

Asynchronous Programming

[Mohamed Atef Elsafty]

Asynchronous programming

In a synchronous programming model, things happen one at a time. When you call a function that performs a long-running action, it returns only when the action has finished and it can return the result. This stops your program for the time the action takes.

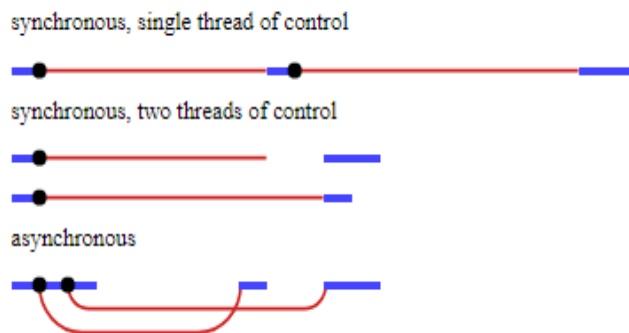
An *asynchronous* model allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result (for example, the data read from disk).

We can compare synchronous and asynchronous programming using a small example: a program that fetches two resources from the network and then combines results.

In a synchronous environment, where the request function returns only after it has done its work, the easiest way to perform this task is to make the requests one after the other. This has the drawback that the second request will be started only when the first has finished. The total time taken will be at least the sum of the two response times.

The solution to this problem, in a synchronous system, is to start additional threads of control. A *thread* is another running program whose execution may be interleaved with other programs by the operating system—since most modern computers contain multiple processors, multiple threads may even run at the same time, on different processors. A second thread could start the second request, and then both threads wait for their results to come back, after which they resynchronize to combine their results.

In the following diagram, the thick lines represent time the program spends running normally, and the thin lines represent time spent waiting for the network. In the synchronous model, the time taken by the network is *part* of the timeline for a given thread of control. In the asynchronous model, starting a network action conceptually causes a *split* in the timeline. The program that initiated the action continues running, and the action happens alongside it, notifying the program when it is finished.



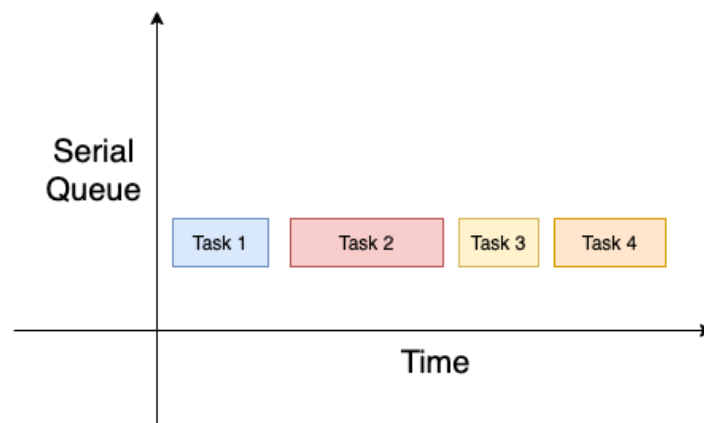
Both of the important JavaScript programming platforms—browsers and Node.js—make operations that might take a while asynchronous, rather than relying on threads. Since programming with threads is notoriously hard (understanding what a program does is much more difficult when it's doing multiple things at once), this is generally considered a good thing.

References:

- [1] [mdn web docs](#)
- [2] [stackify.com](#)
- [3] [eloquentjavascript.net](#)

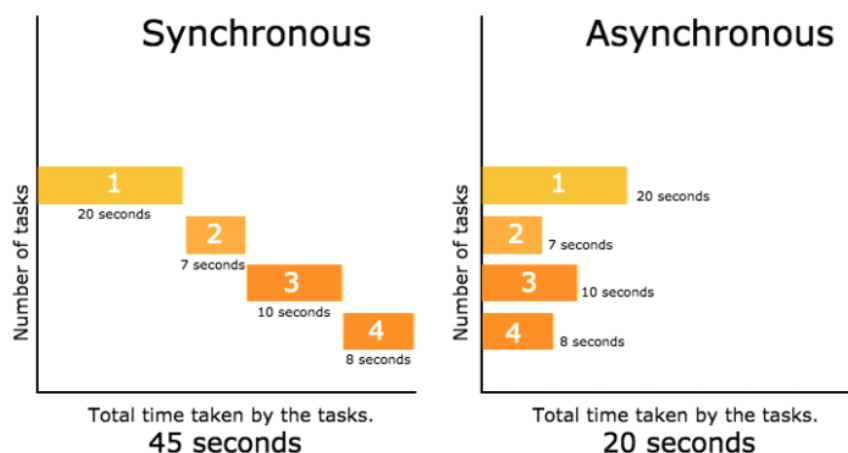
[Youssef Ashraf Sabry]

Synchronous Programming is a technique where all tasks are executed in the order in which they are implemented. Each task has to wait for the previous task to finish execution before it starts execution. However, this can prove particularly very slow especially when independent tasks have to wait for each other to finish before they can execute.



