The Pragmatic Programmers

Pragmatic Express

Async JavaScript

Build More Responsive Apps with Less Code

Trevor Burnham

edited by Jacquelyn Carter

Early Praise for *Async JavaScript*

Async JavaScript is the first full book I've seen dedicated to a key topic in JavaScript development today: how to deal with concurrency and concurrent tasks without going crazy! For the sake of your sanity, check this out.

➤ **Peter Cooper, editor of JavaScript Weekly**

Trevor delivers a concise guide to writing asynchronous JavaScript with a perfect balance of browser and server-side examples. Part guide, part overview, wholly engaging, this book is a must-read for any JavaScript developer looking to level up.

➤ **Wynn Netherland, co-host of The Changelog**

This is a complete guide to the asynchronous realm of JavaScript. The concepts and tools covered by this book are essential to anyone willing to build full-blown, well-structured and efficient JavaScript applications.

➤ **Julien Biezemans, Ruby/JavaScript developer, author of Cucumber.js**

Async JavaScript

Build More Responsive Apps with Less Code

Trevor Burnham

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-27-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2012

Dedicated to Steve Jobs and to the generation
of entrepreneurs he inspired.

Contents

Acknowledgments

It was not love at first sight with me and JavaScript. Yet today, it's one of my two favorite programming languages. The other? Its little brother, CoffeeScript. The story of how I learned to stop worrying and love JavaScripting is a story shared by tens of thousands of programmers. I'd like to thank those who took JavaScript seriously from the start, shaping the rich development ecosystem the language enjoys today: John Resig, for creating the browser's *de facto* standard library, jQuery; Jeremy Ashkenas, for producing CoffeeScript and the rich yet minimalistic Backbone.js framework; Ryan Dahl, for giving the language a robust server environment; and all the other programmers who've proven through their work that JavaScript is a first-class language after all.

Of course, love alone didn't write this book. I'd like to thank the Pragmatic Bookshelf team for helping me thoroughly renovate my original KickStarted manuscript and raise it to the standard of quality that PragProg is famous for. Particular thanks go to managing editor Susannah Pfalzer, head honchos Dave Thomas and Andy Hunt, and most of all my editor, Jackie Carter. Their savvy and motivation have been invaluable.

Thanks also to my technical reviewers for this edition: Julien Biezemans, Christophe Porteneuve, Michael Ficarra, Travis Swicegood, and Lon Ingram. Special thanks to Karl Stolley for going above and beyond in multiple reviews. I'd also like to thank Stan Angeloff and Roly Fentanes for reviewing the original manuscript. Any remaining errors are entirely my fault.

Thanks, finally, to my employer, HubSpot, for supporting me as I brought this book to completion. After years of nomadic freelancing, I've finally found a home.

Trevor Burnham
trevorburnham@gmail.com
November 2012

Introduction

Originally devised to enhance web pages in Netscape 2.0, JavaScript is now faced with being a single-threaded language in a multimedia, multitasking, multicore world. Yet JavaScript has not only persevered since 1995, it's thrived. One after the other, potential rivals in the browser—Flash, Silverlight, and Java applets, to name a few—have come and (more or less) gone.

Meanwhile, when a programmer named Ryan Dahl wanted to build a new framework for event-driven servers, he searched the far reaches of computer science for a language that was both dynamic and single-threaded before realizing that the answer was right in front of him. And so, Node.js was born, and JavaScript became a force to be reckoned with in the server world.

How did this happen? As recently as 2001, Paul Graham wrote the following in his essay "The Other Road Ahead":[1]

> I would not even use JavaScript, if I were you... Most of the JavaScript I see on the Web isn't necessary, and much of it breaks.

Today, Graham is the lead partner at Y Combinator, the investment group behind Dropbox, Heroku, and hundreds of other start-ups—nearly all of which use JavaScript. As he put it in a revised version of the essay, "JavaScript now works."

