Callbacks, setTimeout, & the Event Loop

Learn how JavaScript is both single-threaded and asynchronous. We'll go over how the event loop creates an event-based system that is at the core of how modern websites work. We'll see how to respond to events efficiently and how to write asynchronous code.

What Callbacks Do

Callbacks are absolutely vital to asynchronous programming. The only way for us to tell JavaScript to do something in the future is to provide a callback that will be invoked later.

JavaScript allows us to *attach* a callback to a particular event. We can tell JavaScript that when some event happens, we want to run some function. That's the gist of the whole concept.

The event can be anything. In a browser, it can be a button click from a user. On a file system, it can be a file read operation. We can attach handlers to these events, saying that every time this event happens, we want something else to happen.

For example, when a user clicks a button, perhaps we want to load more content or open a pop-up. When a file read operation occurs, perhaps we want to log the event for security and auditing purposes.

To set up an event listener on a button, we pass in a callback to a function such as jQuery. That function binds our callback to the button. When the button is clicked, our callback is called and whatever changes need to happen to the page occur through it.

How to Use them

DOM

If you've ever used jQuery you've seen that it's entirely based on callbacks.

```
$('#submitButton').on('click', function() {
    console.log('You clicked me!');
});
```

The second parameter to `$('#submitButton').on` is an anonymous function. If we had an HTML page with a button with an id of `submitButton`, this code block would set that function as the event handler on it. It'll simply call the function and log `'You clicked me!'` every time we click the button.

setTimeout

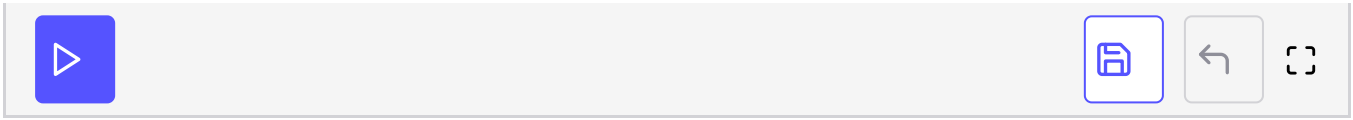
`setTimeout` is an asynchronous function that we can explore right away built into JavaScript. We don't need a browser. It'll run in any JavaScript environment. It takes in a callback function as a parameter and a number as the second parameter.

The number we pass in is how long `setTimeout` will wait before calling our function, in milliseconds. So if we want to wait 1 second, we would pass in `1000`.

Unfortunately, we can't view asynchronous code working in this code editor - the output loads only after everything has run, including asynchronous code. Feel free to copy and paste the code snippets here to try them out in another JavaScript evaluator such as repl.it.

```
function printCharacters() {
    console.log('Logged!');
}

setTimeout(printCharacters, 1000);
// (After one second:) Logged!
```

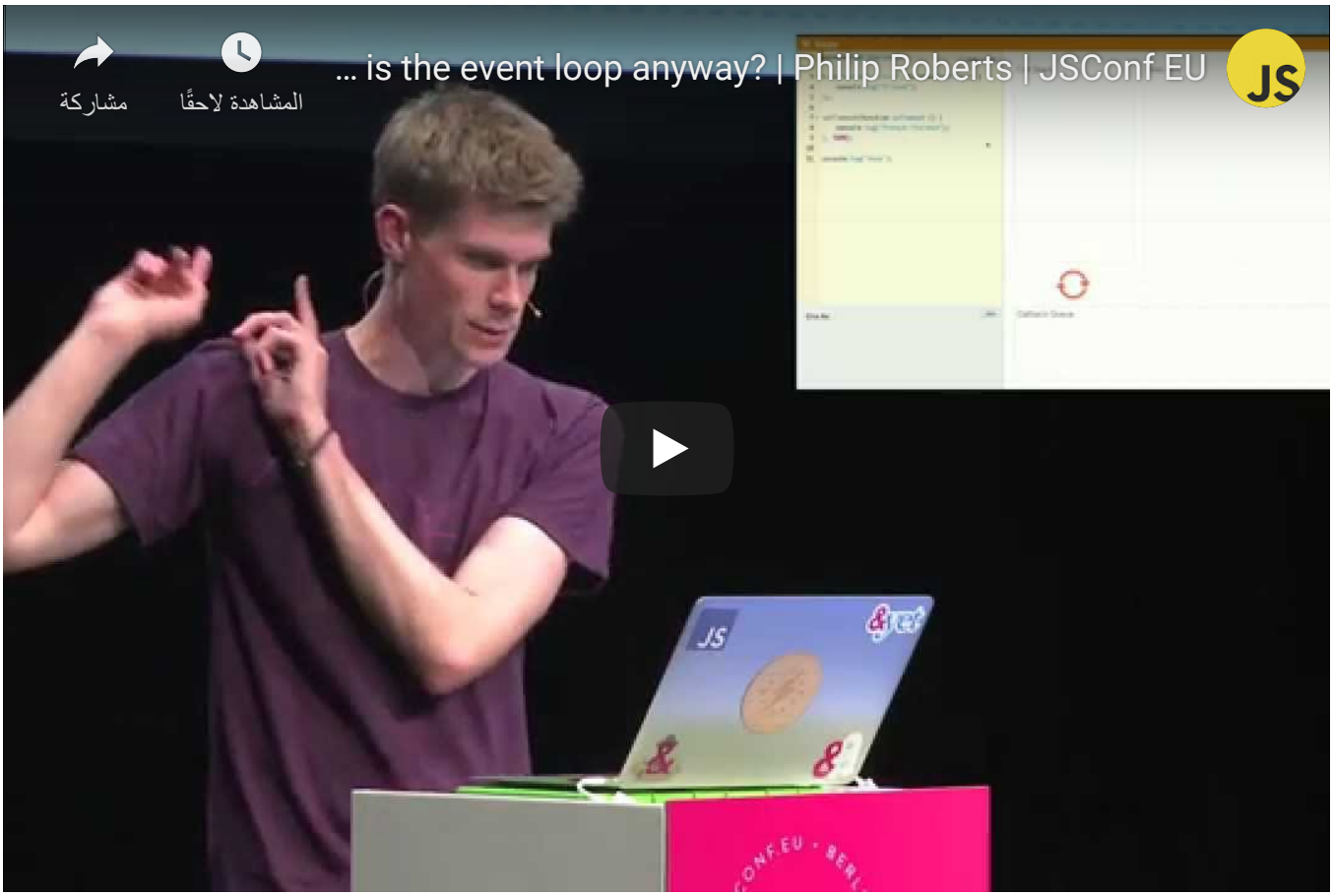


Event Loop

The implementation of asynchronous code under the hood is an artifact called the *event loop*. I could try to explain it but this video does a better job than I ever could.

It's not vital that you understand every mechanism of the event loop. Try to gain an understanding of how events are scheduled asynchronously.

Understand what `setTimeout(fn, 0)` does and why we'd want to use it.



In summary, `setTimeout` throws our callback on the event loop. When the engine is idle and the amount of time we specified has passed, it'll run our callback.

Even `setTimeout(callback, 0)` is asynchronous. It's going to run the callback after the engine is done doing whatever it's doing at the moment. Only after all current function calls have returned will this callback run.

We'll cover common use cases in the next lesson.