alert + setTimeout

- Does alert() block the timer?
- Google returns discussions 10+ years ago; discussions were inconsistent; some said alert() blocks setTimeout() but some said it could be browser dependent

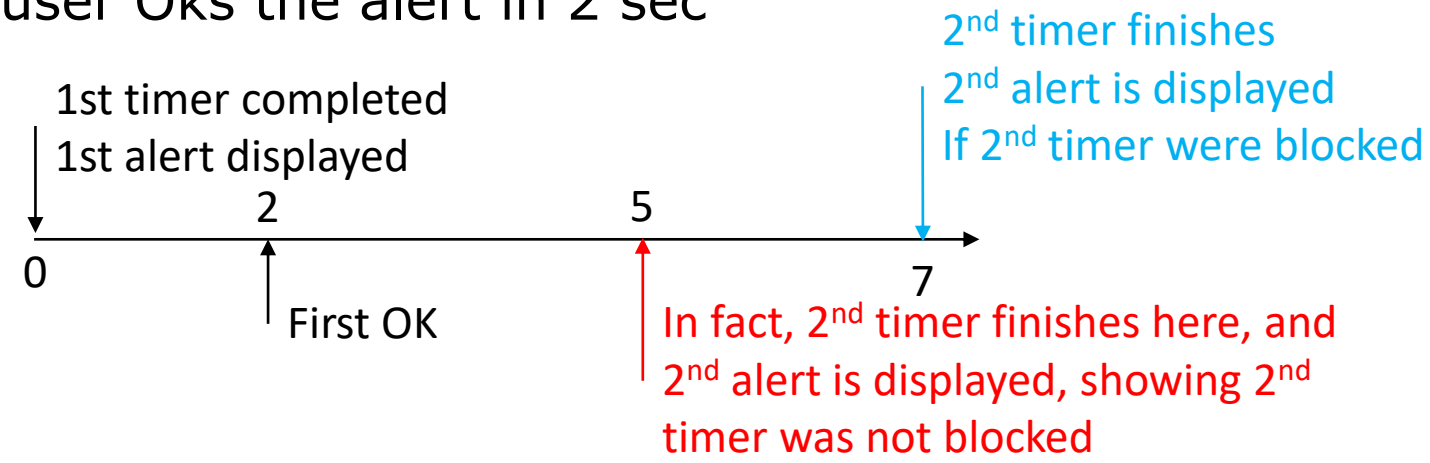
- Now, alert() no longer blocks thanks to a consistent Web API architecture; but Safari was reported to block timer when alert box is displayed

alert() does not Block Timer

- setTimeout("alert('First Alert')", 0);
  setTimeout("alert('Second Alert')", 5000);
- When the 1st alert is displayed, click OK after 10 sec
- If 1st alert blocks the 2nd timer, then the 2nd alert will be displayed after 5 sec
- In fact, the 2nd alert is displayed immediately
- Conclusion:
  - 2nd timer runs while first alert() is running
  - When the 2nd timer is done, the 2nd alert box cannot be displayed (i.e., blocked) if the 1st alert is still waiting for "OK"

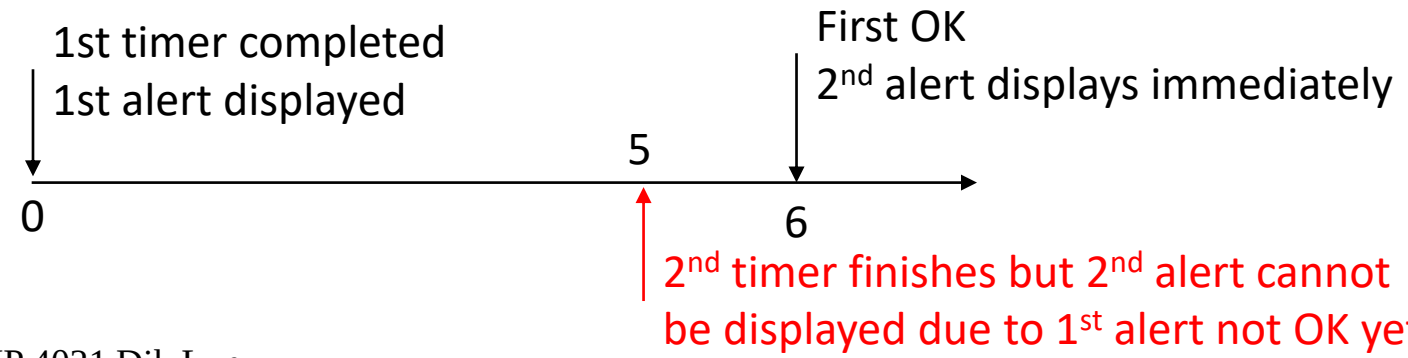
Timing Diagrams

- If user Oks the alert in 2 sec



- If user Oks the firstalert in 6 sec

Demo

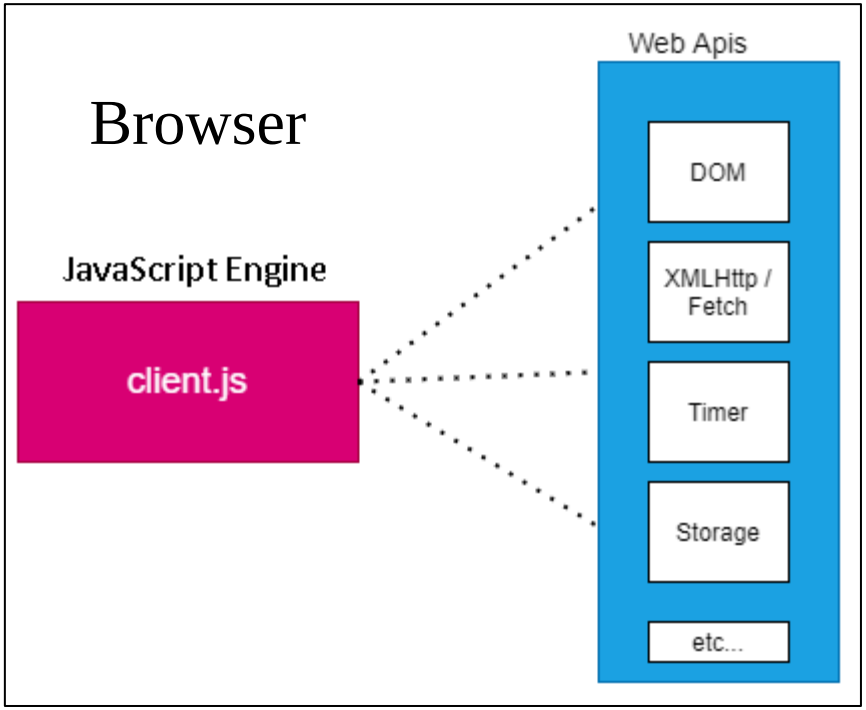


Running Timers in Parallel

- We know now that two timers can run at the same time, and, in fact, they can run simultaneously with the main JS thread
- But JavaScript is single thread; how can the timers be run in parallel?
- Web API comes to the rescue

What is Web API?

