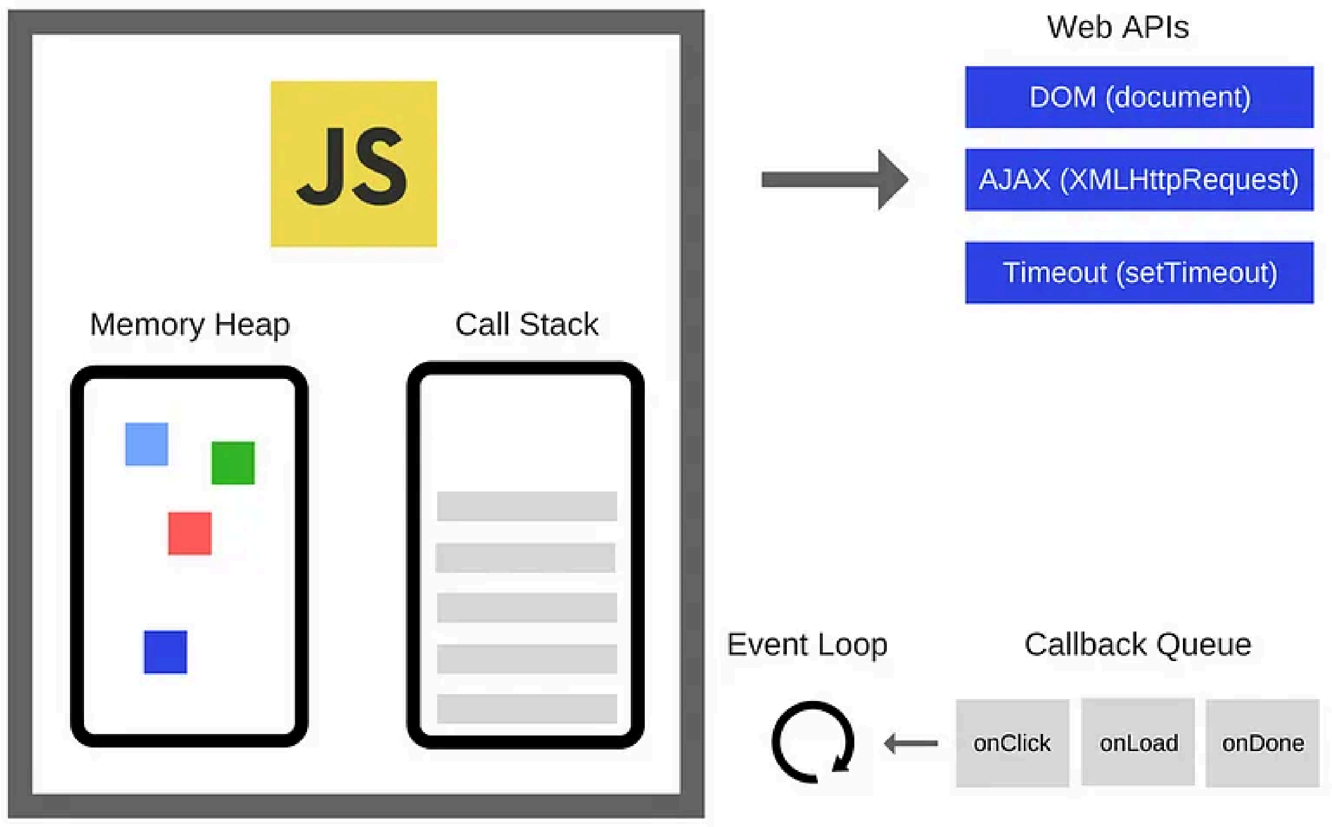
While different engines may implement this process in various ways, the fundamentals remain the same for modern runtime compilation.

The JavaScript runtime environment

Next, we'll discuss the JavaScript runtime environment in the browser, which is crucial to understand.

Picture the runtime environment as a container encompassing everything needed to run JavaScript within the browser. At the core of the runtime environment lies the JavaScript engine, which we explored in the previous section. Without an engine, there is no runtime, and without a runtime, JavaScript cannot function.

However, the engine alone is insufficient. For it to operate correctly within the browser, access to Web APIs is necessary. Web APIs encompass everything related to the DOM, timers, and numerous other APIs that we can access in the browser.



JavaScript runtime in the browser

For instance, console.log, which we frequently use, is not a part of the language itself but an implementation of an API that JavaScript utilizes. Web APIs are interfaces that offer functionality to the engine in the browser environment but are not part of the JavaScript language itself. JavaScript accesses these APIs through the global Window object in the browser.

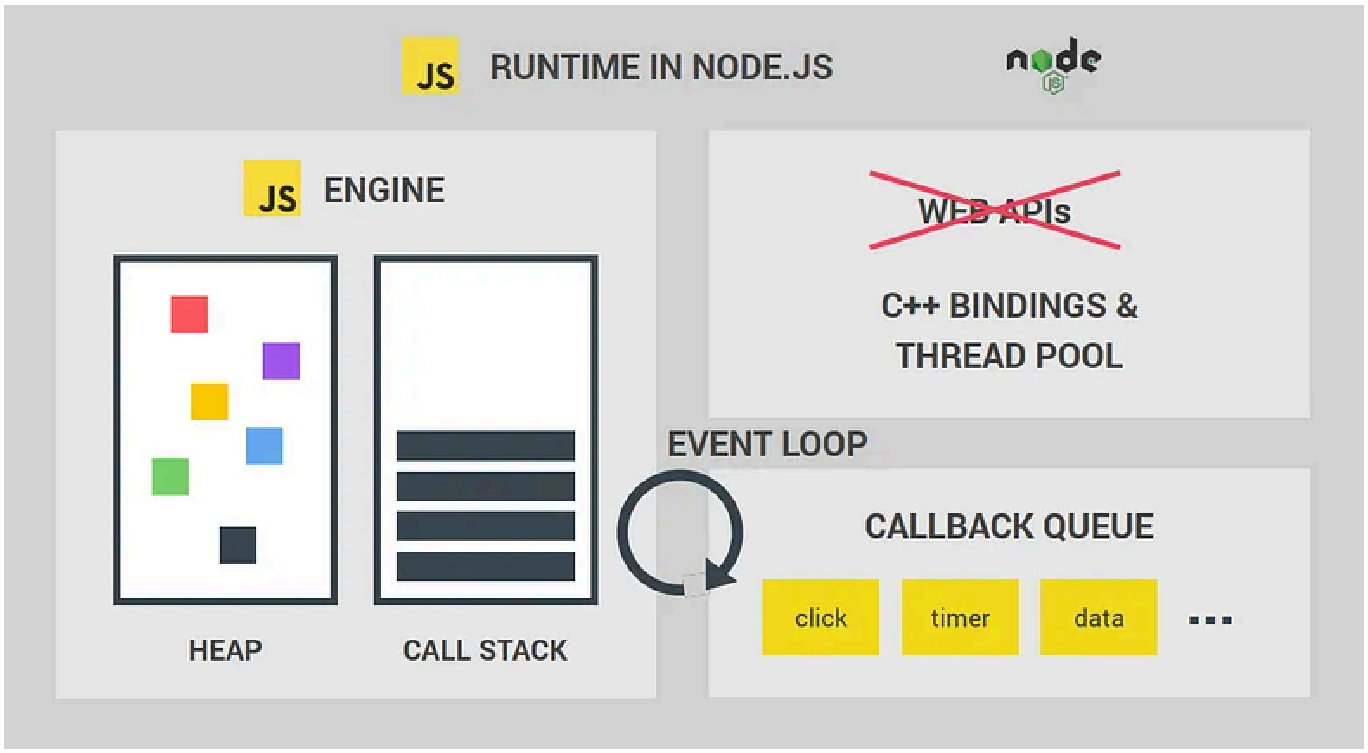
As mentioned earlier, Web APIs are not part of the engine, but they are components of the runtime environment. The runtime environment, as discussed, is a container that holds everything JavaScript needs to function in the browser, in the context of this article. The JavaScript runtime also comprises the Callback Queue, a data structure containing all the callback functions that are ready to execute. For example, when attaching an Event Listener to a DOM element like a button, it responds to a click event. We will see this in action later.

