17 SEPTEMBER 2018 / #JAVASCRIPT

JavaScript Timers: Everything you need to know

by Samer Buna

A few weeks ago, I tweeted this interview question:

****** Answer the question in your head now before you proceed ******

About half the replies to the Tweet were wrong. The answer is **NOT** V8 (or 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. Timers are also implemented natively by the Node.js runtime itself.

In browsers, the main timer functions are part of the `Window` interface, which has a few other functions and objects. That interface makes all of its elements available globally in the main JavaScript scope. This is why you can execute `setTimeout` directly in your browser's console.

In Node, timers are part of the `global` object, which behaves similarly to the browser's `Window` interface. You can see the source code of timers in Node here.

Some might think this is a bad interview question — why does knowing this matter anyway?! As a JavaScript developer, I think you're expected to know this because if you don't, that might be a sign that you don't completely understand how V8 (and other VMs) interacts with browsers and Node.

Let's do a few examples and challenges about timer functions, shall we?

> **Update:** This article is now part of my "Complete Introduction to Node.js".
> You can read the updated version of it at here.

Delaying the execution of a function

Timer functions are higher-order functions that can be used to delay first argument).

Here's an example about delaying:

```
// example1.js
setTimeout(
  () => {
    console.log('Hello after 4 seconds');
  },
  4 * 1000
);
```

This example uses `setTimeout` to delay the printing of the greeting message by 4 seconds. The second argument to `setTimeout` is the delay (in ms). This is why I multiplied 4 by 1000 to make it into 4 seconds.

The first argument to `setTimeout` is the function whose execution will be delayed.

If you execute the `example1.js` file with the `node` command, Node will pause for 4 seconds and then it'll print the greeting message (and exit after that).

Note that the first argument to `setTimeout` is just a function **reference**. It does not have to be an inline function like what `example1.js` has. Here's the same example without using an inline function:

```
const func = () => {
  console.log('Hello after 4 seconds');
};
setTimeout(func, 4 * 1000);
```

Passing Arguments

If the function that uses `setTimeout` to delay its execution accepts any arguments, we can use the remaining arguments for `setTimeout` itself (after the 2 we learned about so far) to relay the argument values to the delayed function.

```
// For: func(arg1, arg2, arg3, ...)
// We can use: setTimeout(func, delay, arg1, arg2, arg3, ...)
```

Here's an example:

```
// example2.js
const rocks = who => {
  console.log(who + ' rocks');
};
setTimeout(rocks, 2 * 1000, 'Node.js');
```

The `rocks` function above, which is delayed by 2 seconds, accepts a `who` argument and the `setTimeout` call relays the value "*Node.js*" as that `who` argument.

Executing `example2.js` with the `node` command will print out "*Node.js rocks*" after 2 seconds.

Timers Challenge #1

Using what you learned so far about `setTimeout`, print the following 2

- Print the message "*Hello after 4 seconds*" after 4 seconds
- Print the message "*Hello after 8 seconds*" after 8 seconds.

Constraints:
You can define only a single function in your solution, which includes inline functions. This means many `setTimeout` calls will have to use the exact same function.

Solution

Here's how I'd solve this challenge:

```
// solution1.js
const theOneFunc = delay => {
  console.log('Hello after ' + delay + ' seconds');
};
setTimeout(theOneFunc, 4 * 1000, 4);
setTimeout(theOneFunc, 8 * 1000, 8);
```

I've made `theOneFunc` receive a `delay` argument and used the value of that `delay` argument in the printed message. This way, the function can print different messages based on whatever delay value we pass to it.

I then used `theOneFunc` in two `setTimeout` calls, one that fires after 4 seconds and another that fires after 8 seconds. Both of these `setTimeout` calls also get a **3rd** argument to represent the `delay` argument for `theOneFunc`.

Executing the `solution1.js` file with the `node` command will print the second message after 8 seconds.

Repeating the execution of a function

What if I asked you to print a message every 4 seconds, forever?

While you can put `setTimeout` in a loop, the timers API offers the `setInterval` function as well, which would accomplish the requirement of doing something forever.

Here's an example of setInterval:

```
// example3.js
setInterval(
  () => console.log('Hello every 3 seconds'),
  3000
);
```

This example will print its message every 3 seconds. Executing `example3.js` with the `node` command will make Node print this message forever, until you kill the process (with *CTRL+C*).