- Web API refers to the functions executed by the browser (not JS engine)
- SetTimeout() calls are executed by the browser, not by JS engine
- Multiple timers can be run on the browser

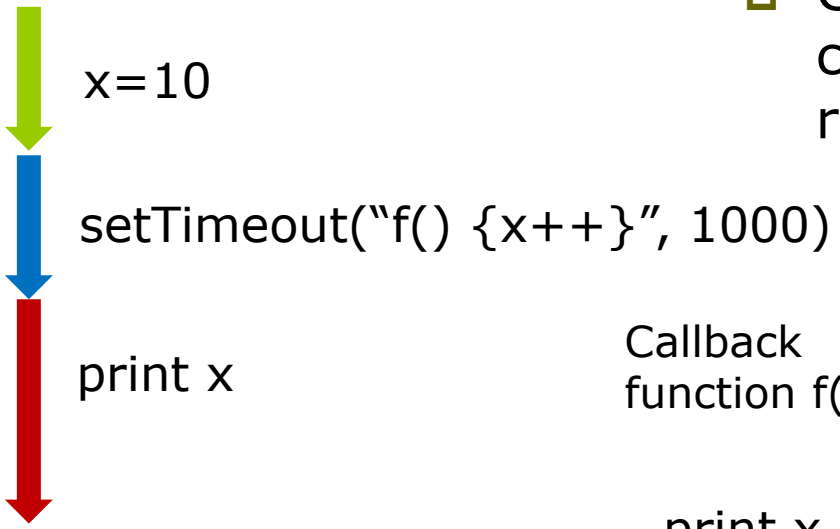


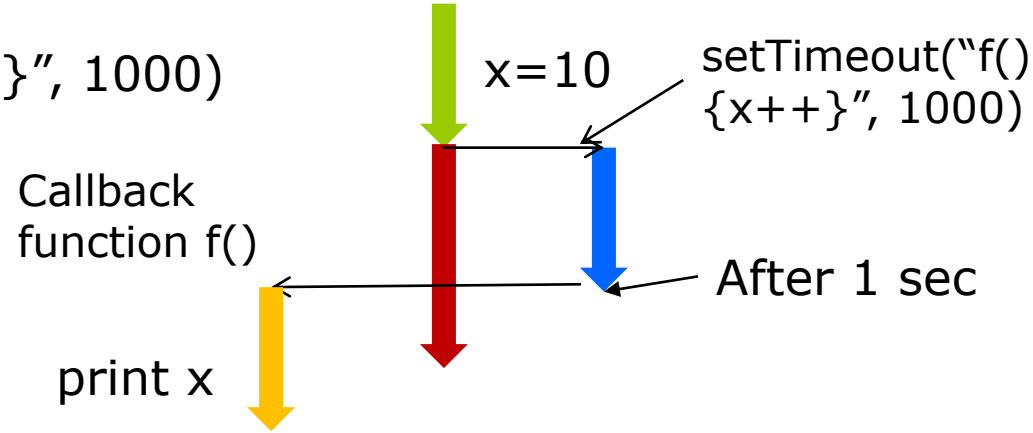
alert() Blocks JavaScript Thread

- JavaScript is single threaded (up to now)
- alert() blocks the thread, and the whole page halts; the only action allowed is to click "OK"
- Some people say this is desirable because it forces the user to focus
- If you do not want blocking (e.g., video continues to play), use DIV to emulate the alert box
  - The displayed message and style can be customized
  - jQuery provides customizable "alert" box

Synchronous vs Asynchronous Execution

- Synchronous operation: the thread executes setTimeout() and when it is done continues with the next operation
- Asynchronous operation: JS thread asks Web API to perform setTimeout(…)
- JS thread continues with next operations
- Calling process provides a callback function to "receive" the result





Alert() in Single-Threaded JavaScript

- The timer is run "outside" the JavaScript thread as an asynchronous operation
- "…functions like setTimeout and setInterval are not part of the ECMAScript specs or any JavaScript engine implementations. Timer functions are implemented by browsers and their implementations will be different among different browsers."

  In "JavaScript Timers: Everything you need to know" (2018)
- This explains why timers are not blocked by alert()
- You may say: timers are run asynchronously by browser

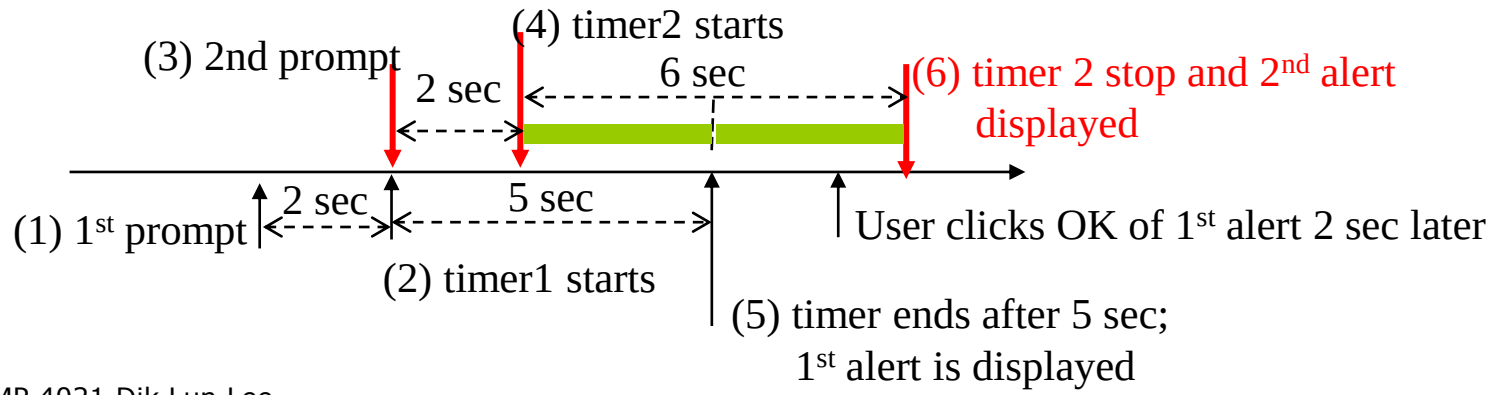
Fill in the Timeline

```
function set_things_up() {
        wait_duration=prompt("How long would you like to sleep?", "");       // (1)
        timer1=setTimeout("show_wake_up_message()", wait_duration );  // (2)

        wait_duration=prompt("How long until your next lecture?", "");       // (3)
        timer2=setTimeout("show_lecture_message()", wait_duration );  } // (4)

function show_wake_up_message() {
        alert("WAKE UP! WAKE UP! WAKE UP!!");   }     // (5)
function show_lecture_message() {
        alert("GO TO LECTURE! GO TO LECTURE!");   } // (6)
```

- The first timeout triggers an alert() blocking execution until "OK" is clicked
- Question: If timer1 is 5 sec and timer2 is 6 sec, user responds to a prompt in 2 sec, can you put 1-6 above on the following timeline?