When did JavaScript become a respectable language? Some say the turning point was Gmail (2004), which showed the world that with a heavy dose of Ajax you could run a first-class email client in the browser. Others say that it was jQuery (2006), which abstracted the rival browser APIs of the time to create a *de facto* standard. (As of 2011, 48 percent of the top 17,000 websites use jQuery.[2])

1. A revised version of this essay can be found at http://paulgraham.com/road.html. The original footnote can be found in the book *Hackers & Painters*.
2. http://appendto.com/jquery-overtakes-flash

Whatever the reason, JavaScript is here to stay. Apple got behind JavaScript with WebKit and Safari. Microsoft is getting behind JavaScript with Metro. Even Adobe is getting behind JavaScript with tools to generate HTML5 instead of Flash. What began as a humble browser feature has become arguably the most important programming language in the world.

Thanks to the ubiquity of web browsers, JavaScript has come closer than any other language to fulfilling Java's old promise of "write once, run anywhere." In 2007, Jeff Atwood coined *Atwood's law:*

> Any application that can be written in JavaScript will eventually be written in JavaScript.[3]

Trouble in Paradise

JavaScript was conceived to be a single-threaded language where asynchronous tasks are handled with events. When there are only a few potential events, event-based code is much simpler than multithreaded code. It's conceptually elegant, and it eliminates the need to wrap up data in mutexes and semaphores to make it thread-safe. But when a number of events are expected, with state that needs to be carried from one event to the next, that simplicity often gives way to a code structure so terrifying that it's been dubbed the *Pyramid of Doom*.

```
step1(function(result1) {
  step2(function(result2) {
    step3(function(result3) {
      // and so on...
    });
  });
});
```

"I love async, but I can't code like this," one developer famously complained on the Node.js Google Group.[4] But the problem isn't with the language itself; it's with the way programmers use the language. Dealing with complex sets of events in an elegant way is still frontier territory in JavaScript.

So, let's push the frontier forward! Let's prove to the world that even the most complex problems can be tackled with clean, maintainable JavaScript code.

3. http://www.codinghorror.com/blog/2007/07/the-principle-of-least-power.html
4. https://groups.google.com/forum/#!topic/nodejs/wzSUdkPlCWg

Who Is This Book For?

This book is aimed at intermediate JavaScripters. You should know how variables are scoped. Keywords like typeof, arguments, and this shouldn't faze you. Perhaps most importantly, you should understand that

```
func(function(arg) { return next(arg); });
```

is just a needlessly verbose way of writing

```
func(next);
```

except in rare cases. (See Reg Braithwaite's excellent article "Captain Obvious on JavaScript" for more examples of small but important functional idioms.)[5]

What you *don't* need to know is how asynchronous events are scheduled in JavaScript. We'll cover that in the next chapter.

Resources for Learning JavaScript

As JavaScript has become the *lingua franca* of the Web (not to mention mobile devices), a vast number of informative books, courses, and sites devoted to it have appeared. Here are a few that I recommend:

- If you're new to programming altogether, check out the interactive tutorial site Codecademy.[6]
- If you're coming from another language and want to get up and running with JavaScript as a language for scripting the browser, take the interactive jQuery Air courses on CodeSchool.[7]
- If you want a more formal introduction to the JavaScript language, absorb Marijn Haverbeke's *Eloquent JavaScript*.[8]
- If you're a JavaScript beginner who wants to level up and avoid common pitfalls, spend some time in the JavaScript Garden.[9]

Where to Turn for Help?

When pondering questions like "Should I use typeof or instanceof here?" steer clear of the dated W3Schools site (which, regrettably, tends to be favored by Google searches). Instead, head to the Mozilla Developer Network (MDN).[10]

5. https://github.com/raganwald/homoiconic/blob/master/2012/01/captain-obvious-on-javascript.md
6. http://www.codecademy.com/
7. http://www.codeschool.com/
8. http://eloquentjavascript.net/
9. http://javascriptgarden.info/
10. https://developer.mozilla.org/

The Mozilla Foundation (you may have heard of its browser, Firefox) is headed up by Brendan Eich, the creator of JavaScript. The foundation knows its stuff.

