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***`);`

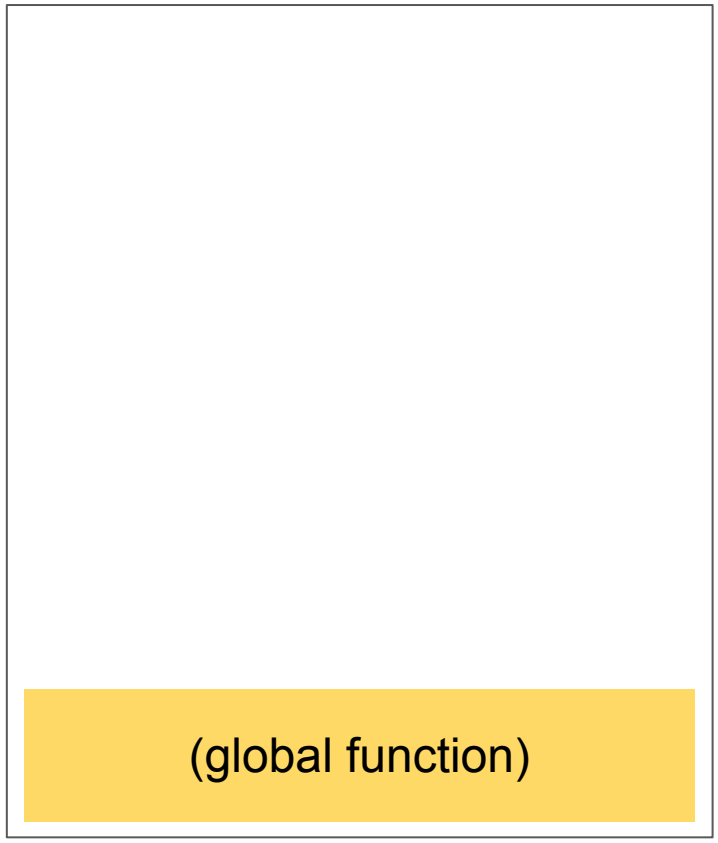
- ***function*** will fire after ***delay*** milliseconds
- CodePen example

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');
```

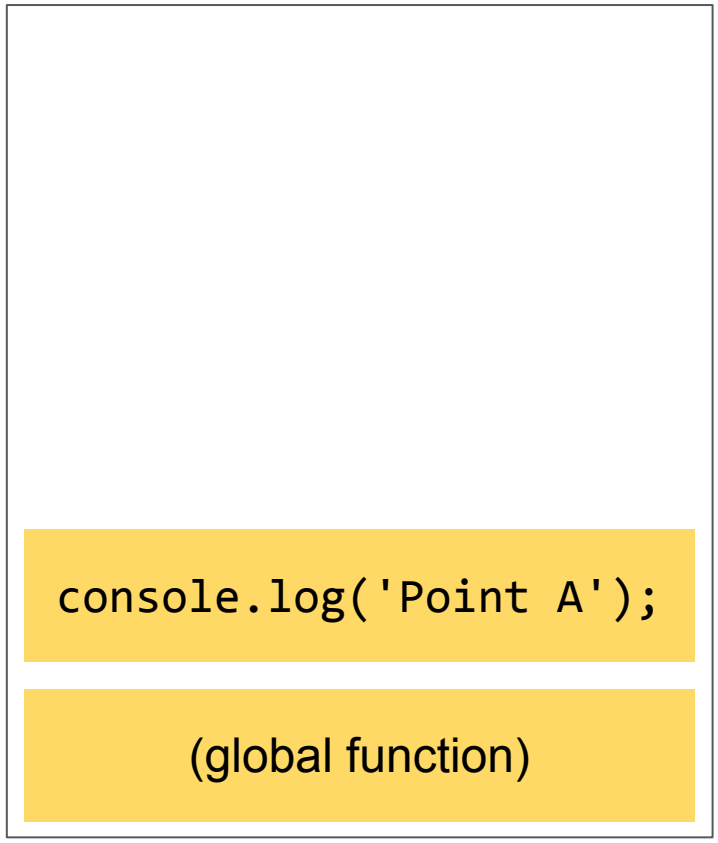
Call Stack

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

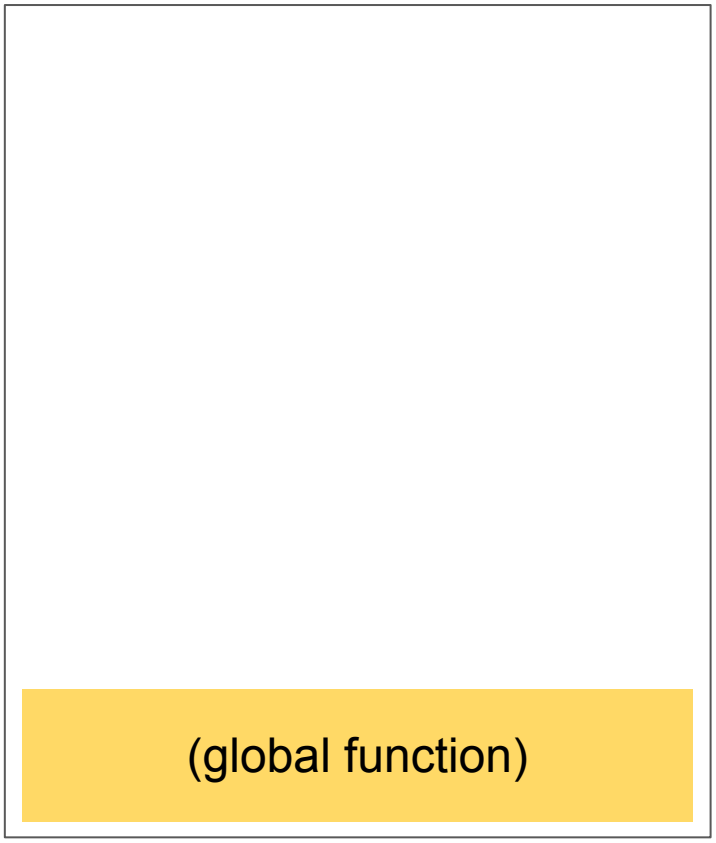


Call stack + `setTimeout`

