Single-threaded?

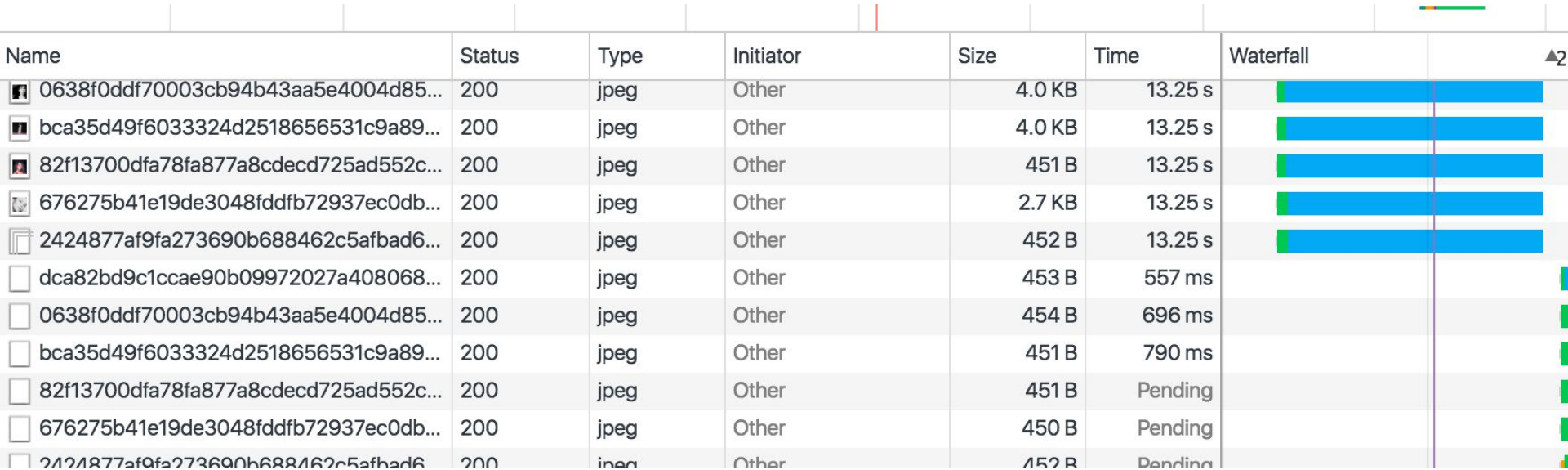
Some hand-wavy definitions:

- **Single-threaded**:
  - When your computer processes one command at a time
  - There is one call stack
- **Multi-threaded**
  - When your computer processes multiple commands simultaneously
  - There is one call stack **per thread**

thread: a linear sequence of instructions; an executable container for instructions

Network tab

If we'd look at Chrome's Network tab while a javascript script load images, we'd see there are several images being loaded simultaneously:



Name	Status	Type	Initiator	Size	Time	Waterfall
0638f0ddf70003cb94b43aa5e4004d85...	200	jpeg	Other	4.0 KB	13.25 s	
bca35d49f6033324d2518656531c9a89...	200	jpeg	Other	4.0 KB	13.25 s	
82f13700dfa78fa877a8cdecd725ad552c...	200	jpeg	Other	451 B	13.25 s	
676275b41e19de3048fddfb72937ec0db...	200	jpeg	Other	2.7 KB	13.25 s	
2424877af9fa273690b688462c5afbad6...	200	jpeg	Other	452 B	13.25 s	
dca82bd9c1ccae90b09972027a408068...	200	jpeg	Other	453 B	557 ms	
0638f0ddf70003cb94b43aa5e4004d85...	200	jpeg	Other	454 B	696 ms	
bca35d49f6033324d2518656531c9a89...	200	jpeg	Other	451 B	790 ms	
82f13700dfa78fa877a8cdecd725ad552c...	200	jpeg	Other	451 B	Pending	
676275b41e19de3048fddfb72937ec0db...	200	jpeg	Other	450 B	Pending	
2424877af9fa273690b688462c5afbad6...	200	jpeg	Other	452 B	Pending	

Q: If JavaScript is single-threaded, i.e. if only one thing happens at a time, how can images be loaded in parallel?

JavaScript event loop

Note: see talk!

(For a perfectly great talk on this, see Philip Roberts' talk: https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=1s

And for a perfectly great deep dive on this, see Jake Archibald's blog post: https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/

These slides are inspired by these resources!)

`setTimeout`

To help us understand the event loop better, let's learn about a new command, `setTimeout`:

`setTimeout(`***function***, ***delay***`);`

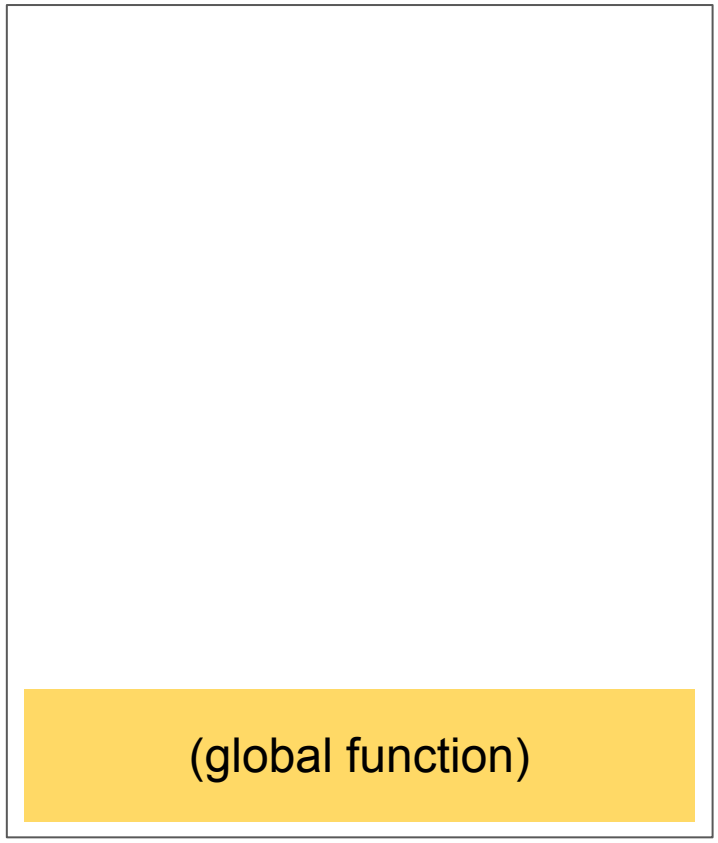
- ***function*** will fire after ***delay*** milliseconds
- CodePen example

Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack



Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

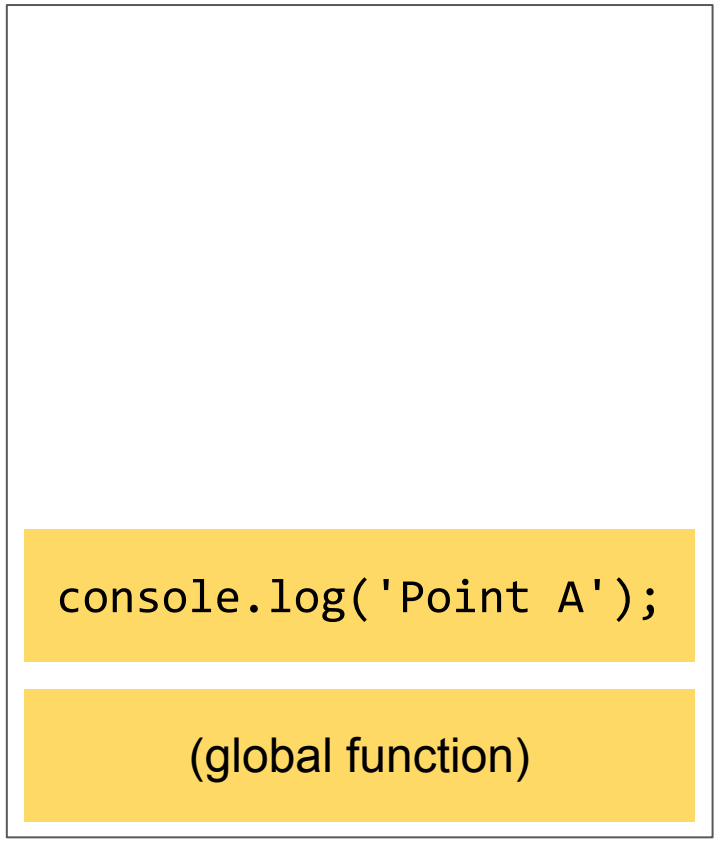
Call Stack

Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

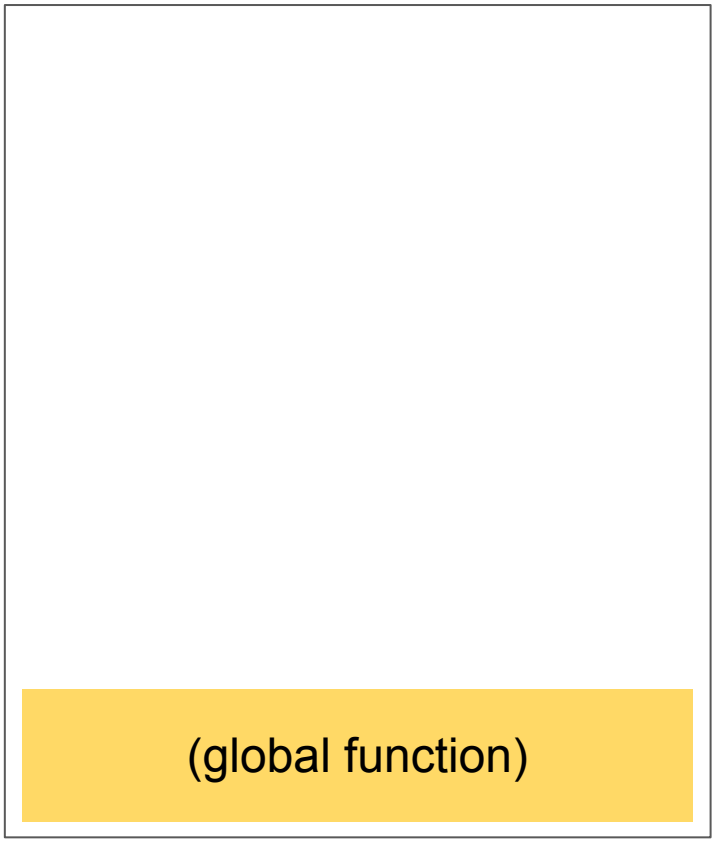


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

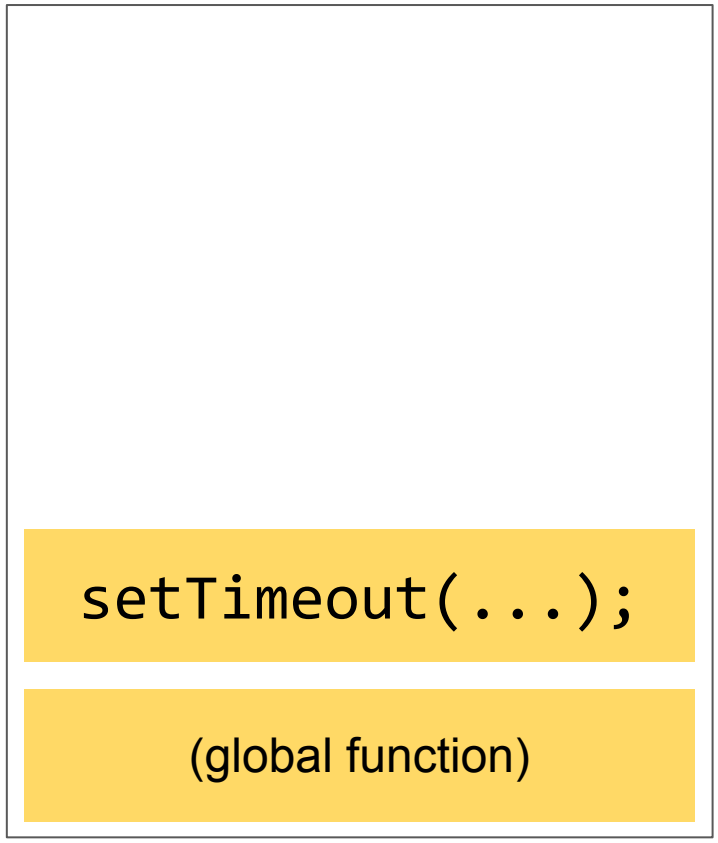


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

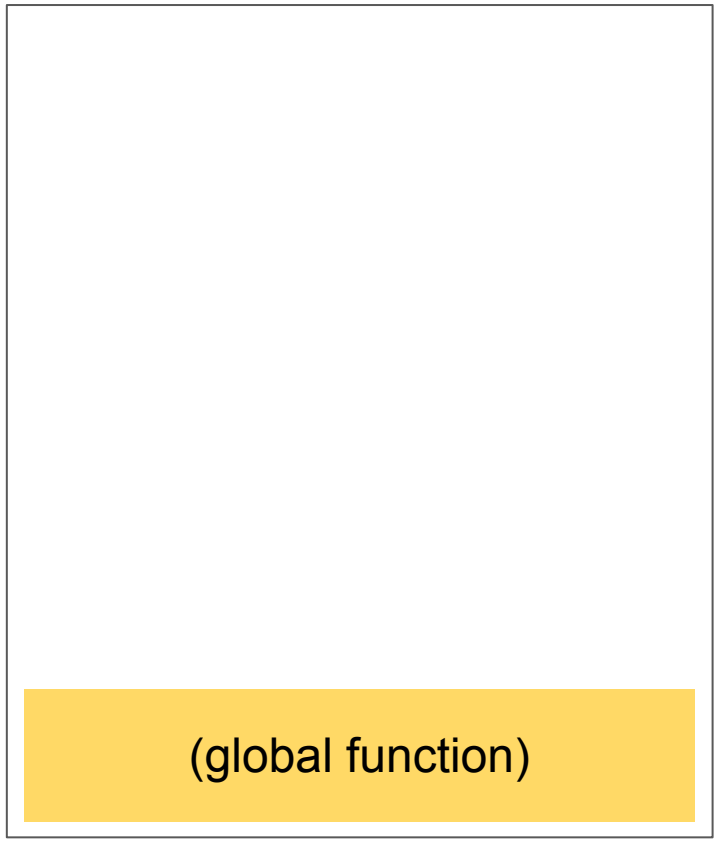


