You Don't Know JS: Async & Performance

Chapter 1: Asynchrony: Now & Later

One of the most important and yet often misunderstood parts of programming in a language like JavaScript is how to express and manipulate program behavior spread out over a period of time.

This is not just about what happens from the beginning of a `for` loop to the end of a `for` loop, which of course takes *some time* (microseconds to milliseconds) to complete. It's about what happens when part of your program runs *now*, and another part of your program runs *later* -- there's a gap between *now* and *later* where your program isn't actively executing.
Practically all nontrivial programs ever written (especially in JS) have in some way or another had to manage this gap, whether that be in waiting for user input, requesting data from a database or file system, sending data across the network and waiting for a response, or performing a repeated task at a fixed interval of time (like animation). In all these various ways, your program has to manage state across the gap in time. As they famously say in London (of the chasm between the subway door and the platform): "mind the gap."

In fact, the relationship between the *now* and *later* parts of your program is at the heart of asynchronous programming.

Asynchronous programming has been around since the beginning of JS, for sure. But most JS developers have never really carefully considered exactly how and why it crops up in their programs, or explored various *other* ways to handle it. The *good enough* approach has always been the humble callback function. Many to this day will insist that callbacks are more than sufficient.

But as JS continues to grow in both scope and complexity, to meet the ever-widening demands of a first-class programming language that runs in browsers and servers and every conceivable device in between, the pains by which we manage asynchrony are becoming increasingly crippling, and they cry out for approaches that are both more capable and more reason-able.

While this all may seem rather abstract right now, I assure you we'll tackle it more completely and concretely as we go on through this book. We'll explore a variety of emerging techniques for async JavaScript programming over the next several chapters.

But before we can get there, we're going to have to understand much more deeply what asynchrony is and how it operates in JS.

A Program in Chunks

You may write your JS program in one *.js* file, but your program is almost certainly comprised of several chunks, only one of which is going to execute *now*, and the rest of which will execute *later*. The most common unit of *chunk* is the `function`.
The problem most developers new to JS seem to have is that *later* doesn't happen strictly and immediately after *now*. In other words, tasks that cannot complete *now* are, by definition, going to complete asynchronously, and thus we will not have blocking behavior as you might intuitively expect or want.

Consider:

```
// ajax(..) is some arbitrary Ajax function given by a library
var data = ajax( "http://some.url.1" );

console.log( data );
// Oops! `data` generally won't have the Ajax results
```

You're probably aware that standard Ajax requests don't complete synchronously, which means the `ajax(..)` function does not yet have any value to return back to be assigned to `data` variable. If `ajax(..)` *could* block until the response came back, then the `data = ..` assignment would work fine.
But that's not how we do Ajax. We make an asynchronous Ajax request *now*, and we won't get the results back until *later*.

The simplest (but definitely not only, or necessarily even best!) way of "waiting" from *now* until *later* is to use a function, commonly called a callback function:

```
// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", function myCallbackFunction(data){

        console.log( data ); // Yay, I gots me some `data`!

} );
```

Warning: You may have heard that it's possible to make synchronous Ajax requests. While that's technically true, you should never, ever do it, under any circumstances,

because it locks the browser UI (buttons, menus, scrolling, etc.) and prevents any user interaction whatsoever. This is a terrible idea, and should always be avoided.

Before you protest in disagreement, no, your desire to avoid the mess of callbacks is *not* justification for blocking, synchronous Ajax.

For example, consider this code:

```
function now() {
	return 21;
}

function later() {
	answer = answer * 2;
	console.log( "Meaning of life:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // Meaning of life: 42
```

There are two chunks to this program: the stuff that will run *now*, and the stuff that will run *later*. It should be fairly obvious what those two chunks are, but let's be super explicit:

Now:

```
function now() {
	return 21;
}

function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

Later:

