Concurrency Model and Event Loop in JavaScript

How JavaScript uses its event loop to emulate concurrency

If you've learned about asynchronous programming, you may wonder how your code can actually be non-blocking and move on to other tasks while it waits for asynchronous operations to complete. This article will remove some of the abstractions about how JavaScript can emulate concurrency by looking at what's going on with the event loop behind the scenes. But what exactly is the event loop? And why do we need it?

Why Do We Need an Event Loop?

JavaScript is a *single-threaded* language, which means that two statements can't be executed simultaneously. For example, if you have a `for` loop that takes a while to process, it'll have to finish executing before the rest of your code runs. That results in blocking code. But as we already learned, we can run non-blocking code in JavaScript, which is where the Event Loop comes in. Input/output (I/O) is handled with events and callbacks so code execution can continue. Let's look at an example of blocking and non-blocking code. Run this block of code yourself locally.

```
console.log("I'm learning about");

for (let idx=0; idx < 999999999; idx++) {}

// The second console.log() statement is
// delayed by the for loop's execution
console.log("the Event Loop");
```

Free response

What happened when you ran the code? What did you notice about the timing of the execution of yo
`console.log()` statements?

Your response

it takes a long time to run second console.log() function

Our answer

The code logged 2 lines to the console. The first line logged, then the `for` loop executed, and after some time, the last line logged to the console.

The example above has synchronous code with a long `for` loop. Here's what happens:

1. The code executes and "I'm learning about" is logged to the console.
2. Next, a `for` loop executes and runs 999999999 loops, which results in blocking code. If you run this locally, this is where the pause happens.
3. Finally, "the Event Loop" is logged.

Now let's take a look at the non-blocking example. There are functions like `setTimeout()` that work differently thanks to the Event Loop. Run the code:

```
console.log("I'm learning about");
setTimeout(() => { console.log("Event Loop");}, 2000);
console.log("the");
```

In this case, the code snippet uses the `setTimeout()` function to demonstrate how JavaScript can be non-blocking with use of the event loop. Here's what happens:

1. A statement is logged.
2. The `setTimeout()` function is executed.
3. A third line of code executes and logs text: "the".

4. Finally, the `setTimeout()` function timer completes and additional text is logged: “Event Loop”.

In this case, JavaScript is still single-threaded, but the event loop is enabling something called concurrency.

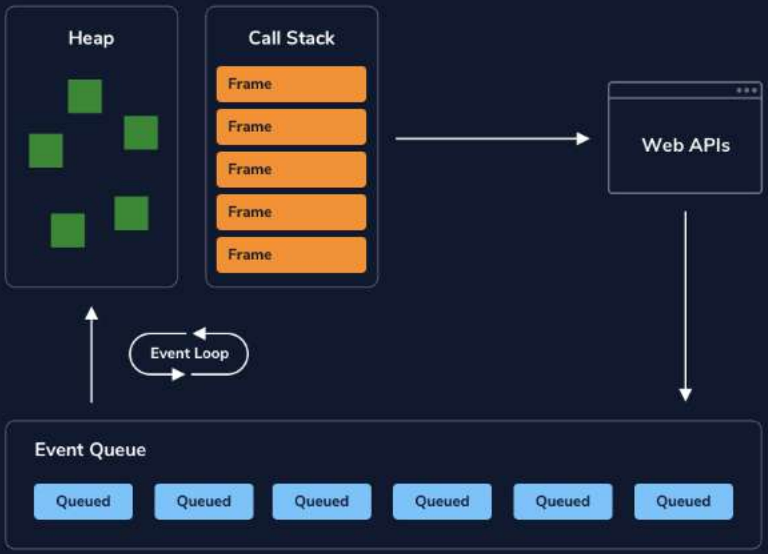
Concurrency in JavaScript

Usually when we think about *concurrency* in programming, it means that two or more procedures are executed at the same time on the same shared resources. Since JavaScript is single-threaded, as we saw in the `for` loop example, we’ll never have that flavor of “true” concurrency. However, we can emulate concurrency using the event loop.

What Is the Event Loop?

At a high level, the event loop is a system for managing code execution. In the diagram, you can see an overview of how the parts that make up the event loop fit together.

