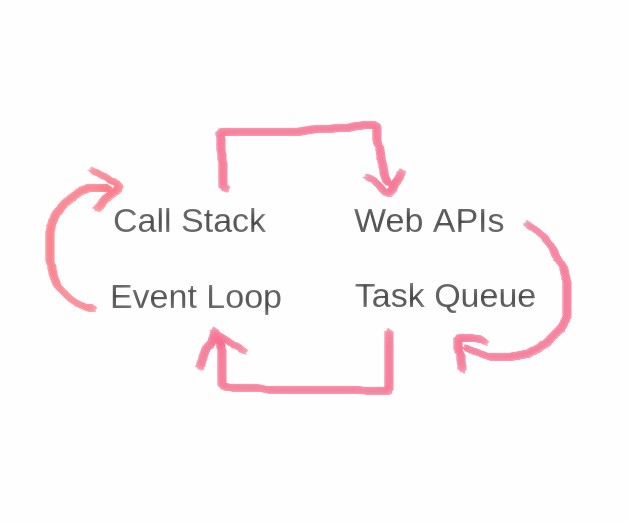
[Siddhartha Chowdhury](https://medium.com/@siddharthac6?source=post_header_lockup)Follow

Art, JavaScript and League of Legends

May 24, 2017



**JavaScript: Execution of Synchronous and Asynchronous codes**

Recently I was in a discussion with a couple of descent JS devs regarding - how JS allocates memory and how a script is parsed and executed. This is one of many (most) important topics which were never a part of our academic career, neither one needs to know to write a JS program. But topics such as these are crucial for those curious developers out there who takes things seriously. I chose to write about this topic because I find it quite ambiguous and people tend to compare things, specially those who are familiar with other programming languages like PHP, C++, Java, etc. but guys JavaScript is a queer beast and it took me quite some time to digest some important aspects of JS initially, like **HOW JavaScript being single-threaded can be non-blocking and concurrent?**

Now before we start diving deep, lets make clear the fundamental concept and difference between JavaScript Engine and JavaScript Run-time Environment.  
- JavaScript engine is a program that is used parse a given script and converts it to a machine executable instructions.   
- On the other hand JavaScript run-time environment, as the name implies is responsible for creating an ecosystem with facilities, services and supports (like array, functions, core libraries, etc.) that are required for the executable instructions to run successfully.

**THE FUNCTIONAL MODEL**

Almost all web browsers has a JavaScript engine. The most popular ones are [V8](https://v8project.blogspot.in/) in Google chrome and Node.js, [Spider monkey](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey) of Mozilla, IE’s [Chakra](https://github.com/Microsoft/ChakraCore), etc. Although all these browser vendors has implemented the JS differently but under the hood they all follows a same old model.

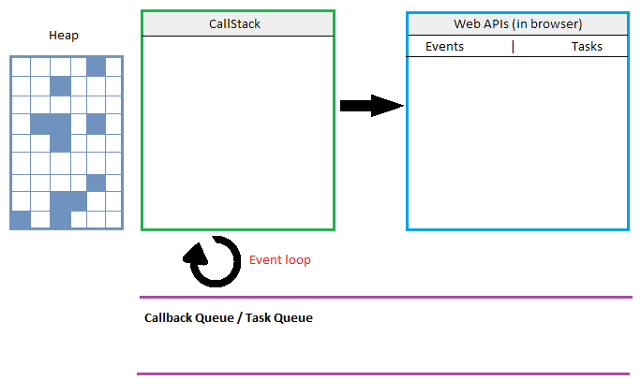


Fig: 1

Call Stack, Web APIs, Event loop, Task Queue, Render Queue, etc. We all hear these buzz terms in our day to day work. Collectively, they all work together to interpret and execute the synchronous and asynchronous blocks of codes we write everyday. Lets look deep into the model and try to understand what they does and most importantly “HOW”.

**SYNCHRONOUS Tasks**

**What does synchronous means?**Say we have 2 lines of codes **Line-1**followed by **Line-2**. Synchronous means Line-2 can not start running until the Line-1 has finished executing.