```
answer = answer * 2;
console.log( "Meaning of life:", answer );
```

The *now* chunk runs right away, as soon as you execute your program.
But `setTimeout(..)` also sets up an event (a timeout) to happen *later*, so the contents of the `later()` function will be executed at a later time (1,000 milliseconds from now).
Any time you wrap a portion of code into a `function` and specify that it should be executed in response to some event (timer, mouse click, Ajax response, etc.), you are creating a *later* chunk of your code, and thus introducing asynchrony to your program.

Async Console

There is no specification or set of requirements around how the `console.*` methods work -- they are not officially part of JavaScript, but are instead added to JS by the *hosting environment* (see the *Types & Grammar* title of this book series).
So, different browsers and JS environments do as they please, which can sometimes lead to confusing behavior.

In particular, there are some browsers and some conditions that `console.log(..)` does not actually immediately output what it's given. The main reason this may happen is because I/O is a very slow and blocking part of many programs (not just JS). So, it may perform better (from the page/UI perspective) for a browser to handle `console` I/O asynchronously in the background, without you perhaps even knowing that occurred.
A not terribly common, but possible, scenario where this could be *observable* (not from code itself but from the outside):

```
var a = {
        index: 1
};

// later
console.log( a ); // ??

// even later
a.index++;
```

We'd normally expect to see the `a` object be snapshotted at the exact moment of the `console.log(..)` statement, printing something like `{ index: 1 }`, such that in the next statement when `a.index++` happens, it's modifying something different than, or just strictly after, the output of `a`.
Most of the time, the preceding code will probably produce an object representation in your developer tools' console that's what you'd expect. But it's possible this same code could run in a situation where the browser felt it needed to defer the console I/O to the background, in which case it's *possible* that by the time the object is represented in the browser console, the `a.index++` has already happened, and it shows `{ index: 2 }`.
It's a moving target under what conditions exactly `console` I/O will be deferred, or even whether it will be observable. Just be aware of this possible asynchronicity in I/O in case you ever run into issues in debugging where objects have been modified *after* a `console.log(..)` statement and yet you see the unexpected modifications show up.
Note: If you run into this rare scenario, the best option is to use breakpoints in your JS debugger instead of relying on `console` output. The next best option would be to force a

"snapshot" of the object in question by serializing it to a `string`, like with `JSON.stringify(..)`.

Event Loop

Let's make a (perhaps shocking) claim: despite clearly allowing asynchronous JS code (like the timeout we just looked at), up until recently (ES6), JavaScript itself has actually never had any direct notion of asynchrony built into it.

What!? That seems like a crazy claim, right? In fact, it's quite true. The JS engine itself has never done anything more than execute a single chunk of your program at any given moment, when asked to.

"Asked to." By whom? That's the important part!

The JS engine doesn't run in isolation. It runs inside a *hosting environment*, which is for most developers the typical web browser. Over the last several years (but by no means exclusively), JS has expanded beyond the browser into other environments, such as servers, via things like Node.js. In fact, JavaScript gets embedded into all kinds of devices these days, from robots to lightbulbs.

But the one common "thread" (that's a not-so-subtle asynchronous joke, for what it's worth) of all these environments is that they have a mechanism in them that handles executing multiple chunks of your program *over time*, at each moment invoking the JS engine, called the "event loop."

In other words, the JS engine has had no innate sense of *time*, but has instead been an on-demand execution environment for any arbitrary snippet of JS. It's the surrounding environment that has always *scheduled* "events" (JS code executions).

So, for example, when your JS program makes an Ajax request to fetch some data from a server, you set up the "response" code in a function (commonly called a "callback"), and the JS engine tells the hosting environment, "Hey, I'm going to suspend execution for now, but whenever you finish with that network request, and you have some data, please *call* this function *back*."

The browser is then set up to listen for the response from the network, and when it has something to give you, it schedules the callback function to be executed by inserting it into the *event loop*.

So what is the *event loop*?

Let's conceptualize it first through some fake-ish code:

```
// `eventLoop` is an array that acts as a queue (first-in, first-out)
var eventLoop = [ ];
var event;

// keep going "forever"
while (true) {
        // perform a "tick"
        if (eventLoop.length > 0) {
                // get the next event in the queue
                event = eventLoop.shift();