The function passed to the Event Listener is a Callback Function. When the event occurs, such as a button click, the callback function is invoked. What happens is that the callback function is first moved to the Callbacks Queue, and once the Call Stack is empty, the Callback Function transitions to the Call Stack for execution.

This occurs through the so-called Event Loop, which takes Callback Functions from the Callback Queue and places them onto the Call Stack for execution. In this manner, the JavaScript runtime implements the Nonblocking Concurrency Model, a non-blocking parallel model. Although JavaScript runs on a single thread, it employs multiple processes using other components in the runtime environment. We'll discuss this further later and explain why it makes the entire JavaScript runtime non-blocking.

We have discussed how JavaScript operates in the browser, but it is important to remember that JavaScript can also exist and run outside of browsers, such as in Node.js.

The Node.js runtime environment is quite similar to the browser's runtime environment. However, since there is no browser, Web APIs are not present, as there is no browser to provide them. Instead, we have C++ Bindings and a Thread Pool. These details are not our primary concern. The essential point to remember is that different JavaScript runtimes exist.



Node.js runtime environment

The Execution Context

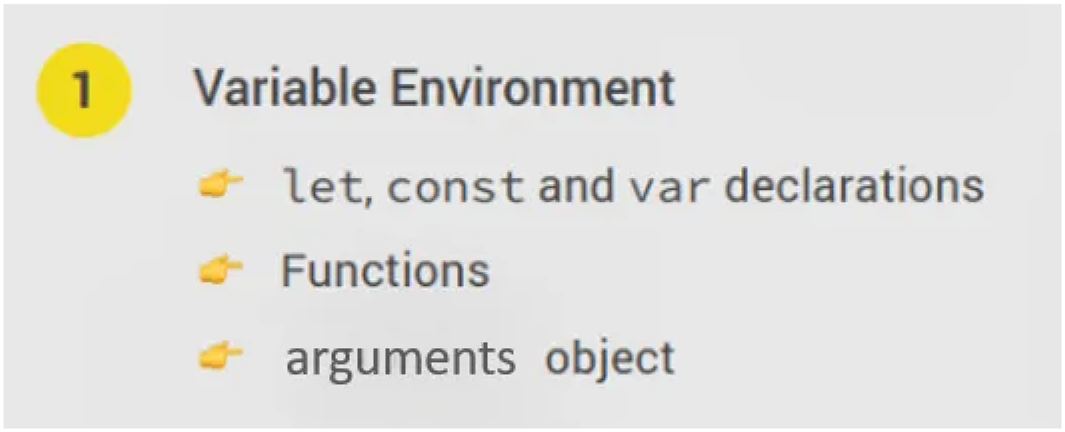
So, how is JavaScript code executed? We will now delve deeper into the topic, starting with the Execution Context. While the Execution Context is an abstract concept, it can be defined as *an environment where segments of JavaScript code are executed*. It serves as a container that holds all the necessary information for executing specific code, such as local variables or arguments passed to functions. There will always be only one Execution Context in operation, and the default Execution Context is the Global Execution Context, where the top-level code is executed.

Once the top-level code execution is complete, functions begin to execute. For each function call, a new Execution Context is created, containing all the information required to run that particular function. The same principle applies to methods, as they are merely functions attached to keys within objects.

After all functions have finished executing, the engine awaits the arrival of Callback Functions to run them, such as those linked to Click Events. As a reminder, it is the Event Loop that supplies these Callback Functions from the Callback Queue when the Call Stack is emptied.

The components of the Execution Context

The first thing we have in the Execution Context is the Variables Environment.


1 Variable Environment

- `let`, `const` and `var` declarations
- Functions
- arguments object

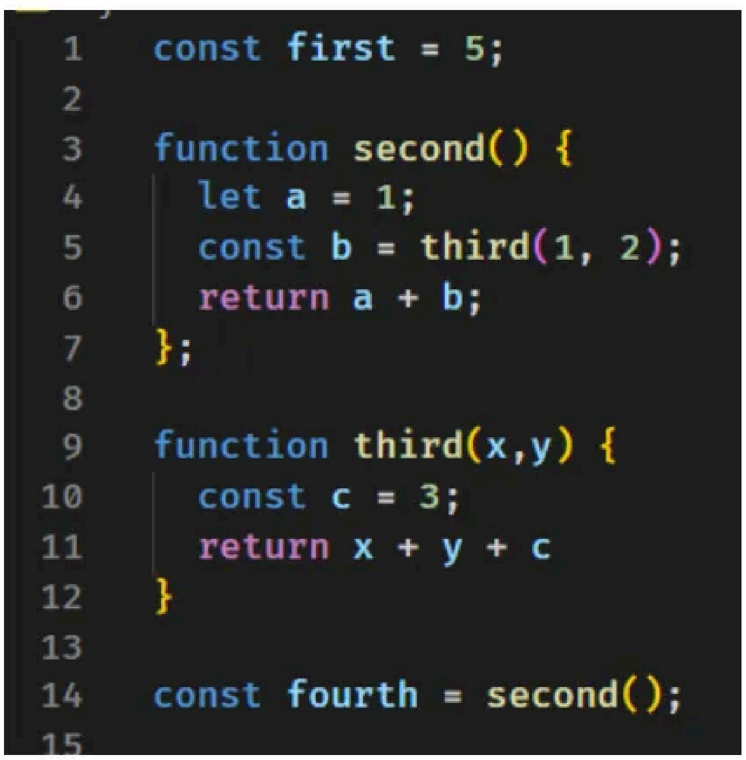
In this environment, all variables and function declarations are stored. Additionally, a special ‘arguments’ object is present. As the name suggests, this object contains all the arguments passed to the function currently in the Execution Context, provided the Execution Context belongs to a function. As mentioned earlier, each function has its own Execution Context, so all variables declared by the function will be located in the function’s variable environment. Furthermore, a function can access variables outside of itself due to the Scope Chain. In short, the Scope Chain contains references to all external variables accessible by the function. Lastly, each Execution Context also receives a special variable called the ‘this’ Keyword.

In summary, the Execution Context comprises the variable environment, the Scope Chain, and the ‘this’ Keyword. These components are created during the Creation Phase, which occurs just before the execution.

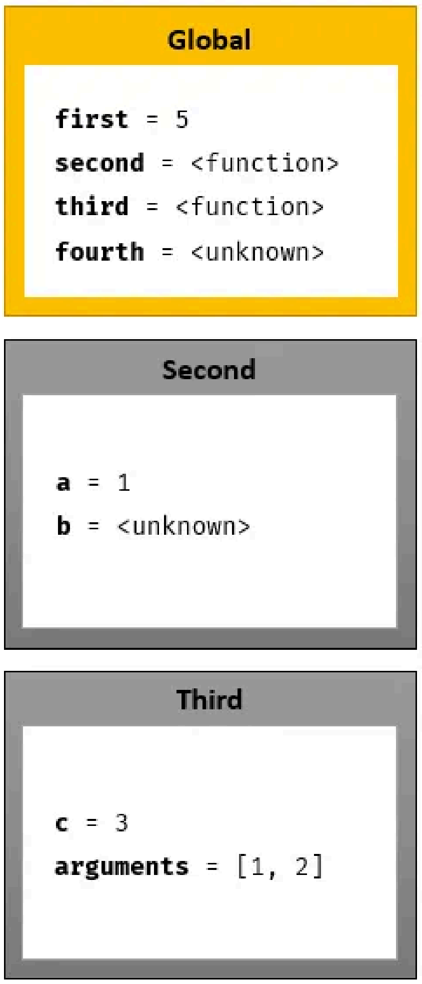
It’s worth mentioning that Arrow Functions do not possess their own ‘arguments’ object or ‘this’ Keyword. Rather, they make use of the ‘arguments’ object and ‘this’ keyword from their nearest parent function or, alternatively, refer to the Global Scope.