Run Example using console.log()

```
<script>
function set_things_up() {
   startTime=Date.now();
   wait_duration=prompt("How long would you like to sleep?", "");
   console.log((Date.now()-startTime)/1000);
   timer1=setTimeout("show_wake_up_message()", wait_duration*1000 );
   wait_duration=prompt("How long until your next lecture?", "");
   console.log((Date.now()-startTime)/1000);
   timer2=setTimeout("show_lecture_message()", wait_duration*1000 );  }
function show_wake_up_message() {
        console.log((Date.now()-startTime)/1000);
        alert("WAKE UP! WAKE UP! WAKE UP!!");   }
function show_lecture_message() {
        console.log((Date.now()-startTime)/1000);
        alert("GO TO LECTURE! GO TO LECTURE!");   }
</script>
```

Run code in TryIt

Output:

```
2.025
4.365
7.028
10.366
```

Execution Order (is counter intuitive)

```
<script>
setTimeout("console.log('1')", 0);
setTimeout("console.log('2')", 0);
console.log('3');
</script>
```

What is the output?

Demo

```
<script>
setTimeout("console.log('1')", 0);
setTimeout("console.log('2')", 0);
var start = Date.now();
while (Date.now() < start + 3000) {};
console.log('After busy wait');
console.log('3');
</script>
```

What is the output?

Demo

```
After busy wait
3
1
2
```

The two setTimeout() were executed outside the main thread. When finished, the two console.log() are queued for the main thread until the third console.log() is finished

Sleep (or Stupid) Sort

```
<script>
var dataSort="";
function appendNum(num) {
dataSort = dataSort + " " + num;
console.log(dataSort);
}

var data=[100, 5, 2, 9];
for (i=0; i<data.length; i++) {
setTimeout(appendNum, data[i], data[i]);
}
</script>
```

What does this script do?

Run it on TryIt

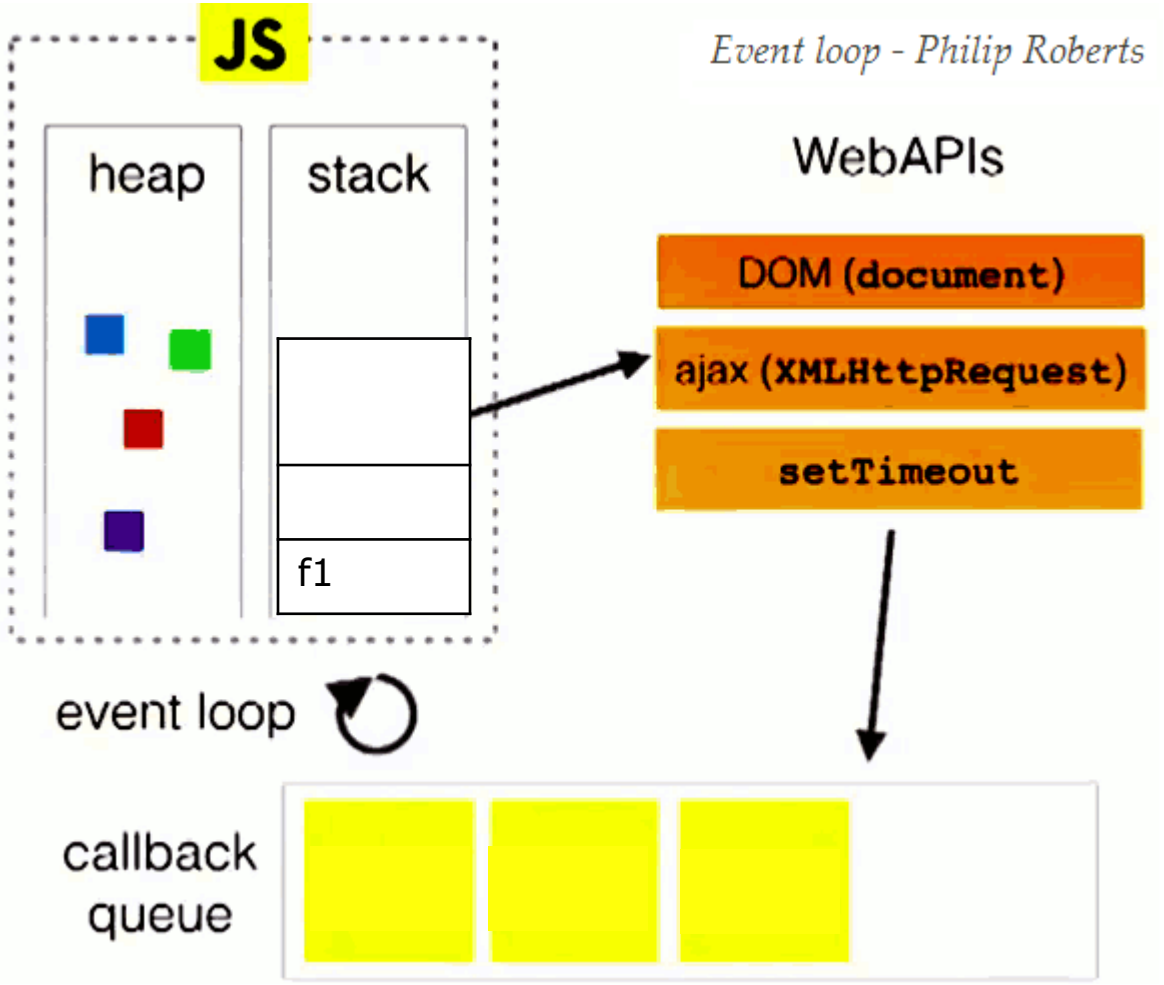
Call Stack and Event Queue

- Like other programming languages, functions in JavaScript are executed within their own contexts
  - A context contains variables, objects, and functions the function has access to
- Whenever a function is called, its Function Execution Context is pushed to the Call Stack
- When a function calls another function, the new function's Function Execution Context is pushed to the call stack
  - ↑ These are basic programming language concepts (not confined to JavaScript)
- All events are queued in Event Queue for JavaScript engine to execute one by one in an Event Loop
  - ↑ This is important JavaScript concept

