Back to Week 2

JS | Async & Callbacks

LESSON

Learning Goals

After this lesson you will be able to:

- understand the concepts of **synchronous** and **asynchronous** code and what is the difference between them,
- understand what a **callback function** is,
- understand how *async* functions execute callbacks,
- use the `setTimeout()` and `clearTimeout()` methods and
- use the `setInterval()` and `clearInterval()` methods.

Introduction

Let's start with the most common interview question:

❓ *Interviewer*: "Hi! Welcome to the job interview. To warm up, can you please tell me all you know about *JavaScript and synchronicity*? Why do we say JavaScript is *asynchronous language*?"

📣 *You*: "Thank you so much for giving me a chance and I'm going to start from your second question and make a small correction, which you probably did purposely to test me - **JavaScript is not an asynchronous language, but synchronous one with some asynchronous behaviors**."

Synchronicity and Asynchronicity

A good approach to understanding these two behaviors is to use an example from real life. Suppose you are at home. You are hungry, and you have to do the dishes, organize your bedroom, and call your significant other. **You can't do all the things at the same time!**

If you would do it in a **synchronous** manner, you would:

1. prepare something to eat
2. do the dishes
3. organize your bedroom
4. do whatever is next on your list.

> This means that each of these activities would take some time and all the others would have to wait till you're done with the previous.

This process would take a looonggg time, or better saying, much longer than if we would take the *asynchronous* approach.

If we would do all the above **asynchronously**, it would look like:

- order something to eat
- while waiting first clean all the dirty dishes
- then organize the room
- food is here 👍

This is what we could consider an **async method** and a **callback function**.

Calling the restaurant to order the meal (the *async method* in this example) will allow you to work on other stuff while you are waiting for the food. Once you get the food, you briefly pause what you're doing to receive it. The interruption is the *callback function*.

JavaScript and (a)synchronicity

🏆 *Potential Interview questions*:
JavaScript, at its core, is a single-threaded and synchronous language, and this means next:

- **single-threaded** - **only one block of code is executed at the time**. You can imagine this as there's always one single actor in the play - ex. you were alone at home in the above example, so only you could've worked on all these tasks. There's no one else who can do it simultaneously at the same time as you. This means that if you (JavaScript engine) work quickly enough and can switch between tasks efficiently enough, you will manage to finish all the tasks like you had a friend or more of them helping you (but remember, in JS case, this is impossible - always one operation at the time - forget about friend's help when you're JavaScript. 😜)
- **synchronous** - **the code gets executed line by line, from top to bottom, in the order in which they are put in** - line 2 can't get executed before line 1, line 3 can't get executed before line 2, and so on.

❓ Okay, now you might ask, if JavaScript is *synchronous* language, how do we get to deal with *asynchronicity* at all? And this is a valid question, so let's demystify this concept a bit.

Async & Callbacks

Asynchronous Programming

Unlike the previous statements, where synchronous programs run line by line from top to bottom, **in asynchronous programs it is possible to have code on line 1 scheduled to be run at some point in the future so in the meantime, code on lines 2,3, and so on can run**.

We can imagine that our "code in line 1", which is scheduled to be run in the future, is some super *time-inefficient* activity, like getting thousands of objects with users' data from some external API. If we ran this program synchronously, the whole app execution would stop until all the data is loaded and then we could proceed to lines 2, 3, 4, ... Doesn't really make sense, you would agree?

To conclude:

Asynchronous programming helps us to avoid performance bottlenecks and **enhance the responsiveness of our applications**. It is especially useful to execute other functions while you wait until a costly function finishes. We usually **use async when we have to execute functions that will take an unpredictable amount of time to finish**.

For example:

```
function getFirstElementOfArray(array) {
  return array[0];
}

const array = ['Madrid', 'Barcelona', 'Miami'];
const firstElement = getFirstElementOfArray(array);

console.log(firstElement); // <== Madrid
```

Explain this code

The execution of this function obviously won't be so expensive in terms of time, so it doesn't have to be async. Let's take a look at a different example:

```
// hypothetical example

function readFile(file) {
  // read the file
  return contentFile.length;
}

const textSize = readFile('odyssey.txt');
console.log(textSize); // => undefined
```

Explain this code

If the file we want to read is a huge book like the Odyssey, which has around 120.000 words, this operation will be more expensive in terms of time.

The problem is - we will have to pause the execution of the rest of the code until this is done. To fix it, we will have to write **an async function.**

> **Async functions allow us to continue executing our code while the async function is being executed in the background**. What this exactly means is: our users will be able to use the app while the data is loading, otherwise, our app would be irresponsive and this would cause a bad user experience.

Let's take a look at this example:

```
normalFunction(); // => "hi", takes 0.1s
asyncFunction(); // => "there", takes 4s
normalFunction2(); // => "ironhackers", takes 0.1s
```

Explain this code

The result of the code above will be:

```
"hi"
"ironhackers"
"there"
```

Explain this code

What would we do if we needed the value of an *asyncFunction()* as an argument in another function?

Suppose we want to concat these three values, passing the outputs as arguments to the next function:

```
//     |------------------------------|
//     |                              |
const text1 = normalFunction(); //    |
//                                -------
//                              |
//                             V
const text2 = asyncFunction(text1);
//     |
//     |-------------------------|
//                               |
//                              V
const text3 = normalFunction2(text2);

console.log(text3);
```

Explain this code

The output may be something different than we expect:

```
"hi undefined there"
```

Explain this code

The `asyncFunction()` takes 4 seconds to be executed, while the other two functions need only 0.1 seconds. The system doesn't know the returned value from the `asyncFunction()` until it finishes its execution.