If you can't find your answer among MDN's pages, take your question to Stack Overflow.[11] The site has fostered an amazingly helpful developer community, and it's a safe bet that any coherent question tagged JavaScript will receive a punctual response.

Running the Code Examples

This book is a bit unusual, in that I discuss both client-side (browser) and server-side (Node.js) code. That reflects the uniquely portable nature of JavaScript. The central concepts apply to all JavaScript environments, but certain examples are aimed at one or the other.

Even if you have no interest in writing Node applications, I hope you'll follow along by running these code snippets locally. See *Running Code in Node.js*, on page xiv for directions.

Which Examples Are Runnable?

When you see a code snippet with a filename, that means it's self-contained and can be run without modification. Here's an example:

Preface/stringConstructor.js

```
console.log('str'.constructor.name);
```

The surrounding context should make it clear whether the code is runnable in the browser, in Node.js, or in both.

When a code snippet doesn't have a filename, that means it's not self-contained. It may be part of a larger example, or it may be a hypothetical. Here's an example:

```
var tenSeconds = 10 * 1e3;
setTimeout(launchSatellite, tenSeconds);
```

These examples are meant to be read, not run.

Running Code in Node.js

Node is very easy to install and use: just head to http://nodejs.org/, click Download, and run the Windows or OS X installer (or build from source on *nix). You can then run node from the command line to open a JavaScript REPL (analogous to Ruby's irb environment).

11. http://stackoverflow.com/

```
$ node
> Math.pow(5, 6)
15625
```

You can run a JavaScript file by giving its name as an argument to the `node` command.

```
$ echo "console.log(typeof NaN)" > foo.js
$ node foo.js
number
```

Running Code in the Browser

Every modern browser provides a nice little REPL that lets you run JavaScript code in the context of the current page. But for playing with multiline code examples, you're better off using a web sandbox like jsFiddle.[12]

With jsFiddle, you can enter JavaScript, HTML, and CSS, and then click Run (or press Ctrl+Enter) to see the result. (`console` output will go to your developer console.) You can bring in a framework like jQuery by choosing it in the left sidebar. And you can save your work, giving you a shareable URL.

Code Style in This Book

JavaScript has no official style guide, but maintaining a consistent style within a project is important. For this book, I've adopted the following (very common) conventions:

- Two-space indentation
- camelCase identifiers
- Semicolons at the end of every expression, except function definitions

More esoterically, I've adopted a special convention for indentation in a chain of function calls, based on a proposal by Reg Braithwaite. The rule is, essentially, that two function calls in a chain have the same indentation level if and only if they return the same object. So, for instance, I might write the following:

```
$('#container > ul li.inactive')
.slideUp();
```

jQuery's `slideUp` method returns the same object that it was called on. Thus, it isn't indented. By contrast:

```
var $paragraphClone = $('p:last')
  .clone();
```

12. http://jsfiddle.net/

Here, the clone method is indented because it returns a different object.

The advantage of this convention is that it clarifies what each function in a chain is returning. Here's a more complex example:

```
$('h1')
  .first()
  .addClass('first')
.end()
  .last()
  .addClass('last');
```

jQuery's first and last filter a set down to its first and last elements, while end undoes the last filter. So, end is unindented because it returns the same value as $('h1'). (last is allowed to occupy the same indentation level as first because the chain was reset.)

This approach to indentation is especially useful when we're doing functional programming, as we'll see in Chapter 4, *Flow Control with Async.js*, on page 47.

```
[1, 2, 3, 4, 5]
  .filter(function(int) { return int % 2 === 1; })
    .forEach(function(odd) { console.log(odd); })
```

A Word on altJS

A number of languages compile to JavaScript, making code easier to write. (You can find a fairly comprehensive list at http://altjs.org.) This book isn't about them. It's about writing the best JavaScript code we can without the use of a precompiler. I have nothing against altJS (see the next section), but I believe it's important to understand the underlying language.