The tasks are executed sequentially in Synchronous Programming.

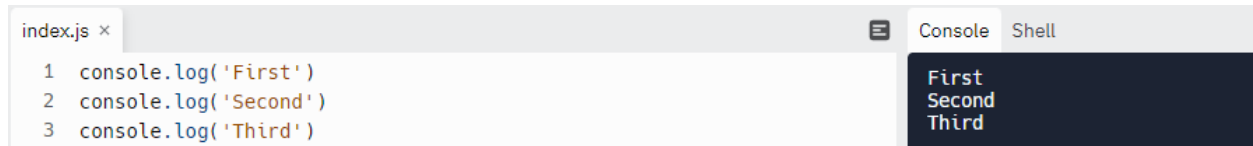
Asynchronous Programming is another technique that allows multiple tasks to run in parallel without having to wait for each other to finish execution. This is particularly useful in decreasing program execution time.



Difference in execution times between Synchronous and Asynchronous.

Synchronous Programming in JavaScript

Most of the code we write is probably synchronous. For example, in calculating operations that depend on each others' values to calculate the final value—or maybe printing a few statements in order.

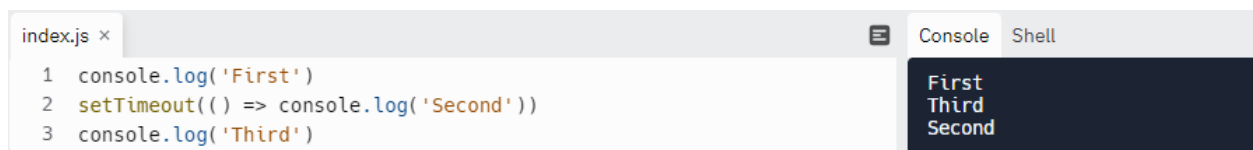


The screenshot shows a code editor with a file named 'index.js'. The code contains three lines: `1 console.log('First')`, `2 console.log('Second')`, and `3 console.log('Third')`. To the right of the code editor is a 'Console' tab showing the output: 'First', 'Second', and 'Third' printed in order.

All statements are printed in the order they were implemented

Asynchronous Programming in JavaScript

In JavaScript, we have multiple ways to implement asynchronous programming. Thankfully, the team behind JavaScript has improved the ways to implement them over time to reduce the complexities of using Asynchronous Programming. Here's an example.



The screenshot shows a code editor with a file named 'index.js'. The code contains three lines: `1 console.log('First')`, `2 setTimeout(() => console.log('Second'))`, and `3 console.log('Third')`. To the right of the code editor is a 'Console' tab showing the output: 'First', 'Third', and 'Second' printed in order.

Although the 'Second' statement came before the third, it was executed before the third statement

Async/Await in JavaScript

Async/Await is one of the most common ways to implement asynchronous programming. Behind the scenes, it works just like Promises in work but is more readable and easier to use, hence why developers call it ‘Syntactic Sugar.’

First, we start by defining an async function by using the **async** keyword before the function declaration. This is essential when we call an asynchronous function, for example, a function that makes a Database query so we can **await** the result.

```
1 ▼ async function printStatements() {  
2  
3 }  
4  
5 printStatements();  
6
```

Second, sometimes when we make external requests to a Database, for example, the request fails for many reasons. So we must surround our main code with a try/catch block to detect any failure.

```
async function printStatements() {  
  try {  
  
  } catch (err) {  
  
  }  
}
```

Third, we write a request to an external database. We are going to use **await** before the request. This is the reason we included **async** keyword before function declaration because we can’t use **await** without **async**.



The screenshot shows a code editor with a file named 'index.js'. The code defines an async function 'printStatements()' which uses 'try/catch' to handle errors. Inside the 'try' block, it logs 'First', calls a pseudo function 'getDatabaseQuery()' (commented as 'a pseudo function that imitates a returned query from DB'), and logs the result. It then logs 'Third'. The 'catch' block returns a message 'Something went wrong! Try again;'. The function is then called 'printStatements()'. To the right, the 'Console' tab shows the output: 'First', 'Third', and 'Second'.

```
1 ▼ async function printStatements() {  
2 ▼   try {  
3     console.log('First')  
4     // getDatabaseQuery is a pseudo function that imitates a returned query from DB  
5     const result = await getDatabaseQuery();  
6     console.log(result);  
7  
8     console.log('Third')  
9 ▼   } catch (err) {  
10    return 'Something went wrong! Try again;'  
11  }  
12 }  
13  
14 printStatements();  
15
```

Console
First
Third
Second

References

- [\[1\] Medium User Article: By Viviam Jim](#)
- [\[2\] Medium User Article: By Srinivas Prayag](#)
- [\[3\] MDN Web Docs](#)