You Don't Know JS: Async & Performance

Chapter 2: Callbacks

In Chapter 1, we explored the terminology and concepts around asynchronous programming in JavaScript. Our focus is on understanding the single-threaded (one-at-a-time) event loop queue that drives all "events" (async function invocations). We also explored various ways that concurrency patterns explain the relationships (if any!) between *simultaneously* running chains of events, or "processes" (tasks, function calls, etc.).

All our examples in Chapter 1 used the function as the individual, indivisible unit of operations, whereby inside the function, statements run in predictable order (above the compiler level!), but at the function-ordering level, events (aka async function invocations) can happen in a variety of orders.

In all these cases, the function is acting as a "callback," because it serves as the target for the event loop to "call back into" the program, whenever that item in the queue is processed.

As you no doubt have observed, callbacks are by far the most common way that asynchrony in JS programs is expressed and managed. Indeed, the callback is the most fundamental async pattern in the language.

Countless JS programs, even very sophisticated and complex ones, have been written upon no other async foundation than the callback (with of course the concurrency interaction patterns we explored in Chapter 1). The callback function is the async work horse for JavaScript, and it does its job respectably.

Except... callbacks are not without their shortcomings. Many developers are excited by the *promise* (pun intended!) of better async patterns. But it's impossible to effectively use any abstraction if you don't understand what it's abstracting, and why.

In this chapter, we will explore a couple of those in depth, as motivation for why more sophisticated async patterns (explored in subsequent chapters of this book) are necessary and desired.

Continuations

Let's go back to the async callback example we started with in Chapter 1, but let me slightly modify it to illustrate a point:

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

`// A` and `// B` represent the first half of the program (aka the *now*), and `// C` marks the second half of the program (aka the *later*). The first half executes right away, and then there's a "pause" of indeterminate length. At some future moment, if the Ajax call completes, then the program will pick up where it left off, and *continue* with the second half.

In other words, the callback function wraps or encapsulates the *continuation* of the program.

Let's make the code even simpler:

```
// A
setTimeout( function(){
    // C
}, 1000 );
// B
```

Stop for a moment and ask yourself how you'd describe (to someone else less informed about how JS works) the way that program behaves. Go ahead, try it out loud. It's a good exercise that will help my next points make more sense.

Most readers just now probably thought or said something to the effect of: "Do A, then set up a timeout to wait 1,000 milliseconds, then once that fires, do C." How close was your rendition?

You might have caught yourself and self-edited to: "Do A, setup the timeout for 1,000 milliseconds, then do B, then after the timeout fires, do C." That's more accurate than the first version. Can you spot the difference?

Even though the second version is more accurate, both versions are deficient in explaining this code in a way that matches our brains to the code, and the code to the JS engine. The disconnect is both subtle and monumental, and is at the very heart of understanding the shortcomings of callbacks as async expression and management.

As soon as we introduce a single continuation (or several dozen as many programs do!) in the form of a callback function, we have allowed a divergence to form between how our brains work and the way the code will operate. Any time these two diverge (and this is by far not the only place that happens, as I'm sure you know!), we run into the inevitable fact that our code becomes harder to understand, reason about, debug, and maintain.

Sequential Brain

I'm pretty sure most of you readers have heard someone say (even made the claim yourself), "I'm a multitasker." The effects of trying to act as a multitasker range from humorous (e.g., the silly patting-head-rubbing-stomach kids' game) to mundane (chewing gum while walking) to downright dangerous (texting while driving).

But are we multitaskers? Can we really do two conscious, intentional actions at once and think/reason about both of them at exactly the same moment? Does our highest level of brain functionality have parallel multithreading going on?

The answer may surprise you: **probably not.**

That's just not really how our brains appear to be set up. We're much more single taskers than many of us (especially A-type personalities!) would like to admit. We can really only think about one thing at any given instant.

I'm not talking about all our involuntary, subconscious, automatic brain functions, such as heart beating, breathing, and eyelid blinking. Those are all vital tasks to our sustained life, but we don't intentionally allocate any brain power to them. Thankfully, while we obsess about checking social network feeds for the 15th time in three minutes, our brain carries on in the background (threads!) with all those important tasks.