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

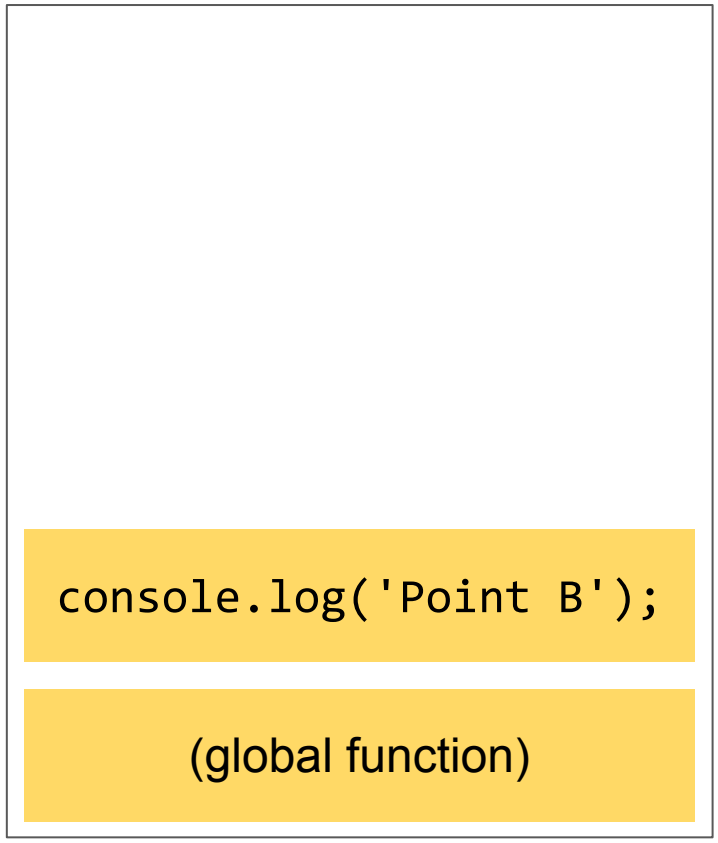


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

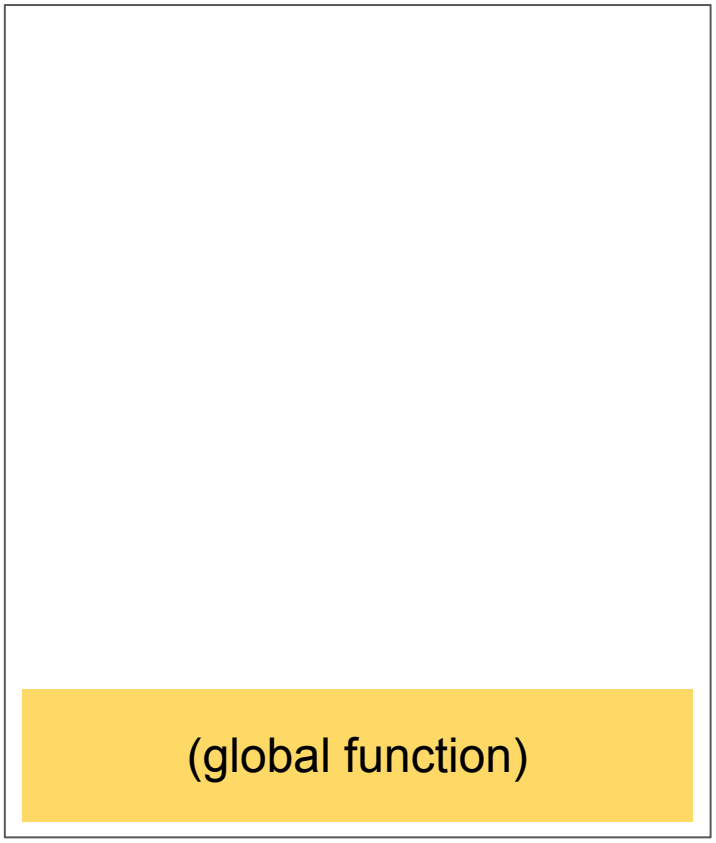


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

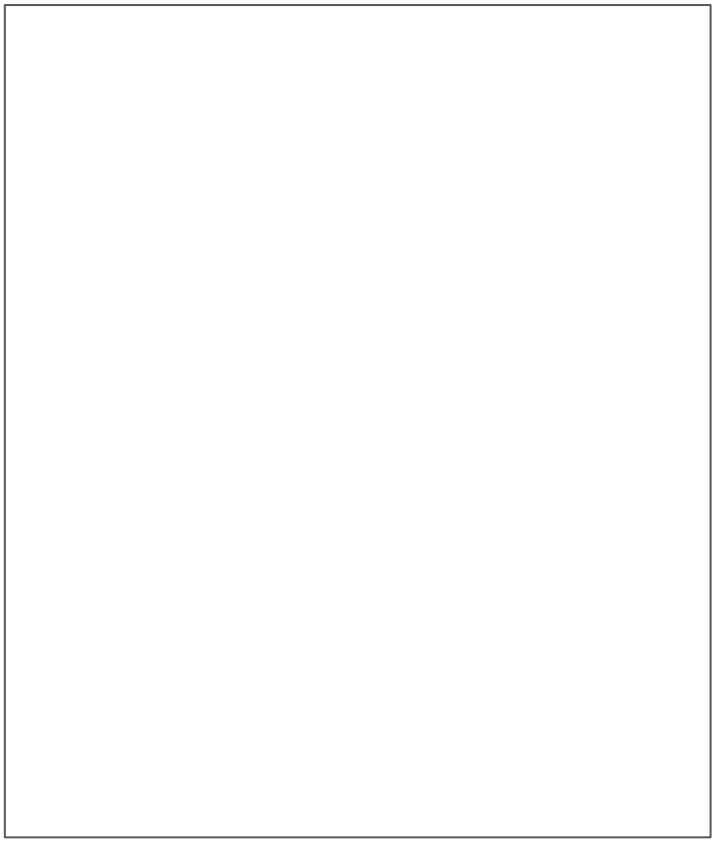


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

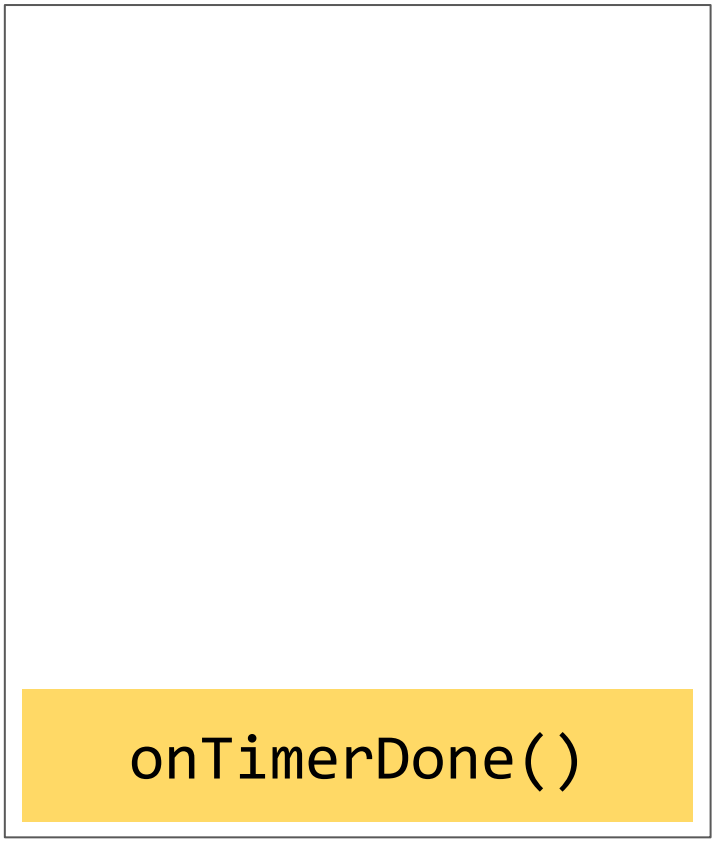


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

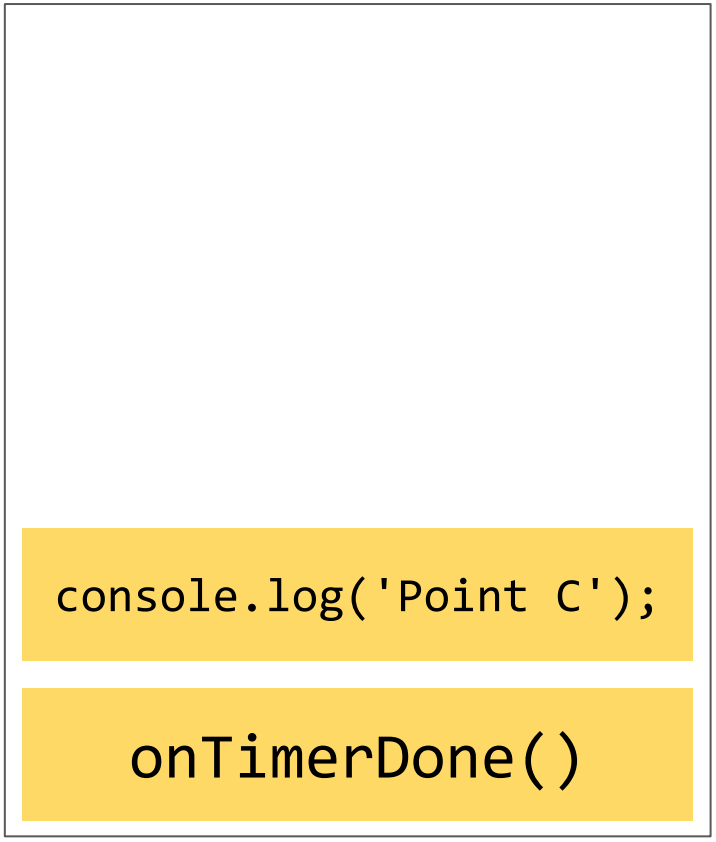


