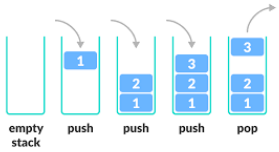
Asynchronous JS and Event Loop:

Sunday, November 6, 2022 6:52 PM

- As JS is single threaded language .
- It has 1 call stack and can perform single operation at a time.
- Call stack present inside the JS engine.
- Let's take an example and determine how call stack works

- **Line 1:** function a() will allocate memory and this function will be store.
- **Line 5**: This is function invocation, EC (execution context) is created for a() to execute the code and it pushed inside the call stack.
- **Line 2**: It will log string to the console
- **Line 3:** JS will reach to this line and it knows there's nothing more to execute inside a(), so it pop out the execution context out of call stack.
- **Line 6**: Now the control move out to line 6 and it execute the console.log and print to the console.
- Now we don't have anything else to execute so our GEC will pop out the call stack.

```
1  function a(){
2    console.log("a");
3  }
4
5  a();
6  console.log("End");
7
8
```



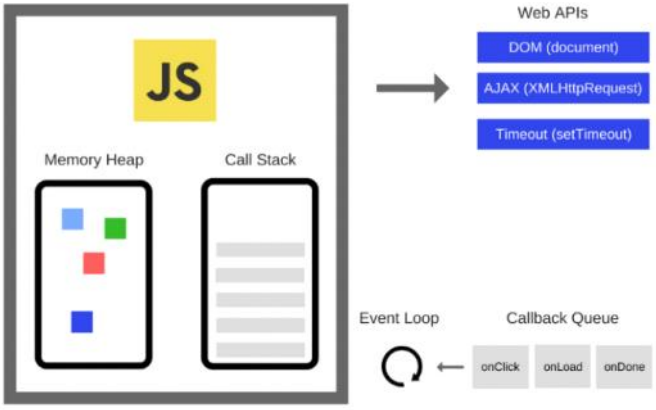
Call stack works on LIFO (last in first out) data structure. Whenever any JS program runs, GEC (global execution context) is created and push inside the call stack.

- GEC => It runs whole JS code line by line.

Q: What happens when you have function call in the Call Stack that take a huge amount of time to be processed.

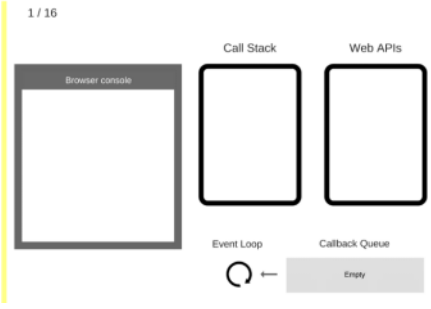
Ans: As we know JS engine doesn't wait for anything whatever comes inside it, it immediately execute it otherwise it will block the main thread and your API is no longer efficient and your application is stuck. For this we provide asynchronous behaviors using callback functions.

```
1  console.log('Hi');
2  setTimeout(function cb1() {
3      console.log('cb1');
4  }, 5000);
5  console.log('Bye');
```



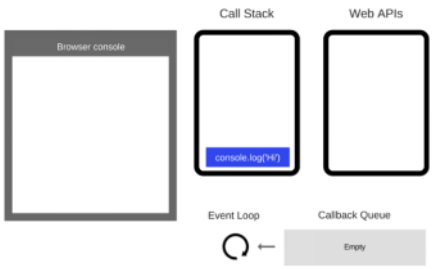
Let's execute this function and see how this works:

STATE 1: The state is clear. The browser console is clear and the call stack is empty.



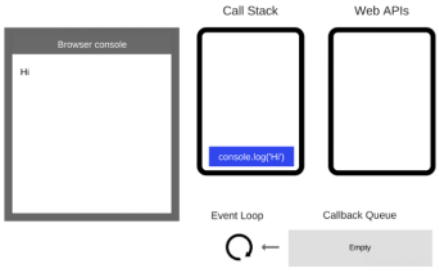
STATE 2: `console.log('Hi');` is added to the call stack.

2 / 16



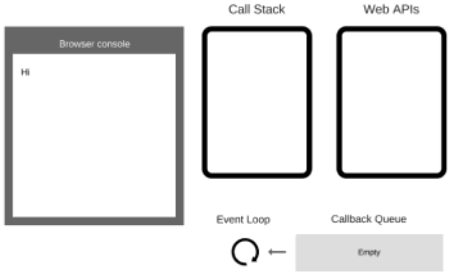
STATE 3: `console.log('Hi');` is executed.

3 / 16



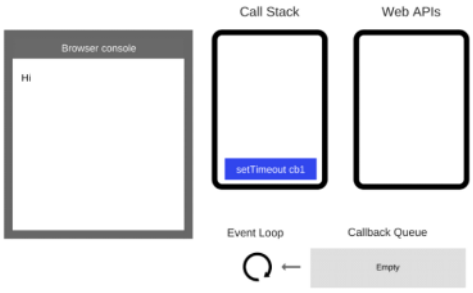
STATE 4: `console.log('Hi');` is removed from the call stack.

4 / 16



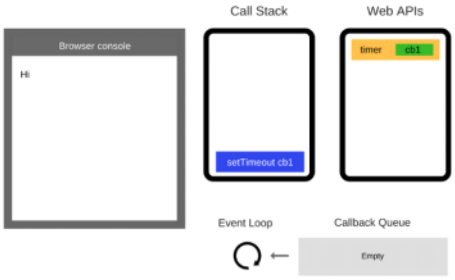
STATE 5: `setTimeout(function cb1() { … }, 5000);` is added to the call stack.