Event Loop, Call Stack,and Callback/Event Queue

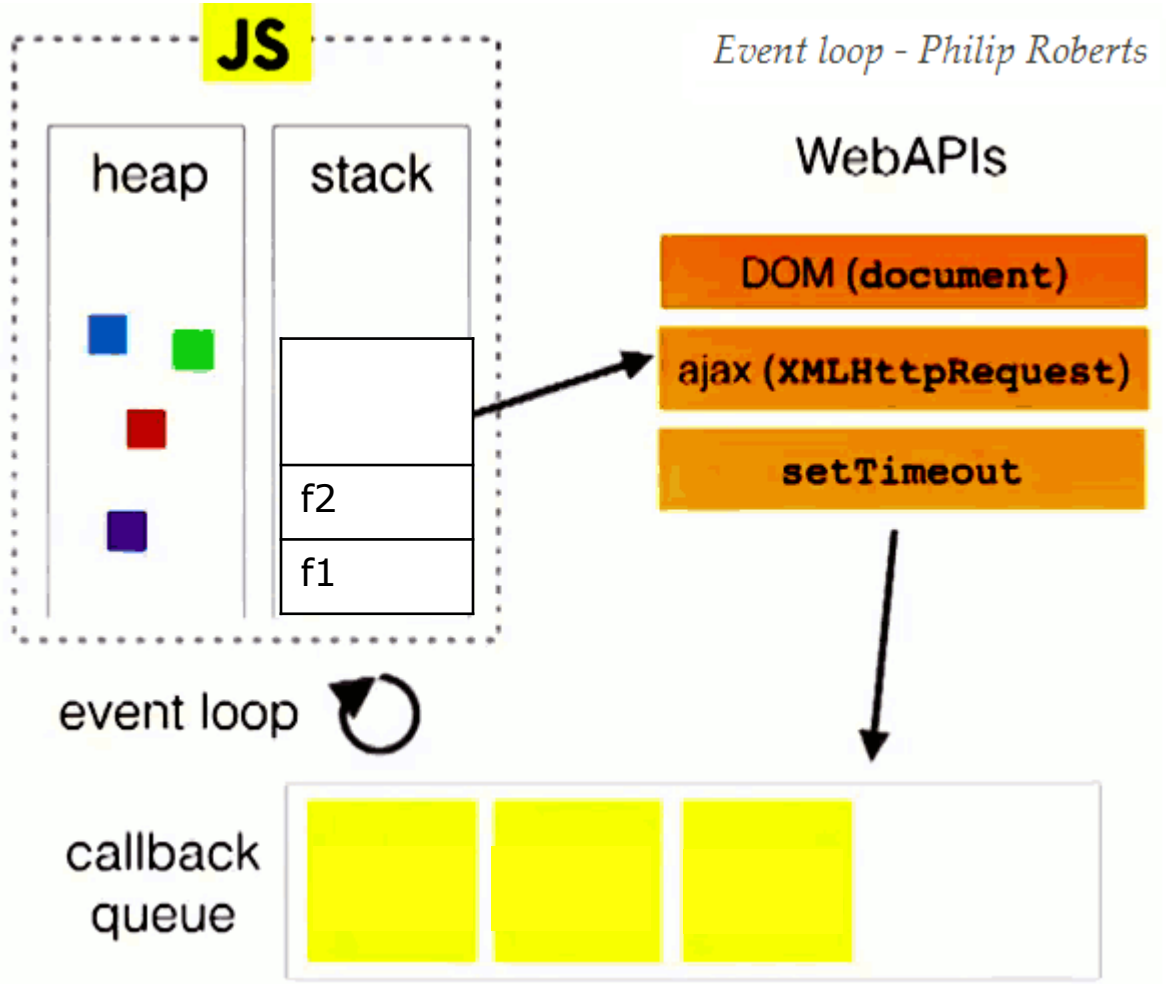
```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }

f3() { … }
```