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

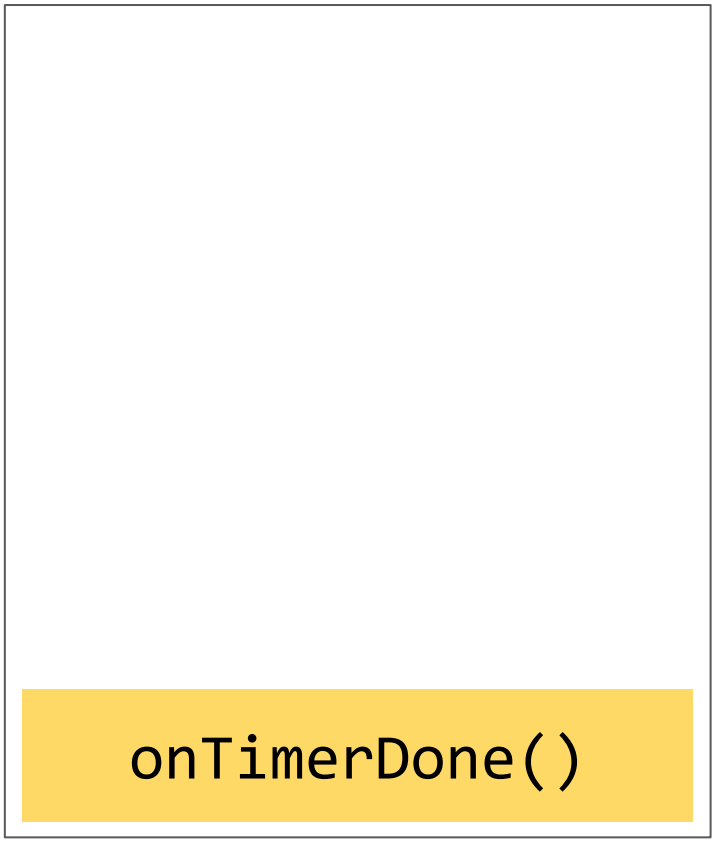


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

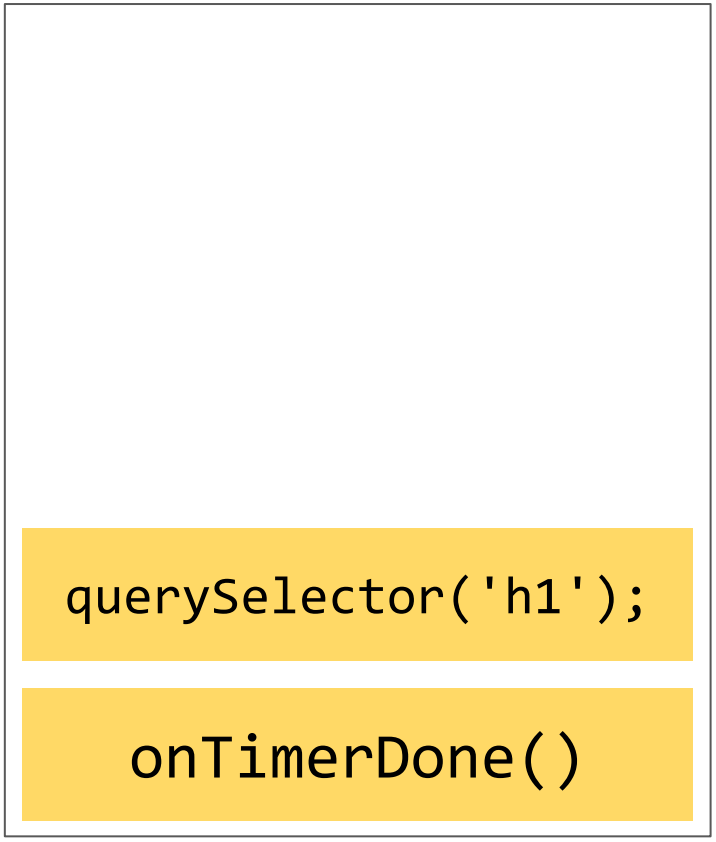


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

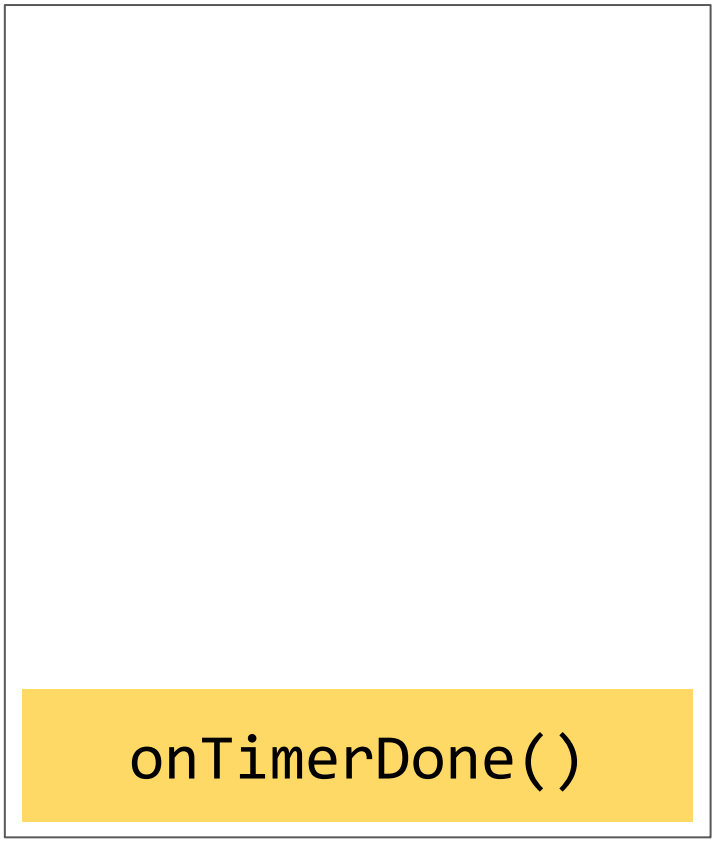


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

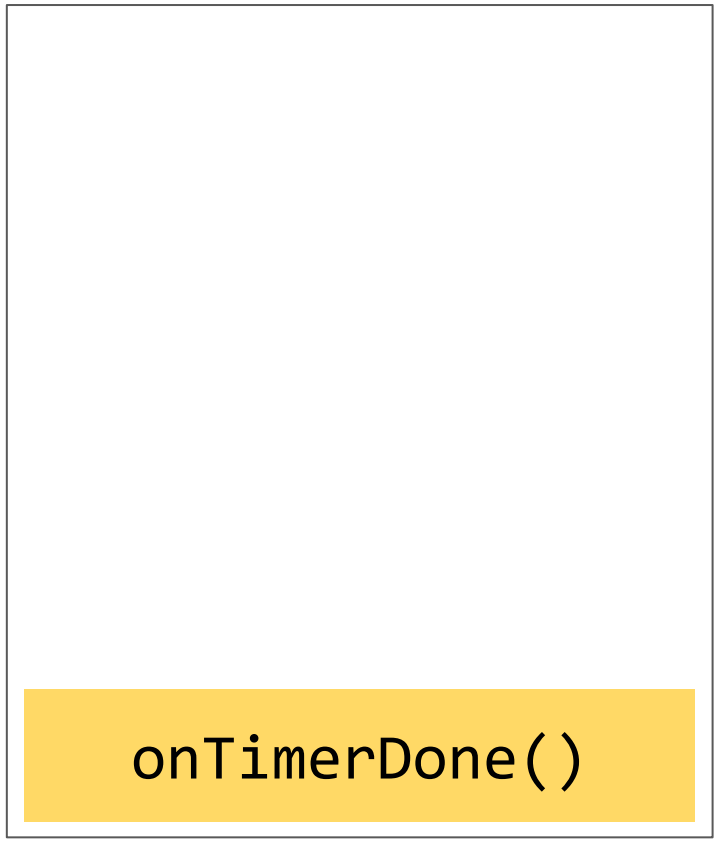


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack

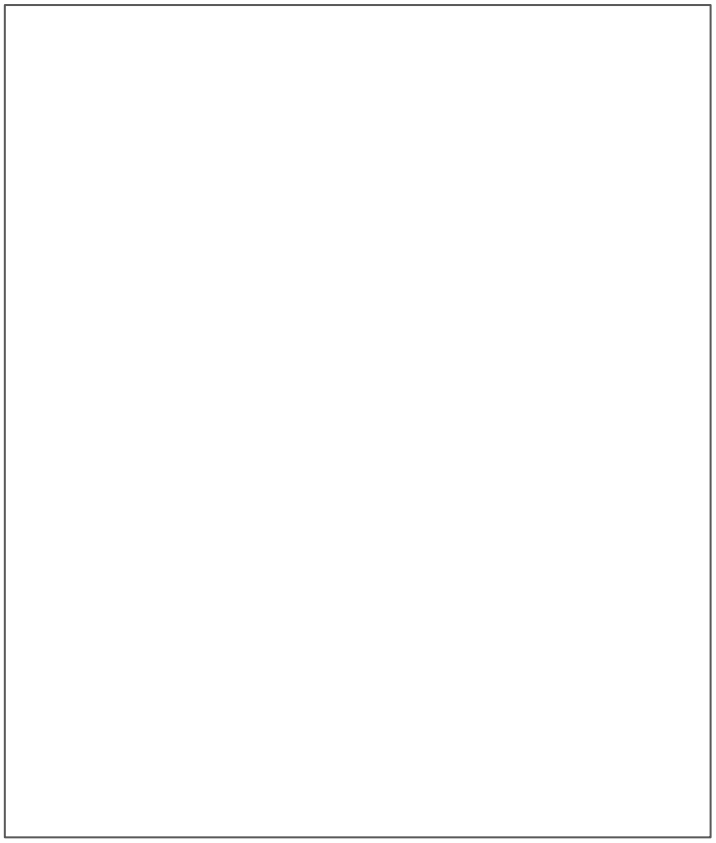