```
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

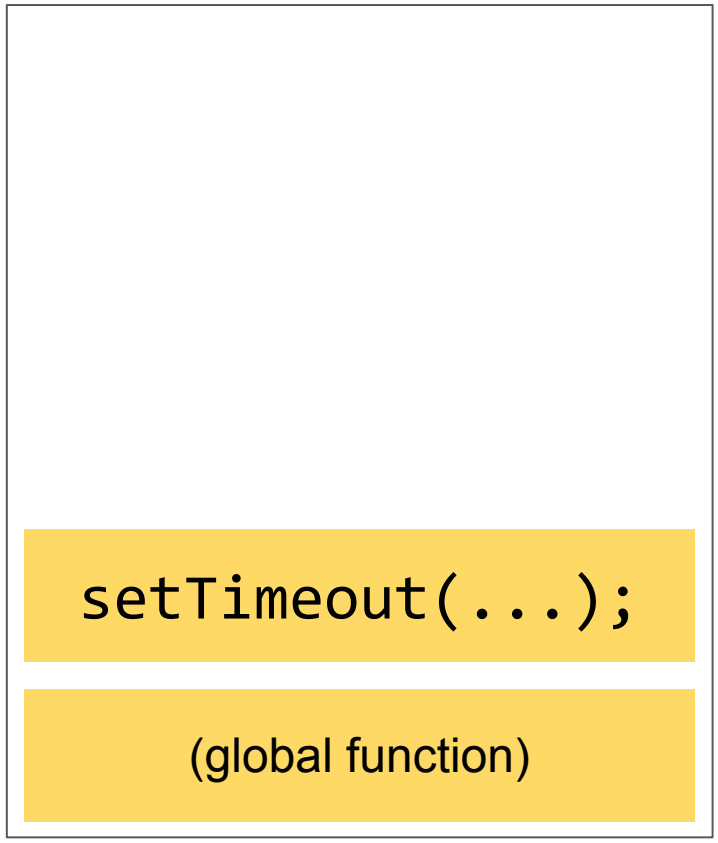


Call stack + `setTimeout`

```
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

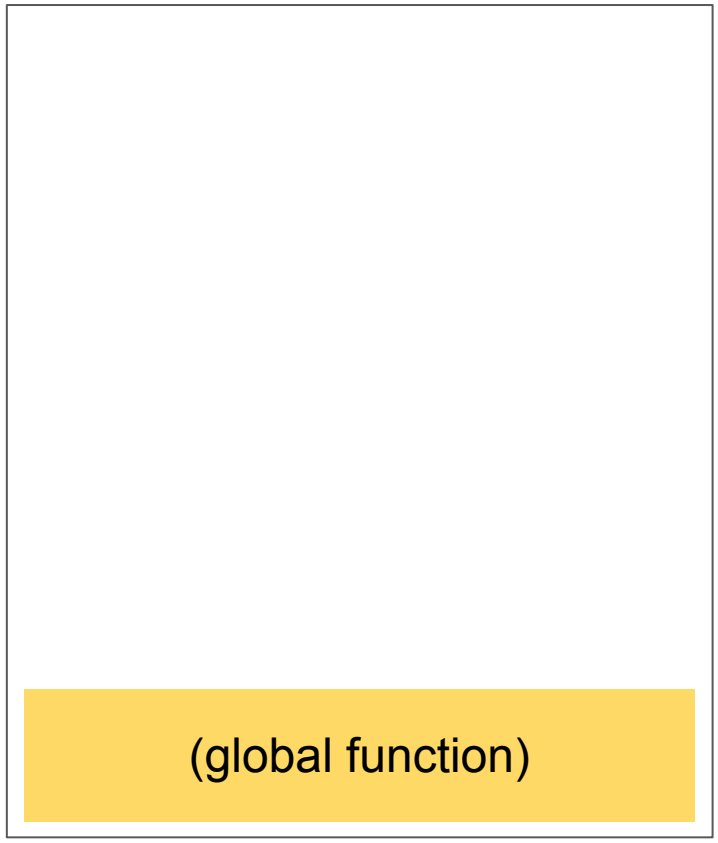


Call stack + `setTimeout`