**JavaScript is single threaded**, that means only one statement is executed at a time. As the JS engine processes our script line by line, it uses this single Call-Stack to keep track of codes that are supposed to run in their respective order. Like what a stack does, a data structure which records lines of executable instructions and executes them in LIFO manner. So say if the engine steps into a function foo(){ it **PUSH-**es ***foo()*** into the stack and when the execution of ***foo()***return; } is over ***foo()*** is **POP-**ped out of the call-stack.

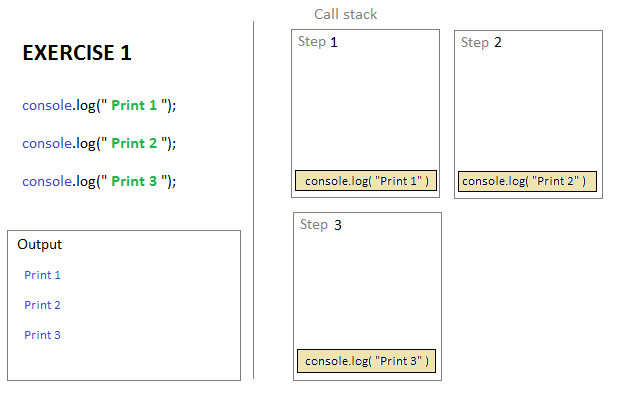


Fig: 2.1 Synchronous execution of tasks Example 1

**EXERCISE 1:** So from the above diagram shows how a typical line by line execution happens. When the script of three console.log() statements is thrown at JS —   
***Step 1***: The console.log("Print 1")is **pushed**into the **call stack** and executed, **once done** with execution, it is then **popped** out of the stack. Now the stack is empty and ready for any *next instruction* to be executed.  
***Step 2:***console.log("Print 2"); // is the next instruction is pushed and the same thing repeats until -***Step 3:*** is executed and there is nothing left to push and execute.

Lets get into our next exercise:

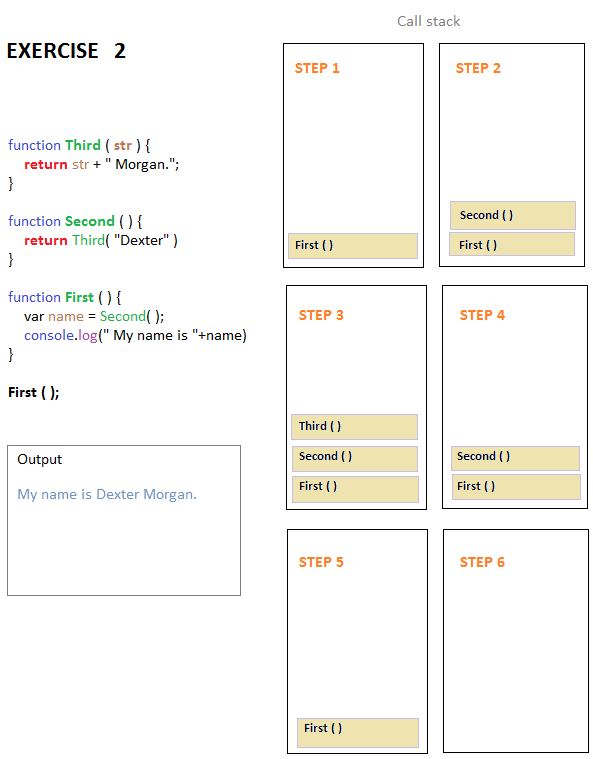


Fig: 2.2 Synchronous Execution of tasks Example 2

**EXERCISE 2:** So whats happening here is as follows:

**Step 1:** Call stack is pushed with the first executable statement of our script the First() function call. While executing through the scope of function First() our engine encounters another function call Second()

**Step 2:** Hence the function call Second() is pushed into the call stack and engine starts executing Second()function’s body *(****Note****: The functionFirst()is still not finished)* , again, there’s another function call Third()inside Second()'s body.

**Step 3:** Likewise the function call Third() is pushed into the call stack and the engine starts processing Third() function definition. While the functions Second() and First()still living in stack waiting for their turn (successor to finish their execution) respectively.

**Step 4:** So when the engine encounters a return ; statement within the function definition of Third() , well that’s the end of Third() . Hence Third() is popped out of the call stack as it has finished execution. At this point engine is back at executing Second() ‘s offerings.