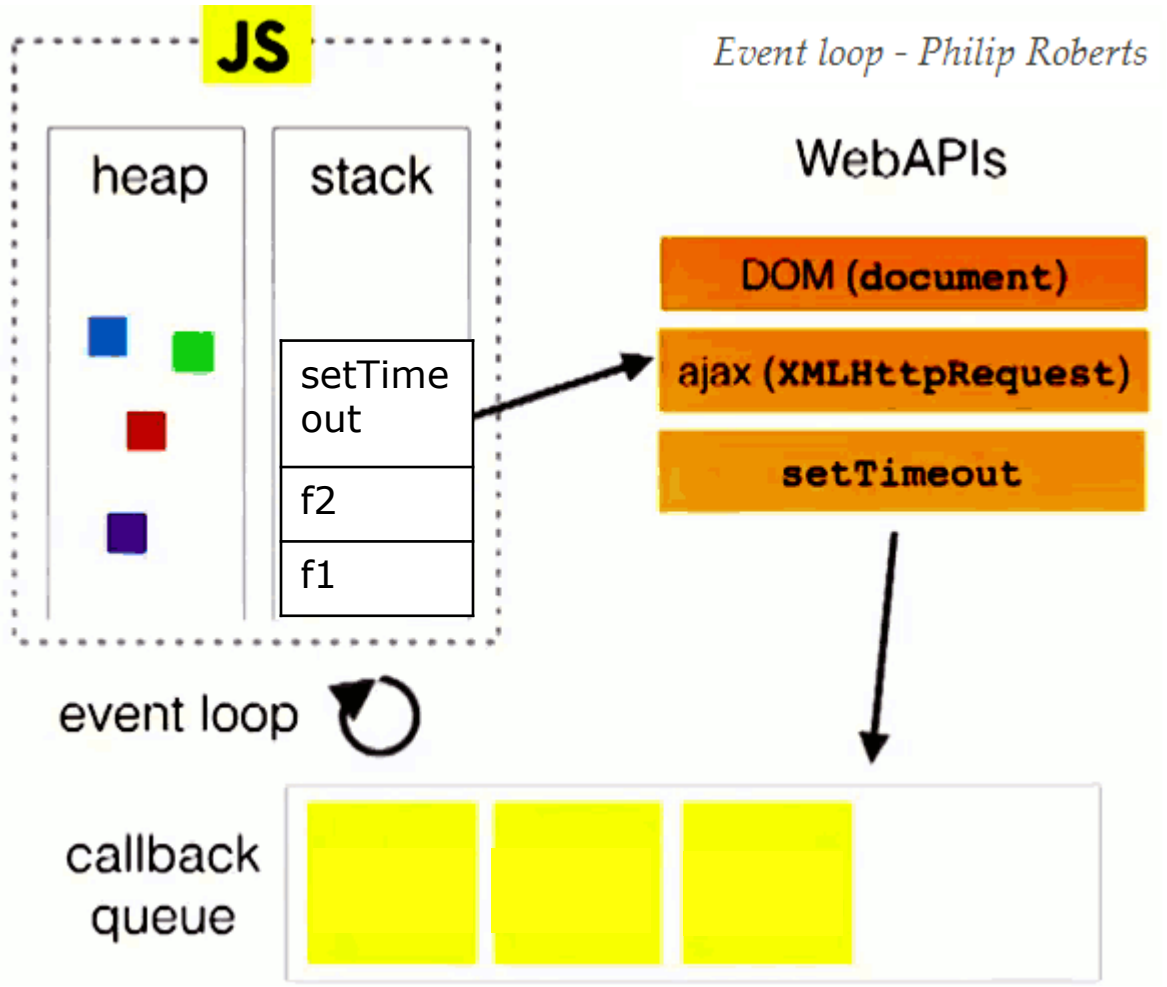
Event Loop, Call Stack,and Callback/Event Queue

```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```



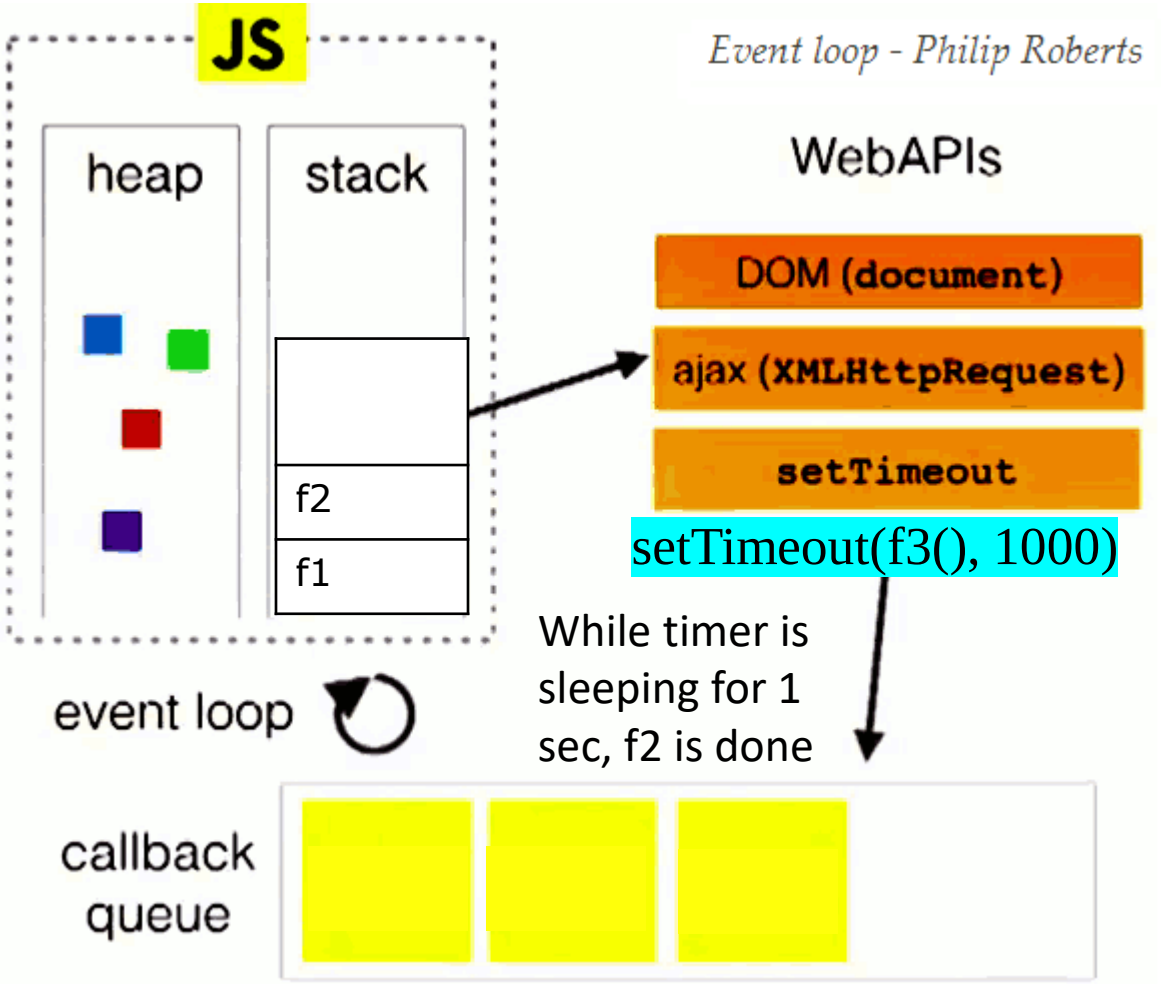
Event Loop, Call Stack,and Callback/Event Queue