```
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

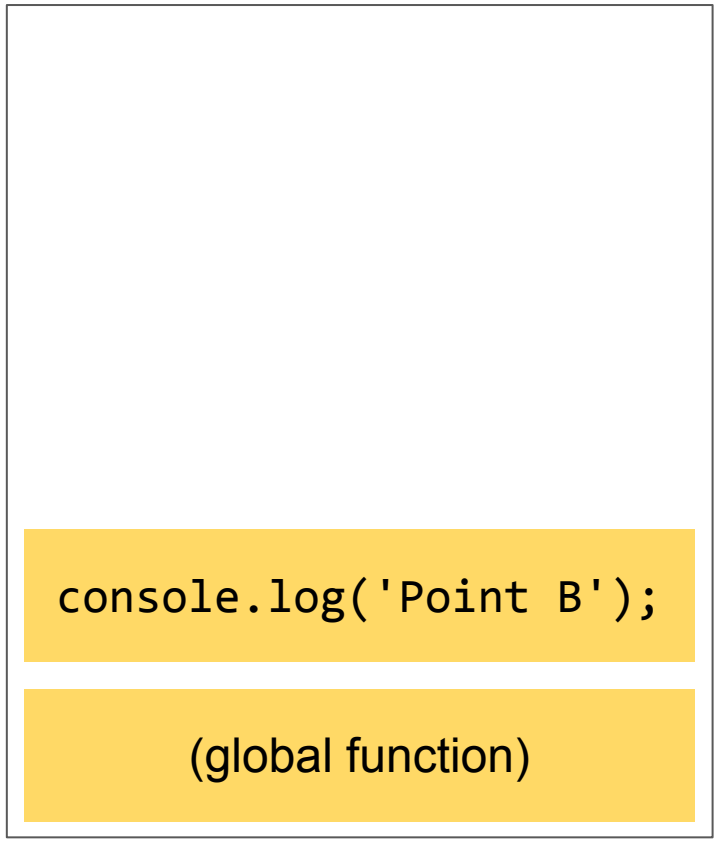


Call stack + `setTimeout`

```
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

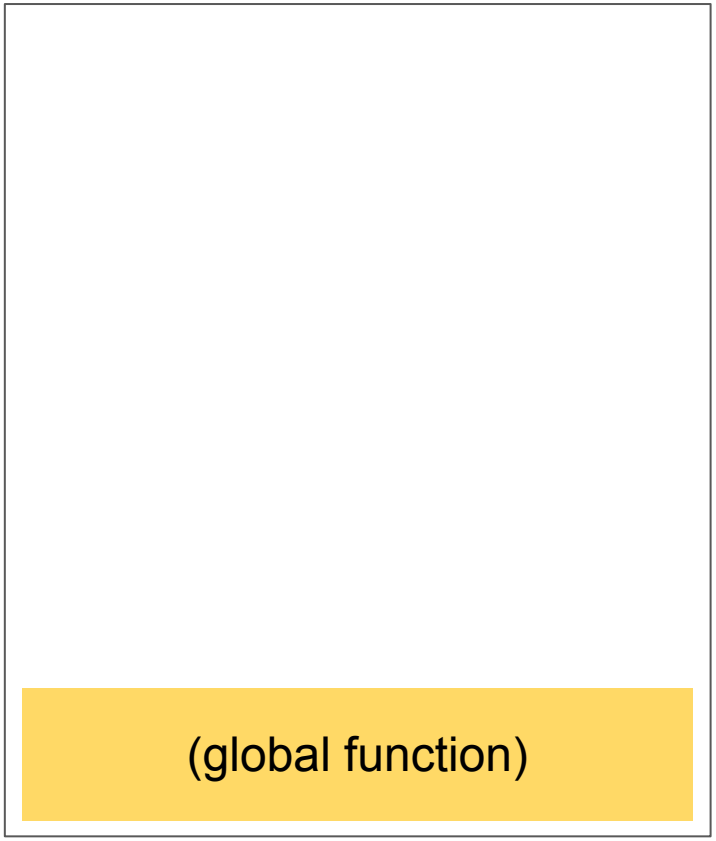


Call stack + `setTimeout`

```
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

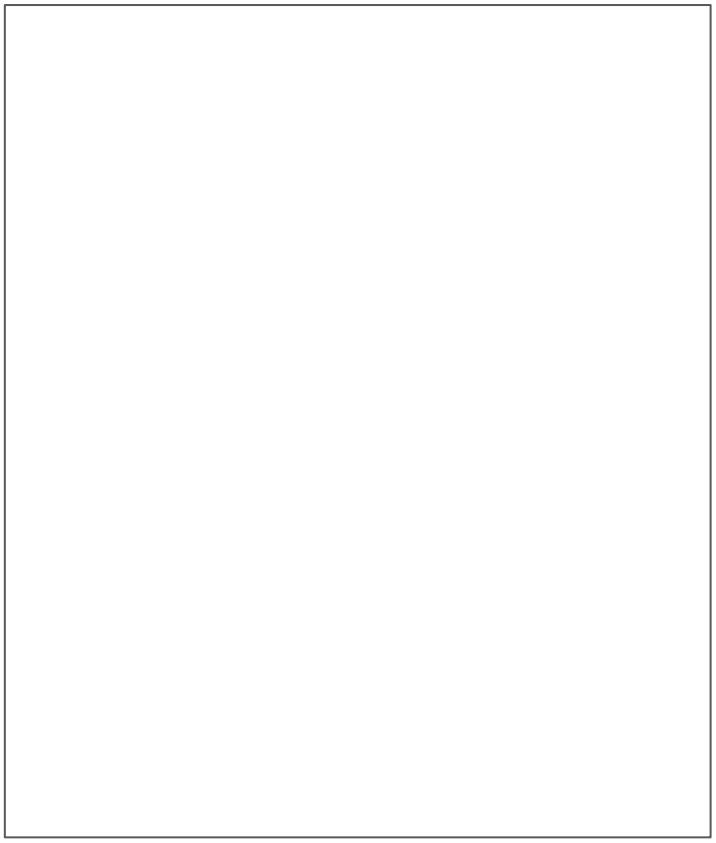


Call stack + `setTimeout`

```
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

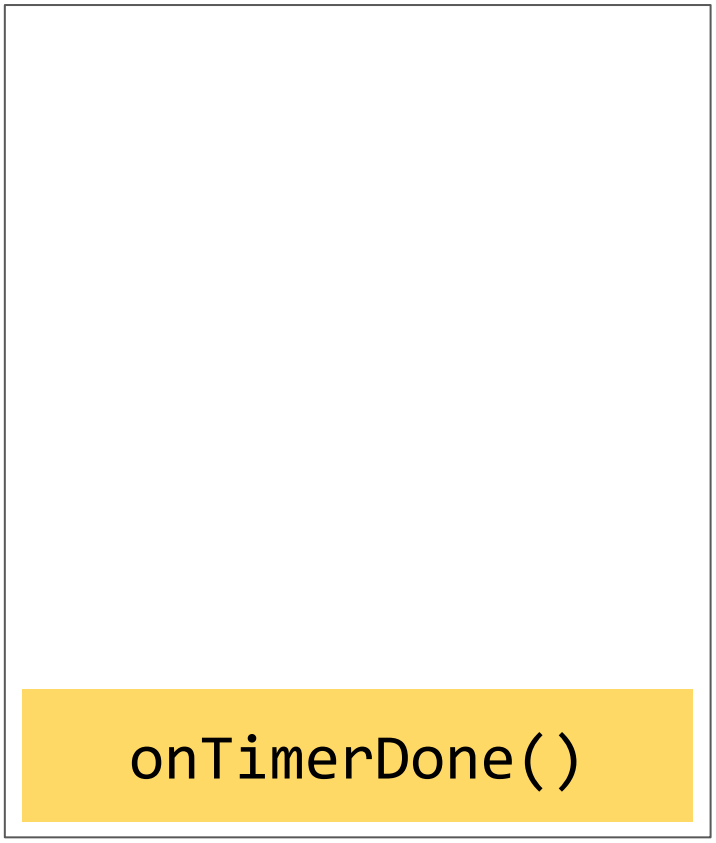


Call stack + `setTimeout`