The Execution Context in practice

Let’s look at an example to understand the topic better.



```
1   const first = 5;
2
3   function second() {
4     let a = 1;
5     const b = third(1, 2);
6     return a + b;
7   };
8
9   function third(x,y) {
10    const c = 3;
11    return x + y + c
12  }
13
14  const fourth = second();
15
```



Initially, a Global Execution Context is created for the Top Level code, which is any code not contained within a function. So, at first, only this code is executed. This makes sense because functions only run when they are called or triggered. In this example, the '*first*' variable is in the Top Level Code, so it's executed in the Global Execution Context. It's known that the value of '*first*' is 5, '*second*' and '*third*' are functions, and the value of '*fourth*' is still unknown.

In line 14, to get the value of '*fourth*', the '*second*' function is activated and its Execution Context is created. Here, the value of '*a*' is known to be 1, but the value of '*b*' is still unknown, as it must be obtained from the '*third*' function.

So, the '***third***' function runs, and its Execution Context is created with the value of '***c***' being 3 and its 'arguments' object containing 1 and 2, which were passed when the function was executed. Now, '***third***' returns 6 (1 + 2 + 3), we return to the Execution Context of '***second***', and the value of '***b***' is now 6. Finally, '***second***' returns 7 (6 + 1), and the value of '***fourth***' is determined as 7 in the Global Execution Context.

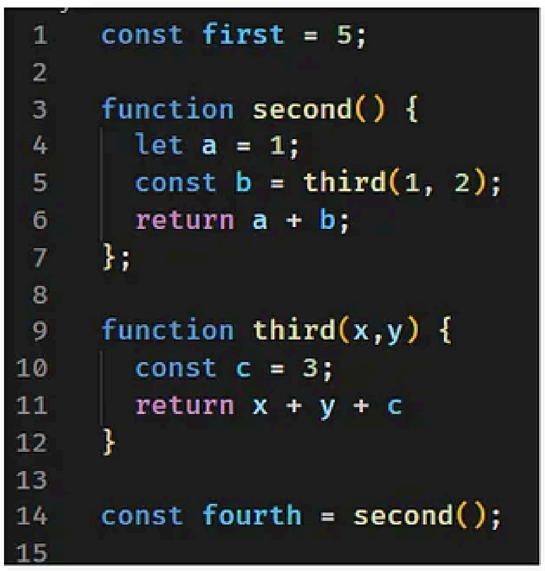
In this small and seemingly simple example, we observed a rather complex process. This raises the question of how the JavaScript engine knows when to execute functions, and which Execution Context should be the current one, especially in more intricate code examples. This is where the **Call Stack** comes into play.

The Call Stack

As previously mentioned, the JavaScript runtime includes the JavaScript engine, which itself comprises the Memory Heap and the Call Stack. So, what exactly is the Call Stack?

The Call Stack is where Execution Contexts are stacked on top of one another to keep track of the program's execution. At any given moment, the topmost Execution Context is the one being executed, following the Last-In-First-Out (LIFO) method.

Referring back to the previous example, the Global Execution Context initially occupies the Call Stack. When the '***second***' function is called, it moves to the top of the Call Stack, followed by the '***third***' function when it is called. Once the '***third***' function finishes and returns a value, it is removed from the Call Stack. Similarly, when the '***second***' function concludes, it is also removed, leaving the Global Execution Context with all the values it was waiting for.



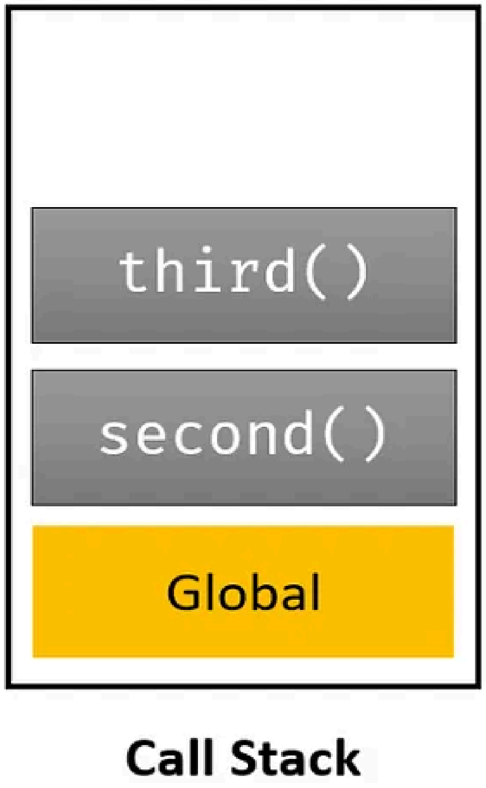
```
const first = 5;

function second() {
  let a = 1;
  const b = third(1, 2);
  return a + b;
};

function third(x,y) {
  const c = 3;
  return x + y + c
}

const fourth = second();

```

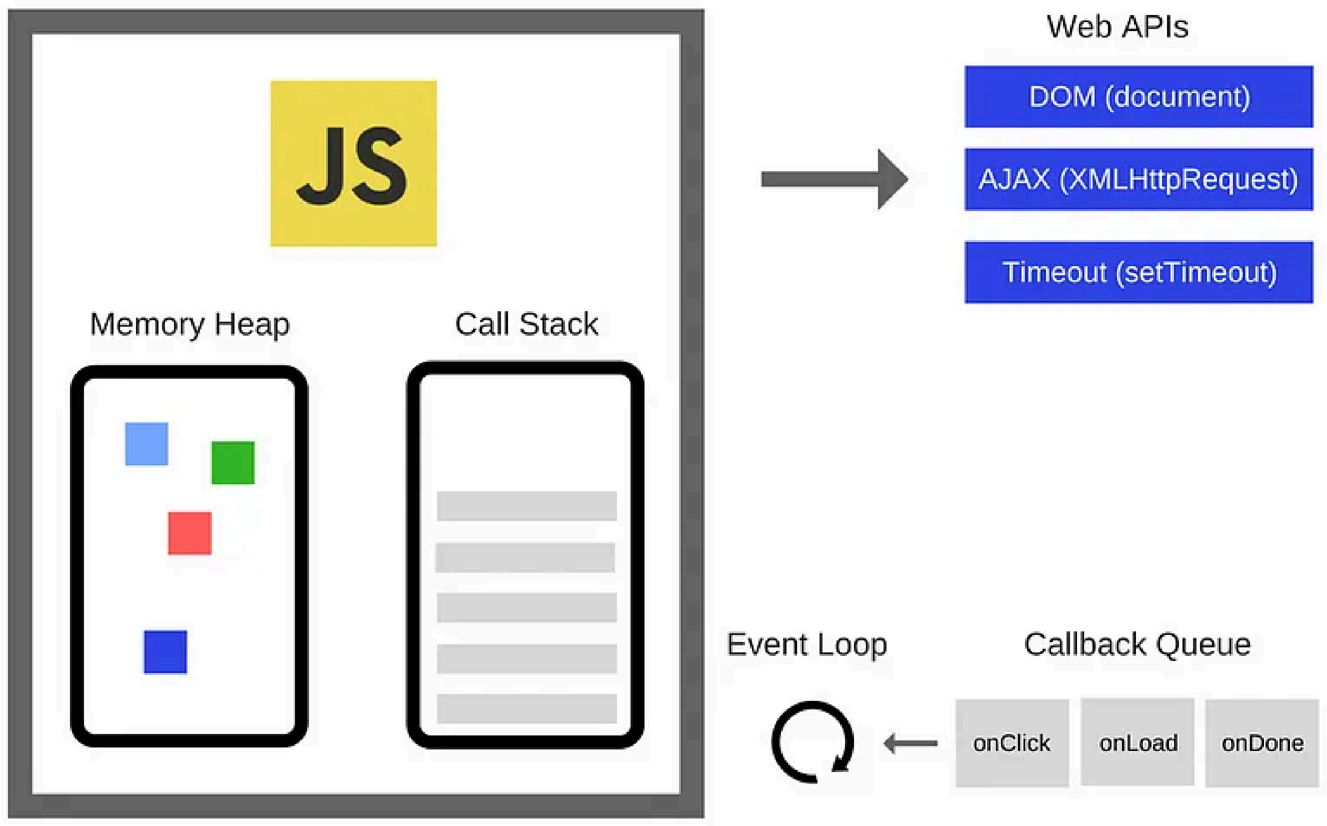


It’s crucial to understand that while any Execution Context is running and positioned at the top of the Call Stack, all other Execution Contexts are suspended and waiting for the current one to complete. This occurs because JavaScript operates on a single thread, requiring operations to be performed sequentially rather than in parallel. The Call Stack is an essential component of the JavaScript engine itself and not a separate part of the runtime environment.