                // now, execute the next event
                try {
                        event();
                }
                catch (err) {
                        reportError(err);
                }
        }
}
```

This is, of course, vastly simplified pseudocode to illustrate the concepts. But it should be enough to help get a better understanding.

As you can see, there's a continuously running loop represented by the `while` loop, and each iteration of this loop is called a "tick." For each tick, if an event is waiting on the queue, it's taken off and executed. These events are your function callbacks.
It's important to note that `setTimeout(..)` doesn't put your callback on the event loop queue. What it does is set up a timer; when the timer expires, the environment places your callback into the event loop, such that some future tick will pick it up and execute it.
What if there are already 20 items in the event loop at that moment? Your callback waits. It gets in line behind the others -- there's not normally a path for preempting the queue and skipping ahead in line. This explains why `setTimeout(..)` timers may not fire with perfect temporal accuracy. You're guaranteed (roughly speaking) that your callback won't fire *before* the time interval you specify, but it can happen at or after that time, depending on the state of the event queue.
So, in other words, your program is generally broken up into lots of small chunks, which happen one after the other in the event loop queue. And technically, other events not related directly to your program can be interleaved within the queue as well.

Note: We mentioned "up until recently" in relation to ES6 changing the nature of where the event loop queue is managed. It's mostly a formal technicality, but ES6 now specifies how the event loop works, which means technically it's within the purview of the JS engine, rather than just the *hosting environment*. One main reason for this change is the

introduction of ES6 Promises, which we'll discuss in Chapter 3, because they require the ability to have direct, fine-grained control over scheduling operations on the event loop queue (see the discussion of `setTimeout(..0)` in the "Cooperation" section).

Parallel Threading

It's very common to conflate the terms "async" and "parallel," but they are actually quite different. Remember, async is about the gap between *now* and *later*. But parallel is about things being able to occur simultaneously.

The most common tools for parallel computing are processes and threads. Processes and threads execute independently and may execute simultaneously: on separate processors, or even separate computers, but multiple threads can share the memory of a single process.

An event loop, by contrast, breaks its work into tasks and executes them in serial, disallowing parallel access and changes to shared memory. Parallelism and "serialism" can coexist in the form of cooperating event loops in separate threads.

The interleaving of parallel threads of execution and the interleaving of asynchronous events occur at very different levels of granularity.

For example:

```
function later() {
        answer = answer * 2;
        console.log( "Meaning of life:", answer );
}
```

While the entire contents of `later()` would be regarded as a single event loop queue entry, when thinking about a thread this code would run on, there's actually perhaps a dozen different low-level operations. For example, `answer = answer * 2` requires first loading the current value of `answer`, then putting `2` somewhere, then performing the multiplication, then taking the result and storing it back into `answer`.
In a single-threaded environment, it really doesn't matter that the items in the thread queue are low-level operations, because nothing can interrupt the thread. But if you have a parallel system, where two different threads are operating in the same program, you could very likely have unpredictable behavior.

Consider:

```
var a = 20;

function foo() {
```

```
        a = a + 1;
}

function bar() {
        a = a * 2;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

In JavaScript's single-threaded behavior, if `foo()` runs before `bar()`, the result is that `a` has `42`, but if `bar()` runs before `foo()` the result in `a` will be `41`.

If JS events sharing the same data executed in parallel, though, the problems would be much more subtle. Consider these two lists of pseudocode tasks as the threads that could respectively run the code in `foo()` and `bar()`, and consider what happens if they are running at exactly the same time:

Thread 1 (`X` and `Y` are temporary memory locations):

```
foo():
  a. load value of `a` in `X`
  b. store `1` in `Y`
  c. add `X` and `Y`, store result in `X`
  d. store value of `X` in `a`
```

Thread 2 (`X` and `Y` are temporary memory locations):

```
bar():
  a. load value of `a` in `X`
  b. store `2` in `Y`
  c. multiply `X` and `Y`, store result in `X`
  d. store value of `X` in `a`
```

Now, let's say that the two threads are running truly in parallel. You can probably spot the problem, right? They use shared memory locations `X` and `Y` for their temporary steps. What's the end result in `a` if the steps happen like this?

```
1a  (load value of `a` in `X`   ==> `20`)
2a  (load value of `a` in `X`   ==> `20`)
1b  (store `1` in `Y`   ==> `1`)
2b  (store `2` in `Y`   ==> `2`)
1c  (add `X` and `Y`, store result in `X`   ==> `22`)
1d  (store value of `X` in `a`   ==> `22`)
2c  (multiply `X` and `Y`, store result in `X`   ==> `44`)
2d  (store value of `X` in `a`   ==> `44`)
```

The result in `a` will be `44`. But what about this ordering?

```
1a  (load value of `a` in `X`   ==> `20`)
2a  (load value of `a` in `X`   ==> `20`)
2b  (store `2` in `Y`   ==> `2`)
1b  (store `1` in `Y`   ==> `1`)
2c  (multiply `X` and `Y`, store result in `X`   ==> `20`)
1c  (add `X` and `Y`, store result in `X`   ==> `21`)
1d  (store value of `X` in `a`   ==> `21`)
2d  (store value of `X` in `a`   ==> `21`)
```

The result in `a` will be `21`.

