

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Product](#)

[Pricing](#)

[Customers](#)

[Resources](#)

[Blog](#)

[Podcast](#)

[Software Engineering](#)

[Product News](#)

[Podcast](#)

[Greatest Hits](#)

[write with us](#)

1 Jun 2023 · Software Engineering

Asynchronous JavaScript for Beginners



Written by:
Daniel Aganem



Reviewed by:
Tomas Fernandez

14 min read

Share this



Contents

Imagine a chef working in a busy restaurant kitchen. In a synchronous kitchen, the chef would prepare one dish at a time, waiting for each dish to finish cooking before starting the next one. This can be slow and inefficient, especially when many orders are in the queue. In an asynchronous kitchen, however, the chef could work on multiple dishes simultaneously, checking on each one periodically and adjusting the cooking times as necessary.

This allows the chef to fulfill orders quickly and ensures that customers receive their food as soon as possible. Similarly, asynchronous programming enables computer programs to perform multiple tasks concurrently, resulting in faster and more efficient execution.

Learn CI/CD

Level up your skills and use CI/CD at

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Blog](#)[Software Engineering](#)[Product](#)[Podcast](#)[Greatest Hits](#)[write with us](#)

[Product](#)[Pricing](#)[Customers](#)[Resources](#)[Blog](#)[Podcast](#)

In this tutorial, we'll go over the asynchronous components of JavaScript.

What is asynchronous Javascript?

Asynchronicity means that if JavaScript has to wait for an operation to complete, it will execute the rest of the code while waiting.

Note that JavaScript is [single-threaded](#). This means that it carries out asynchronous operations via the callback [queue](#) and [event loop](#).

In synchronous JavaScript, each function is performed in turn, waiting for the previous one to complete before executing the subsequent one. Synchronous code is written from top to bottom.

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Product](#)

[Pricing](#)

[Customers](#)

[Resources](#)

[Blog](#)

[Podcast](#)

[Software Engineering](#)

[Product News](#)

[Podcast](#)

[Greatest Hits](#)

[write with us](#)



The nature of JavaScript and why async programming is required

JavaScript is single-threaded and has a global [execution context](#), as we learned earlier.

As a result, by design, JavaScript is synchronous and has a single [call stack](#). Code will be executed in a [last-in, first-out](#) (LIFO) order in which it is called.

To understand what synchronous JavaScript means, let us consider the code snippet below.

```
//synchronous javascript  
  
console.log("synchronous.");  
  
console.log("synchronous javascript!")  
  
console.log("synchronous again!")
```

If we go ahead and run the above code, we get:

No More Seat Costs: [Semaphore Plans Just Got Better!](#)



bottom, as we can observe the console running the function linearly from top to bottom.

With the code above, the console first logs synchronous to the terminal, then synchronous javascript! and synchronous again!. This is an example of synchronous JavaScript since the program executes code in the order it is written.

Async programming is vital because it allows multiple processes to run concurrently without interfering with the [main thread](#).

This is significant because the main thread is in charge of managing the call stack, which is a data structure that holds the current sequence of function calls.

Blockage of the main thread leads to decreased performance because async programming permits the main thread to stay unblocked, and additional tasks can be completed while the asynchronous task is ongoing.

Let's look at a JavaScript function call called the `setTimeout()` function, which allows us to run javascript code after a certain amount of time.

After the specified time, the `setTimeout()` method executes a block of code. The method only runs the code once.

To work with the `setTimeout()` function, we need to use an

No More Seat Costs: [Semaphore Plans Just Got Better!](#)



1. function: a function that includes a code block.
2. milliseconds: the time it takes for the function to be executed

To explain this, let's look at the code below:

```
console.log("asynchronous.");  
setTimeout(() =>  
  console.log("asynchronous javascript!"), 3000);  
console.log("asynchronous again!");
```

The code above won't run synchronously on the JavaScript engine, in contrast to our prior example.

Let's look at the output displayed below:

```
// asynchronous.  
// asynchronous again!  
// asynchronous javascript!
```

Step 1: The first line of code will be executed, logging the string asynchronous to the console.