**Step 5:** Well as the engine encounters a return ; statement, the function Second() is popped out and engine starts executing First() . Now there’s no return statement within the First() ‘s scope so the engine executes its body until the end of scope and pops First() out of the stack at **Step 6.**

That’s how a script of synchronous tasks gets handled by our browser without involving anything other than the “legendary” Call Stack. But things gets a little bit complex when JS engine encounters an asynchronous task.

**ASYNCHRONOUS Tasks**

**What does asynchronous means?**Unlike synchronous, asynchronous is a behavior. Say if we have two lines of code Line-1 followed by Line-2. Line-1 is a **time consuming** instruction. So Line-1 starts executing its instruction in the background (like a daemon process), allowing Line-2 to start executing without having to wait for Line-1 to finish.

We need this behavior when things are slow. Synchronous executions of code may seem straightforward but can be **slow.**Tasks like image processingcan be slow, file operations can be really slow, making network request and waiting for response is definitely slow, making huge calculations like over a 100 million for-loop iteration is somewhat slow. So such slow things in Call stack results in “**Blocking**”. When call stack is blocked, browser prevents user’s interrupts and other code statements from executing until the blocking statement is executed and call stack is freed. Hence **Asynchronous callbacks** are used to handle such situations.

**Example**: Well the function setTimeout() is probably the simplest and easiest way to demonstrate asynchronous behavior.

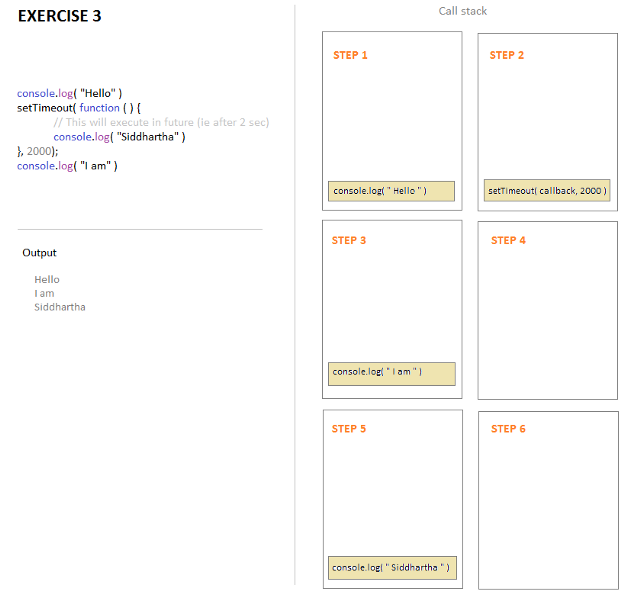


Fig: 3.1 SetTimeout exampl

**EXERCISE 3:** If we trace the call-stack we can see:  
at ***Step 1***: As usual console.log("Hello ") gets pushed into the stack first and is executed then popped out when done.  
***Step 2:***setTimeout() gets pushed into the stack, but Note- console.log("Siddhartha") can not be executed (or pushed to the stack) immediately because it is meant to execute in some future time (i.e. after **minimum**of 2 seconds). So it disappears for now (will explain later where it disappears to).  
***Step 3:***Naturally next line console.log(" I am ") is pushed into the stack, gets executed and is popped out immediately.  
***Step 4:*** Call stack is empty and waiting.  
***Step 5:***Suddenly console.log( "Siddhartha" ) is found pushed into the stack after 2 seconds as setTimeout() has timed out. It is then executed and once done is popped out of the Stack at ***Step 6:*** Stack is empty again.

*Hence it proves, even though JavaScript is single threaded, we can achieve concurrency through asynchronously handling the tasks.*

Now we are left with three important questions:  
**Question 1.***WHAT happened to thesetTimeout() ?*  
**Question 2.** *From WHERE did it came back ? And*  
**Question 3.** *HOW did it happen?*

So this is where **Event Loop, Callback Queue and Web APIs** (in browser) kicks in. Lets introduce each of the above pieces and answer the above 3 questions through our next diagram.