Some altJS languages are aimed specifically at "taming" async callbacks by allowing them to be written in a more synchronous style. I've included an overview of these languages in Appendix 1, *Tools for Taming JavaScript*, on page 81.

CoffeeScript

It's no secret that I ♥ CoffeeScript, a beautiful and expressive language that compiles to JavaScript. I use it extensively in my day-to-day work at HubSpot. I've given talks on it at conferences like Railsconf and Øredev. And it was the subject of my first book, *CoffeeScript: Accelerated JavaScript Development*.[13]

13. http://pragprog.com/book/tbcoffee/coffeescript

But when I started writing the book you're reading now, I decided that doing it in CoffeeScript would needlessly limit its appeal. By and large, CoffeeScripters understand JavaScript perfectly well, whereas code like `square = (x) => x * x` might as well be hieroglyphics to JavaScript purists.[14]

So, if you're a CoffeeScripter, my apologies for the curly braces. Rest assured that the lessons you draw from this book will carry over to any altJS language.

Resources for This Book

This book has a website at http://pragprog.com/book/tbajs/async-javascript. There you can download the example code used in the book, get up-to-date information, and ask book-related questions in a friendly forum.

For more general JavaScript-related questions, I (again) heartily recommend Stack Overflow.[15] I have no affiliation with the site, but I am an avid fan with a proud 23,000 reputation points (and counting). Coherent, well-formatted questions there are almost always answered promptly.

Finally, if you want to contact me directly, you can reach me at trevorburnham@gmail.com or on Twitter: @trevorburnham. I'm always happy to hear from my readers.

Enough introduction. Let's get our async on!

14. However, maybe not for long: http://wiki.ecmascript.org/doku.php?id=harmony:arrow_function_syntax.
15. http://stackoverflow.com/

CHAPTER 1

Understanding JavaScript Events

Events. How *do* they work? Confusion about JavaScript's asynchronous event model is as old as JavaScript itself. Confusion leads to bugs, bugs lead to anger, and Yoda taught us the rest....

Yet at heart, JavaScript events are both conceptually elegant and practical. Once you've accepted the language's single-threaded design, it feels like a feature rather than a limitation. It means that your code is uninterruptible and that the events you schedule line up in an orderly fashion.

In this chapter, we'll take a tour of JavaScript's asynchronous mechanisms and dispel some common misconceptions. We'll see what setTimeout *really* does. Then we'll discuss handling errors in callbacks. Finally, we'll set up the main theme of this book: organizing async code for clarity and maintainability.

1.1 Scheduling Events

When we want to make a piece of code run in the future in JavaScript, we put it in a *callback*. A callback is just an ordinary function, except that it's passed to a function like setTimeout or bound as a property like document.onready. When a callback runs, we say that an event (e.g., the timeout elapsing or the document becoming ready) has fired.

Of course, the devil is in the details, even for something as seemingly simple as setTimeout. A common description of setTimeout goes something like this:

> Given a callback and a delay of *n* milliseconds, setTimeout runs that callback *n* milliseconds later.

But as we'll see in this section, and throughout this chapter, that description is seriously flawed. In most cases, it's only approximately true. In others, it's flat-out wrong. To truly understand setTimeout, we have to understand the JavaScript event model as a whole.

Now or Later?

To begin our exploration of setTimeout, let's look at a simple example of a situation that often mystifies new JavaScripters, especially those coming from multithreaded languages like Java and Ruby.

EventModel/loopWithTimeout.js

```
for (var i = 1; i <= 3; i++) {
  setTimeout(function(){ console.log(i); }, 0);
};
```

```
4
4
4
```

Most newcomers to the language would expect the loop to produce the output 1, 2, 3, or perhaps a juxtaposition of those three numbers as the three timeouts (each scheduled to go off in 0 milliseconds) race to fire first.

To understand why the output is 4, 4, 4 instead, there are three things you need to know.

- There's only one variable named i, scoped by the declaration var i (which, incidentally, scopes it not within the loop but within the closest function containing the loop).
- After the loop, i === 4, having been incremented until it failed the condition i <= 3.
- JavaScript event handlers don't run until the thread is free.