Callback Function

✔ The **callback function** contains the code that will be executed when the async function finishes its execution. The syntax to define this function will change depending on the async function.

We will understand callbacks better by taking a look at our first _async methods: `setTimeout()` and `setInterval()`.

`setTimeout()`

`setTimeout()` sets a timer that executes a callback function once the timer expires.

Syntax:

```
const timeoutId = setTimeout (callbackFunction [, 'delay]);

```

Explain this code

Parameters:

- `callbackFunction`: the function that will be executed once the timer expires
- `delay` (optional): the time (in milliseconds) the timer should wait before the given callback function is executed

The method returns a numeric `timeoutId`, which identifies the timer created by the call to `setTimeout()`. We can cancel the timeout by passing this `id` to the `clearTimeout()` method.

Let's take a look at the following example to see how `setTimeout()` works. You can try this code in a new Repl.it note:

```
// ES5
function someCallbackFunction() {
  console.log('Hey there, Ironhackers!');
}
const timeoutId = setTimeout(someCallbackFunction, 1000);
```

Explain this code

After one second, in the console, it will print "Hey there, Ironhackers!". **Cool, huh?**

Let's try to change the delay and set it up to five seconds.

```
const timeoutId = setTimeout(someCallbackFunction, 5000);
```

Explain this code

Okay, so now the message will be shown five seconds after calling the function. Let's create the timer and avoid the execution of the callback function by doing the following:

```
const timeoutId = setTimeout(someCallbackFunction, 5000);

clearTimeout(timeoutId);
```

Explain this code

Sidenote: You might see this way of writing `setTimeout()` much often:

```
// ES5
const timeoutId = setTimeout(function () {
  console.log('Hey there, Ironhackers!');
}, 1000);

// ES6
const timeoutId = setTimeout(() => {
  console.log('Hey there, Ironhackers!');
}, 1000);
```

Explain this code

We broke it into 2 steps previously just to make it more clear how it is a callback really.

📝 **Time to practice**

Let's create a counter that will print a number in a sequence each second.

```
let counter = 1;
const callbackFunction = function () {
  console.log(counter);
  setTimeout(callbackFunction, 1000);

  counter += 1;
};

let timeoutId = setTimeout(callbackFunction, 1000);
```

Explain this code

Could you stop the counting with the `clearTimeout()` method? No! Every time the callback function is executed, a new timeoutId is created but we are not saving it.

Let's modify the code above to stop the timeout after 10 iterations.

```
let counter = 1;
const callbackFunction = function () {
  console.log(counter);
  timeoutId = setTimeout(callbackFunction, 1000);

  counter += 1;

  if (counter > 10) {
    clearTimeout(timeoutId);
  }
};

let timeoutId = setTimeout(callbackFunction, 1000);
```

Explain this code

Though the code above works executing the function every second, **not everything that works is a good solution**. In this case, we are using `setTimeout()` to do something that we could better do with another method: `setInterval()`.

`setInterval()`

`setInterval()` calls a function repeatedly with a fixed delayed time between each call.

Syntax:

```
const intervalId = setInterval(callbackFunction, delay);
```

Explain this code

Parameters:

Parameters:

- `callbackFunction`: the function that will be executed once the timer expires
- `delay`: the time (milliseconds) the timer should delay in between executions of the specified callback function

The method returns a numeric `intervalId`, which identifies the timer created by the call to `setInterval()`. We can cancel the interval by passing this `id` to the `clearInterval()` method.

The usage is very similar than `setTimeout()`. Now we can create the counter from the example above as follows:

```
let i = 1;
const intervalId = setInterval(function () {
  console.log(i);

  i++;

  if (i > 10) {
    clearInterval(intervalId);
  }
}, 1000);
```

Explain this code

Time to practice

Let's do a reverse countdown from 10 to 0. When the countdown is zero, it should show "Pop!" and stop the interval.

```
let i = 10;
const intervalId = setInterval(function () {
  if (i > 0) {
    console.log(i);
  } else {
    console.log('Pop!');
    clearInterval(intervalId);
  }

  i--;
}, 1000);
```

Explain this code

While `setTimeout()` executes the function just once, `setInterval()` executes the given function repeatedly until the `clearInterval()` function is called.

A bit later in the course, we will cover other ways to deal with the asynchronicity of JavaScript such are *promises* and *async/await*.

Summary

Potential Interview questions:
What is the difference between synchronous and asynchronous code in JavaScript?

In short, **synchronous** means the operation needs to be executed in a certain order, and each operation has to wait for the previous one to complete.
Asynchronous means the opposite of the previous - an operation can occur while another operation is still being processed.

Async JavaScript allows us to execute time-consuming functions without blocking the rest of our code. Once the async function finishes, the callback function is executed.

We have also learned two different methods to perform tasks with a delay. **`setTimeout()` will execute a function with a delayed time just once**. **`setInterval()` will execute a function repeatedly with a fixed delayed time between each call**.

Extra Resources

- Understanding Synchronous and Asynchronous JavaScript – Part 1
- MDN Documentation - setTimeout
- MDN Documentation - setInterval
- Callback hell example

Mark as completed

PREVIOUS
← JS | Debugging, Error Handling and JS Hint

NEXT
LAB | JS Chronometer - iteration 1 →