```
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

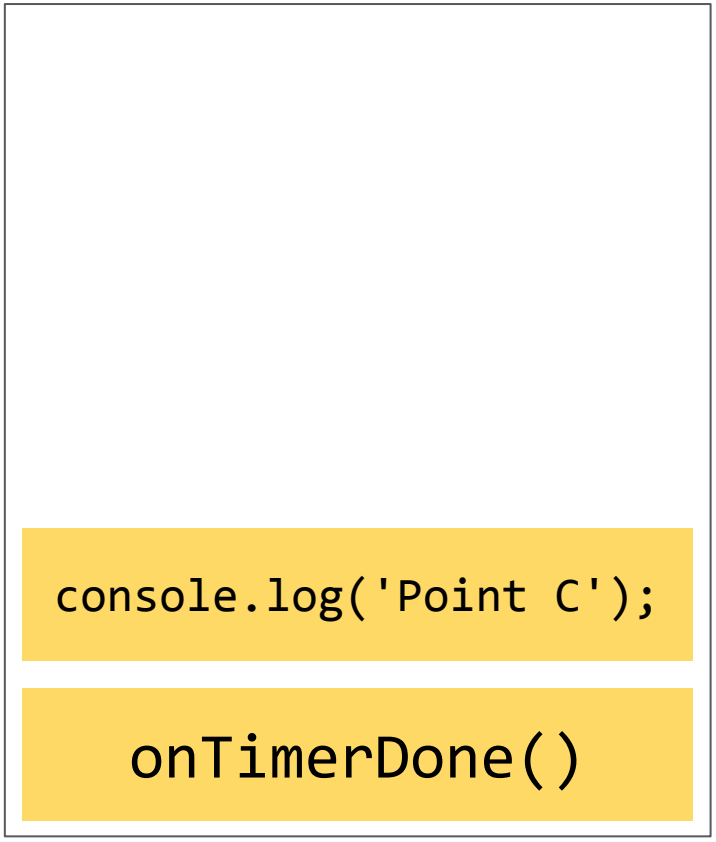


Call stack + `setTimeout`

```
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

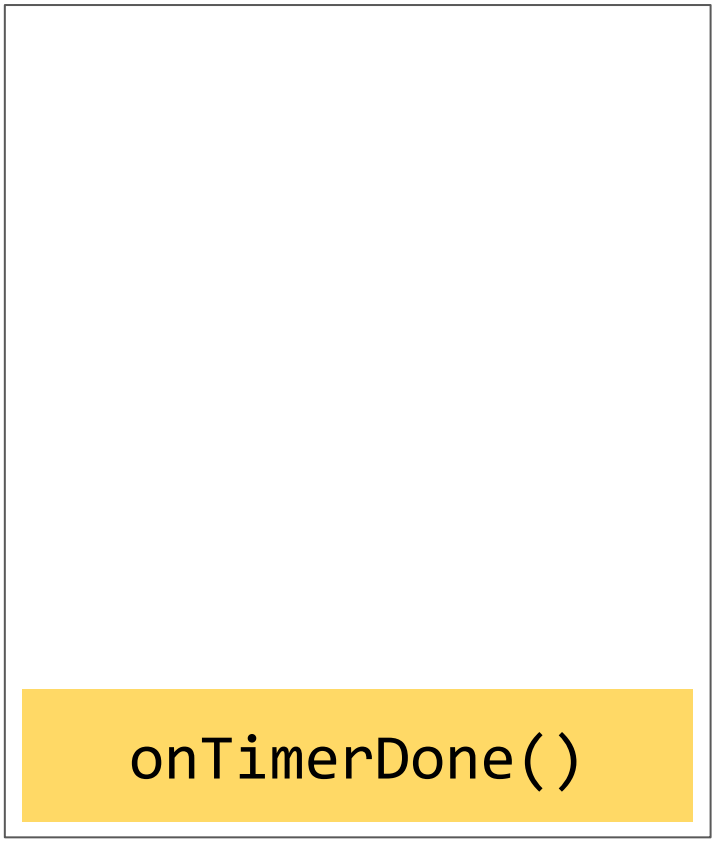


Call stack + `setTimeout`

```
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

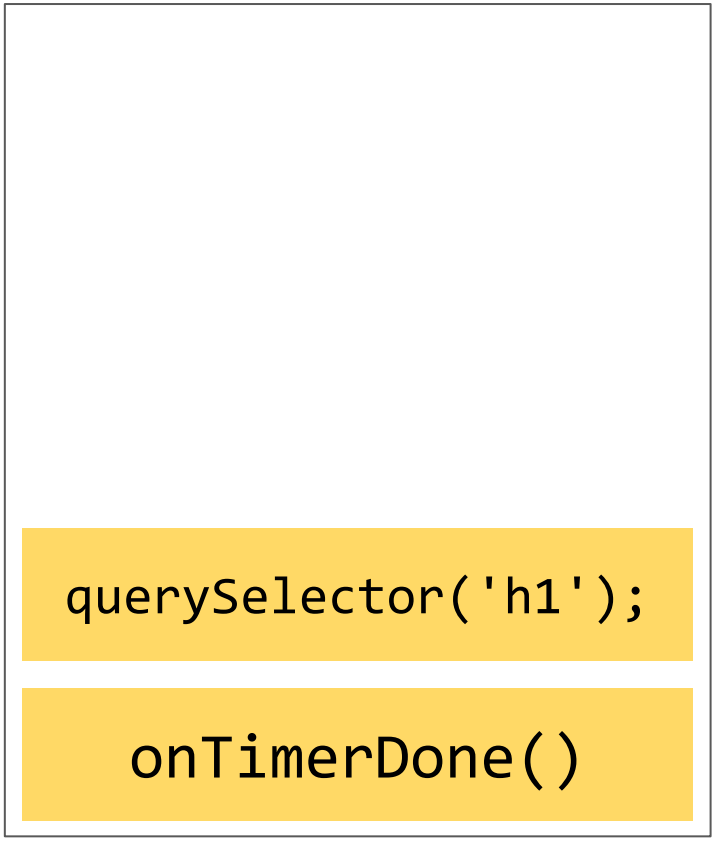


Call stack + `setTimeout`

```
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

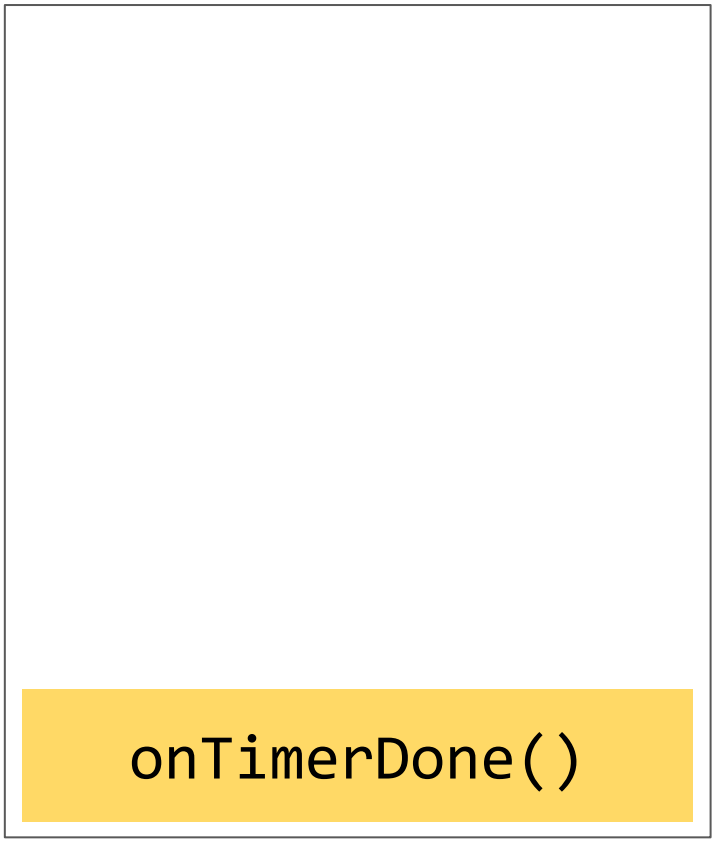


Call stack + `setTimeout`