The first two concepts are in the realm of JavaScript 101, but the third comes as more of a surprise. When I first started using JavaScript, I didn't quite believe it. Java had trained me to fear that my code could be interrupted *at any moment*. A million potential edge cases filled me with anxiety as I wondered, "What if a rare event happened between these two lines of code?"

And then one day, that burden was lifted from me....

Blocking the Thread

This piece of code demolished my preconceptions about JavaScript events:

EventModel/loopBlockingTimeout.js

```
var start = new Date;
setTimeout(function(){
  var end = new Date;
  console.log('Time elapsed:', end - start, 'ms');
}, 500);
while (new Date - start < 1000) {};
```

In my multithreaded mind-set, I'd expected only 500ms to go by before the timed function ran. But that would have required the loop, designed to last a full second, to be interrupted. Instead, if you run the code, you'll get something like this:

```
Time elapsed: 1002ms
```

You'll probably get a slightly different number; setTimeout and setInterval are, alas, a lot less precise than you'd hope (see *Timing Functions*, on page 5). But it will definitely be at least 1000, because the setTimeout callback can't fire until the while loop has finished running.

So, if setTimeout isn't using another thread, then what is it doing?

Meet the Queue

When we call setTimeout, a timeout event is *queued*. Then execution continues: the line after the setTimeout call runs, and then the line after that, and so on, until there are no lines left. Only then does the JavaScript virtual machine ask, "What's on the queue?"

If there's at least one event on the queue that's eligible to "fire" (like a 500ms timeout that was set 1000ms ago), the VM will pick one and call its *handler* (e.g., the function we passed in to setTimeout). When the handler returns, we go back to the queue.

Input events work the same way: when a user clicks a DOM element with a click handler attached, a click event is queued. But the handler won't be executed until all currently running code has finished (and, potentially, until after other events have had their turn). That's why web pages that use JavaScript imprudently tend to become unresponsive.

You might sometimes hear the term *event loop* used to describe how the queue works. It's as if your code is being run from a loop that looks like this:

```
runYourScript();
while (atLeastOneEventIsQueued) {
  fireNextQueuedEvent();
};
```

One implication of this is that each event that fires will be at the root of the stack trace. We'll learn more about that in Section 1.4, *Handling Async Errors*, on page 12.

The ease of event scheduling in JavaScript is one of the language's most powerful features. Async functions like setTimeout make delayed execution simple, without spawning threads. JavaScript code can never be interrupted,

because events can be *queued* only while code is running; they can't *fire* until it's done.

In the next section, we'll take a closer look at the building blocks of async JavaScript.

1.2 Types of Async Functions

Each JavaScript environment comes with its own set of async functions. Some, like setTimeout and setInterval, are ubiquitous. Others are unique to certain browsers or server-side frameworks. The async functions provided by the JavaScript environment generally fall into two categories: I/O and timing. These are the basic building blocks that you'll use to define complex async behaviors in your applications.

I/O Functions

Node.js wasn't created so that people could run JavaScript on the server. It was created because Ryan Dahl wanted an event-driven server framework built on a high-level language. JavaScript just happened to be the right language for the job. Why? Because the language is a perfect fit for *nonblocking I/O*.

In other languages, it's tempting to "block" your application (typically by running a loop) until an I/O request completes. In JavaScript, that approach isn't even possible. A loop like this will run forever:

```
var ajaxRequest = new XMLHttpRequest;
ajaxRequest.open('GET', url);
ajaxRequest.send(null);
while (ajaxRequest.readyState === XMLHttpRequest.UNSENT) {
  // readyState can't change until the loop returns
};
```

Instead, you need to attach a handler and return to the event queue.

```
var ajaxRequest = new XMLHttpRequest;
ajaxRequest.open('GET', url);
ajaxRequest.send(null);
ajaxRequest.onreadystatechange = function() {
  // ...
};
```

That's how it goes. Whether you're waiting for a keypress from the user or a batch of data from a remote server, you need to define a callback—unless your JavaScript environment gives you a synchronous I/O function that does the blocking for you.

