

# Synchronous vs. asynchronous

This article was originally published at

<https://gist.github.com/joepie91/bf3d04febb024da89e3a3e61b164247d>.

You'll run into the terms "synchronous" and "asynchronous" a lot when working with JS. Let's look at what they actually *mean*.

Synchronous code is like what you might be used to already from other languages. You call a function, it does some work, and then returns the result. No other code runs in the meantime. This is simple to understand, but it's also inefficient; what if "doing some work" mostly involves getting some data from a database? In the meantime, our process is sitting around doing nothing, waiting for the database to respond. It could be doing useful work in that time!

And that's what brings us to asynchronous code. Asynchronous code works differently; you still call a function, but it *doesn't* return a result. Instead, you don't just pass the regular arguments to the function, but also give it a piece of code in a function (a so-called "asynchronous callback") to execute when the operation completes. The JS runtime stores this callback alongside the in-progress operation, to retrieve and execute it later when the external service (eg. the database) reports that the operation has been completed.

Crucially, this means that when you call an asynchronous function, it *cannot* wait until the external processing is complete before returning from the function! After all, the intention is to keep running other code in the meantime, so it needs to return from the function so that the 'caller' (the code which originally called the function) can continue doing useful things even while the external operation is in progress.

All of this takes place in what's called the "event loop" - you can pretty much think of it as a huge infinite loop that contains your entire program. Every time you trigger an external process through an asynchronous function call, that external process will eventually finish, and put its result in a 'queue' alongside the callback you specified. On each iteration ("tick") of the event loop, it then goes through that queue, executes all of the callbacks, which can then indirectly cause new items to be put into the queue, and so on. The end result is a program that calls asynchronous callbacks as and when necessary, and that keeps giving new work to the event loop through a chain of those callbacks.

This is, of course, a very simplified explanation - just enough to understand the rest of this page. I strongly recommend reading up on the event loop more, as it will make it much easier to

understand JS in general. Here are some good resources that go into more depth:

1. <https://nodesource.com/blog/understanding-the-nodejs-event-loop> (article)
2. <https://www.youtube.com/watch?v=8aGhZQkoFbQ> (video)
3. <https://www.youtube.com/watch?v=cCOL7MC4PI0> (video)

Now that we understand the what the event loop is, and what a "tick" is, we can define more precisely what "asynchronous" means in JS:

**Asynchronous code is code that happens across more than one event loop tick. An asynchronous function is a function that needs more than one event loop tick to complete.**

This definition will be important later on, for understanding why asynchronous code can be more difficult to write correctly than synchronous code.

## Asynchronous execution order and boundaries

This idea of "queueing code to run at some later tick" has consequences for how you write your code.

Remember how the event loop is a loop, and ticks are iterations - this means that event loop ticks are *distributed across time linearly*. First the first tick happens, then the second tick, then the third tick, and so on. Something that runs in the first tick can *never* execute before something that runs in the third tick; unless you're a time traveller anyway, in which case you probably would have more important things to do than reading this guide ☹️

Anyhow, this means that code will run in a slightly counterintuitive way, if you're used to synchronous code. For example, consider the following code, which uses the asynchronous `setTimeout` function to run something after a specified amount of milliseconds:

```
console.log("one");

setTimeout(() => {
  console.log("two");
}, 300);

console.log("three");
```

You might expect this to print out `one, two, three` - but if you try running this code, you'll see that it doesn't! Instead, you get this:

```
one
three
two
```

What's going on here?!

The answer to that is what I mentioned earlier; the asynchronous callback is *getting queued for later*. Let's pretend for the sake of explanation that an event loop tick only happens when there's actually something to do. The **first tick** would then run this code:

```
console.log("one");

setTimeout(..., 300); // This schedules some code to run in a next tick, about 300ms later

console.log("three");
```

Then 300 milliseconds elapse, with nothing for the event loop to do - and after those 300ms, the callback we gave to `setTimeout` suddenly appears in the event loop queue. Now the **second tick** happens, and it executes this code:

```
console.log("two");
```

... thus resulting in the output that we saw above.

The key insight here is that **code with callbacks does not execute in the order that the code is written**. Only the code *outside* of the callbacks executes in the written order. For example, we can be certain that `three` will get printed after `one` because both are outside of the callback and so they are executed in that order, but because `two` is printed from *inside* of a callback, we can't know when it will execute.

"But hold on", you say, "then how can you know that `two` will be printed after `three` and `one`?"

This is where the earlier definition of "asynchronous code" comes into play! Let's reason through it:

1. `setTimeout` is asynchronous.
2. Therefore, we call `console.log("two")` from within an asynchronous callback.
3. Synchronous code executes within one tick.
4. Asynchronous code needs more than one tick to execute, ie. the asynchronous callback will be called in a *later* tick than the one where we started the operation (eg. `setTimeout`).
5. Therefore, an asynchronous callback will *always* execute after the synchronous code that started the operation, no matter what.
6. Therefore, `two` will *always* be printed after `one` and `three`.

So, we can know when the asynchronous callback will be executed, in terms of relative time. That's useful, isn't it? Doesn't that mean that we can do that for *all* asynchronous code? Well,

unfortunately not - it gets more complicated when there is *more* than one asynchronous operation.

Take, for example, the following code:

```
console.log("one");

someAsynchronousOperation(() => {
  console.log("two");
});

someOtherAsynchronousOperation(() => {
  console.log("three");
});

console.log("four");
```

We have two different asynchronous operations here, and we don't know for certain which of the two will finish faster. We don't even know whether it's always the *same* one that finishes faster, or whether it varies between runs of the program. So while we can determine that `two` and `three` will *always* be printed after `one` and `four` - remember, asynchronous callbacks in synchronous code - we *can't* know whether `two` or `three` will come first.

And this is, fundamentally, what makes asynchronous code more difficult to write; you never know for sure in what order your code will complete. Every real-world program will have at least *some* scenarios where you can't force an order of operations (or, at least, not without horribly bad performance), so this is a problem that you *have* to account for in your code.

The easiest solution to this, is to avoid "shared state". Shared state is information that you store (eg. in a variable) and that gets used by multiple parts of your code independently. This can sometimes be necessary, but it also comes at a cost - if function A and function B both modify the same variable, then if they run in a different order than you expected, one of them might mess up the expected state of the other. This is generally already true in programming, but even more important when working with asynchronous code, as your chunks of code get 'interspersed' much more due to the callback model.

[...]

---

Revision #1

Created 11 December 2024 18:42:39 by joepie91

Updated 11 December 2024 18:44:19 by joepie91