So, threaded programming is very tricky, because if you don't take special steps to prevent this kind of interruption/interleaving from happening, you can get very surprising, nondeterministic behavior that frequently leads to headaches.

JavaScript never shares data across threads, which means *that* level of nondeterminism isn't a concern. But that doesn't mean JS is always deterministic. Remember earlier, where the relative ordering of `foo()` and `bar()` produces two different results (`41` or `42`)?
Note: It may not be obvious yet, but not all nondeterminism is bad. Sometimes it's irrelevant, and sometimes it's intentional. We'll see more examples of that throughout this and the next few chapters.

Run-to-Completion

Because of JavaScript's single-threading, the code inside of `foo()` (and `bar()`) is atomic, which means that once `foo()` starts running, the entirety of its code will finish before any of the code in `bar()` can run, or vice versa. This is called "run-to-completion" behavior.
In fact, the run-to-completion semantics are more obvious when `foo()` and `bar()` have more code in them, such as:

```
var a = 1;
var b = 2;

function foo() {
        a++;
        b = b * a;
        a = b + 3;
}

function bar() {
        b--;
        a = 8 + b;
        b = a * 2;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Because `foo()` can't be interrupted by `bar()`, and `bar()` can't be interrupted by `foo()`, this program only has two possible outcomes depending on which starts running first -- if threading were present, and the individual statements in `foo()` and `bar()` could be interleaved, the number of possible outcomes would be greatly increased!
Chunk 1 is synchronous (happens *now*), but chunks 2 and 3 are asynchronous (happen *later*), which means their execution will be separated by a gap of time.

Chunk 1:

```
var a = 1;
var b = 2;
```

Chunk 2 (`foo()`):

```
a++;
b = b * a;
a = b + 3;
```

Chunk 3 (`bar()`):

```
b--;
a = 8 + b;
b = a * 2;
```

Chunks 2 and 3 may happen in either-first order, so there are two possible outcomes for this program, as illustrated here:

Outcome 1:

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

Outcome 2:

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

Two outcomes from the same code means we still have nondeterminism! But it's at the function (event) ordering level, rather than at the statement ordering level (or, in fact, the expression operation ordering level) as it is with threads. In other words, it's *more deterministic* than threads would have been.

As applied to JavaScript's behavior, this function-ordering nondeterminism is the common term "race condition," as `foo()` and `bar()` are racing against each other to see which runs first. Specifically, it's a "race condition" because you cannot predict reliably how `a` and `b` will turn out.
Note: If there was a function in JS that somehow did not have run-to-completion behavior, we could have many more possible outcomes, right? It turns out ES6 introduces just such a thing (see Chapter 4 "Generators"), but don't worry right now, we'll come back to that!

Concurrency

Let's imagine a site that displays a list of status updates (like a social network news feed) that progressively loads as the user scrolls down the list. To make such a feature work correctly, (at least) two separate "processes" will need to be executing *simultaneously* (i.e., during the same window of time, but not necessarily at the same instant).

Note: We're using "process" in quotes here because they aren't true operating system–level processes in the computer science sense. They're virtual processes, or tasks, that represent a logically connected, sequential series of operations. We'll simply prefer "process" over "task" because terminology-wise, it will match the definitions of the concepts we're exploring.

The first "process" will respond to `onscroll` events (making Ajax requests for new content) as they fire when the user has scrolled the page further down. The second "process" will receive Ajax responses back (to render content onto the page). Obviously, if a user scrolls fast enough, you may see two or more `onscroll` events fired during the time it takes to get the first response back and process, and thus you're going to have `onscroll` events and Ajax response events firing rapidly, interleaved with each other.
Concurrency is when two or more "processes" are executing simultaneously over the same period, regardless of whether their individual constituent operations happen *in parallel* (at the same instant on separate processors or cores) or not. You can think of concurrency then as "process"-level (or task-level) parallelism, as opposed to operation-level parallelism (separate-processor threads).

Note: Concurrency also introduces an optional notion of these "processes" interacting with each other. We'll come back to that later.

For a given window of time (a few seconds worth of a user scrolling), let's visualize each independent "process" as a series of events/operations:

"Process" 1 (`onscroll` events):

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
onscroll, request 7
```