We're instead talking about whatever task is at the forefront of our minds at the moment. For me, it's writing the text in this book right now. Am I doing any other higher level brain function at exactly this same moment? Nope, not really. I get distracted quickly and easily -- a few dozen times in these last couple of paragraphs!

When we *fake* multitasking, such as trying to type something at the same time we're talking to a friend or family member on the phone, what we're actually most likely doing is acting as fast context switchers. In other words, we switch back and forth between two or more tasks in rapid succession, *simultaneously* progressing on each task in tiny, fast little chunks. We do it so fast that to the outside world it appears as if we're doing these things *in parallel*.

Does that sound suspiciously like async evented concurrency (like the sort that happens in JS) to you?! If not, go back and read Chapter 1 again!

In fact, one way of simplifying (i.e., abusing) the massively complex world of neurology into something I can remotely hope to discuss here is that our brains work kinda like the event loop queue.

If you think about every single letter (or word) I type as a single async event, in just this sentence alone there are several dozen opportunities for my brain to be interrupted by some other event, such as from my senses, or even just my random thoughts.

I don't get interrupted and pulled to another "process" at every opportunity that I could be (thankfully -- or this book would never be written!). But it happens often enough that I feel my own brain is nearly constantly switching to various different contexts (aka "processes"). And that's an awful lot like how the JS engine would probably feel.

Doing Versus Planning

OK, so our brains can be thought of as operating in single-threaded event loop queue like ways, as can the JS engine. That sounds like a good match.

But we need to be more nuanced than that in our analysis. There's a big, observable difference between how we plan various tasks, and how our brains actually operate those tasks.

Again, back to the writing of this text as my metaphor. My rough mental outline plan here is to keep writing and writing, going sequentially through a set of points I have ordered in my thoughts. I don't plan to have any interruptions or nonlinear activity in this writing. But yet, my brain is nevertheless switching around all the time.

Even though at an operational level our brains are async evented, we seem to plan out tasks in a sequential, synchronous way. "I need to go to the store, then buy some milk, then drop off my dry cleaning."

You'll notice that this higher level thinking (planning) doesn't seem very async evented in its formulation. In fact, it's kind of rare for us to deliberately think solely in terms of events. Instead, we plan things out carefully, sequentially (A then B then C), and we assume to an extent a sort of temporal blocking that forces B to wait on A, and C to wait on B.

When a developer writes code, they are planning out a set of actions to occur. If they're any good at being a developer, they're **carefully planning** it out. "I need to set `z` to the value of `x`, and then `x` to the value of `y`," and so forth.

When we write out synchronous code, statement by statement, it works a lot like our errands to-do list:

```
// swap `x` and `y` (via temp variable `z`)
z = x;
x = y;
y = z;
```

These three assignment statements are synchronous, so `x = y` waits for `z = x` to finish, and `y = z` in turn waits for `x = y` to finish. Another way of saying it is that these three statements are temporally bound to execute in a certain order, one right after the other. Thankfully, we don't need to be bothered with any async evented details here. If we did, the code gets a lot more complex, quickly!

So if synchronous brain planning maps well to synchronous code statements, how well do our brains do at

planning out asynchronous code?

It turns out that how we express asynchrony (with callbacks) in our code doesn't map very well at all to that synchronous brain planning behavior.

Can you actually imagine having a line of thinking that plans out your to-do errands like this?

> "I need to go to the store, but on the way I'm sure I'll get a phone call, so 'Hi, Mom', and while she starts talking, I'll be looking up the store address on GPS, but that'll take a second to load, so I'll turn down the radio so I can hear Mom better, then I'll realize I forgot to put on a jacket and it's cold outside, but no matter, keep driving and talking to Mom, and then the seatbelt ding reminds me to buckle up, so 'Yes, Mom, I am wearing my seatbelt, I always do!'. Ah, finally the GPS got the directions, now..."

