Covers ES2020

JavaScript®
The New Toys

T. J. Crowder

Table of Contents

List of Tables

List of Illustrations

Chapter 18

FIGURE 18-1

JavaScript®

THE NEW TOYS

T.J. Crowder

INTRODUCTION

THIS BOOK IS FOR ANY JAVASCRIPT (OR TYPESCRIPT) programmer who wants to get up to speed on the latest features added to JavaScript in the last few years, and who wants to know how to stay informed as the language continues to evolve and grow. You can find nearly all of the information in this book *somewhere* on the web if you look hard enough and you're cautious about what sites you trust; this book provides all of the technical details in one place, along with showing you how to keep track of the changes as they continue to be made. You can continue to stay up-to-date via the book's website at https://thenewtoys.dev, where I continue covering new features as they're added.

WHAT DOES THIS BOOK COVER?

Here's a glance at what's in each chapter:

Chapter 1: The New Toys in ES2015–ES2020, and Beyond — begins with introducing various players in the JavaScript world and some important terminology. Then it describes the definition of “The New Toys” for the purposes of the book; covers the way new features are added to JavaScript, how that process is managed, and by whom; and explains how to follow and participate in that process. It finishes by introducing some tools for using the new toys in older environments (or using the very newest toys in current environments).

Chapter 2: Block-Scoped Declarations: let and const — covers the new declaration keywords `let` and `const` and the new scoping concepts they support, including in-depth coverage of scoping in loops, particularly the new handling in `for` loops.

Chapter 3: New Function Features — covers the variety of new features related to functions: arrow functions, default parameter values, rest parameters, the `name` property, and various syntax improvements.

Chapter 4: Classes — covers the new `class` feature: the basics, what it is and what it isn't, subclassing, `super`, subclassing built-ins like `Array` and `Error`, and the `new.target` feature. Chapter 4 leaves coverage of private fields and other features making their way through the proposals process to Chapter 18.

Chapter 5: New Object Features — covers computed property names, shorthand properties, getting and setting an object's prototype, the new Symbol type and how it relates to objects, method syntax, property

order, property spread syntax, and a host of new object functions.

Chapter 6: Iterables, Iterators, for-of, Iterable Spread, Generators — covers iteration, a powerful new tool for collections and lists, and generators, a powerful new way to interact with functions.

Chapter 7: Destructuring — covers this important new syntax and how to use it to extract data from objects and from arrays and other iterables, with defaults, deep picking, and more.

Chapter 8: Promises — dives deep into this important new tool for coping with asynchronous processes.

Chapter 9: Asynchronous Functions, Iterators, and Generators — goes into detail about the new `async`/`await` syntax that lets you use familiar logical flow structures with asynchronous code, as well as how asynchronous iterators and generators work and the new `for-await-of` loop.

Chapter 10: Templates, Tag Functions, and New String Features — describes template literal syntax, tag functions, and lots of new string features like better Unicode support, updates to familiar methods, and lots of new methods.

Chapter 11: New Array Features, Typed Arrays — covers the wide range of new array methods, various other array updates, typed arrays such as `Int32Array`, and advanced features for interacting with typed array data.

Chapter 12: Maps and Sets — teaches you all about the new keyed collections `Map` and `Set`, and also their "weak" cousins `WeakMap` and `WeakSet`.

Chapter 13: Modules — is a deep dive into this exciting and powerful way to organize your code.

Chapter 14: Reflection—Reflect and Proxy — covers the powerful new dynamic metaprogramming features of the `Reflect` and `Proxy` objects and how they relate to one another.

Chapter 15: Regular Expression Updates — describes all of the updates to regular expressions that have been done the past several years such as new flags, named capture groups, lookbehind assertions, and new Unicode features.

Chapter 16: Shared Memory — covers the complex and potentially difficult area of sharing memory across threads in a JavaScript program, including covering `SharedArrayBuffer` and the `Atomics` object as well as fundamental concepts and notes on pitfalls.

Chapter 17: Miscellany — covers a wide range of things that didn't quite fit in elsewhere: BigInt, the new integer literal syntaxes (binary, new octal), optional `catch` bindings, new math methods, the exponentiation operator, various additions to the `Math` object (no pun!), tail call optimization, nullish coalescing, optional chaining, and “Annex B” (browser-only) features defined for compatibility reasons.

Chapter 18: Upcoming Class Features — takes us a bit into the future, describing enhancements to classes that haven't quite been finished yet but which are far along in the process: public field declarations, private fields, and private methods.

Chapter 19: A Look Ahead ... — winds up with looking forward to some further enhancements currently in the pipeline: top-level `await`, WeakRefs and cleanup callbacks, RegExp match indices,

`Atomics.asyncWait`, a couple of new syntax features, legacy RegExp features, and various upcoming standard library additions.