```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```



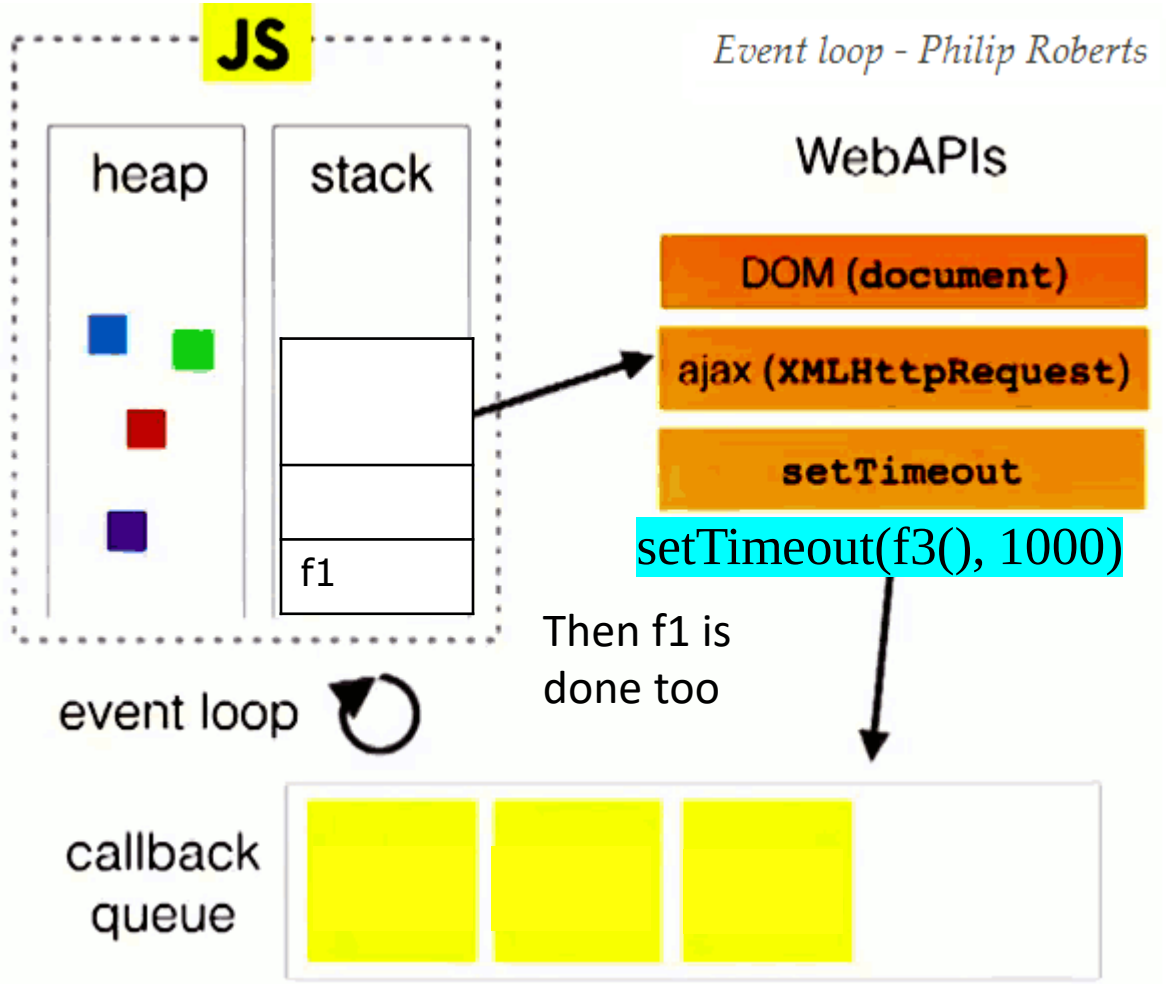
Event Loop, Call Stack,and Callback/Event Queue

```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```



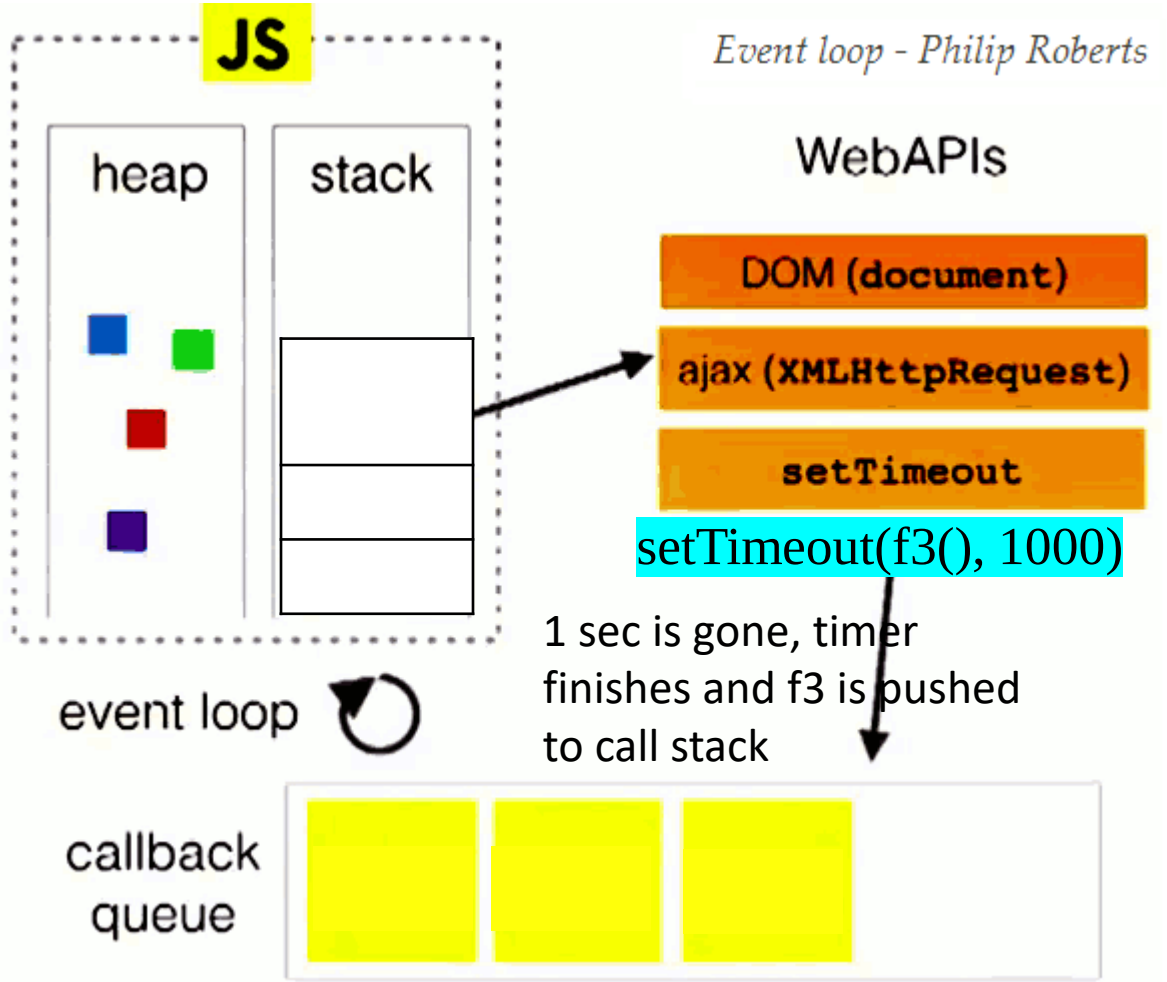
Event Loop, Call Stack,and Callback/Event Queue