In this way, the Call Stack maintains the execution order of Execution Contexts, ensuring that the sequence of execution is always preserved.

How Asynchronous JavaScript works behind the scenes

Let’s briefly review the JavaScript runtime again.



JavaScript runtime

As previously mentioned, the JavaScript runtime environment serves as a container that encompasses all the various components required for executing JavaScript code. At the core of any JavaScript runtime lies the **JavaScript engine**, which is responsible for code execution and object storage in memory, and operates on a single thread. Code execution and object storage occur in the **Call Stack** and the **Memory Heap**. Notably, JavaScript is single-threaded, allowing it to perform only one task at a time, in contrast to languages like Java that can execute multiple code pieces simultaneously.

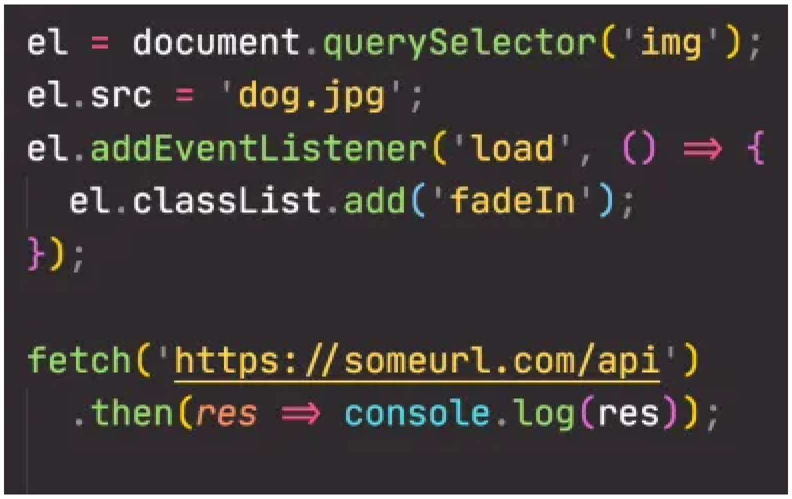
Next, we have the **Web APIs environment**, which provides APIs to the engine that aren't part of the JavaScript language itself. These include timers, the

DOM, the fetch API, the Geolocation API, and so on. Additionally, there's the **Callback Queue**, a data structure containing all the Callback Functions ready to be executed and attached to upcoming events. When the Call Stack is empty, the Event Loop transfers callbacks from the Callback Queue to the Call Stack for execution.

The **Event Loop** is crucial for enabling asynchronous behavior in JavaScript, allowing for the **Non-Blocking Concurrency Model**. The term "Concurrency Model" refers to how a language handles multiple tasks occurring simultaneously.

So, how does JavaScript's Non-Blocking Concurrency Model truly function, and why is the Event Loop so vital? We'll concentrate on the essential components of the runtime environment for this topic, including **the Call Stack, Event Loop, Web APIs,** and **Callback Queue**. Although the JavaScript engine operates on a single thread, asynchronous code can still be executed non-blockingly. We'll soon explore how the JavaScript concurrency model operates behind the scenes, using all the parts of the JavaScript runtime we've already discussed, by examining a real code sample. We've seen how the Call Stack functions, and now we'll focus on the code in **the Web APIs** and **Callback Queue**.

Consider the following image element, '***el***'. In the second line, we define the '***src***' of this image as '***dog.jpg***', and the image begins to load asynchronously in the background (image loading occurs asynchronously in JavaScript).

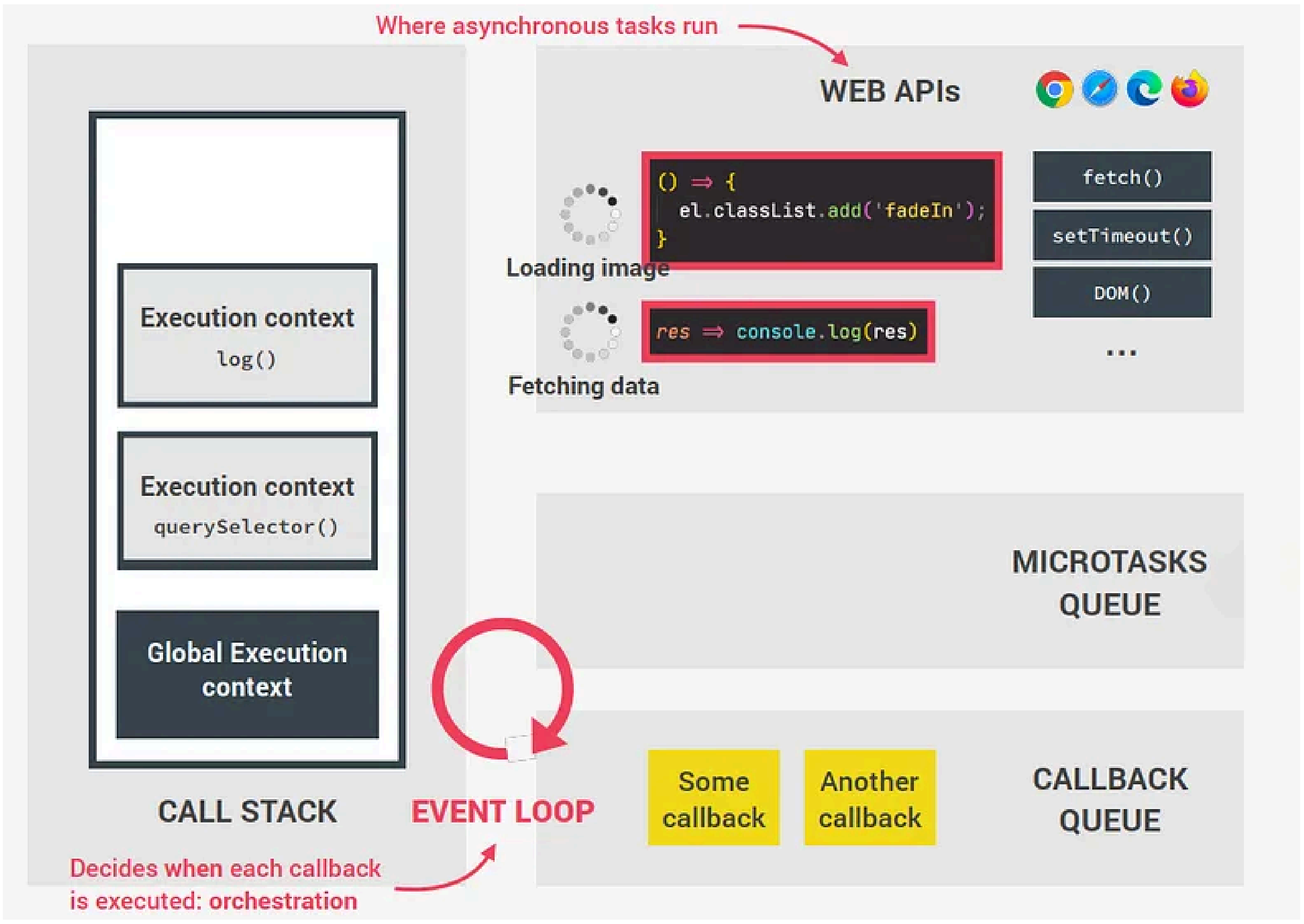


```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));
```

What exactly is this enigmatic “**background**”? As we’re aware, everything associated with the DOM isn’t truly part of JavaScript but belongs to Web APIs. Consequently, asynchronous tasks related to the DOM will operate in the Web APIs environment, **not within the JavaScript engine itself!** In fact, this holds true for timers, AJAX calls, and all other asynchronous tasks as well.