Call stack + `setTimeout`

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

Call Stack



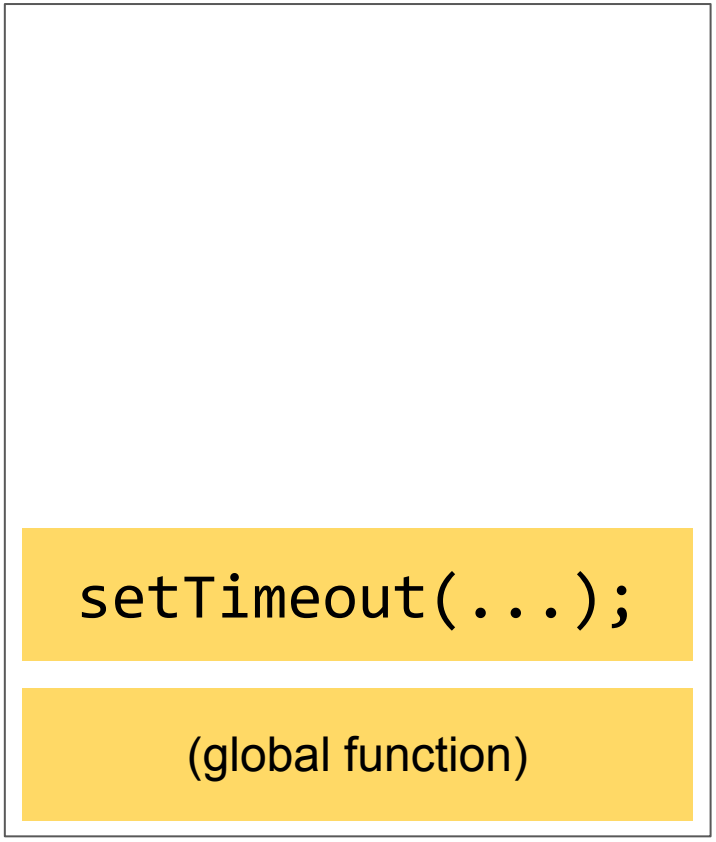
Call stack + setTimeout

```
function onTimerDone() {
  console.log('Point C');
  const h1 = document.querySelector('h1');
  h1.textContent = 'loaded';
}

console.log('Point A');
setTimeout(onTimerDone, 3000);
console.log('Point B');
```

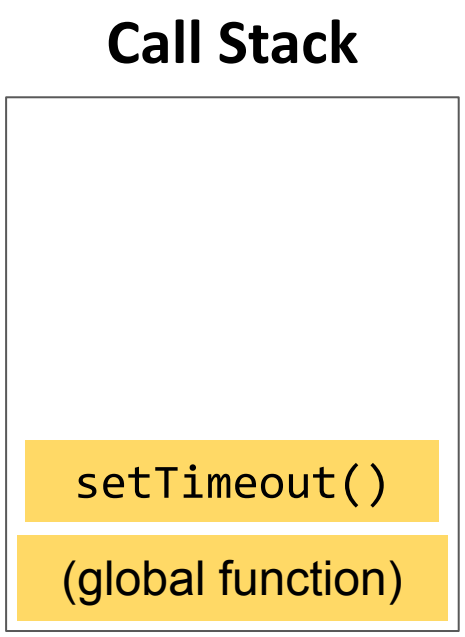
What "enqueues" onTimerDone?
How does it get fired?