Cancelling Timers

Because calling a timer function schedules an action, that action can also be cancelled before it gets executed.

A call to `setTimeout` returns a timer "ID" and you can use that timer ID with a `clearTimeout` call to cancel that timer. Here's an example:

```
// example4.js
```

```
    0
  );
  clearTimeout(timerId);
```

This simple timer is supposed to fire after `0` ms (making it immediate), but it will not because we are capturing the `timerId` value and canceling it right after with a `clearTimeout` call.

When we execute `example4.js` with the `node` command, Node will not print anything and the process will just exit.

By the way, in Node.js, there is another way to do `setTimeout` with `0` ms. The Node.js timer API has another function called `setImmediate`, and it's basically the same thing as a `setTimeout` with a `0` ms but we don't have to specify a delay there:

```
setImmediate(
  () => console.log('I am equivalent to setTimeout with 0 ms'),
);
```

The `setImmediate` function is not available in all browsers. Don't use it for front-end code.

Just like `clearTimeout`, there is also a `clearInterval` function, which does the same thing but for `setInerval` calls, and there is also a `clearImmediate` call as well.

A timer delay is not a guaranteed thing

In the previous example, did you notice how executing something with setTimeout line), but rather execute it right away after everything else in the script (including the clearTimeout call)?

Let me make this point clear with an example. Here's a simple `setTimeout` call that should fire after half a second, but it won't:

```
// example5.js
setTimeout(
  () => console.log('Hello after 0.5 seconds. MAYBE!'),
  500,
);
for (let i = 0; i < 1e10; i++) {
  // Block Things Synchronously
}
```

Right after defining the timer in this example, we block the runtime synchronously with a big `for` loop. The `1e10` is `1` with `10` zeros in front of it, so the loop is a `10` Billion ticks loop (which basically simulates a busy CPU). Node can do nothing while this loop is ticking.

This of course is a very bad thing to do in practice, but it'll help you here to understand that `setTimeout` delay is not a guaranteed thing, but rather a **minimum** thing. The `500` ms means a minimum delay of `500` ms. In reality, the script will take a lot longer to print its greeting line. It will have to wait on the blocking loop to finish first.

Timers Challenge #2

Write a script to print the message "**Hello World**" every second, but only 5 times. After 5 times, the script should print the message "*Done*" and let the Node process exit.

Hint: You need a counter.

Solution

Here's how I'd solve this one:

```
let counter = 0;
const intervalId = setInterval(() => {
  console.log('Hello World');
  counter += 1;
if (counter === 5) {
    console.log('Done');
    clearInterval(intervalId);
  }
}, 1000);
```

I initiated a `counter` value as `0` and then started a `setInterval` call capturing its id.

The delayed function will print the message and increment the counter each time. Inside the delayed function, an `if` statement will check if we're at `5` times by now. If so, it'll print "*Done*" and clear the interval using the captured `intervalId` constant. The interval delay is `1000` ms.

Who exactly "calls" the delayed functions?

When you use the JavaScript `this` keyword inside a regular function, like this:

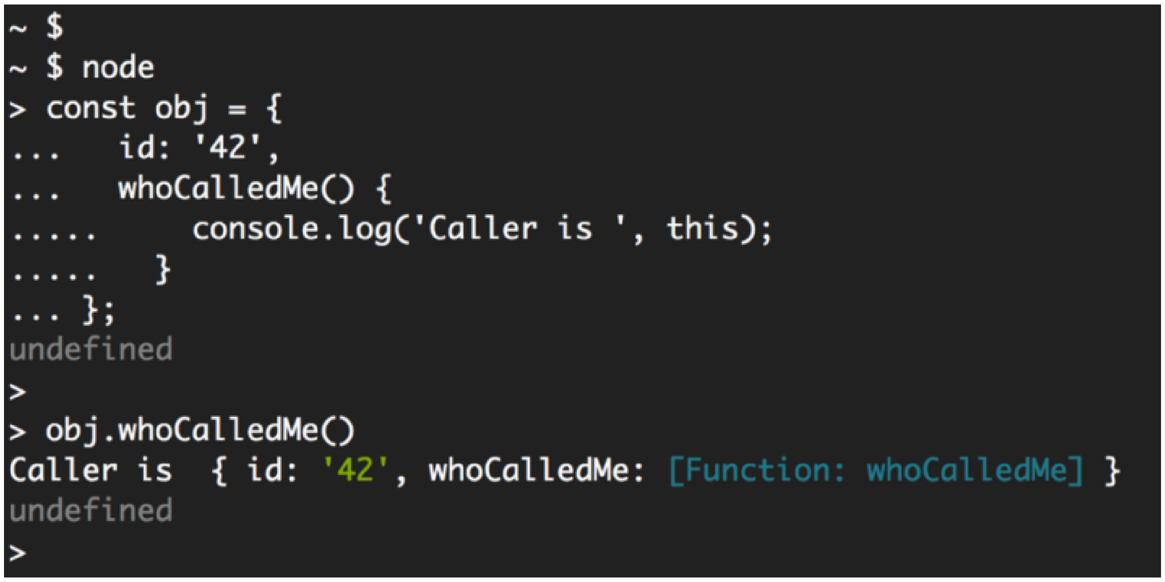
```
function whoCalledMe() {
  console.log('Caller is', this);
```