Thus, these asynchronous tasks will execute in the browser’s Web APIs environment.

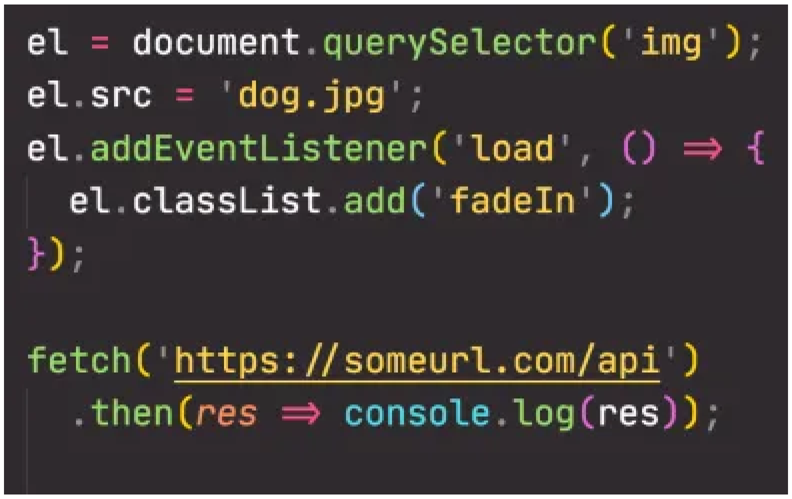


```
() => {
  el.classList.add('fadeIn');
}
```

```
res => console.log(res)
```

Should the image load within the Call Stack, it would result in obstructing the execution of the remaining code. Consequently, image loading in JavaScript is asynchronous and does not occur in the Call Stack or the Main Thread of Execution but rather in the separate Web APIs environment, as previously mentioned.

Now, if we desire to perform an action after the image completes loading, we must listen for the loading event. This is precisely what we do in the following line of code.



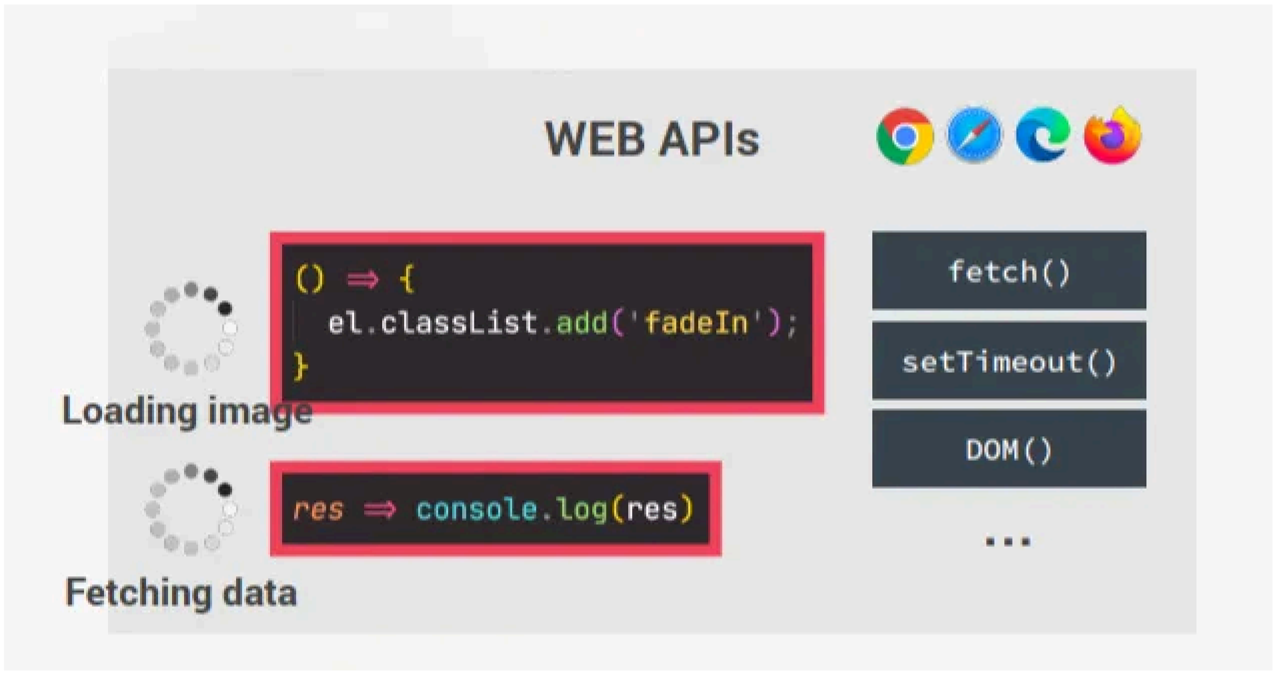
```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));
```

Here, we attach an eventListener to the load event of the image and pass it a callback function, as usual. In practice, this involves registering the callback function in the Web APIs environment, precisely where the image is being loaded, and this callback function will remain there until the loading event concludes.

In the subsequent line, we execute an AJAX call using the fetch API, and the asynchronous fetch operation also occurs in the Web APIs environment to avoid blocking the aforementioned Call Stack. Finally, we employ the ‘then’ method on the Promise returned by the fetch function, which registers a Callback Function in the Web APIs environment, allowing us to respond to the future resolved value of the Promise. This Callback Function is associated with a Promise that retrieves data from the API, a detail that will be significant later on.

