O'REILLY®

"Kyle wrangles the asynchronous nature of JavaScript and shows you how to straighten it out with promises and generators."
–**MARC GRABANSKI**, CEO & UI Developer, Frontend Masters

KYLE SIMPSON

ASYNC & PERFORMANCE

YOU DON'T KNOW JS

The *YOU DON'T KNOW JS* series includes:

- *Up & Going*
- *Scope & Closures*
- *this & Object Prototypes*
- *Types & Grammar*
- *Async & Performance*
- *ES6 & Beyond*

ASYNC & PERFORMANCE

No matter how much experience you have with JavaScript, odds are you don't fully understand the language. As part of the *You Don't Know JS* series, this concise yet in-depth guide focuses on new asynchronous features and performance techniques–including Promises, generators, and Web Workers–that let you create sophisticated single-page web applications *and* escape callback hell in the process.

Like other books in this series, *You Don't Know JS: Async & Performance* dives into trickier parts of the language that many JavaScript programmers simply avoid. Armed with this knowledge, you can become a true JavaScript master.

WITH THIS BOOK YOU WILL:

- **Explore old and new JavaScript methods for handling asynchronous programming**
- **Escape from "callback hell," using JavaScript Promises to fix "inversion of control."**
- **Use generators to express async flow in a sequential, synchronous-looking fashion**
- **Tackle program-level performance with Web Workers, SIMD, and asm.js**
- **Learn valuable resources and techniques for benchmarking and tuning your expressions and statements**

KYLE SIMPSON is an Open Web Evangelist who's passionate about all things JavaScript. He's an author, workshop trainer, tech speaker, and OSS contributor/leader.

JAVASCRIPT

US $24.99 **CAN $28.99**

ISBN: 978-1-491-90422-0

9 781491 904220 52499

O'REILLY®

oreilly.com

YouDontKnowJS.com

Async & Performance

Kyle Simpson

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Async & Performance
by Kyle Simpson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Brian MacDonald
Production Editor: Kristen Brown

Copyeditor: Jasmine Kwityn
Proofreader: Phil Dangler
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen

March 2015: First Edition

Revision History for the First Edition
2015-02-19: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491904220* for release details.

978-1-491-90422-0

[LSI]

Table of Contents

Foreword

Over the years, my employer has trusted me enough to conduct interviews. If we're looking for someone with JavaScript skills, my first line of questioning…well, actually, is to check if the candidate needs the bathroom and/or a drink, because comfort is important. But once I'm past the bit about the candidate's fluid intake/output, I set about determining if the candidate knows JavaScript, or just jQuery.

Not that there's anything wrong with jQuery. It lets you do a lot without really knowing JavaScript, and that's a feature—not a bug. But if the job calls for advanced skills in JavaScript performance and maintainability, you need someone who knows how libraries such as jQuery are put together. You need to be able to harness the core of JavaScript the same way they do.

If I want to get a picture of someone's core JavaScript skill, I'm most interested in what they make of closures (you've read the *You Don't Know JS: Scope & Closures* title of this series already, right?) and how to get the most out of asynchronicity, which brings us to this book.

For starters, you'll be taken through callbacks, the bread and butter of asynchronous programming. Of course, bread and butter does not make for a particularly satisfying meal, but the next course is full of tasty, tasty Promises!

If you don't know Promises, now is the time to learn. Promises are now the official way to provide async return values in both JavaScript and the DOM. All future async DOM APIs will use them, and many already do, so be prepared! At the time of writing, Promises have shipped in most major browsers, with IE shipping soon. Once

you've finished savoring Promises, I hope you left room for the next course, Generators.

Generators snuck their way into stable versions of Chrome and Firefox without too much pomp and ceremony, because, frankly, they're more complicated than they are interesting. Or, that's what I thought until I saw them combined with Promises. There, they become an important tool in readability and maintenance.

For dessert, well, I won't spoil the surprise, but prepare to gaze into the future of JavaScript! This book covers features that give you more and more control over concurrency and asynchronicity.

Well, I won't block your enjoyment of the book any longer—on with the show! If you've already read part of the book before reading this foreword, give yourself 10 asynchronous points! You deserve them!

—Jake Archibald (http://jakearchibald.com, @jaffathecake),
Developer Advocate at Google Chrome

Preface

I'm sure you noticed, but "JS" in the series title is not an abbreviation for words used to curse about JavaScript, though cursing at the language's quirks is something we can probably all identify with!