```
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

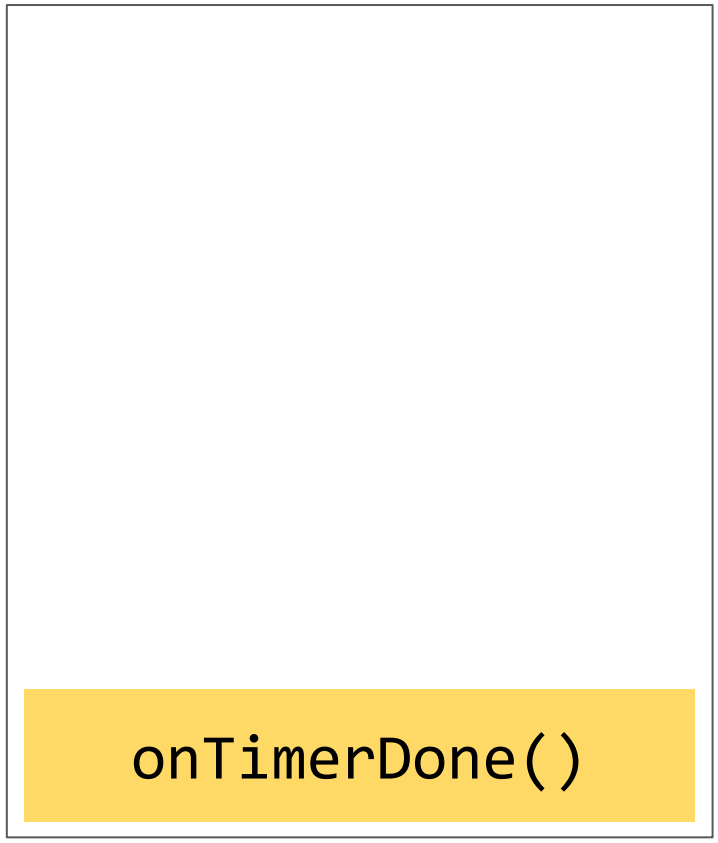


Call stack + `setTimeout`

```
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

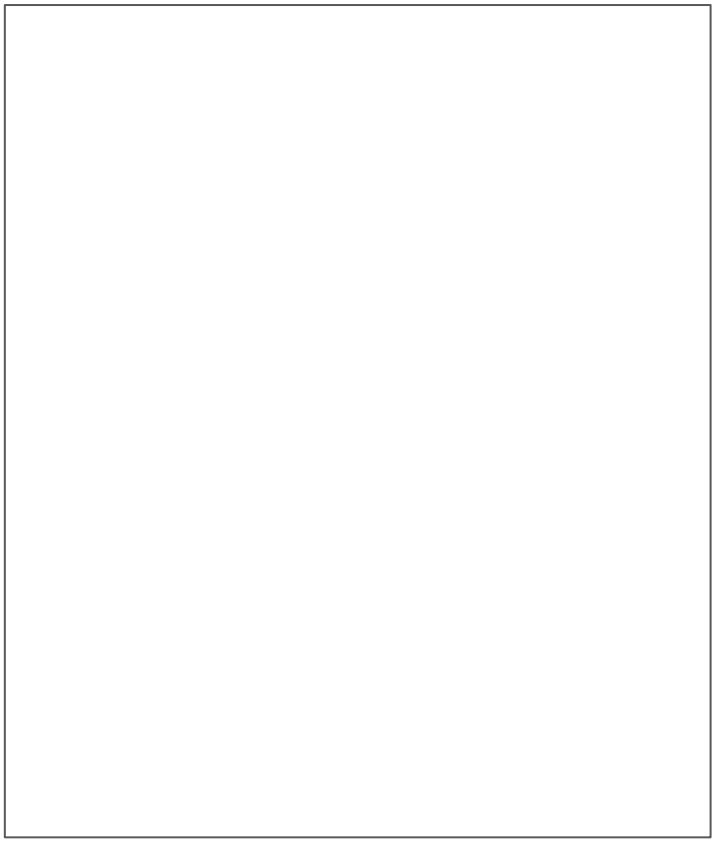


Call stack + `setTimeout`

```
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



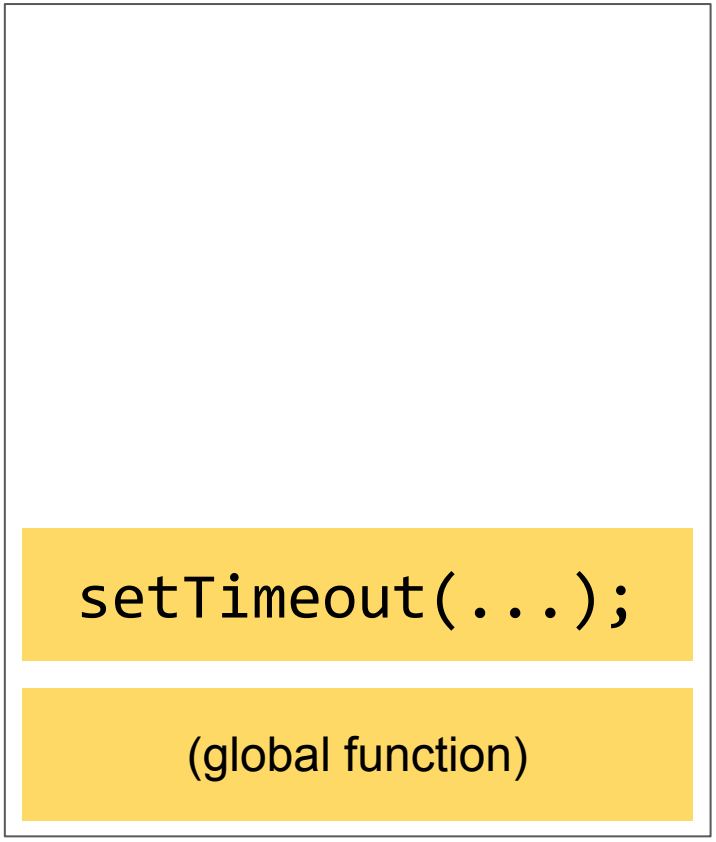
Call stack + `setTimeout`

```
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');
```