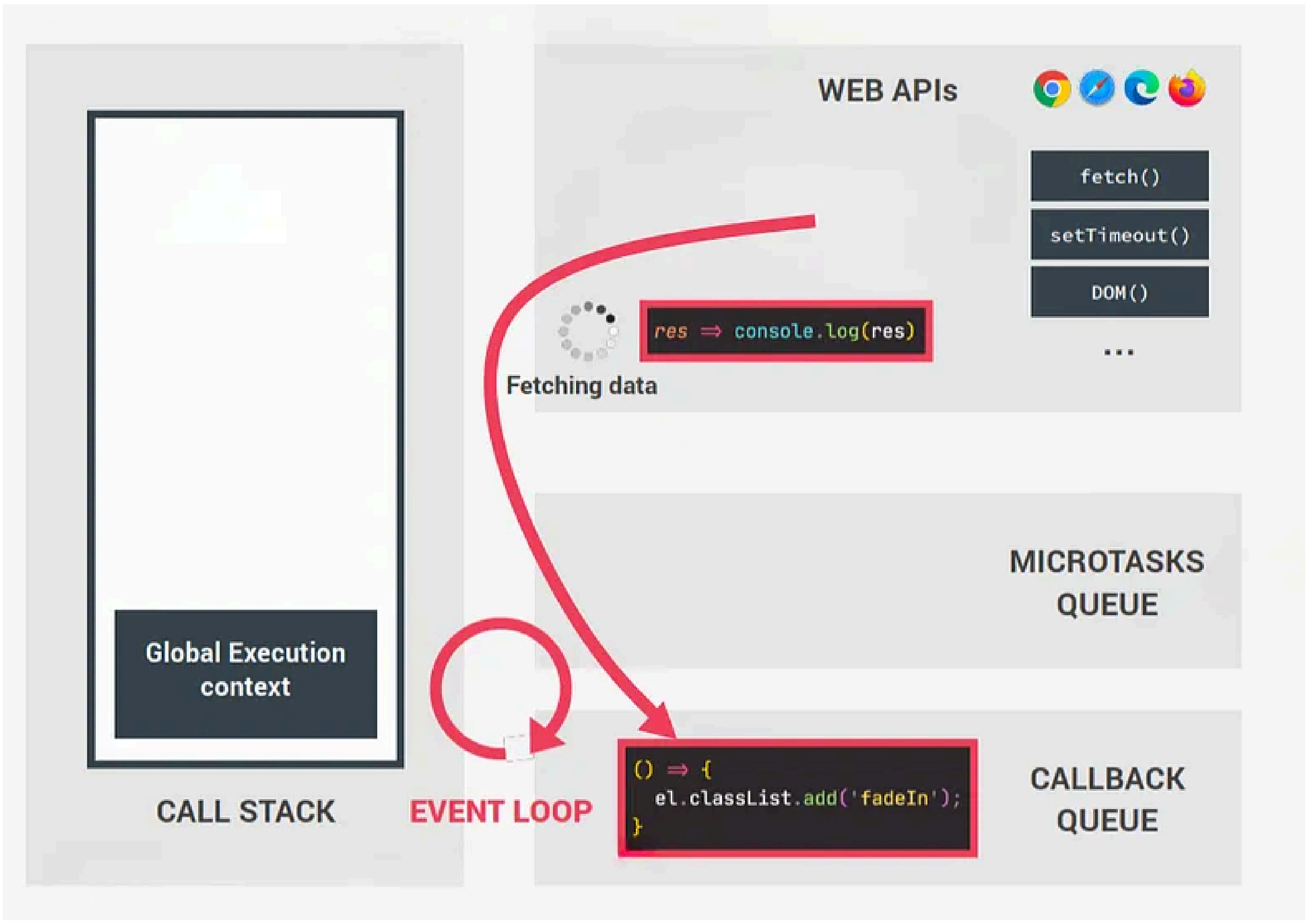
Thus, we have an image loading in the background and data being fetched from the API, all taking place within the Web APIs environment.



```
() => {
  el.classList.add('fadeIn');
}
```

```
res => console.log(res)
```

Now we’ve arrived at a particularly intriguing point. Imagine the image has completed loading, and the Load event occurs for that same image. What follows is the callback for this event being sent to the Callback Queue.



Essentially, the Callback Queue is an ordered list of all the Callback Functions waiting in line for execution. You can think of the Callback Queue as a to-do list containing all tasks to be carried out, with the Call Stack being the one to perform those tasks in the end.

In our example, there are no other callbacks in the Callback Queue, but there could be. If there were other callbacks in the queue, then the new callback would be placed at the end and wait patiently for its turn to run.

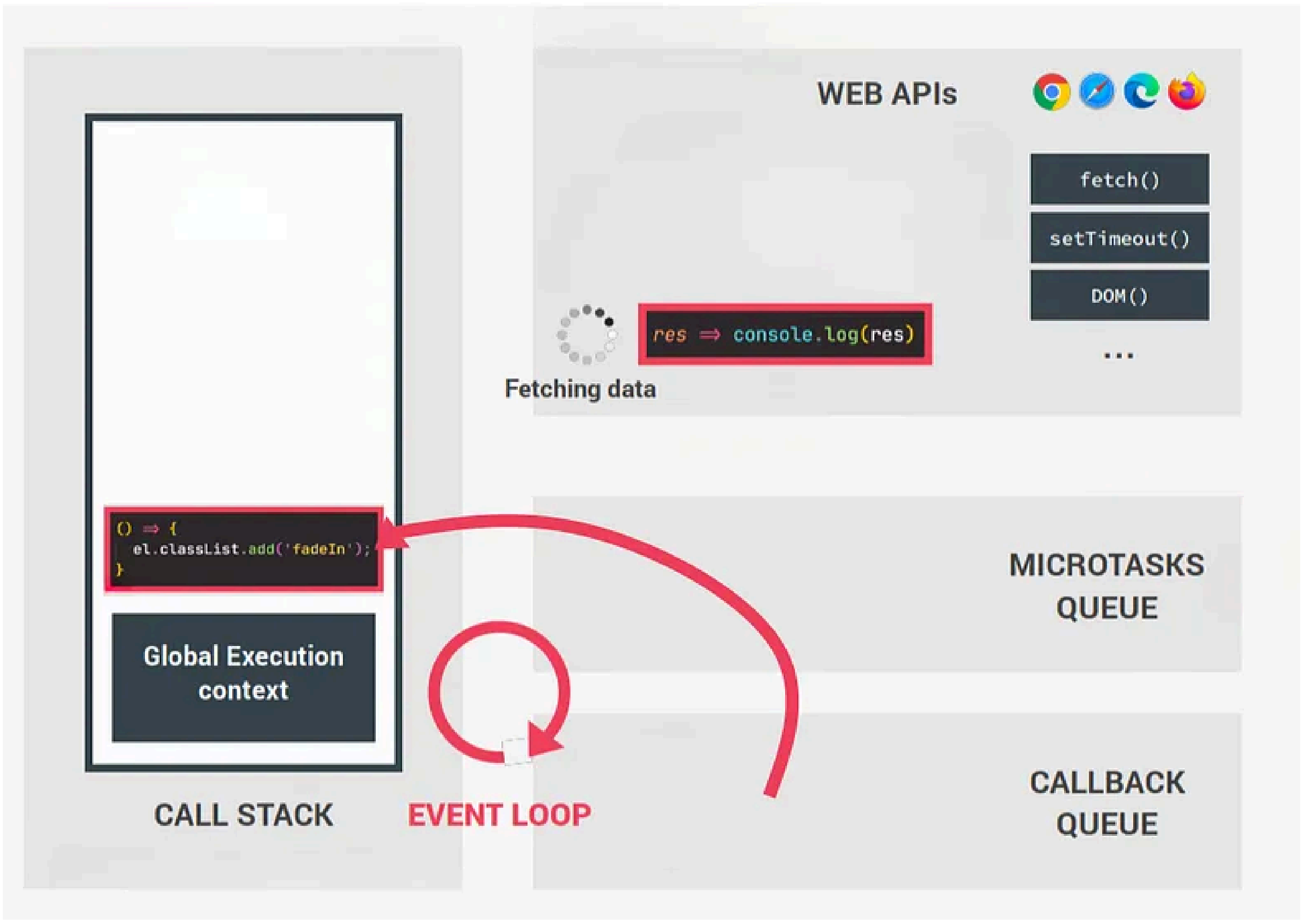
This has significant implications that are crucial to note. Consider a situation where you set setTimeOut to five seconds. After five seconds, the callback in setTimeOut will not immediately go to the Call Stack but to the Callback Queue. If there are already other callbacks in the Callback Queue waiting to be executed, and it takes one second to execute all of them, **the setTimeOut**

callback will run only after six seconds, not five! These six seconds consist of the five seconds elapsed for the timer, plus the additional second it took to execute all the other queued callbacks. **This means that the time set for setTimeOut functions is not a guarantee.** The only certainty is that setTimeOuts will not run before the specified time, but they may run after it. This all depends on the state of the Callback Queue and the Call Stack (which also needs to be clear to receive callbacks).

Another important point to mention is that the Callback Queue also holds callbacks resulting from DOM events, such as mouse clicks, key presses, or others. It's worth noting that many DOM events, like Click Events, are not asynchronous, but they still utilize the Callback Queue to trigger their callbacks. Therefore, if a button with an added Event Listener is clicked, the same process will occur as with the asynchronous loading event.