From the earliest days of the Web, JavaScript has been a foundational technology that drives interactive experience around the content we consume. While flickering mouse trails and annoying pop-up prompts may be where JavaScript started, nearly two decades later, the technology and capability of JavaScript has grown many orders of magnitude, and few doubt its importance at the heart of the world's most widely available software platform: the Web.

But as a language, it has perpetually been a target for a great deal of criticism, owing partly to its heritage but even more to its design philosophy. Even the name evokes, as Brendan Eich once put it, "dumb kid brother" status next to its more mature older brother Java. But the name is merely an accident of politics and marketing. The two languages are vastly different in many important ways. "JavaScript" is as related to "Java" as "Carnival" is to "Car."

Because JavaScript borrows concepts and syntax idioms from several languages, including proud C-style procedural roots as well as subtle, less obvious Scheme/Lisp-style functional roots, it is exceedingly approachable to a broad audience of developers, even those with little to no programming experience. The "Hello World" of JavaScript is so simple that the language is inviting and easy to get comfortable with in early exposure.

While JavaScript is perhaps one of the easiest languages to get up and running with, its eccentricities make solid mastery of the language a vastly less common occurrence than in many other lan-

guages. Where it takes a pretty in-depth knowledge of a language like C or C++ to write a full-scale program, full-scale production JavaScript can, and often does, barely scratch the surface of what the language can do.

Sophisticated concepts that are deeply rooted into the language tend instead to surface themselves in *seemingly* simplistic ways, such as passing around functions as callbacks, which encourages the JavaScript developer to just use the language as-is and not worry too much about what's going on under the hood.

It is simultaneously a simple, easy-to-use language that has broad appeal, and a complex and nuanced collection of language mechanics that without careful study will elude *true understanding* even for the most seasoned of JavaScript developers.

Therein lies the paradox of JavaScript, the Achilles' heel of the language, the challenge we are presently addressing. Because JavaScript *can* be used without understanding, the understanding of the language is often never attained.

Mission

If at every point that you encounter a surprise or frustration in JavaScript, your response is to add it to the blacklist (as some are accustomed to doing), you soon will be relegated to a hollow shell of the richness of JavaScript.

While this subset has been famously dubbed "The Good Parts," I would implore you, dear reader, to instead consider it the "The Easy Parts," "The Safe Parts," or even "The Incomplete Parts."

This *You Don't Know JS* series offers a contrary challenge: learn and deeply understand *all* of JavaScript, even and especially "The Tough Parts."

Here, we address head-on the tendency of JS developers to learn "just enough" to get by, without ever forcing themselves to learn exactly how and why the language behaves the way it does. Furthermore, we eschew the common advice to retreat when the road gets rough.

I am not content, nor should you be, at stopping once something just works and not really knowing *why*. I gently challenge you to journey down that bumpy "road less traveled" and embrace all that JavaScript is and can do. With that knowledge, no technique, no framework, no popular buzzword acronym of the week will be beyond your understanding.

These books each take on specific core parts of the language that are most commonly misunderstood or under-understood, and dive very deep and exhaustively into them. You should come away from reading with a firm confidence in your understanding, not just of the theoretical, but the practical "what you need to know" bits.

The JavaScript you know right now is probably parts handed down to you by others who've been burned by incomplete understanding. *That* JavaScript is but a shadow of the true language. You don't really know JavaScript *yet*, but if you dig into this series, you will. Read on, my friends. JavaScript awaits you.

Review

JavaScript is awesome. It's easy to learn partially, and much harder to learn completely (or even *sufficiently*). When developers encounter confusion, they usually blame the language instead of their lack of understanding. These books aim to fix that, inspiring a strong appreciation for the language you can now, and *should*, deeply know.

Many of the examples in this book assume modern (and future-reaching) JavaScript engine environments, such as ES6. Some code may not work as described if run in older (pre-ES6) engines.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://bit.ly/ydkjs-async-code*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code

does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*You Don't Know JavaScript: Async & Performance* by Kyle Simpson (O'Reilly). Copyright 2015 Getify Solutions, Inc., 978-1-491-90422-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/ydkjs-async-performance*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

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 each *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 the `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`!

} );
```

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 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.

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 your clearly being able to write 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.

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 exactly 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 "Cooperation" on page 21).

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 load-

ing 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`)?

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 has only 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 reliably predict how `a` and `b` will turn out.

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), 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).

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 use "process" instead of "task" because terminology-wise, it matches 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). You can think of concurrency then as "process"-level (or task-level) parallelism, as opposed to operation-level parallelism (separate-processor threads).

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 can handle only 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:

```
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.