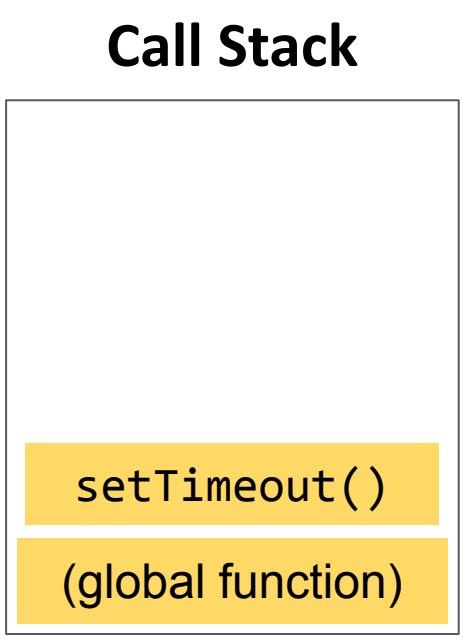
What "enqueues" `onTimerDone`?
How does it get fired?

Call Stack



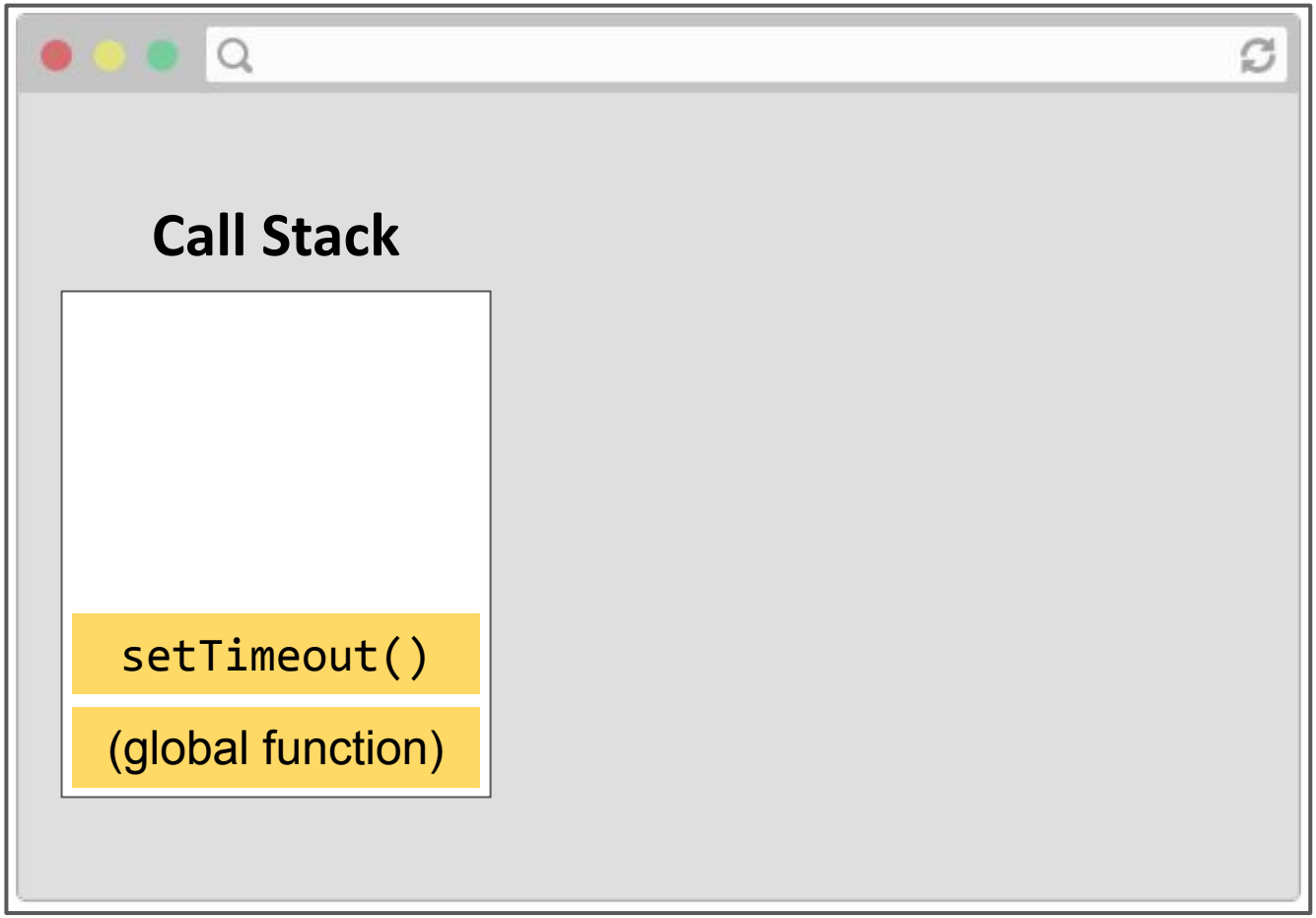
Tasks, Micro-tasks, and the Event Loop

Tasks and the Event Loop

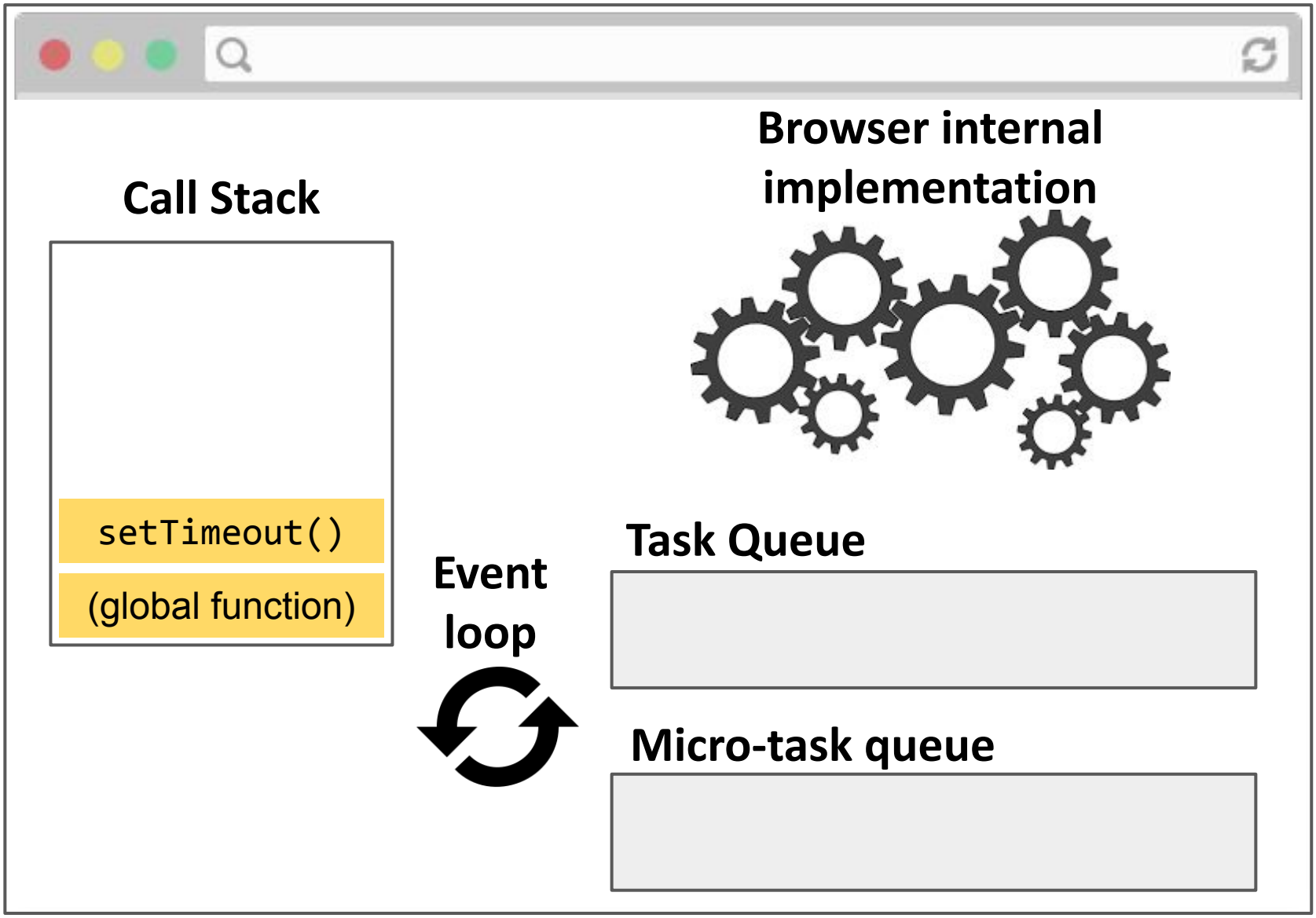


The JavaScript runtime can do only one thing at a time...