Step 2: The setTimeout method is invoked, which will execute the anonymous function after 3 seconds (3,000 milliseconds). The anonymous function will log asynchronous javascript! to the console.

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Product](#)

[Pricing](#)

[Customers](#)

[Resources](#)

[Blog](#)

[Podcast](#)


[Software Engineering](#)

[Product News](#)

[Podcast](#)

[Greatest Hits](#)

[write with us](#)



In other words, with asynchronous JavaScript, the JavaScript doesn't wait for answers before continuing to execute subsequent functions.

This is useful for applications that require a lot of processing power since it permits numerous tasks to be completed at the same time.

Techniques for writing asynchronous JavaScript

JavaScript can behave in an asynchronous way. Let's have a look at some techniques that are useful for asynchronous JavaScript:

We will discuss:

- promises
- async/await
- callbacks

Callbacks: this allows for asynchronous code to be written in a synchronous fashion.

Promises: writing asynchronous JavaScript code is made easy with promises. They enable you to create code that runs after a predetermined period of time or when a predetermined condition is satisfied.

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Product](#)

[Pricing](#)

[Customers](#)

[Resources](#)

[Blog](#)

[Podcast](#)

[Software Engineering](#)

[Product News](#)

[Podcast](#)

[Greatest Hits](#)

[write with us](#)



In JavaScript, functions are first-class citizens. Therefore, you can pass a function to another function as an argument. By definition, a callback is a function that executes after the outer code call has finished running. It is supplied as an input to another function. When a function has completed its purpose, it is utilized to enable it to invoke another function.

Let's take a look at the code example below:

```
function incrementDigits(callback) {  
    callback();  
}
```

The `incrementDigits()` function takes another function as a parameter and calls it inside. This is valid in JavaScript and we call it a callback. So, a function that is passed to another function as a parameter is a callback function.

In order to carry out asynchronous actions in JavaScript, such as executing an AJAX request or anticipating a user click, callbacks are widely utilized. Having learned about the `setTimeout()` function in the previous section, You can also write callback functions as an ES6 arrow function, which is a newer type of function in JavaScript:

```
setTimeout(() => {  
    console.log("output is initiated after 5 seconds");  
}, 5000);
```

The `setTimeout()` method is being used in this code to

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Product](#)[Pricing](#)[Customers](#)[Resources](#)[Blog](#)[Podcast](#)

[Software Engineering](#)[Product News](#)[Podcast](#)[Greatest Hits](#)[write with us](#)

```
function incrementDigits(num, callback) {
  setTimeout(function() {
    num++;
    console.log(num);
    if (num < 10) {
      incrementDigits(num, callback);
    } else {
      callback();
    }
  }, 1000);
}

incrementDigits(0, function() {
  console.log('done!');
});
```

This code is an example of [callback hell](#). It is a recursive function that calls itself until the number is greater than 10. Then, it executes the callback function. The `setTimeout()` function is used to delay the execution of the code in order to simulate a longer process. The function will print out each number, starting at 0, until it reaches 10, and then the `done!` message will be printed out. As a result, the code can run asynchronously, which means that the functions can run concurrently rather than waiting for one another to complete. We have already seen how to create an asynchronous callback in the previous section.

How asynchronous functions and callbacks corks under the hood

When an asynchronous function encounters the term `await`s, it stops execution until the promise is resolved, while an asynchronous callback is a higher-order function

No More Seat Costs: [Semaphore Plans Just Got Better!](#)



Handling callback hell

In JavaScript, the way to create a callback function is to pass it as a parameter to another function and then call it back after the task is completed.

There are ways to handle callbacks.