The value inside the `this` keyword will represent the **caller** of the function. If you define the function above inside a Node REPL, the caller will be the `global` object. If you define the function inside a browser's console, the caller will be the `window` object.

Let's define the function as a property on an object to make this a bit more clear:

```
const obj = {
  id: '42',
  whoCalledMe() {
    console.log('Caller is', this);
  }
};
// The function reference is now: obj.whoCallMe
```

Now when you call the `obj.whoCallMe` function using its reference directly, the caller will be the `obj` object (identified by its id):



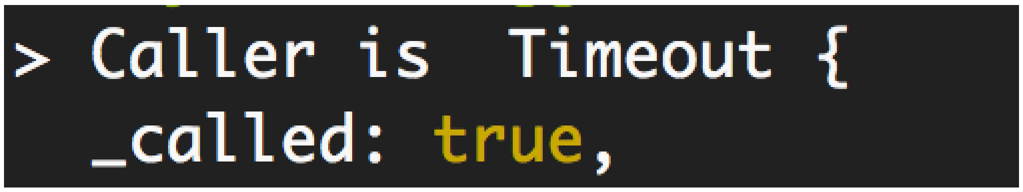
```
~ $
~ $ node
> const obj = {
...   id: '42',
...   whoCalledMe() {
.....     console.log('Caller is ', this);
.....   }
... };
undefined
>
> obj.whoCalledMe()
Caller is  { id: '42', whoCalledMe: [Function: whoCalledMe] }
undefined
>
```

Now, the question is, what would the caller be if we pass the reference of `obj.whoCallMe` to a `setTimetout` call?

```
// What will this print??
setTimeout(obj.whoCalledMe, 0);
```

Who will the caller be in that case?

The answer is different based on where the timer function is executed. You simply can't depend on who the caller is in that case. You lose control of the caller because the timer implementation will be the one invoking your function now. If you test it in a Node REPL, you'd get a `Timetout` object as the caller:



```
> Caller is  Timeout {
  _called: true,
```

Note that this only matters if you're using JavaScript's `this` keyword inside regular functions. You don't need to worry about the caller at all if you're using arrow functions.

Timers Challenge #3

Write a script to continuously print the message "*Hello World*" with varying delays. Start with a delay of 1 second and then increment the delay by 1 second each time. The second time will have a delay of 2 seconds. The third time will have a delay of 3 seconds, and so on.

include the delay in the printed message. Expected output looks like:

```
Hello World. 1
Hello World. 2
Hello World. 3
...
```

Constraints: You can only use `const` to define variables. You can't use `let` or `var`.

Solution

Because the delay amount is a variable in this challenge, we can't use `setInterval` here, but we can manually create an interval execution using `setTimeout` within a recursive call. The first executed function with setTimeout will create another timer, and so on.

Also, because we can't use let/var, we can't have a counter to increment the delay in each recursive call, but we can instead use the recursive function arguments to increment during the recursive call.

Here's one possible way to solve this challenge:

```
const greeting = delay =>
  setTimeout(() => {
    console.log('Hello World. ' + delay);
    greeting(delay + 1);
  }, delay * 1000);
greeting(1);
```

Write a script to continuously print the message "*Hello World*" with the same varying delays concept as challenge #3, but this time, in groups of 5 messages per main-delay interval. Starting with a delay of 100ms for the first 5 messages, then a delay of 200ms for the next 5 messages, then 300ms, and so on.