"Process" 2 (Ajax response events):

```
response 1
response 2
response 3
response 4
response 5
response 6
response 7
```

It's quite possible that an `onscroll` event and an Ajax response event could be ready to be processed at exactly the same *moment*. For example, let's visualize these events in a timeline:

```
onscroll, request 1
onscroll, request 2          response 1
onscroll, request 3          response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6          response 4
onscroll, request 7
response 6
response 5
response 7
```

But, going back to our notion of the event loop from earlier in the chapter, JS is only going to be able to handle one event at a time, so either `onscroll, request 2` is going to happen first or `response 1` is going to happen first, but they cannot happen at literally the same moment. Just like kids at a school cafeteria, no matter what crowd they form outside the doors, they'll have to merge into a single line to get their lunch!

Let's visualize the interleaving of all these events onto the event loop queue.

Event Loop Queue:

```
onscroll, request 1   <--- Process 1 starts
onscroll, request 2
response 1            <--- Process 2 starts
```

```
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7   <--- Process 1 finishes
response 6
response 5
response 7            <--- Process 2 finishes
```

"Process 1" and "Process 2" run concurrently (task-level parallel), but their individual events run sequentially on the event loop queue.

By the way, notice how `response 6` and `response 5` came back out of expected order? The single-threaded event loop is one expression of concurrency (there are certainly others, which we'll come back to later).

Noninteracting

As two or more "processes" are interleaving their steps/events concurrently within the same program, they don't necessarily need to interact with each other if the tasks are unrelated. **If they don't interact, nondeterminism is perfectly acceptable.**

For example:

```
var res = {};

function foo(results) {
        res.foo = results;
}

function bar(results) {
        res.bar = results;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

`foo()` and `bar()` are two concurrent "processes," and it's nondeterminate which order they will be fired in. But we've constructed the program so it doesn't matter what order they fire in, because they act independently and as such don't need to interact.
This is not a "race condition" bug, as the code will always work correctly, regardless of the ordering.

Interaction

More commonly, concurrent "processes" will by necessity interact, indirectly through scope and/or the DOM. When such interaction will occur, you need to coordinate these interactions to prevent "race conditions," as described earlier.

Here's a simple example of two concurrent "processes" that interact because of implied ordering, which is only *sometimes broken*:

```
var res = [];

function response(data) {
        res.push( data );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

The concurrent "processes" are the two `response()` calls that will be made to handle the Ajax responses. They can happen in either-first order.
Let's assume the expected behavior is that `res[0]` has the results of the `"http://some.url.1"` call, and `res[1]` has the results of the `"http://some.url.2"` call. Sometimes that will be the case, but sometimes they'll be flipped, depending on which call finishes first. There's a pretty good likelihood that this nondeterminism is a "race condition" bug.
Note: Be extremely wary of assumptions you might tend to make in these situations. For example, it's not uncommon for a developer to observe that `"http://some.url.2"` is "always" much slower to respond than `"http://some.url.1"`, perhaps by virtue of what tasks they're doing (e.g., one performing a database task and the other just fetching a static file), so the observed ordering seems to always be as expected. Even if both requests go to the same server, and *it* intentionally responds in a certain order, there's no *real* guarantee of what order the responses will arrive back in the browser.
So, to address such a race condition, you can coordinate ordering interaction:

```
var res = [];

function response(data) {
        if (data.url == "http://some.url.1") {
                res[0] = data;
        }
        else if (data.url == "http://some.url.2") {
                res[1] = data;
        }
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

Regardless of which Ajax response comes back first, we inspect the `data.url` (assuming one is returned from the server, of course!) to figure out which position the response data should occupy in the `res` array. `res[0]` will always hold the `"http://some.url.1"` results and `res[1]` will always hold the `"http://some.url.2"` results. Through simple coordination, we eliminated the "race condition" nondeterminism.

The same reasoning from this scenario would apply if multiple concurrent function calls were interacting with each other through the shared DOM, like one updating the contents of a `<div>` and the other updating the style or attributes of the `<div>` (e.g., to make the DOM element visible once it has content). You probably wouldn't want to show the DOM element before it had content, so the coordination must ensure proper ordering interaction.

Some concurrency scenarios are *always broken* (not just *sometimes*) without coordinated interaction. Consider:

```
var a, b;

function foo(x) {
        a = x * 2;
        baz();
}

function bar(y) {
        b = y * 2;
        baz();
}

function baz() {
        console.log(a + b);
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

In this example, whether `foo()` or `bar()` fires first, it will always cause `baz()` to run too early (either `a` or `b` will still be `undefined`), but the second invocation of `baz()` will work, as both `a` and `b` will be available.

There are different ways to address such a condition. Here's one simple way:

```
var a, b;

function foo(x) {
        a = x * 2;
        if (a && b) {
                baz();
        }
}
```

```
function bar(y) {
        b = y * 2;
        if (a && b) {
                baz();
        }
}

function baz() {
        console.log( a + b );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

The `if (a && b)` conditional around the `baz()` call is traditionally called a "gate," because we're not sure what order `a` and `b` will arrive, but we wait for both of them to get there before we proceed to open the gate (call `baz()`).
Another concurrency interaction condition you may run into is sometimes called a "race," but more correctly called a "latch." It's characterized by "only the first one wins" behavior. Here, nondeterminism is acceptable, in that you are explicitly saying it's OK for the "race" to the finish line to have only one winner.

Consider this broken code:

```
var a;

function foo(x) {
        a = x * 2;
        baz();
}

function bar(x) {
        a = x / 2;
        baz();
}

function baz() {
        console.log( a );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Whichever one (`foo()` or `bar()`) fires last will not only overwrite the assigned `a` value from the other, but it will also duplicate the call to `baz()` (likely undesired).
So, we can coordinate the interaction with a simple latch, to let only the first one through:

```
var a;
```

```
function foo(x) {
        if (a == undefined) {
                a = x * 2;
                baz();
        }
}

function bar(x) {
        if (a == undefined) {
                a = x / 2;
                baz();
        }
}

function baz() {
        console.log( a );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

The `if (a == undefined)` conditional allows only the first of `foo()` or `bar()` through, and the second (and indeed any subsequent) calls would just be ignored. There's just no virtue in coming in second place!

Note: In all these scenarios, we've been using global variables for simplistic illustration purposes, but there's nothing about our reasoning here that requires it. As long as the functions in question can access the variables (via scope), they'll work as intended. Relying on lexically scoped variables (see the *Scope & Closures* title of this book series), and in fact global variables as in these examples, is one obvious downside to these forms of concurrency coordination. As we go through the next few chapters, we'll see other ways of coordination that are much cleaner in that respect.

Cooperation

Another expression of concurrency coordination is called "cooperative concurrency." Here, the focus isn't so much on interacting via value sharing in scopes (though that's obviously still allowed!). The goal is to take a long-running "process" and break it up into steps or batches so that other concurrent "processes" have a chance to interleave their operations into the event loop queue.

For example, consider an Ajax response handler that needs to run through a long list of results to transform the values. We'll use `Array#map(..)` to keep the code shorter:

```
var res = [];

// `response(..)` receives array of results from the Ajax call
function response(data) {
```

```
        // add onto existing `res` array
        res = res.concat(
                // make a new transformed array with all `data` values doubled
                data.map( function(val){
                        return val * 2;
                } )
        );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

If `"http://some.url.1"` gets its results back first, the entire list will be mapped into `res` all at once. If it's a few thousand or less records, this is not generally a big deal. But if it's say 10 million records, that can take a while to run (several seconds on a powerful laptop, much longer on a mobile device, etc.).

While such a "process" is running, nothing else in the page can happen, including no other `response(..)` calls, no UI updates, not even user events like scrolling, typing, button clicking, and the like. That's pretty painful.

So, to make a more cooperatively concurrent system, one that's friendlier and doesn't hog the event loop queue, you can process these results in asynchronous batches, after each one "yielding" back to the event loop to let other waiting events happen.