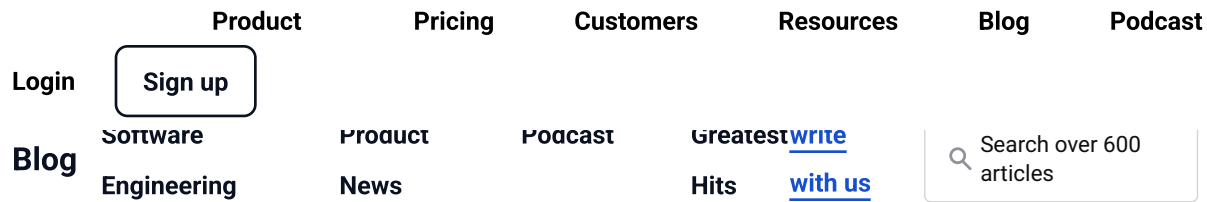
1. Use of promise: a promise can be generated for each callback. When the callback is successful, the promise is resolved, and if the callback fails, the promise is rejected.
2. Use of async-await: asynchronous functions are executed sequentially with the use of await; execution stops until the promise is resolved and function execution is successful.

Promise

An async promise operation's eventual success or failure is represented as a JavaScript object. It enables the creation of asynchronous code that works and appears synchronous. A promise, in our context, is something that will take some time to do. A promise has three possible states: pending, fulfilled, or rejected.

An asynchronous operation is still in progress while a promise is still unfulfilled. promise fulfillment indicates the successful completion of an asynchronous operation. Using the promise constructor, you could make a promise in the following way:

No More Seat Costs: [Semaphore Plans Just Got Better!](#)



This function is called the executor function.

```
// Promise constructor as an argument

function(resolve, reject) {
  // doSomethingHere
}
```

The executor function takes two arguments, resolve and reject. Your logic goes inside the executor function, which runs automatically when a new promise is created and processes the code as follows:

Step 1: A new promise is created using the promise constructor and two arguments, resolve and reject.

Step 2: The `.then()` method is called on the promise, which takes two callback functions, one for if the promise is resolved and one for if the promise is rejected.

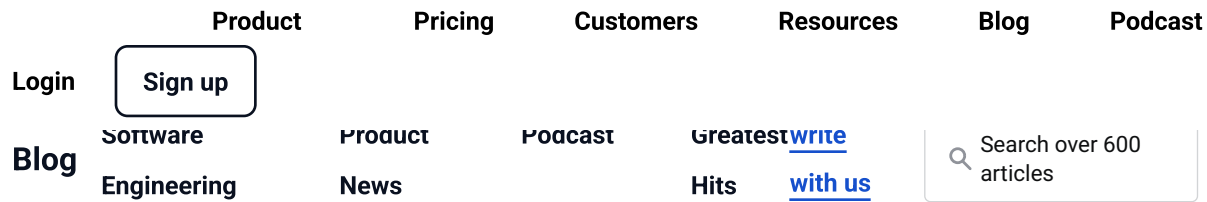
Step 3: The first callback function will be executed if the promise is successfully resolved, while the second callback function will be executed if the promise is rejected.

Successful call completions are indicated by the resolve function call, and a rejected promise indicates that the asynchronous operation has failed.

Converting a callback to a promise

To convert a callback to a promise, the first step is to create a new promise object. The promise constructor takes two

No More Seat Costs: [Semaphore Plans Just Got Better!](#)



object is returned after the callback has been activated so that it can be used in subsequent processing.

By converting callbacks to promises, developers are able to write code that is more concise and easier to understand.

```
const incrementDigits = num => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      num++;
      console.log(num);
      if (num < 10) {
        resolve(incrementDigits(num));
      } else {
        resolve('done!');
      }
    }, 1000);
  });
};

incrementDigits(0).then(res => console.log(res));
```

Promises are used in this code to carry out an asynchronous operation. Through a technique called [promise chaining](#), promises are a clever way to address issues caused by callback hell.

With less code and simpler techniques, you can use this method to sequentially retrieve data from multiple endpoints. The code above creates a function called `incrementDigits()` which takes an argument of `num` (assumed to be a number). It then makes a new promise and passes it two arguments, `resolve` and `reject` (both of which

No More Seat Costs: [Semaphore Plans Just Got Better!](#)



logged.