Call Stack



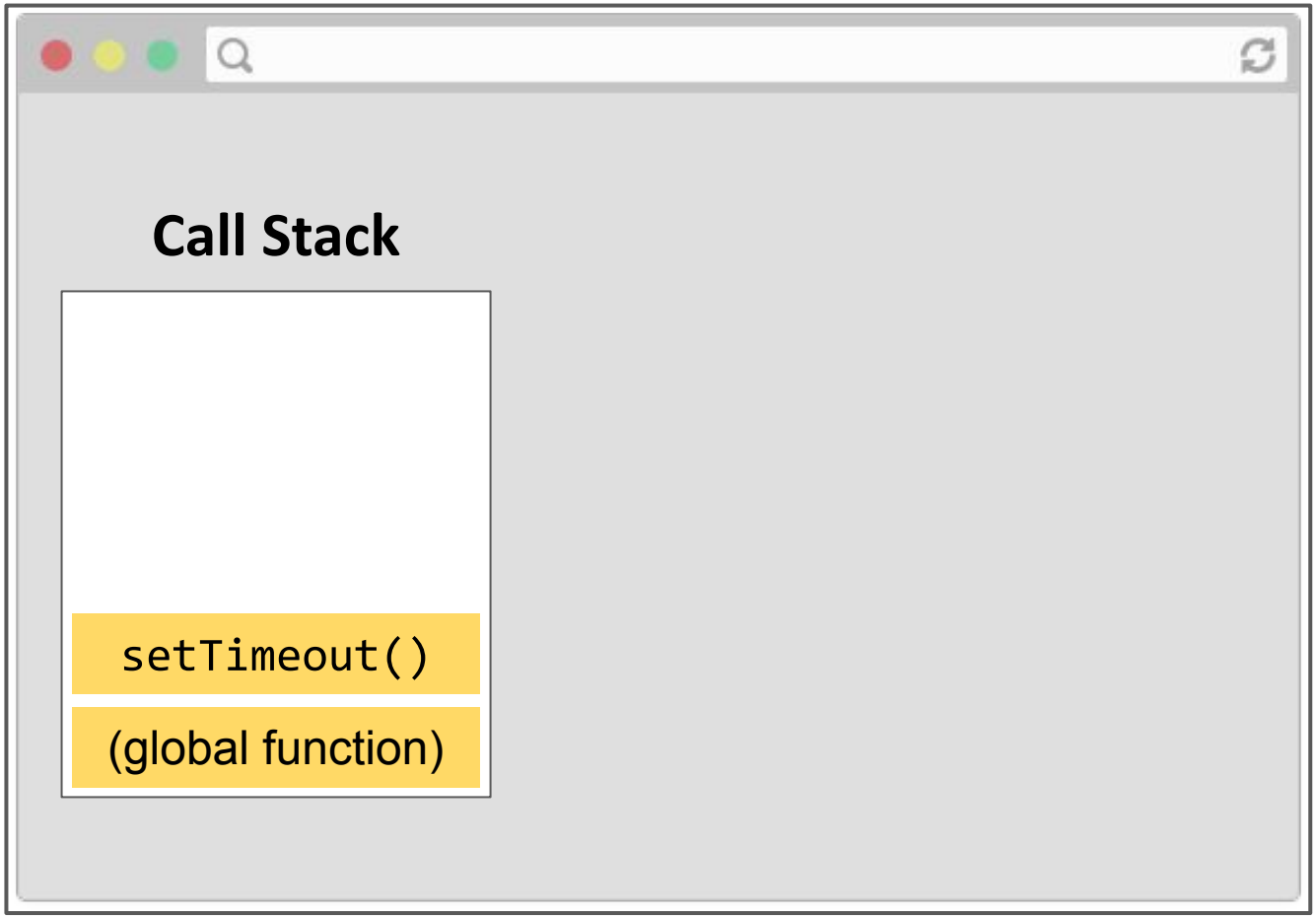
Tasks, Micro-tasks, and the Event Loop

Tasks and the Event Loop

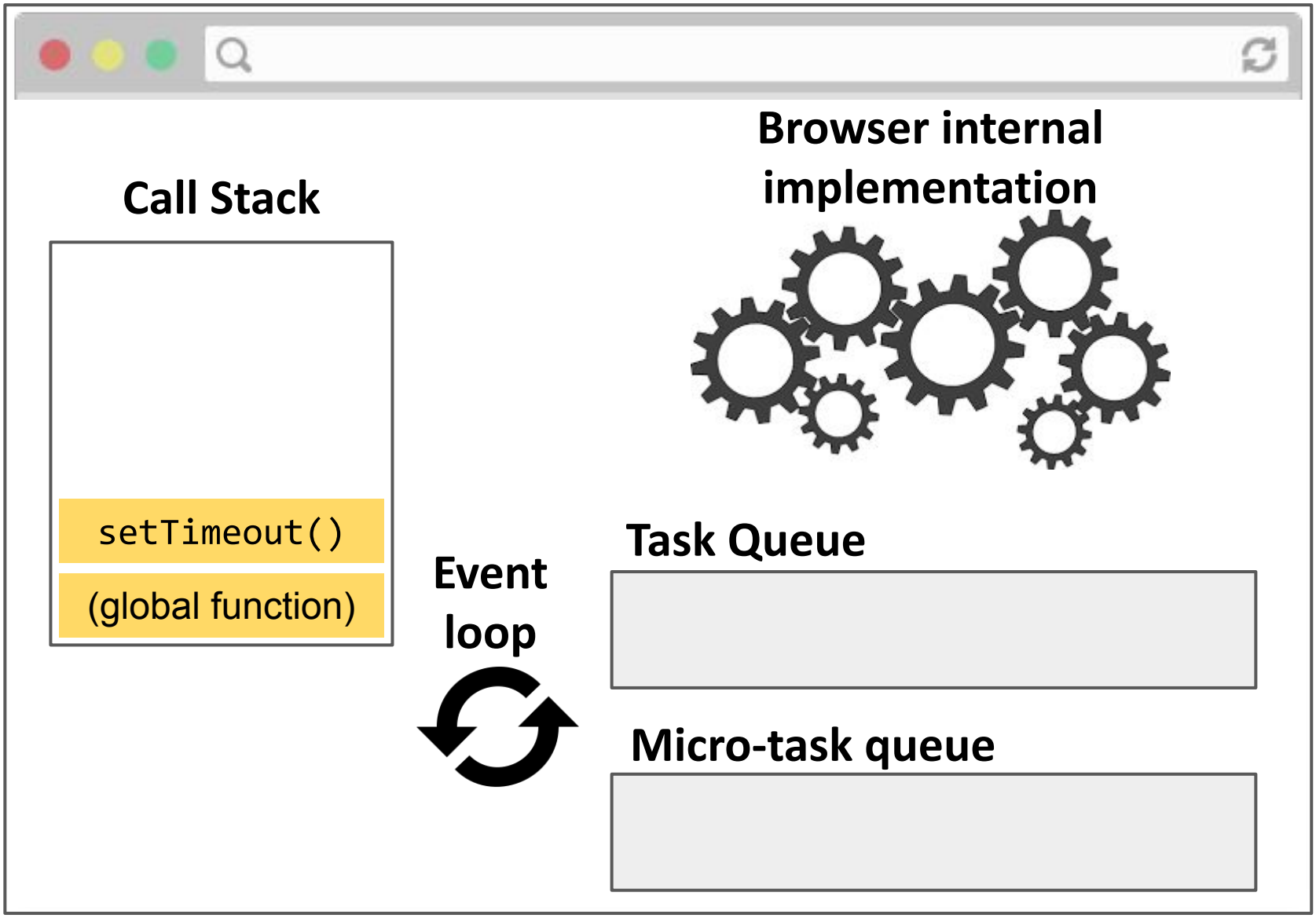


The JavaScript runtime can do only one thing at a time...

Tasks and the Event Loop

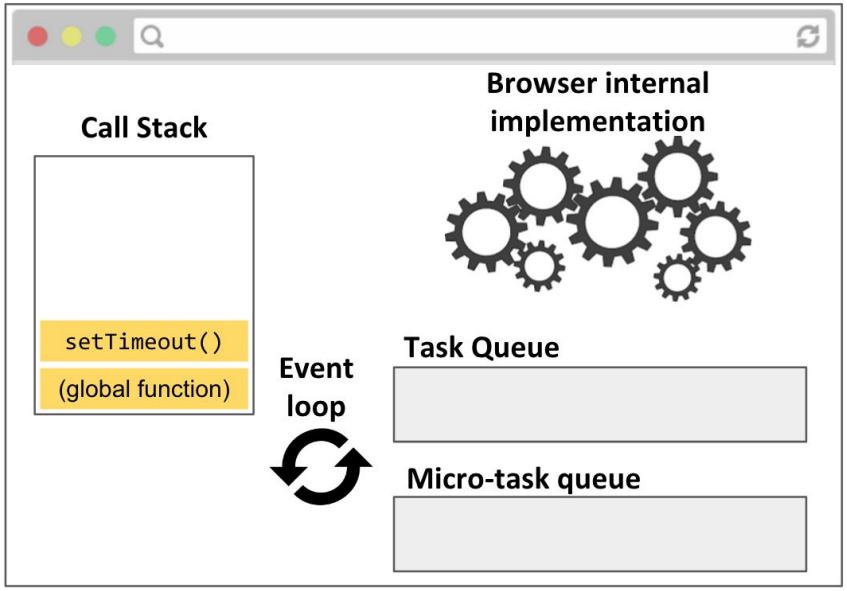


But the JS runtime runs within a browser, which can do multiple things at a time.