Here's a very simple approach:

```
var res = [];

// `response(..)` receives array of results from the Ajax call
function response(data) {
        // let's just do 1000 at a time
        var chunk = data.splice( 0, 1000 );

        // add onto existing `res` array
        res = res.concat(
                // make a new transformed array with all `chunk` values doubled
                chunk.map( function(val){
                        return val * 2;
                } )
        );

        // anything left to process?
        if (data.length > 0) {
                // async schedule next batch
                setTimeout( function(){
                        response( data );
                }, 0 );
        }
}

// ajax(..) is some arbitrary Ajax function given by a library
```

```
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

We process the data set in maximum-sized chunks of 1,000 items. By doing so, we ensure a short-running "process," even if that means many more subsequent "processes," as the interleaving onto the event loop queue will give us a much more responsive (performant) site/app.

Of course, we're not interaction-coordinating the ordering of any of these "processes," so the order of results in `res` won't be predictable. If ordering was required, you'd need to use interaction techniques like those we discussed earlier, or ones we will cover in later chapters of this book.
We use the `setTimeout(..0)` (hack) for async scheduling, which basically just means "stick this function at the end of the current event loop queue."
Note: `setTimeout(..0)` is not technically inserting an item directly onto the event loop queue. The timer will insert the event at its next opportunity. For example, two subsequent `setTimeout(..0)` calls would not be strictly guaranteed to be processed in call order, so it *is* possible to see various conditions like timer drift where the ordering of such events isn't predictable. In Node.js, a similar approach is `process.nextTick(..)`. Despite how convenient (and usually more performant) it would be, there's not a single direct way (at least yet) across all environments to ensure async event ordering. We cover this topic in more detail in the next section.

Jobs

As of ES6, there's a new concept layered on top of the event loop queue, called the "Job queue." The most likely exposure you'll have to it is with the asynchronous behavior of Promises (see Chapter 3).

Unfortunately, at the moment it's a mechanism without an exposed API, and thus demonstrating it is a bit more convoluted. So we're going to have to just describe it conceptually, such that when we discuss async behavior with Promises in Chapter 3, you'll understand how those actions are being scheduled and processed.

So, the best way to think about this that I've found is that the "Job queue" is a queue hanging off the end of every tick in the event loop queue. Certain async-implied actions that may occur during a tick of the event loop will not cause a whole new event to be added to the event loop queue, but will instead add an item (aka Job) to the end of the current tick's Job queue.

It's kinda like saying, "oh, here's this other thing I need to do *later*, but make sure it happens right away before anything else can happen."

Or, to use a metaphor: the event loop queue is like an amusement park ride, where once you finish the ride, you have to go to the back of the line to ride again. But the Job queue is like finishing the ride, but then cutting in line and getting right back on.

A Job can also cause more Jobs to be added to the end of the same queue. So, it's theoretically possible that a Job "loop" (a Job that keeps adding another Job, etc.) could spin indefinitely, thus starving the program of the ability to move on to the next event loop tick. This would conceptually be almost the same as just expressing a long-running or infinite loop (like `while (true) ..`) in your code.
Jobs are kind of like the spirit of the `setTimeout(..0)` hack, but implemented in such a way as to have a much more well-defined and guaranteed ordering: **later, but as soon as possible**.
Let's imagine an API for scheduling Jobs (directly, without hacks), and call it `schedule(..)`. Consider:

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

setTimeout( function(){
        console.log( "B" );
}, 0 );

// theoretical "Job API"
schedule( function(){
        console.log( "C" );

        schedule( function(){
                console.log( "D" );
        } );
} );
```

You might expect this to print out `A B C D`, but instead it would print out `A C D B`, because the Jobs happen at the end of the current event loop tick, and the timer fires to schedule for the *next* event loop tick (if available!).
In Chapter 3, we'll see that the asynchronous behavior of Promises is based on Jobs, so it's important to keep clear how that relates to event loop behavior.

Statement Ordering

The order in which we express statements in our code is not necessarily the same order as the JS engine will execute them. That may seem like quite a strange assertion to make, so we'll just briefly explore it.