To manage the promise and return the result of the `incrementDigits()` asynchronous operation, use the `.then()` method. If an error condition arises inside a promise, you “reject” the promise by calling the `reject()` function with an error. To handle a promise rejection, you pass a callback to the `catch()` function.

Promise handler methods

These handlers are simple functions that help to create the link between the executor and the consumer functions so that they can be in synchronization when a promise is resolved or rejected. Instead of just returning the value, however, we return an object that contains this data.

The major promise handler methods are:

- `.then()`
- `.catch()`
- `.finally()`

There are other promise handler methods as well, such as `.all()`, which creates a single promise that resolves once all of the promises in an array have been fulfilled, and `.race()`, which creates a single promise that resolves or rejects as soon as one of the promises in an array has been fulfilled, respectively.

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Product](#)[Pricing](#)[Customers](#)[Resources](#)[Blog](#)[Podcast](#)

[Software Engineering](#)[Product News](#)[Podcast](#)[Greatest Hits](#)[write with us](#)

tuitioned, allows us to write asynchronous functions as though they were synchronous (executed sequentially). Async/await is a technique for building asynchronous code that looks and behaves like synchronous code. It enables you to construct code that appears to run in sequence but operates asynchronously.

This function simplifies the reading and writing of code that employs asynchronous operations. Async/await is a JavaScript technology that allows you to create asynchronous code more synchronously. Similar to callbacks, chaining promises together can become quite clumsy and confusing. For this reason, async and await were developed.

Async functions are defined as follows:

```
//async/await

const newAsyncFunc = async () => {

};
```

The `await` keyword can then be used inside of this `async` function to instruct it to wait for something:

```
const result = asyncFunc();

const asyncFunc = async() => {
  const result = await asyncFunc();
  //doSomethingNow
}
```

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Blog](#)[Software Engineering](#)[Product News](#)[Podcast](#)[Greatest Hits](#)[write with us](#)

[Product](#)[Pricing](#)[Customers](#)[Resources](#)[Blog](#)[Podcast](#)

```
        console.log(num);
        if (num < 10) {
            await incrementDigits(num);
        } else {
            return 'done!';
        }
    };

    (async () => {
        const res = await incrementDigits(0);
        console.log(res);
    })();
```

The `async/await` syntax is used in this code fragment to increment a number until it reaches 10. The `async` keyword, which comes before the declaration of the `incrementDigits()` function, denotes that a promise will be returned by the function.

Then, to halt the `incrementDigits()` function's execution until the promise is fulfilled, the `await` keyword is used. When the promise is resolved, the code following the `await` keyword will be executed.

Error handling in asynchronous JavaScript

Errors can be coding errors made by the programmer, input errors, or other unforeseeable things.

Error handling in asynchronous JavaScript involves `try/catch` blocks, which are used in JavaScript to catch

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Blog](#)[Software Engineering](#)[Product News](#)[Pricing](#)[Customers](#)[Podcast](#)[Resources](#)[Blog](#)[Podcast](#)

[Greatest Hits](#)[write with us](#)

A try statement lets you test a block of code for errors.

A catch statement lets you handle that error. For example:

```
try{

    alert("This is an error function!")
}catch(e){

    alert(e)
}
```

Including a `.catch()` method at the conclusion of the asynchronous action will enable the management of any potential errors.

A catch block can have parameters that will give you error information. Generally, the catch block is used to log an error or display specific messages to the user. Now that the ``incrementDigits()`` asynchronous function has an error handler function, we can include it as follows:

```
const incrementDigits = async num => {
    try{
        num++;
        console.log(num);
        if (num < 10) {
            await incrementDigits(num);
        } else {
            return 'done!';
        }
    }
}
```

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#)

[Sign up](#)

[Product](#)

[Pricing](#)

[Customers](#)

[Resources](#)

[Blog](#)

[Podcast](#)

[Blog](#)

[Software Engineering](#)

[Product News](#)

[Podcast](#)

[Greatest Hits](#)

[write with us](#)

Conclusion

In this post, we explored asynchronous JavaScript, examined the internal workings of the JavaScript asynchronous functions, and learned how to build asynchronous JavaScript using promises and `async/await`. We have seen that callbacks are simple functions passed to other functions and are only executed when an event is completed.