```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```



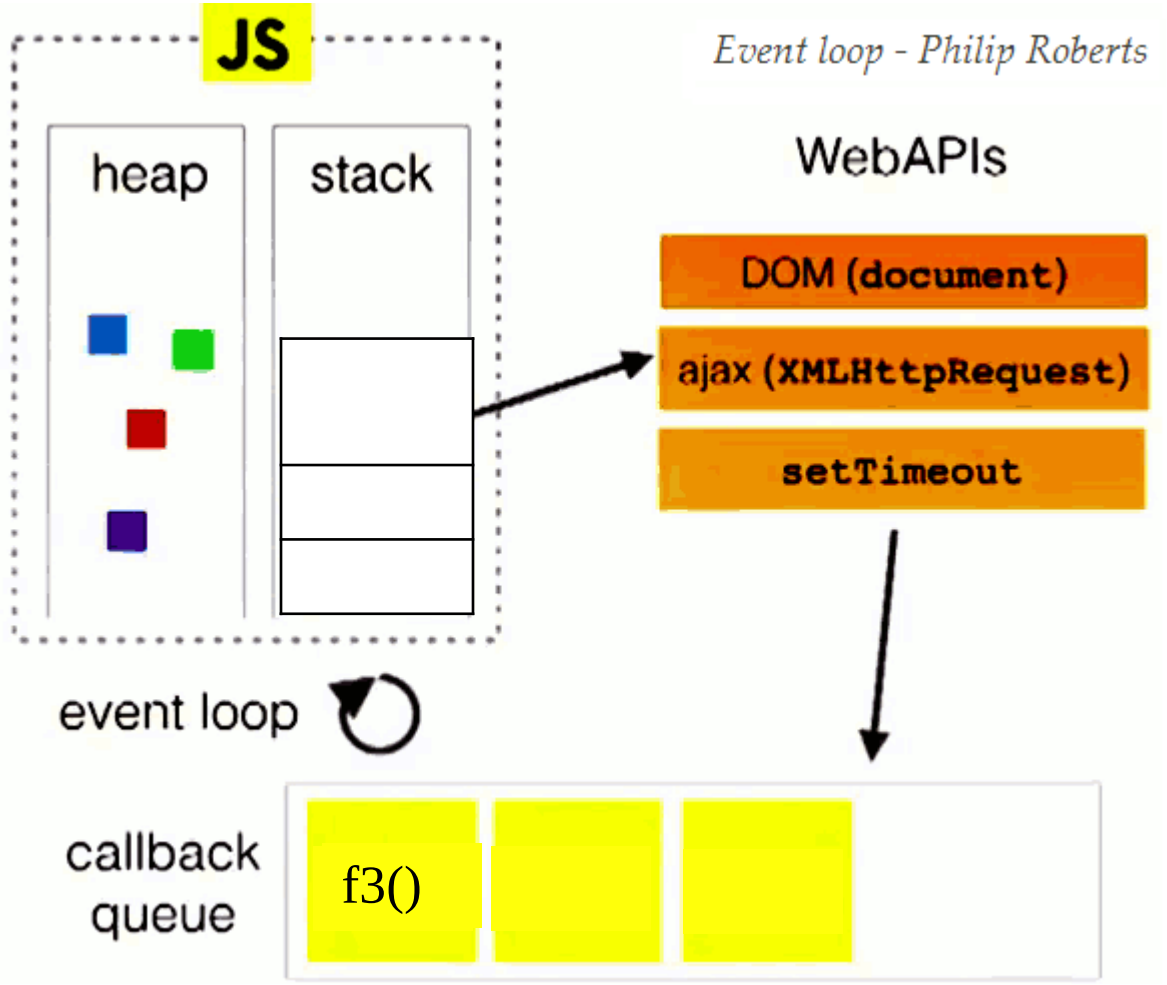
Event Loop, Call Stack,and Callback/Event Queue

```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```



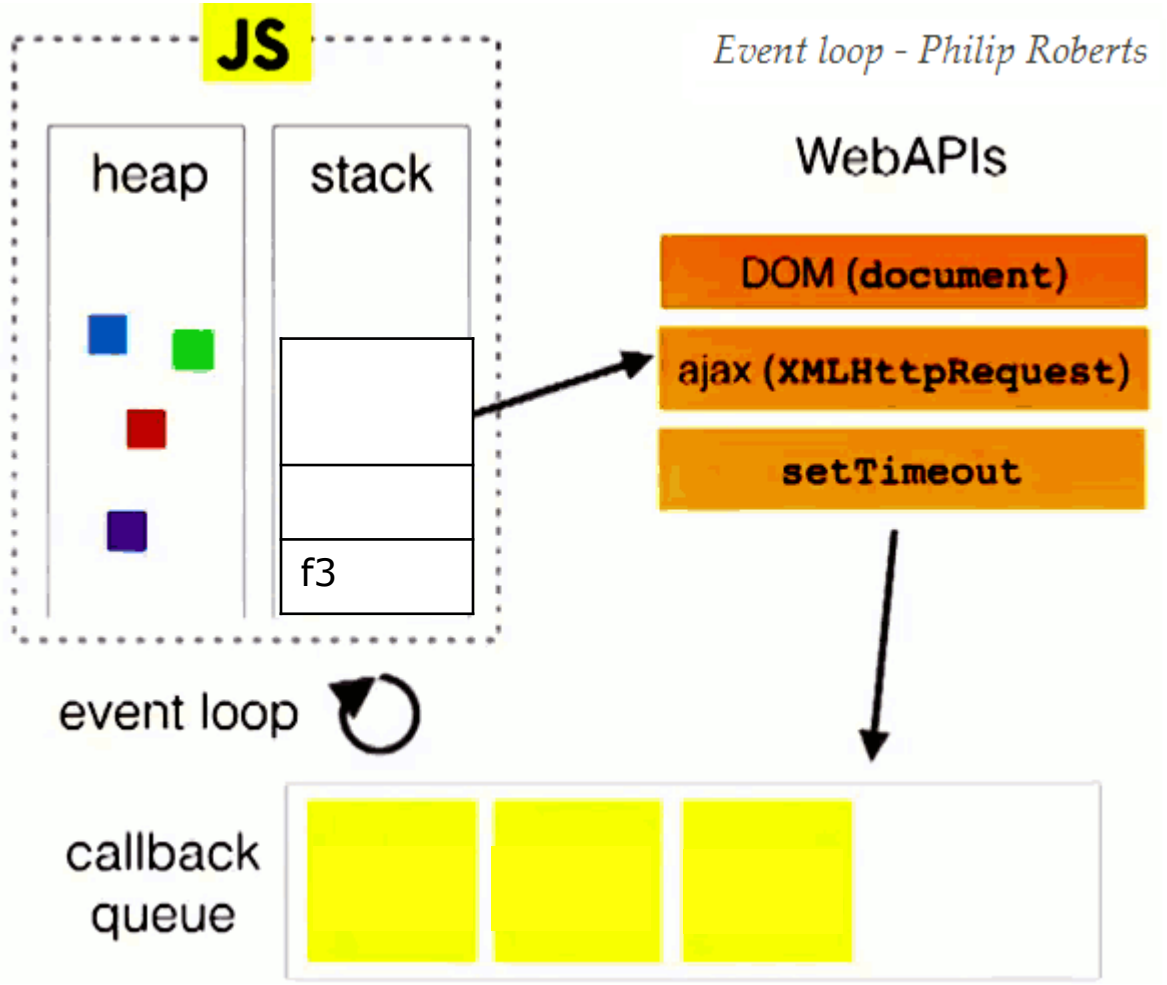
Event Loop, Call Stack,and Callback/Event Queue