Here's how the script should behave:

- At the 100ms point, the script will start printing "Hello World" and do that 5 times with an interval of 100ms. The 1st message will appear at 100ms, 2nd message at 200ms, and so on.
- After the first 5 messages, the script should increment the main delay to 200ms. So 6th message will be printed at 500ms + 200ms (700ms), 7th message will be printed at 900ms, 8th message will be printed at 1100ms, and so on.
- After 10 messages, the script should increment the main delay to 300ms. So the 11th message should be printed at 500ms + 1000ms + 300ms (18000ms). The 12th message should be printed at 21000ms, and so on.
- Continue the pattern forever.

Include the delay in the printed message. The expected output looks like this (without the comments):

```
Hello World. 100  // At 100ms
Hello World. 100  // At 200ms
Hello World. 100  // At 300ms
Hello World. 100  // At 400ms
Hello World. 100  // At 500ms
Hello World. 200  // At 700ms
Hello World. 200  // At 900ms
```

Constraints: You can use only `setInterval` calls (not `setTimeout`) and you can use only ONE if statement.

Solution

Because we can only use `setInterval` calls, we'll need recursion here as well to increment the delay of the next `setInterval` call. In addition, we need an if statement to control doing that only after 5 calls of that recursive function.

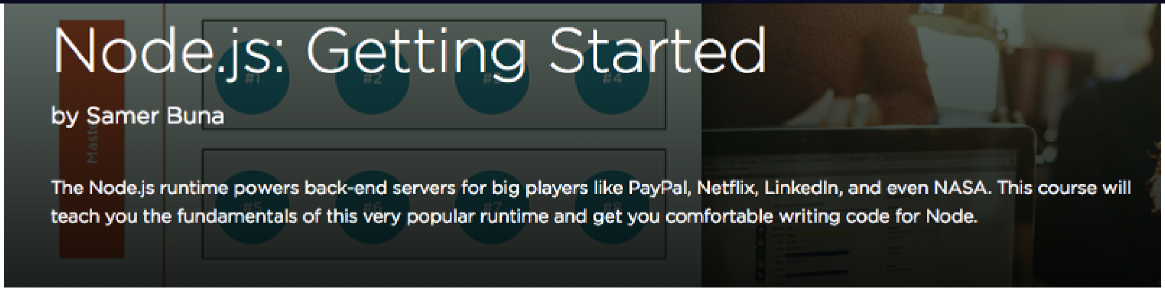
Here's one possible solution:

```
let lastIntervalId, counter = 5;
const greeting = delay => {
  if (counter === 5) {
    clearInterval(lastIntervalId);
    lastIntervalId = setInterval(() => {
      console.log('Hello World. ', delay);
      greeting(delay + 100);
    }, delay);
    counter = 0;
  }
counter += 1;
};
greeting(100);
```

Thanks for reading.

If you're just beginning to learn Node.js, I recently published a **first-steps course at Pluralsight**, check it out:

freeCodeCamp(🔥) Donate


Node.js: Getting Started
by Samer Buna
The Node.js runtime powers back-end servers for big players like PayPal, Netflix, LinkedIn, and even NASA. This course will teach you the fundamentals of this very popular runtime and get you comfortable writing code for Node.

https://jscomplete.com/c/nodejs-getting-started

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

Trending Guides

freeCodeCamp(🔥) Donate

CSS Box Shadow	What is GitHub?
Python List Append	Python Sort List
JavaScript Array Sort	Comments in JSON
Symlink in Linux	What is Kanban?
Linux Grep Command	Python Write to File
What is DNS?	CSS Media Queries
Primary Key SQL	HTML Entities
SQL Update Statement	Excel VBA
Screenshot on PC	LOOKUP in Excel
What is a Proxy Server?	Arrow Function JavaScript
Cat Command in Linux	Remove Duplicates in Excel
CSS Background Image	dllhost.exe COM Surrogate
HTML Background Color	Boolean Algebra Truth Table
CSS Comment Example	Video Chat for Android

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty

Code of Conduct Privacy Policy Terms of Service Copyright Policy