When dealing with time-consuming processes like network queries or file operations, asynchronous JavaScript is especially helpful. By carrying out these activities in an asynchronous manner, the computer can run other pieces of code while it waits for the results, increasing resource efficiency and speeding up execution.

5 thoughts on “Asynchronous JavaScript for Beginners”

Asynchronous programming is a key concept in modern web development. It enables programmers to react quickly to user events without causing the user interface to halt/slow.

June 3, 2023 at 1:21 am

Solid article and nice examples. Asynchronous programming techniques enable the seamless loading of data in the background while the user interacts

with the interface, ensuring a responsive and engaging user experience. I have one suggestion. In the `async/await` example, we can drop usage of `async` and `await` on the `incrementDigits` function. Asynchronous programming is becoming more and more important due to the rise of single-page

applications and the need for smooth user interactions. Example:



// Example using `async/await`
[asynchronous JavaScript](#). For further learning, here are some resources on

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Login](#) [Sign up](#)

[Product](#) [Pricing](#) [Customers](#) [Resources](#) [Blog](#) [Podcast](#)

[Blog](#) [Software Engineering](#) [Product News](#) [Podcast](#) [Greatest Hits](#) [write with us](#)

 Search over 600 articles 

[Reply](#)

Vijay says:

June 3, 2023 at 9:59 am

Excellent Sir!

[Reply](#)

Daniel Agantem says:

June 6, 2023 at 3:03 am

Thanks for the feedback

[Reply](#)

Mike says:

June 4, 2023 at 9:18 pm

Nice one

[Reply](#)

Santosh Kumar says:

January 11, 2024 at 8:33 pm

awesome


[Reply](#)

No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Product](#) [Pricing](#) [Customers](#) [Resources](#) [Blog](#) [Podcast](#)

[Login](#) [Sign up](#)

[Blog](#) [Software Engineering](#) [Product News](#) [Podcast](#) [Greatest Hits](#) [write with us](#)

Search over 600 articles 

Name *

Email *

Written by:

Daniel Agantem

I'm a front-end developer with 2+ years of experience. Able to effectively self-manage during independent projects, as well as collaborate as part of a productive team. I am both driven and self-motivated, and I am constantly experimenting with new technologies and techniques. I am very passionate about Software Development and strive to better myself as a developer, and the development community as a whole.




No More Seat Costs: [Semaphore Plans Just Got Better!](#)

[Product](#) [Pricing](#) [Customers](#) [Resources](#) [Blog](#) [Podcast](#)

[Login](#) [Sign up](#)

[Blog](#) [Software Engineering](#) [Product News](#) [Podcast](#) [Greatest Hits](#) [write with us](#)





CI/CD Weekly Newsletter

Tutorials, interviews, and tips for you to become a well-rounded developer.

Email

Subscribe

No More Seat Costs: Semaphore Plans Just Got Better!

[Login](#)

Sign up

[Product](#)

[Pricing](#)

[Customers](#)

[Resources](#)

[Blog](#)

[Podcast](#)

[Software Engineering](#)

[Product News](#)

[Podcast](#)

[Greatest Hits](#)

[write with us](#)

Search over 600 articles

[2023 Year in Review](#)

[Press Kit](#)

[Get in touch](#)

[Terms of Service](#)

[Privacy Policy](#)

[Security](#)

[Observability](#)

[Deployments & Automation](#)

[Documentation](#)

[Product News](#)

[System Status](#)

[Customers](#)

[Premium Support](#)

[Sign up](#)

[Android](#)

[Top Features](#)

[Test reports](#)

[Monorepos](#)

[Self-hosted agents](#)

[On-Premise](#)

[vs Bitbucket](#)

[Pipelines](#)

[Uncut Podcast](#)

[Blog](#)

[Write With Us](#)

[Connect with us on](#)

[Discord](#)