As ridiculous as that sounds as a formulation for how we plan our day out and think about what to do and in what order, nonetheless it's exactly how our brains operate at a functional level. Remember, that's not multitasking, it's just fast context switching.

The reason it's difficult for us as developers to write async evented code, especially when all we have is the callback to do it, is that stream of consciousness thinking/planning is unnatural for most of us.

We think in step-by-step terms, but the tools (callbacks) available to us in code are not expressed in a step-by-step fashion once we move from synchronous to asynchronous.

And **that** is why it's so hard to accurately author and reason about async JS code with callbacks: because it's not how our brain planning works.

Note: The only thing worse than not knowing why some code breaks is not knowing why it worked in the first place! It's the classic "house of cards" mentality: "it works, but not sure why, so nobody touch it!" You may have heard, "Hell is other people" (Sartre), and the programmer meme twist, "Hell is other people's code." I believe truly: "Hell is not understanding my own code." And callbacks are one main culprit.

Nested/Chained Callbacks

Consider:

```
listen( "click", function handler(evt){
    setTimeout( function request(){
        ajax( "http://some.url.1", function response(text){
            if (text == "hello") {
                handler();
            }
            else if (text == "world") {
                request();
            }
        } );
    }, 500) ;
} );
```

There's a good chance code like that is recognizable to you. We've got a chain of three functions nested together, each one representing a step in an asynchronous series (task, "process").

This kind of code is often called "callback hell," and sometimes also referred to as the "pyramid of doom" (for its sideways-facing triangular shape due to the nested indentation).

But "callback hell" actually has almost nothing to do with the nesting/indentation. It's a far deeper problem than that. We'll see how and why as we continue through the rest of this chapter.

First, we're waiting for the "click" event, then we're waiting for the timer to fire, then we're waiting for the Ajax response to come back, at which point it might do it all again.

At first glance, this code may seem to map its asynchrony naturally to sequential brain planning.

First (*now*), we:

```
listen( "..", function handler(..){
    // ..
} );
```

Then *later*, we:

```
setTimeout( function request(..){
    // ..
}, 500) ;
```

Then still *later*, we:

```
ajax( "..", function response(..){
    // ..
} );
```

And finally (most *later*), we:

```
if ( .. ) {
    // ..
}
else ..
```

But there's several problems with reasoning about this code linearly in such a fashion.

First, it's an accident of the example that our steps are on subsequent lines (1, 2, 3, and 4...). In real async JS programs, there's often a lot more noise cluttering things up, noise that we have to deftly maneuver past in our brains as we jump from one function to the next. Understanding the async flow in such callback-laden code is not impossible, but it's certainly not natural or easy, even with lots of practice.

But also, there's something deeper wrong, which isn't evident just in that code example. Let me make up another scenario (pseudocode-ish) to illustrate it:

```
doA( function(){
    doB();

    doC( function(){
        doD();
    } )

    doE();
} );

doF();
```

While the experienced among you will correctly identify the true order of operations here, I'm betting it is more than a little confusing at first glance, and takes some concerted mental cycles to arrive at. The operations will happen in this order:

- `doA()`
- `doF()`
- `doB()`
- `doC()`
- `doE()`

- `doD()`

Did you get that right the very first time you glanced at the code?

OK, some of you are thinking I was unfair in my function naming, to intentionally lead you astray. I swear I was just naming in top-down appearance order. But let me try again:

```
doA( function(){
    doC();

    doD( function(){
        doF();
    } )

    doE();
} );

doB();
```

Now, I've named them alphabetically in order of actual execution. But I still bet, even with experience now in this scenario, tracing through the `A -> B -> C -> D -> E -> F` order doesn't come natural to many if any of you readers. Certainly your eyes do an awful lot of jumping up and down the code snippet, right?

But even if that all comes natural to you, there's still one more hazard that could wreak havoc. Can you spot what it is?

What if `doA(..)` or `doD(..)` aren't actually async, the way we obviously assumed them to be? Uh oh, now the order is different. If they're both sync (and maybe only sometimes, depending on the conditions of the program at the time), the order is now `A -> C -> D -> F -> E -> B`.