The Event Loop

Now, let's observe the Event Loop in action. The Event Loop examines the Call Stack to determine if it is empty, excluding the ever-present Global Execution Context, of course. If the Call Stack is indeed empty, indicating that no code is currently being executed, the Event Loop will take the first callback from the Callback Queue and place it onto the Call Stack.



This process is called an Event Loop Tick. Each time the Event Loop retrieves a Callback Function from the Callback Queue and places it on the Call Stack, we say an Event Loop Tick has occurred. The Event Loop plays a crucial role in coordinating the Call Stack with the Callbacks in the Callback Queue. Essentially, the Event Loop decides when each Callback Function will be executed, orchestrating the entire JavaScript runtime environment.

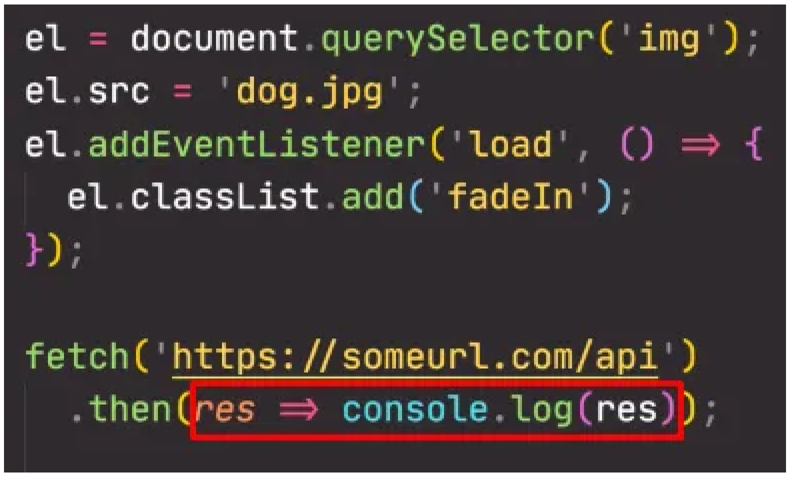
It's important to note that JavaScript itself doesn't have a sense of time since all asynchronous actions occur outside its engine. The rest of the runtime manages the asynchronous behavior, with the Event Loop determining when and where the JavaScript engine will execute the code it receives.

To summarize the process, the image began loading asynchronously in the Web APIs environment, not in the Call Stack. We then used addEventListener to attach a callback function to the image load event. The callback function represents the deferred asynchronous code, which we only want to execute after the image has loaded, allowing the rest of the code to continue running. addEventListener registered the callback in the Web APIs environment, where it waited for the loading event. Once the event occurred, the Web APIs environment moved the callback to the Callback Queue. The callback then waited in the Callback Queue for the Event Loop to transfer it to the Call Stack. This transfer occurred when the callback was first in line and the Call Stack was empty. The entire process ensures that the image loads in the background in a non-blocking manner.

In conclusion, the Web APIs environment, Callback Queue, and Event Loop work together to enable asynchronous code execution in a non-blocking way, even with only one thread in the JavaScript engine.

The Microtasks Queue

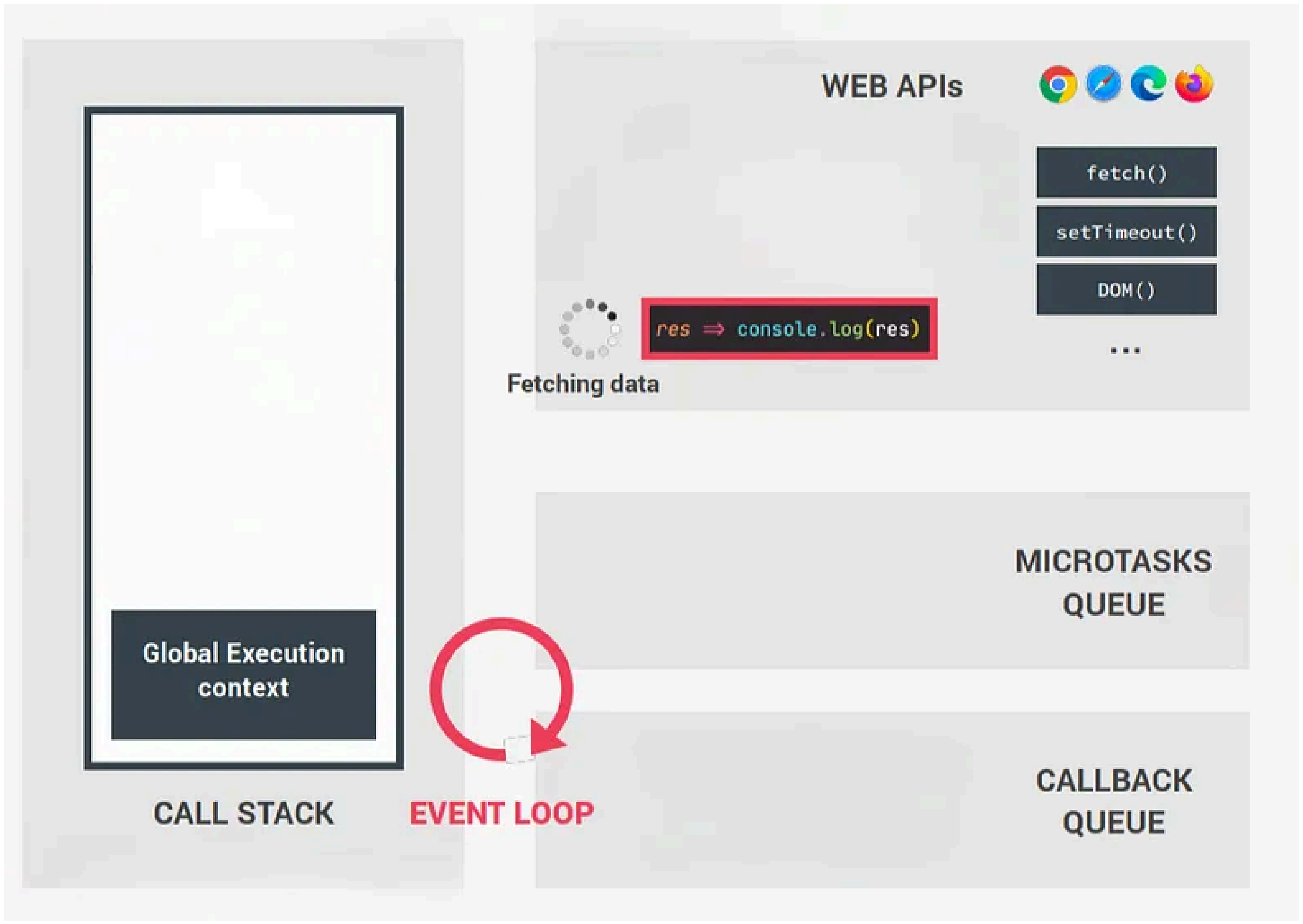
Alright, we still need to address the function waiting for the AJAX call in the background.