We have data structures that we call the heap and the call stack, which are part of the JavaScript engine. The heap and call stack interact with Node and Web APIs, which pass messages back to the stack via an event queue. The event queue’s interaction with the call stack is managed by an event loop. All together, those parts maintain the order of code execution when we run asynchronous functions. Don’t worry about understanding what those terms mean yet—we’ll dive into them shortly.



Note: You can click on the image to enlarge it.

Understand the Components of the Event Loop

The *event loop* is made up of these parts:

- Memory Heap
- Call Stack
- Event Queue
- Event Loop
- Node or Web APIs

Let's take a closer look at each part before we put it all together.

The Heap

The *heap* is a block of memory where we store objects in an unordered manner. JavaScript variables and objects that are currently in use are stored in the heap.

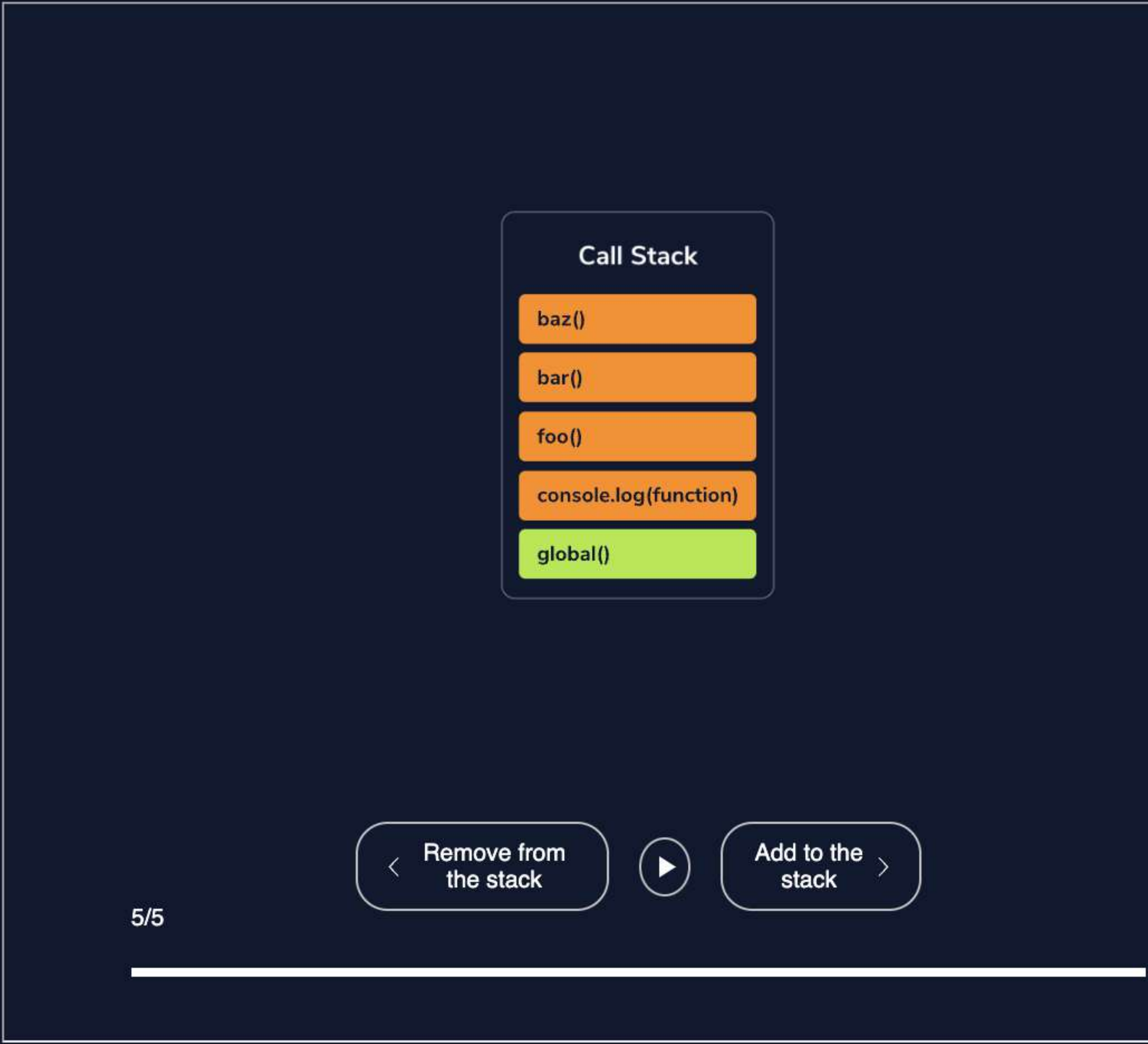
The Call Stack

The *stack*, or *call stack*, tracks what function is currently being run in your code.

When you invoke a function, a *frame* is added to the stack. Frames connect that function's arguments and local variables from the heap. Frames enter the stack in a *last in, first out* (LIFO) order. In the code snippet below, a series of nested functions are declared, then `foo()` is called and logged.

```
function foo() {
 return function bar() {
   return function baz() {
     return 'I love CodeCademy'
   }
 }
}
console.log(foo()()());
```

The function executing at any given point in time is at the top of the stack. In our example code, since we have nested functions, they will all be added to the stack until the innermost function has been executed. When the function finishes executing e.g. returns, its frame is removed from the stack. When we execute `console.log(foo()()())`, we'd see the stack build as follows:



You might have noticed that `global()` is at the bottom of the stack–when you first initiate a program, the *global execution context* is added to the call stack, which contains the global variable and lexical environment. Each subsequent frame for a called function has a function execution context that includes the function's lexical and variable environment.

So when we say the call stack tracks what function is currently being run in our code, what we are tracking is the current execution context. When a function runs to completion, it is popped off of the call stack. The memory, or the frame, is cleared.

The Event Queue