That sound you just heard faintly in the background is the sighs of thousands of JS developers who just had a face-in-hands moment.

Is nesting the problem? Is that what makes it so hard to trace the async flow? That's part of it, certainly.

But let me rewrite the previous nested event/timeout/Ajax example without using nesting:

```
listen( "click", handler );

function handler() {
    setTimeout( request, 500 );
}

function request(){
    ajax( "http://some.url.1", response );
}

function response(text){
    if (text == "hello") {
        handler();
    }
    else if (text == "world") {
        request();
    }
}
```

This formulation of the code is not hardly as recognizable as having the nesting/indentation woes of its previous form, and yet it's every bit as susceptible to "callback hell." Why?

As we go to linearly (sequentially) reason about this code, we have to skip from one function, to the next, to the next, and bounce all around the code base to "see" the sequence flow. And remember, this is simplified code in sort of best-case fashion. We all know that real async JS program code bases are often fantastically more jumbled, which makes such reasoning orders of magnitude more difficult.

Another thing to notice: to get steps 2, 3, and 4 linked together so they happen in succession, the only affordance callbacks alone gives us is to hardcode step 2 into step 1, step 3 into step 2, step 4 into step 3, and so on. The hardcoding isn't necessarily a bad thing, if it really is a fixed condition that step 2 should always lead to step 3.

But the hardcoding definitely makes the code a bit more brittle, as it doesn't account for anything going wrong that might cause a deviation in the progression of steps. For example, if step 2 fails, step 3 never gets reached, nor does step 2 retry, or move to an alternate error handling flow, and so on.

All of these issues are things you *can* manually hardcode into each step, but that code is often very repetitive and not reusable in other steps or in other async flows in your program.

Even though our brains might plan out a series of tasks in a sequential type of way (this, then this, then this), the evented nature of our brain operation makes recovery/retry/forking of flow control almost effortless. If you're out running errands, and you realize you left a shopping list at home, it doesn't end the day because you didn't plan that ahead of time. Your brain routes around this hiccup easily: you go home, get the list, then head right

back out to the store.

But the brittle nature of manually hardcoded callbacks (even with hardcoded error handling) is often far less graceful. Once you end up specifying (aka pre-planning) all the various eventualities/paths, the code becomes so convoluted that it's hard to ever maintain or update it.

That is what "callback hell" is all about! The nesting/indentation are basically a side show, a red herring.

And as if all that's not enough, we haven't even touched what happens when two or more chains of these callback continuations are happening *simultaneously*, or when the third step branches out into "parallel" callbacks with gates or latches, or... OMG, my brain hurts, how about yours!?

Are you catching the notion here that our sequential, blocking brain planning behaviors just don't map well onto callback-oriented async code? That's the first major deficiency to articulate about callbacks: they express asynchrony in code in ways our brains have to fight just to keep in sync with (pun intended!).

Trust Issues

The mismatch between sequential brain planning and callback-driven async JS code is only part of the problem with callbacks. There's something much deeper to be concerned about.

Let's once again revisit the notion of a callback function as the continuation (aka the second half) of our program:

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

`// A` and `// B` happen *now*, under the direct control of the main JS program. But `// C` gets deferred to happen *later*, and under the control of another party -- in this case, the `ajax(..)` function. In a basic sense, that sort of hand-off of control doesn't regularly cause lots of problems for programs.

But don't be fooled by its infrequency that this control switch isn't a big deal. In fact, it's one of the worst (and yet most subtle) problems about callback-driven design. It revolves around the idea that sometimes `ajax(..)` (i.e., the "party" you hand your callback continuation to) is not a function that you wrote, or that you directly control. Many times it's a utility provided by some third party.

We call this "inversion of control," when you take part of your program and give over control of its execution to another third party. There's an unspoken "contract" that exists between your code and the third-party utility -- a

set of things you expect to be maintained.

Tale of Five Callbacks

It might not be terribly obvious why this is such a big deal. Let me construct an exaggerated scenario to illustrate the hazards of trust at play.