In the browser, Ajax methods have an async option that can (but *should never, ever*) be set to false, bringing the entire browser pane to a halt until a response is received. In Node.js, synchronous API methods are clearly indicated with names like fs.readFileSync. These are convenient when writing short scripts but should be avoided when writing applications that need to handle multiple requests or operations in parallel. And these days, which applications don't?

Some I/O functions have both synchronous and async effects. For instance, when you manipulate the DOM in a modern browser, the changes are immediate from your script's perspective but aren't rendered until you return to the event queue. That prevents the DOM from being rendered in an inconsistent state. You can see a simple demonstration of this at http://jsfiddle.net/TrevorBurnham/SNBYV/.

Is console.log Async?

WebKit's console.log has surprised many a developer by behaving asynchronously. In Chrome or Safari, this code will log {foo: bar}:

EventModel/log.js

```
var obj = {};
console.log(obj);
obj.foo = 'bar';
```

How does this happen? Rather than taking a snapshot of the object immediately, WebKit's console.log stores a reference to the object and then takes a snapshot when the code returns to the event queue.

Node's console.log, on the other hand, is strictly synchronous, so the same code yields the output {}.

Adapting to nonblocking I/O is one of the biggest hurdles that newcomers to JavaScript face, but it's also one of the language's key strengths. It makes writing efficient, event-based code feel natural.

Timing Functions

We've seen how async functions are a natural fit for I/O operations, but sometimes we want asynchronicity for its own sake. That is, we want to make a function run at some point in the future, perhaps for an animation or a simulation. The well-known functions for time-based events are setTimeout and its repeating cousin, setInterval.

Unfortunately, these well-known timer functions have their flaws. As we saw in *Blocking the Thread*, on page 2, one of those flaws is insurmountable: no JavaScript timing function can cause code to run while other code is running

in the same JavaScript process. But even with that limitation in mind, setTimeout and setInterval are alarmingly imprecise. Here's a demonstration:

EventModel/fireCount.js

```
var fireCount = 0;
var start = new Date;
var timer = setInterval(function() {
  if (new Date - start > 1000) {
    clearInterval(timer);
    console.log(fireCount);
    return;
  }
  fireCount++;
}, 0);
```

When we schedule an event with setInterval and a 0ms delay, it should run as often as possible, right? So, in a modern browser powered by a speedy Intel i7 processor, at what rate does the event fire?

About 200/sec. That's across Chrome, Safari, and Firefox. Under Node, the event fired at a rate of about 1000/sec. (Using setTimeout to schedule each iteration yields similar results.) By comparison, replacing setInterval with a simple while loop brings that rate to 4,000,000/sec in Chrome and 5,000,000/sec in Node!

What's going on? It turns out that setTimeout and setInterval are slow by design. In fact, the HTML spec (which all major browsers respect) mandates a *minimum* timeout/interval of 4ms![1]

So, what do you do when you need finer-grained timing? Some runtimes offer alternatives.

- In Node, process.nextTick lets you schedule an event to fire ASAP. On my system, process.nextTick events fire at a rate of over 100,000/sec.
- Modern browsers (including IE9+) have a requestAnimationFrame function, which serves a dual purpose: it allows JavaScript animations to run at 60+ frames/sec, and it conserves CPU cycles by preventing those animations from running in background tabs. In the latest Chrome, you can even get submillisecond precision.[2]

Though they're the bread and butter of async JavaScript, never forget that setTimeout and setInterval are imprecise tools. When you just want to produce a

1. http://www.whatwg.org/specs/web-apps/current-work/multipage/timers.html#dom-windowtimers-settimeout
2. http://updates.html5rocks.com/2012/05/requestAnimationFrame-API-now-with-sub-millisecond-precision

short delay in Node, use process.nextTick. In the browser, try to use a shim[3] that defers to requestAnimationFrame in browsers that support it and falls back on setTimeout in those that don't.