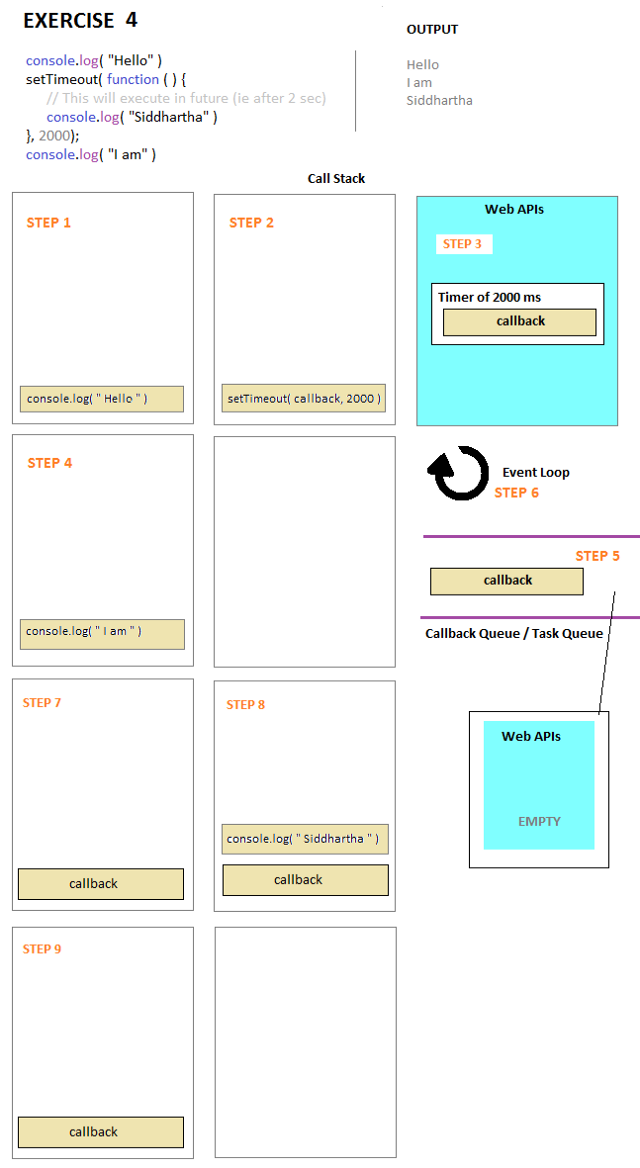


Fig: 3.2 The final demonstration of the entire model

**EXERCISE 4:** Lets jump right into

***Step 2:*** At this point setTimeout(callback, 2000) gets pushed into the call stack. As we can see it has to components a callback and a delay of 2000ms . Now setTimeout() is NOT a part of any JavaScript engine, it’s in fact a**Web API**included in the browser environment as extra feature.

***Step 3:***So the **browser Web API** takes responsibility of the callbackprovided and fires up the timer of 2000 ms leaving behind setTimeout()statement which has done its job, so is popped out of the stack.*[ Question 1 is answered]*

***Step 4:*** The next line in our script console.log( "I am" ) is pushed into the stack and popped out after its execution.

***Step 5:*** Now we have a callback in the WebAPIs which is going to get triggered after 2000 ms . But **WebAPIs directly can not PUSH things randomly into the call-stack**, because it might create an interrupt to some other code being executed by the JavaScript engine at that moment. So instead the callback is inserted into the **Callback Queue/Task Queue**after the timer of 2000 msis over. WebAPI is now empty and freed

***Step 6:*** **Event Loop —***it is responsible for taking out the first element from the Callback/Task Queue and PUSH it into the Call-Stack only when the stack is empty or free* , so at this point of our equation, the Call-Stack is empty.

***Step 7:*** So callback is pushed into the Call-Stack (as it was free and empty) from the **Callback/Task** **queue**by the **Event Loop** , and callback is executed. *[Question 2 is answered]*

***Step 8:***So another executable statement console.log("Siddhartha") is found inside the callback ‘s scope, therefore console.log("Siddhartha") is pushed into the Call-Stack

***Step 9:*** Once console.log("Siddhartha") is executed, it is then popped out of the Call-Stack, and JavaScript engine comes back to finish executing the callback ‘s remaining body. Which when done, callback is popped out of the Call-Stack. The End of the story *[Answers the “HOW”, i.e Question 3]*.

Well this was one very easy demonstration, but things gets messy and complex with several situations like when there are multiple setTimeout()’s getting queued — results differs than what’s normally expected (This is another exciting topic to discuss). I don’t know how accurately I am able to demonstrate this topic but there are a lot in between that could have been elaborated or explained better, but am in a rush and it’s already too long already. Hope it helps someone.