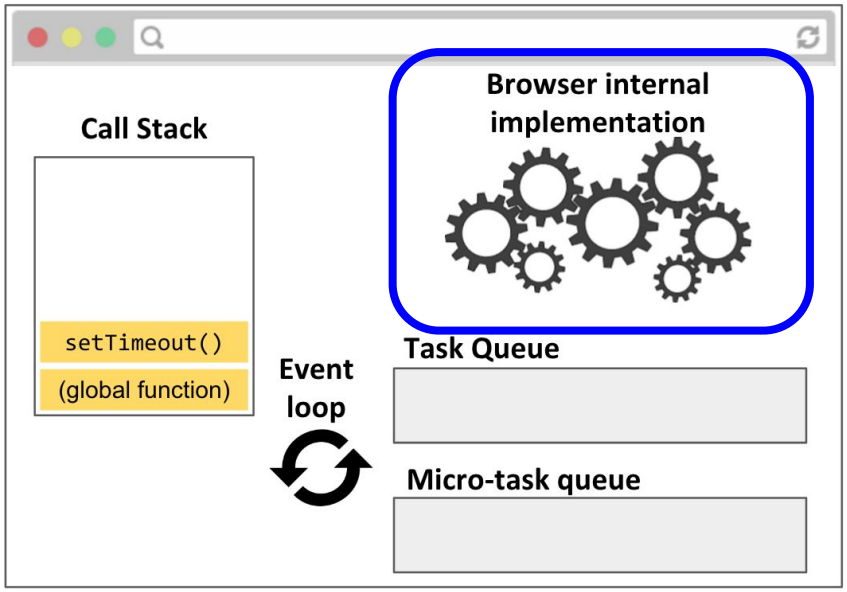
Here's a picture of the major pieces involved in executing JavaScript code in the browser.

JS execution



- **Call stack:** JavaScript runtime call stack. Executes the JavaScript commands, functions.
- **Browser internal implementation:** The C++ code that executes in response to native JavaScript commands, e.g. `setTimeout`, `element.classList.add('style')`, etc.