Imagine you're a developer tasked with building out an ecommerce checkout system for a site that sells expensive TVs. You already have all the various pages of the checkout system built out just fine. On the last page, when the user clicks "confirm" to buy the TV, you need to call a third-party function (provided say by some analytics tracking company) so that the sale can be tracked.

You notice that they've provided what looks like an async tracking utility, probably for the sake of performance best practices, which means you need to pass in a callback function. In this continuation that you pass in, you will have the final code that charges the customer's credit card and displays the thank you page.

This code might look like:

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
    displayThankyouPage();
} );
```

Easy enough, right? You write the code, test it, everything works, and you deploy to production. Everyone's happy!

Six months go by and no issues. You've almost forgotten you even wrote that code. One morning, you're at a coffee shop before work, casually enjoying your latte, when you get a panicked call from your boss insisting you drop the coffee and rush into work right away.

When you arrive, you find out that a high-profile customer has had his credit card charged five times for the same TV, and he's understandably upset. Customer service has already issued an apology and processed a refund. But your boss demands to know how this could possibly have happened. "Don't we have tests for stuff like this!?"

You don't even remember the code you wrote. But you dig back in and start trying to find out what could have gone awry.

After digging through some logs, you come to the conclusion that the only explanation is that the analytics utility somehow, for some reason, called your callback five times instead of once. Nothing in their documentation mentions anything about this.

Frustrated, you contact customer support, who of course is as astonished as you are. They agree to escalate it to their developers, and promise to get back to you. The next day, you receive a lengthy email explaining what they found, which you promptly forward to your boss.

Apparently, the developers at the analytics company had been working on some experimental code that, under certain conditions, would retry the provided callback once per second, for five seconds, before failing with a timeout. They had never intended to push that into production, but somehow they did, and they're totally embarrassed and apologetic. They go into plenty of detail about how they've identified the breakdown and what they'll do to ensure it never happens again. Yadda, yadda.

What's next?

You talk it over with your boss, but he's not feeling particularly comfortable with the state of things. He insists, and you reluctantly agree, that you can't trust *them* anymore (that's what bit you), and that you'll need to figure out how to protect the checkout code from such a vulnerability again.

After some tinkering, you implement some simple ad hoc code like the following, which the team seems happy with:

```
var tracked = false;

analytics.trackPurchase( purchaseData, function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
} );
```

Note: This should look familiar to you from Chapter 1, because we're essentially creating a latch to handle if there happen to be multiple concurrent invocations of our callback.

But then one of your QA engineers asks, "what happens if they never call the callback?" Oops. Neither of you had thought about that.

You begin to chase down the rabbit hole, and think of all the possible things that could go wrong with them calling your callback. Here's roughly the list you come up with of ways the analytics utility could misbehave:

- Call the callback too early (before it's been tracked)
- Call the callback too late (or never)
- Call the callback too few or too many times (like the problem you encountered!)
- Fail to pass along any necessary environment/parameters to your callback

- Swallow any errors/exceptions that may happen
- ...

That should feel like a troubling list, because it is. You're probably slowly starting to realize that you're going to have to invent an awful lot of ad hoc logic **in each and every single callback** that's passed to a utility you're not positive you can trust.

Now you realize a bit more completely just how hellish "callback hell" is.

Not Just Others' Code

Some of you may be skeptical at this point whether this is as big a deal as I'm making it out to be. Perhaps you don't interact with truly third-party utilities much if at all. Perhaps you use versioned APIs or self-host such libraries, so that its behavior can't be changed out from underneath you.

So, contemplate this: can you even *really* trust utilities that you do theoretically control (in your own code base)?

Think of it this way: most of us agree that at least to some extent we should build our own internal functions with some defensive checks on the input parameters, to reduce/prevent unexpected issues.

Overly trusting of input: ```js function addNumbers(x,y) { // + is overloaded with coercion to also be // string concatenation, so this operation // isn't strictly safe depending on what's // passed in. return x + y; }

addNumbers(21, 21); // 42 addNumbers(21, "21"); // "2121" ```

Defensive against untrusted input: ```js function addNumbers(x,y) { // ensure numerical input if (typeof x != "number" || typeof y != "number") { throw Error("Bad parameters"); }