```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```



Event Loop, Call Stack,and Callback/Event Queue

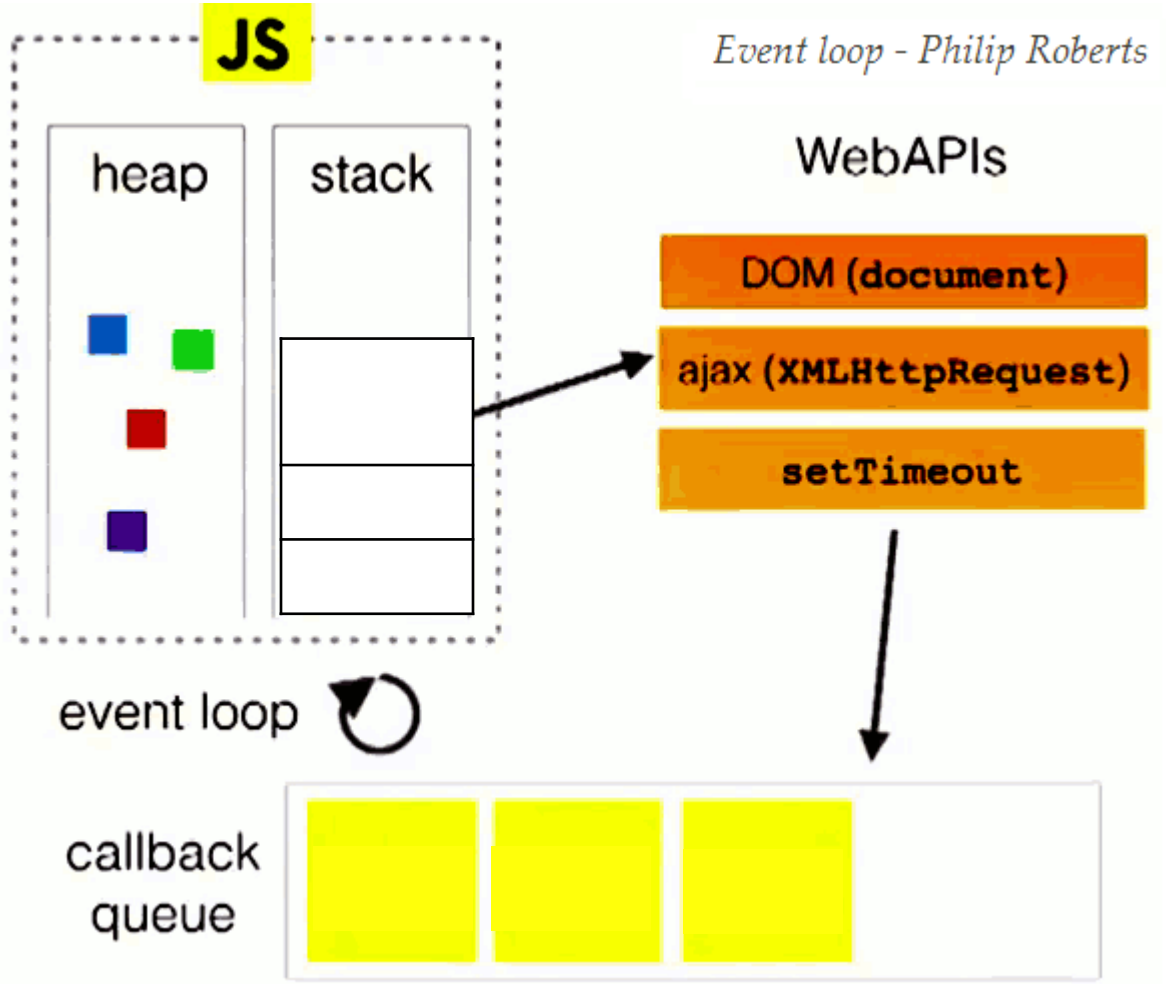
```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```



Event Loop, Call Stack,and Callback/Event Queue

```
f1() {… f2() …}
    f2() {
      setTimeout(f3(), 1000);
      }
f3() { … }
```

All functions DONE!
Repeat using the console.log and sleep sort examples



Call Stack and Event Queue

- Like other programming languages, functions in JavaScript are executed within their own contexts
  - A context contains variables, objects, and functions the function has access to
- Whenever a function is called, its Function Execution Context is pushed to the Call Stack
- When a function calls another function, the new function's Function Execution Context is pushed to the call stack
  - ↑ These are basic programming language concepts (not confined to JavaScript)
- All events are queued in Event Queue for JavaScript engine to execute one by one in an Event Loop
  - ↑ This is important JavaScript concept

Why is Event Queue Important?

- Multiple events can happen any time in any order
- Events can read/write/update the DOM simultaneously, causing the famous concurrency control problem
- JavaScript is single thread, so all event handlers (callback functions) are lined up in the event queue
  - Event loop executes the "jobs" one by one
- JavaScript engine process the functions from the queue after it finishes all the work in the main thread
- Now can you explain the outputs of the previous examples more clearly?
  - Note: In setTimeout(), when the wait time is reached, the callback function console.log() is executed, which is treated as an action generated by an event and got lined up in the event queue!

Why is setTimeout() Useful?

- Sure, setTimeout() is useful for games
- setTimeout() can be used to explicitly impose execution order of two functions:

```
setTimeout(f1, 0);
setTimeout(f2,0);
```

- E.g., you want to display a DIV (containing a message), then reduce its size by 50%
  - The reduction myDidv.width=my.Div.width*0.5 must be done after the DIV is fully loaded (why?)