But before we do, we should be crystal clear on something: the rules/grammar of the language (see the *Types & Grammar* title of this book series) dictate a very predictable and reliable behavior for statement ordering from the program point of view. So what

we're about to discuss are **not things you should ever be able to observe** in your JS program.

Warning: If you are ever able to *observe* compiler statement reordering like we're about to illustrate, that'd be a clear violation of the specification, and it would unquestionably be due to a bug in the JS engine in question -- one which should promptly be reported and fixed! But it's vastly more common that you *suspect* something crazy is happening in the JS engine, when in fact it's just a bug (probably a "race condition"!) in your own code -- so look there first, and again and again. The JS debugger, using breakpoints and stepping through code line by line, will be your most powerful tool for sniffing out such bugs in *your code*.

Consider:

```
var a, b;

a = 10;
b = 30;

a = a + 1;
b = b + 1;

console.log( a + b ); // 42
```

This code has no expressed asynchrony to it (other than the rare `console` async I/O discussed earlier!), so the most likely assumption is that it would process line by line in top-down fashion.
But it's *possible* that the JS engine, after compiling this code (yes, JS is compiled -- see the *Scope & Closures* title of this book series!) might find opportunities to run your code faster by rearranging (safely) the order of these statements. Essentially, as long as you can't observe the reordering, anything's fair game.

For example, the engine might find it's faster to actually execute the code like this:

```
var a, b;

a = 10;
a++;

b = 30;
b++;

console.log( a + b ); // 42
```

Or this:

```
var a, b;
```

```
a = 11;
b = 31;

console.log( a + b ); // 42
```

Or even:

```
// because `a` and `b` aren't used anymore, we can
// inline and don't even need them!
console.log( 42 ); // 42
```

In all these cases, the JS engine is performing safe optimizations during its compilation, as the end *observable* result will be the same.

But here's a scenario where these specific optimizations would be unsafe and thus couldn't be allowed (of course, not to say that it's not optimized at all):

```
var a, b;

a = 10;
b = 30;

// we need `a` and `b` in their preincremented state!
console.log( a * b ); // 300

a = a + 1;
b = b + 1;

console.log( a + b ); // 42
```

Other examples where the compiler reordering could create observable side effects (and thus must be disallowed) would include things like any function call with side effects (even and especially getter functions), or ES6 Proxy objects (see the *ES6 & Beyond* title of this book series).

Consider:

```
function foo() {
        console.log( b );
        return 1;
}

var a, b, c;

// ES5.1 getter literal syntax
c = {
        get bar() {
                console.log( a );
                return 1;
        }
};
```

```
a = 10;
b = 30;

a += foo();                             // 30
b += c.bar;                             // 11

console.log( a + b );    // 42
```

If it weren't for the `console.log(..)` statements in this snippet (just used as a convenient form of observable side effect for the illustration), the JS engine would likely have been free, if it wanted to (who knows if it would!?), to reorder the code to:

```
// ...

a = 10 + foo();
b = 30 + c.bar;

// ...
```

While JS semantics thankfully protect us from the *observable* nightmares that compiler statement reordering would seem to be in danger of, it's still important to understand just how tenuous a link there is between the way source code is authored (in top-down fashion) and the way it runs after compilation.

Compiler statement reordering is almost a micro-metaphor for concurrency and interaction. As a general concept, such awareness can help you understand async JS code flow issues better.

Review

A JavaScript program is (practically) always broken up into two or more chunks, where the first chunk runs *now* and the next chunk runs *later*, in response to an event. Even though the program is executed chunk-by-chunk, all of them share the same access to the program scope and state, so each modification to state is made on top of the previous state.

Whenever there are events to run, the *event loop* runs until the queue is empty. Each iteration of the event loop is a "tick." User interaction, IO, and timers enqueue events on the event queue.

At any given moment, only one event can be processed from the queue at a time. While an event is executing, it can directly or indirectly cause one or more subsequent events.

Concurrency is when two or more chains of events interleave over time, such that from a high-level perspective, they appear to be running *simultaneously* (even though at any given moment only one event is being processed).

It's often necessary to do some form of interaction coordination between these concurrent "processes" (as distinct from operating system processes), for instance to ensure ordering or to prevent "race conditions." These "processes" can also *cooperate* by breaking themselves into smaller chunks and to allow other "process" interleaving.