```
// if we get here, + will safely do numeric addition
return x + y;
```

}

addNumbers(21, 21); // 42 addNumbers(21, "21"); // Error: "Bad parameters" ```

Or perhaps still safe but friendlier: ```js function addNumbers(x,y) { // ensure numerical input x = Number(x); y = Number(y);

```
// + will safely do numeric addition
return x + y;
```

}

addNumbers(21, 21); // 42 addNumbers(21, "21"); // 42 ```

However you go about it, these sorts of checks/normalizations are fairly common on function inputs, even with code we theoretically entirely trust. In a crude sort of way, it's like the programming equivalent of the geopolitical principle of "Trust But Verify."

So, doesn't it stand to reason that we should do the same thing about composition of async function callbacks, not just with truly external code but even with code we know is generally "under our own control"? **Of course we should.**

But callbacks don't really offer anything to assist us. We have to construct all that machinery ourselves, and it often ends up being a lot of boilerplate/overhead that we repeat for every single async callback.

The most troublesome problem with callbacks is *inversion of control* leading to a complete breakdown along all those trust lines.

If you have code that uses callbacks, especially but not exclusively with third-party utilities, and you're not already applying some sort of mitigation logic for all these *inversion of control* trust issues, your code *has* bugs in it right now even though they may not have bitten you yet. Latent bugs are still bugs.

Hell indeed.

Trying to Save Callbacks

There are several variations of callback design that have attempted to address some (not all!) of the trust issues we've just looked at. It's a valiant, but doomed, effort to save the callback pattern from imploding on itself.

For example, regarding more graceful error handling, some API designs provide for split callbacks (one for the success notification, one for the error notification):

```
function success(data) {
    console.log( data );
}

function failure(err) {
    console.error( err );
}

ajax( "http://some.url.1", success, failure );
```

In APIs of this design, often the `failure()` error handler is optional, and if not provided it will be assumed you want the errors swallowed. Ugh.

Note: This split-callback design is what the ES6 Promise API uses. We'll cover ES6 Promises in much more detail in the next chapter.

Another common callback pattern is called "error-first style" (sometimes called "Node style," as it's also the convention used across nearly all Node.js APIs), where the first argument of a single callback is reserved for an error object (if any). If success, this argument will be empty/falsy (and any subsequent arguments will be the success data), but if an error result is being signaled, the first argument is set/truthy (and usually nothing else is passed):

```
function response(err,data) {
    // error?
    if (err) {
        console.error( err );
    }
    // otherwise, assume success
    else {
        console.log( data );
    }
}

ajax( "http://some.url.1", response );
```

In both of these cases, several things should be observed.

First, it has not really resolved the majority of trust issues like it may appear. There's nothing about either callback that prevents or filters unwanted repeated invocations. Moreover, things are worse now, because you may get both success and error signals, or neither, and you still have to code around either of those conditions.

Also, don't miss the fact that while it's a standard pattern you can employ, it's definitely more verbose and boilerplate-ish without much reuse, so you're going to get weary of typing all that out for every single callback in your application.

What about the trust issue of never being called? If this is a concern (and it probably should be!), you likely will need to set up a timeout that cancels the event. You could make a utility (proof-of-concept only shown) to help you with that:

```
function timeoutify(fn,delay) {
    var intv = setTimeout( function(){
            intv = null;
            fn( new Error( "Timeout!" ) );
        }, delay )
    ;

    return function() {
        // timeout hasn't happened yet?
        if (intv) {
            clearTimeout( intv );
            fn.apply( this, [ null ].concat( [].slice.call( arguments ) ) );
        }
    };
}
```

Here's how you use it:

```
// using "error-first style" callback design
function foo(err,data) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( data );
    }
}

ajax( "http://some.url.1", timeoutify( foo, 500 ) );
```

Another trust issue is being called "too early." In application-specific terms, this may actually involve being called before some critical task is complete. But more generally, the problem is evident in utilities that can either invoke the callback you provide *now* (synchronously), or *later* (asynchronously).

This nondeterminism around the sync-or-async behavior is almost always going to lead to very difficult to track down bugs. In some circles, the fictional insanity-inducing monster named Zalgo is used to describe the sync/async nightmares. "Don't release Zalgo!" is a common cry, and it leads to very sound advice: always invoke callbacks asynchronously, even if that's "right away" on the next turn of the event loop, so that all callbacks are predictably async.

Note: For more information on Zalgo, see Oren Golan's "Don't Release Zalgo!" (https://github.com/oren/oren.github.io/blob/master/posts/zalgo.md) and Isaac Z. Schlueter's "Designing APIs for Asynchrony" (http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony).

Consider:

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", result );
a++;
```

Will this code print `0` (sync callback invocation) or `1` (async callback invocation)? Depends... on the conditions.

You can see just how quickly the unpredictability of Zalgo can threaten any JS program. So the silly-sounding "never release Zalgo" is actually incredibly common and solid advice. Always be asyncing.

What if you don't know whether the API in question will always execute async? You could invent a utility like this `asyncify(..)` proof-of-concept:

```
function asyncify(fn) {
    var orig_fn = fn,
        intv = setTimeout( function(){
            intv = null;
            if (fn) fn();
        }, 0 )
    ;

    fn = null;

    return function() {
        // firing too quickly, before `intv` timer has fired to
        // indicate async turn has passed?
        if (intv) {
            fn = orig_fn.bind.apply(
                orig_fn,
                // add the wrapper's `this` to the `bind(..)`
                // call parameters, as well as currying any
                // passed in parameters
                [this].concat( [].slice.call( arguments ) )
            );
        }
        // already async
        else {
            // invoke original function
            orig_fn.apply( this, arguments );
        }
    };
}
```

You use `asyncify(..)` like this:

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", asyncify( result ) );
a++;
```

Whether the Ajax request is in the cache and resolves to try to call the callback right away, or must be fetched over the wire and thus complete later asynchronously, this code will always output `1` instead of `0` -- `result(..)` cannot help but be invoked asynchronously, which means the `a++` has a chance to run

before `result(..)` does.

Yay, another trust issued "solved"! But it's inefficient, and yet again more bloated boilerplate to weigh your project down.

That's just the story, over and over again, with callbacks. They can do pretty much anything you want, but you have to be willing to work hard to get it, and oftentimes this effort is much more than you can or should spend on such code reasoning.

You might find yourself wishing for built-in APIs or other language mechanics to address these issues. Finally ES6 has arrived on the scene with some great answers, so keep reading!

Review

Callbacks are the fundamental unit of asynchrony in JS. But they're not enough for the evolving landscape of async programming as JS matures.

First, our brains plan things out in sequential, blocking, single-threaded semantic ways, but callbacks express asynchronous flow in a rather nonlinear, nonsequential way, which makes reasoning properly about such code much harder. Bad to reason about code is bad code that leads to bad bugs.

We need a way to express asynchrony in a more synchronous, sequential, blocking manner, just like our brains do.

Second, and more importantly, callbacks suffer from *inversion of control* in that they implicitly give control over to another party (often a third-party utility not in your control!) to invoke the *continuation* of your program. This control transfer leads us to a troubling list of trust issues, such as whether the callback is called more times than we expect.

Inventing ad hoc logic to solve these trust issues is possible, but it's more difficult than it should be, and it produces clunkier and harder to maintain code, as well as code that is likely insufficiently protected from these hazards until you get visibly bitten by the bugs.

We need a generalized solution to **all of the trust issues**, one that can be reused for as many callbacks as we create without all the extra boilerplate overhead.

We need something better than callbacks. They've served us well to this point, but the *future* of JavaScript demands more sophisticated and capable async patterns. The subsequent chapters in this book will dive into those emerging evolutions.