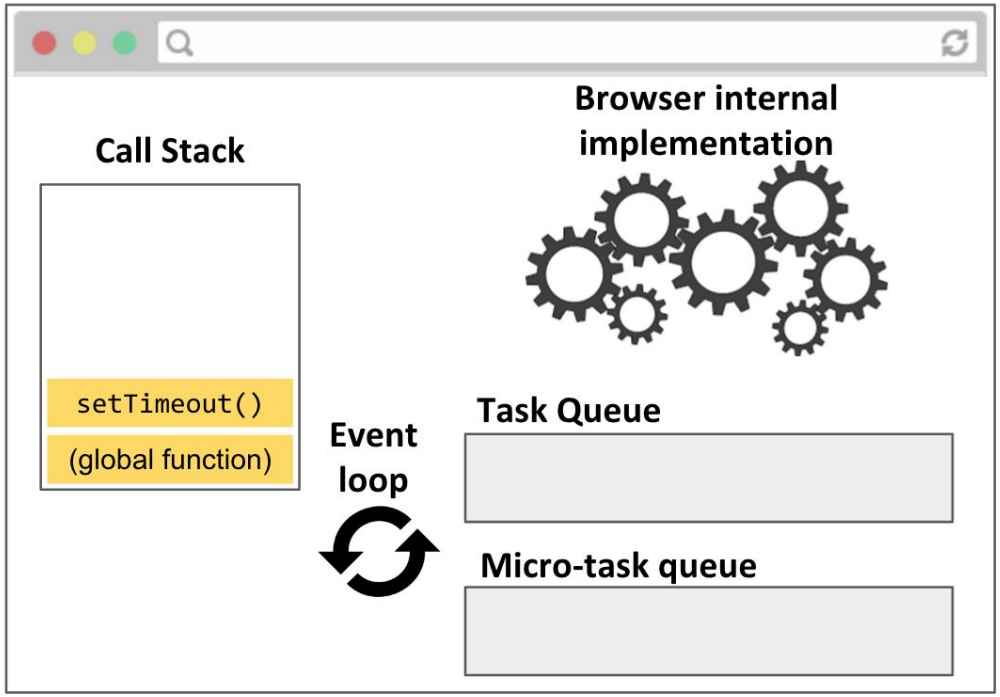
JS execution



The browser itself is multi-threaded and multi-process!

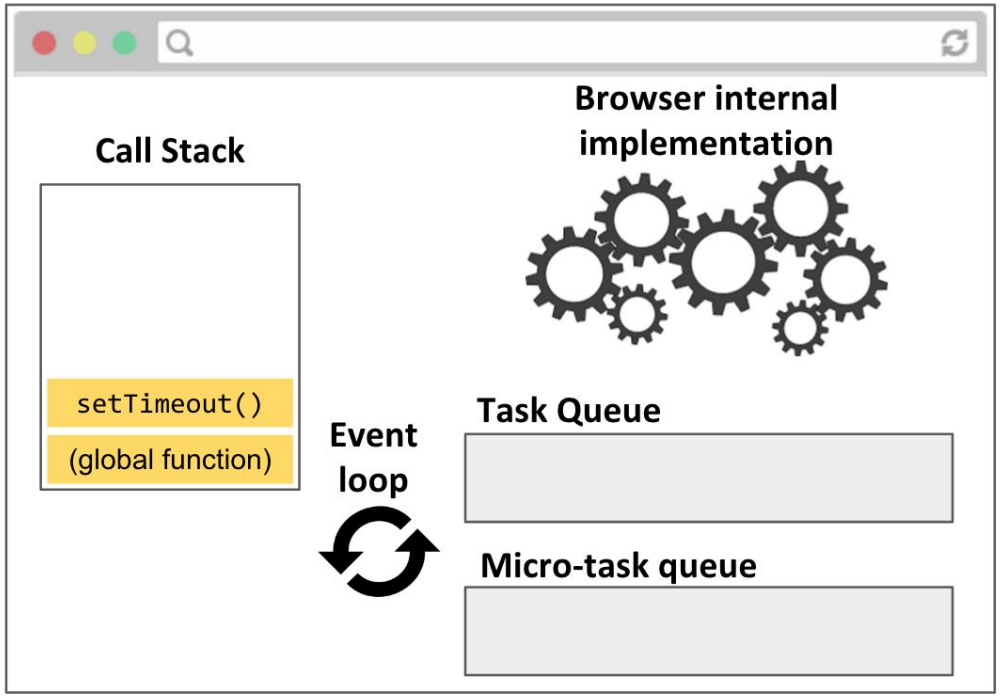
- **Call stack:** JavaScript runtime call stack. Executes the JavaScript commands, functions.
- **Browser internal implementation:** The C++ code that executes in response to native JavaScript commands, e.g. `setTimeout`, `element.classList.add('style')`, etc.

JS execution



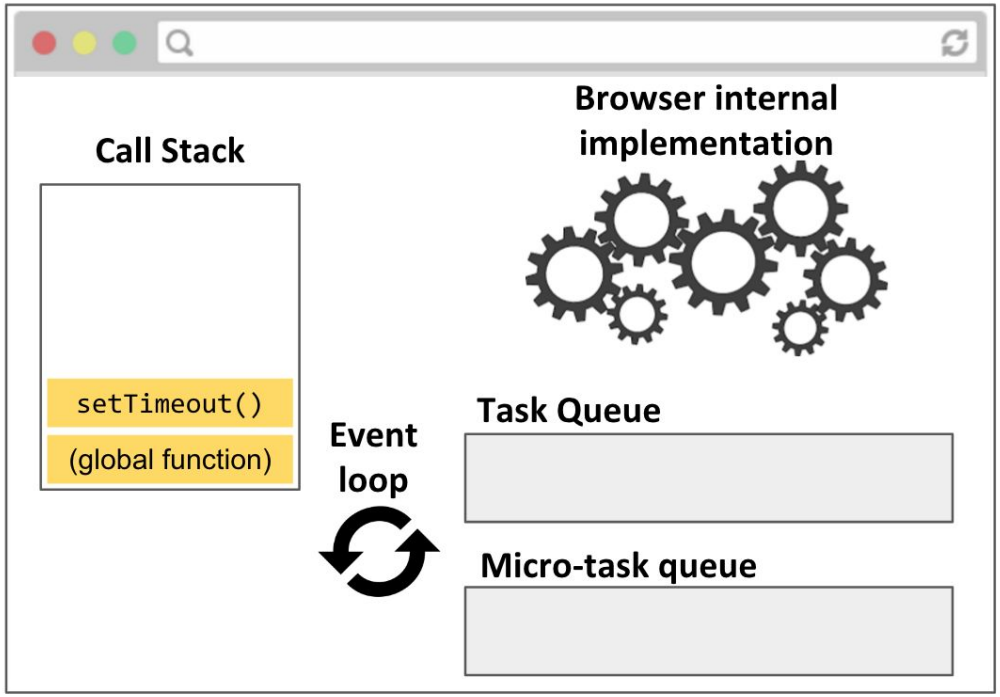
- **Task Queue:** When the browser internal implementation notices a callback from something like setTimeout or addEventListener is should be fired, it creates a Task and enqueues it in the Task Queue

JS execution



- **Micro-task Queue:** Promises are special tasks that execute with higher priority than normal tasks, so they have their own special queue. (see details here)

JS execution



Event loop: Processes the task queues.

- When the call stack is empty, the event loop pulls the next task from the task queues and puts it on the call stack.
- The Micro-task queue has higher priority than the Task Queue.

Demo

Philip Roberts wrote a nice visualizer for the JS event loop:

- setTimeout
- With click