That concludes our brief overview of basic async functions in JavaScript. But how do we tell when a function is async anyway? In the next section, we'll ponder that question as we write our own async functions.

1.3 Writing Async Functions

Every async function in JavaScript is built on some other async function(s). It's async functions all the way down (to native code)!

The converse is also true: any function that uses an async function has to provide the result of that operation in an async way. As we learned from *Blocking the Thread*, on page 2, JavaScript doesn't provide a mechanism for preventing a function from returning until an async operation has finished. In fact, until the function returns, no async events will fire.

In this section, we'll look at some common patterns in async function design. We'll see that functions can be mercurial, deciding to be async only some of the time. But first, let's define exactly what an async function is.

When Is a Function Async?

The term *async function* is a bit of a misnomer: if you call a function, your program simply won't continue until that function returns. What JavaScripters mean when they call a function "async" is that it can cause another function (called a *callback* when it's passed as an argument to the function) to run later, from the event queue. So, an async function that takes a callback will never fail this test:

```
var functionHasReturned = false;
asyncFunction(function() {
  console.assert(functionHasReturned);
});
functionHasReturned = true;
```

Another term for async functions is *nonblocking*. The term emphasizes how speedy they are: a query made with an async MySQL driver may take an hour, but the function that sent the query will return in a matter of microseconds —a boon to web servers that need to quickly process a high volume of incoming requests.

3. http://paulirish.com/2011/requestanimationframe-for-smart-animating/

Typically, functions that take a callback take it as their last argument. (Regrettably, the venerable setTimeout and setInterval are exceptions to this convention.) But some async functions take callbacks indirectly, by returning a Promise or using PubSub. We'll learn about those patterns later in the book.

Unfortunately, the only way to be sure whether a function is async or not is to inspect its source code. Some functions that are synchronous have an API that looks async, either because they might become async in the future or because callbacks provide a convenient way to return multiple arguments. When in doubt, don't depend on a function being async.

Sometimes-Async Functions

There are functions that are async sometimes but not at other times. For instance, jQuery's eponymous function (typically aliased as $) can be used to delay a function until the DOM has finished loading. But if the DOM has already finished loading, there's no delay; its callback fires immediately.

This unpredictable behavior can get you in a lot of trouble if you aren't careful. One mistake I've seen (and made myself) is assuming that $ will run a function after other scripts on the page have loaded.

```
// application.js
$(function() {
  utils.log('Ready');
});

// utils.js
window.utils = {
  log: function() {
    if (window.console) console.log.apply(console, arguments);
  }
};
```

```
<script src="application.js"></script>
<script src="util.js"></script>
```

This code works fine—unless the browser loads the page from the cache, making the DOM ready before the script runs. When that happens, the callback passed to $ runs before utils.log is set, causing an error. (We could avoid this situation by taking a more modern approach to client-side dependency management. See Chapter 6, *Async Script Loading*, on page 69.)

Let's look at another example.

sample content of Async JavaScript: Build More Responsive Apps with Less Code (Pragmatic Express)

- *Great God Aâ€™mighty! The Dixie Hummingbirds: Celebrating the Rise of Soul Gospel Music pdf, azw (kindle)*
- Historia de las alcobas for free
- Desperate Sons: Samuel Adams, Patrick Henry, John Hancock, and the Secret Bands of Radicals Who Led the Colonies to War for free
- read online The Cookbook Library: Four Centuries of the Cooks, Writers, and Recipes That Made the Modern Cookbook pdf

- http://creativebeard.ru/freebooks/Great-God-A---mighty--The-Dixie-Hummingbirds--Celebrating-the-Rise-of-Soul-Gospel-Music.pdf
- http://diy-chirol.com/lib/Historia-de-las-alcobas.pdf
- http://www.shreesaiexport.com/library/Santa-Claus--A-Biography.pdf
- http://studystrategically.com/freebooks/The-Cookbook-Library--Four-Centuries-of-the-Cooks--Writers--and-Recipes-That-Made-the-Modern-Cookbook.pdf