Tasks and the Event Loop

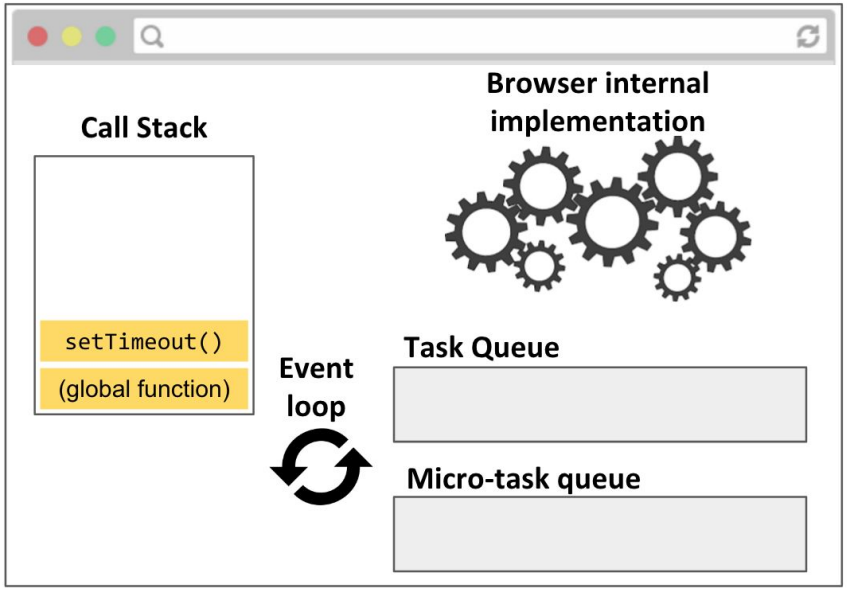


But the JS runtime runs within a browser, which can do multiple things at a time.



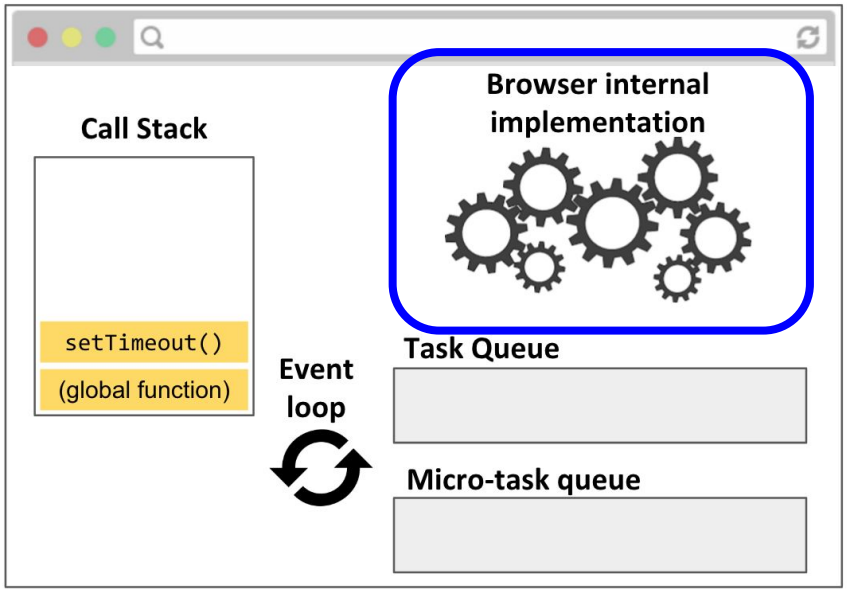
Here's a picture of the major pieces involved in executing JavaScript code in the browser.

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.