The *event queue* is a list of messages corresponding to functions that are waiting to be processed. In the diagram, these messages are entering the event queue from sources such as various web APIs or async functions that were called and are returning additional events to be handled by the stack. Messages enter the queue in a first in, first out (FIFO) order. No code is executed in the event queue; instead, it holds functions that are waiting to be added back into the stack.

The Event Queue in Context

This event queue is a specific part of our overall event loop concept. Messages that are waiting in the event queue to be added back into the stack are added back via the event loop. When the call stack is empty, if there is anything in the event queue, the event loop can add those one at a time to the stack for execution.

1. First the event loop will poll the stack to see if it is empty.
2. It will add the first waiting message.
3. It will repeat steps 1 and 2 until the stack has cleared.

The Event Loop in Action

Now that we know all of the pieces of the event loop, let's walk through some code to understand the event loop in action.

```
console.log("This is the first line of code in app.js.");

function usingsetTimeout() {
    console.log("I'm going to be queued in the Event Loop.");
}
setTimeout(usingsetTimeout, 3000);

console.log("This is the last line of code in app.js.");
```

1. `console.log("This is the first line of code in app.js.");` is added to the stack, executes, then pops off of the stack.
2. `setTimeout()` is added to the stack.
3. `setTimeout()`'s callback is passed to be executed by a web API. The timer will run for 3 seconds. After 3 seconds elapse, the callback function, `usingsetTimeout()` is pushed to the Event Queue.
4. The Event Loop, meanwhile, will check periodically if the stack is cleared to handle any messages in the Event Queue.
5. `console.log("This is the last line of code in app.js.");` is added to the stack, executes, then pops off of the stack.
6. The stack is now empty, so the event loop pushes `usingsetTimeout` onto the stack.
7. `console.log("I'm going to be queued in the Event Loop.");` is added to the stack, executes, gets popped
8. `usingsetTimeout` pops off of the stack.

Summary

Thanks to the event loop, JavaScript is a single-threaded, event-driven language that can run non-blocking code asynchronously. The Event Loop can be summarized as: when code is executed, it is handled by the heap and call stack, which interact with Node and Web APIs. Those APIs enable concurrency and pass asynchronous messages back to the stack via an event queue. The event queue's interaction with the call stack is managed by an event loop. All together, those parts maintain the order of code execution when we run asynchronous functions.