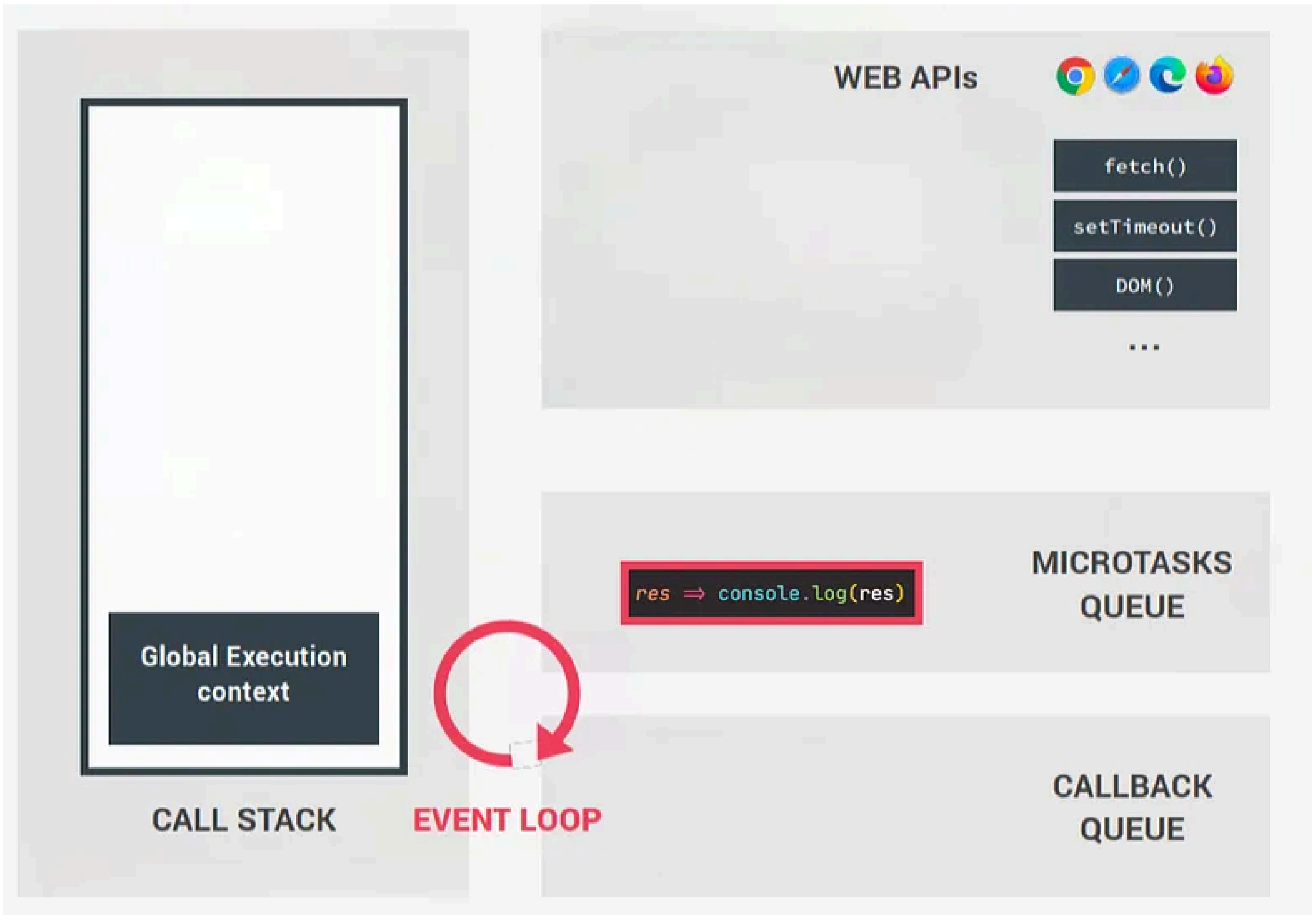
```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));
```

This is where Promises come into play. With Promises, the process works slightly differently. Suppose the data has finally arrived. The callbacks associated with Promises, such as the one we registered with the 'then' method of the Promise, don't go to the Callback Queue. Instead, Promise-related Callback Functions have their own dedicated queue, called the **Microtasks Queue**.



The Microtasks Queue has a unique feature: it takes priority over the Callback Queue. The Event Loop will first check for any callbacks in the Microtasks Queue, and if there are any, it will execute them before running additional callbacks from the Callback Queue. Promise-related Callback Functions are referred to as Microtasks, hence the name Microtasks Queue. While there are other types of Microtasks, they are not relevant at the moment. In our example, we currently have a Microtask in the Microtasks Queue, and the Call Stack is empty.



The Event Loop will now take the Microtask and place it on the Call Stack, just like it did with the Callback Function from the Callback Queue, regardless of whether the Callback Queue is empty or not. This behavior would be the same even if multiple callbacks were in the Callback Queue, because Microtasks always take precedence. In practical terms, this means that Microtasks can effectively skip ahead of regular callbacks in the queue.

If another Microtask arrives, it will also be executed before any Callback Function in the Callback Queue. This situation implies that the Microtasks Queue can potentially starve the Callback Queue, as continuously adding more Microtasks may prevent Callback Functions in the Callback Queue from ever being executed. While this scenario is unlikely to occur, it is still important to be aware of this possibility. The execution of asynchronous code with both normal callbacks and microtasks originating from promises is quite similar, with the only difference being that they enter distinct queues and the Event Loop prioritizes microtasks over normal callbacks.

In conclusion, we explored the inner workings of asynchronous JavaScript, the runtime environment, the Call Stack, Web APIs, the Callback Queue, the Event Loop, and the Microtasks Queue.

JavaScript's Non-Blocking Concurrency Model, despite operating on a single thread, has been made possible by the intricate interplay between these components. We have demonstrated how the Event Loop manages and coordinates asynchronous code execution by transferring callbacks from the Callback Queue and Microtasks Queue to the Call Stack, with microtasks taking precedence over regular callbacks. This intricate system enables JavaScript to efficiently handle multiple tasks simultaneously, ensuring a smooth and responsive user experience.

Conclusion

In conclusion, in this article we got an overview of the inner workings of JavaScript, delving into the mechanics of its engine, compilation techniques, and runtime environment. We have explored the evolution from interpretation to just-in-time (JIT) compilation, which has revolutionized JavaScript's performance by optimizing code execution based on runtime information. Furthermore, we have examined the intricacies of the JavaScript runtime environment, such as the Execution Context and the Call Stack, and their critical roles in managing code execution. The article has also shed light on the non-blocking concurrency model, detailing the Event Loop, the Web APIs environment and the Microtasks Queue as essential components in managing asynchronous JavaScript. By understanding these fundamental aspects and these core concepts of JavaScript, we can better grasp the power of asynchronous JavaScript and optimize our code to enhance performance and efficiency.

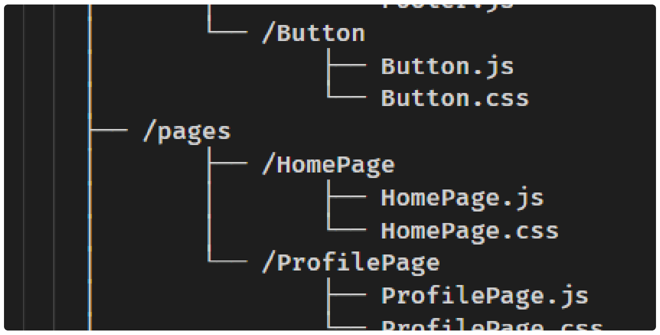
JavaScript Inner Workings Javascript Event Loop Under The Hood
Javascript Engine

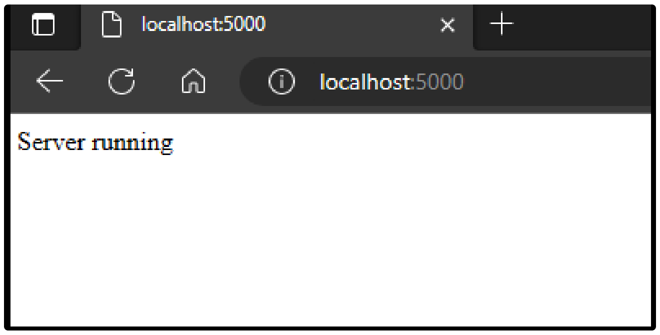
Written by Ori Baram

455 Followers · 7 Following

A programmer & instructor in JavaScript, HTML, CSS, dedicated to learning, teaching, evolving, and sharing a passion for coding with the world.

More from Ori Baram


```
        └── /Button
                ├── Button.js
                └── Button.css
    ├── /pages
        ├── /HomePage
                ├── HomePage.js
                └── HomePage.css
        └── /ProfilePage
                ├── ProfilePage.js
```



Server running