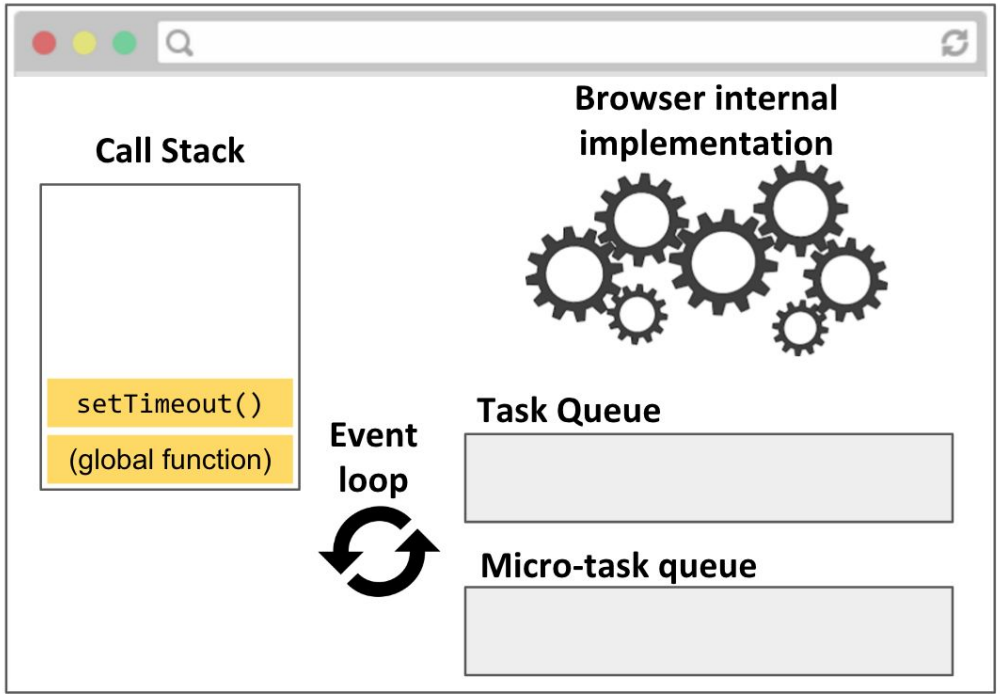
JS execution



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

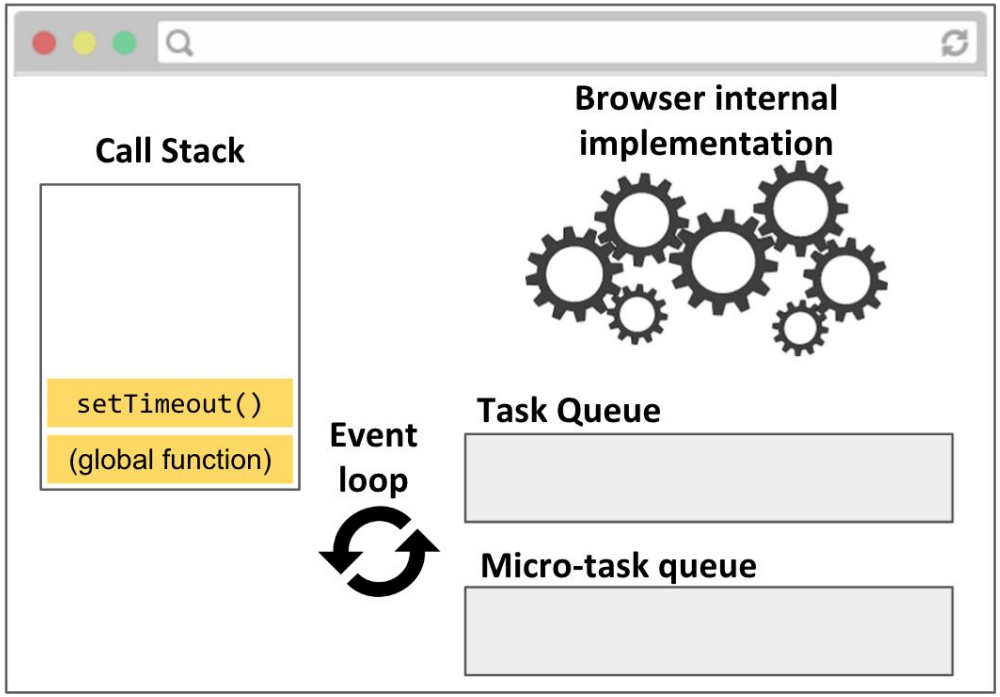
- **Call stack:** JavaScript runtime call stack. Executes the JavaScript commands, functions.
- **Browser internal implementation:** The C++ code that executes in response to native JavaScript commands, e.g. `setTimeout`, `element.classList.add('style')`, etc.

JS execution



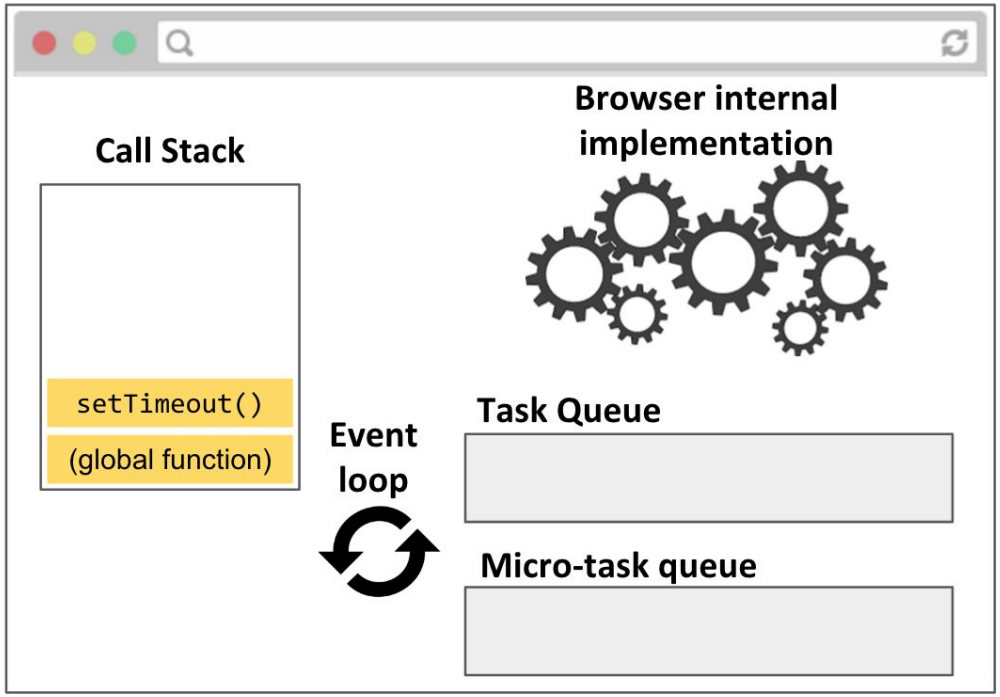
- **Task Queue:** When the browser internal implementation notices a callback from something like setTimeout or addEventListener is should be fired, it creates a Task and enqueues it in the Task Queue

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.