5 / 16



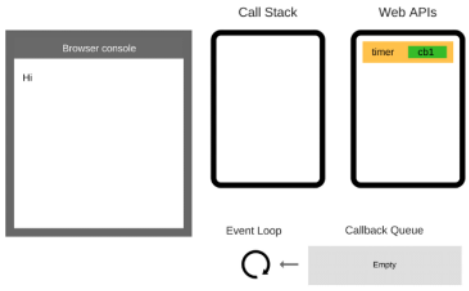
STATE 6: `setTimeout(function cb1() { … }, 5000);` is executed. The browser creates a times as part of the Web APIs. It is going to handle the countdown for you.

6 / 16



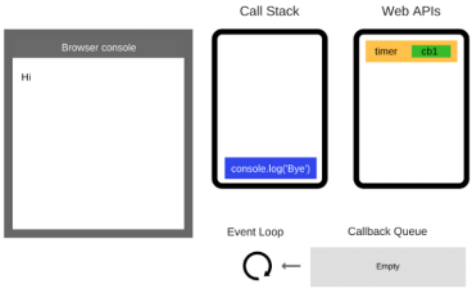
STATE 7: `setTimeout(function cb1() { … }, 5000);` itself is completed and is removed from the call stack.

7 / 16



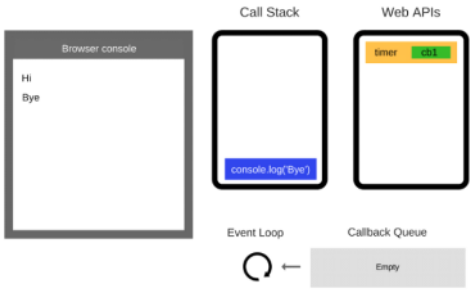
STATE 8: `console.log('Bye');` is added to the call stack.

8 / 16

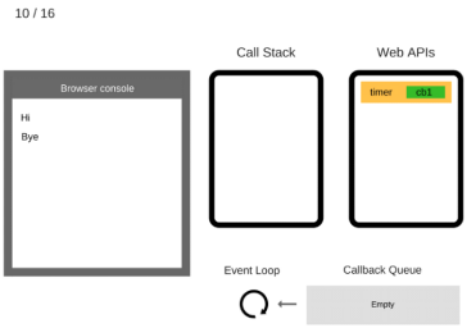


STATE 9: `console.log('Bye');` is executed.

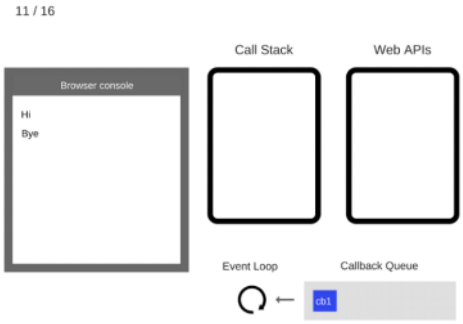
9 / 16



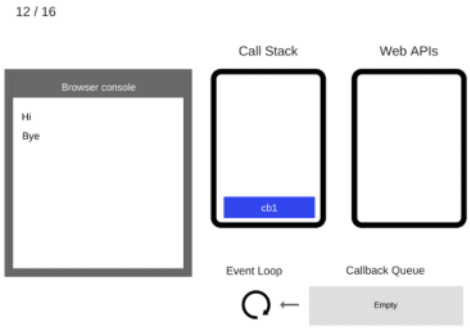
STATE 10: `console.log('Bye');` is removed from the call stack.



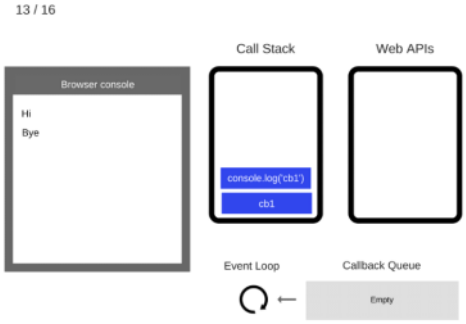
STATE 11: After at least 5000ms the timer completes and it pushes the cb1 callback to the callback queue.



STATE 12: The Event Loop takes cb1 from the Callback Queue and pushes it to the call stack.

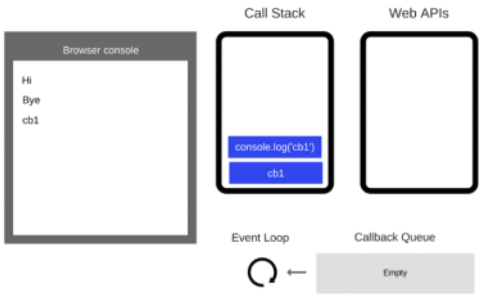


STATE 13: cb1 is executed and adds `console.log('cb1');` to the call stack.



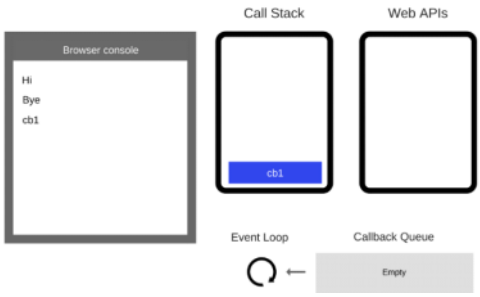
STATE 14: `console.log('cb1');` is executed.

14 / 16



STATE 15: console.log('cb1'); is removed from the call stack

15 / 16



STATE 16: cb1 is removed from the call stack.

16 / 16