Appendix: Fantastic Features and Where to Find Them — *(with apologies to J.K. Rowling)* provides lists of the new toys and tells you which chapter covers each of them. The lists are: Features in Alphabetical Order; New Fundamentals; New Syntax, Keywords, Operators, Loops, and Similar; New Literal Forms; Standard Library Additions and Changes; and Miscellaneous.

WHO SHOULD READ THIS BOOK

You should read this book if:

- You have at least a basic understanding of JavaScript, and
- You want to learn the new features added in the last several years.

This isn't an academic book just for experts. It's a pragmatic book for everyday JavaScript programmers.

Almost anyone picking up this book is already going to know *some* of what's in it, and almost no one picking up this book will already know *all* of it. Maybe you're already clear on the basics of `let` and `const`, but you don't quite get `async` functions yet. Maybe promises are old hat to you, but you saw some syntax you didn't recognize in some modern code you ran across. You'll find all of it from ES2015 through ES2020 (and beyond) covered here.

HOW TO USE THIS BOOK

Read Chapter 1. Chapter 1 defines a lot of terms I use throughout the rest of the book. Skipping Chapter 1 will likely trip you up.

After that, you have a choice to make: read the chapters in order, or jump around?

I've laid out the chapters in the order I have for a reason; and I build on the content of earlier chapters in later chapters. For instance, you'll learn about promises in Chapter 8, which is important to understanding `async` functions in Chapter 9. Naturally, I suggest reading them in the order I've arranged them. But I bet you're a smart person who knows their own mind; if you jump around instead, you'll be just fine.

I do suggest that you read—or at least skim—all of the chapters (with the possible exception of Chapter 16; more on that in a minute). Even if you think you know a feature, there's probably something here that you don't know, or only think you know. For instance: maybe you're planning to skip Chapter 2 because you already know all there is to know about `let` and `const`. Maybe you even know why

```
for (let i = 0; i < 10; ++i) { /*…*/
    setTimeout(() => console.log(i));
}
```

creates ten *different* variables called `i`, and why using

```
let a = "ay";
var b = "bee";
```

at global scope creates a `window.b` property but doesn't create a `window.a` property. Even if you do, I'd skim Chapter 2 just to be sure there aren't other wrinkles you haven't picked up.

Chapter 16 is a bit of a special case: it's about sharing memory between threads. Most JavaScript programmers don't need to share memory between threads. *Some* of you will, and that's why Chapter 16 is there, but most won't and if you're in that category, just tuck away the fact that *if* you think you need shared memory at some point in the future you can come back and learn about it in Chapter 16. That's just fine.

Beyond all that: run the examples, experiment with them, and above all enjoy yourself.

HOW TO CONTACT THE AUTHOR

I'd love to hear your feedback. Please feel free to reach out:

- If you think you've found an error, please write to errata@wiley.com.
- You can download the examples and listings at https://thenewtoys.dev/bookcode or https://www.wiley.com/go/javascript-newtoys.
- You can ask me questions and/or talk with other readers on https://thenewtoys.dev.
- You can reach me on Twitter at `@tjcrowder`.
- I'm usually hanging out in the JavaScript tag on https://stackoverflow.com/.

Enjoy!

1
The New Toys in ES2015–ES2020, and Beyond

WHAT'S IN THIS CHAPTER?

- Definitions, who's who, and terminology
- Explanation of JavaScript versions (ES6? ES2020?)
- What are the “new toys”?
- The process driving new JavaScript features
- Tools for using next-generation JavaScript

CODE DOWNLOADS FOR THIS CHAPTER

You can download the code for this chapter at https://thenewtoys.dev/bookcode or https://www.wiley.com/go/javascript-newtoys.

JavaScript has changed a lot in the last few years.

If you were an active JavaScript developer in the 2000s, for a while there you could be forgiven for thinking JavaScript was standing still. After the 3rd Edition specification in December 1999, developers were waiting fully 10 years for the next edition of the specification. From the outside, it seemed like nothing was happening. In fact, lots of work

was being done; it just wasn't finding its way into the official specification and multiple JavaScript engines. We could (but won't) spend an entire chapter—if not a book—on what various different groups in important positions vis-à-vis JavaScript were doing and how they couldn't agree on a common path forward for some time. The key thing is that they ultimately *did* agree on a way forward, at a fateful meeting in Oslo in July 2008 after much advance negotiation. That common ground, which Brendan Eich (creator of JavaScript) later called Harmony, paved the way for the 5th Edition specification in December 2009 (the 4th Edition was never completed), and laid the basis for ongoing progress.

And my, how things have progressed.

In this chapter, you'll get an overview of the new features since 2009 (which the rest of the book covers in detail). You'll learn who's in charge of moving JavaScript forward, what process is now used to do so, and how you can keep track of what's coming and get involved if you like. You'll learn about tools you can use to write modern JavaScript today, even if you have to target environments that haven't kept up.

DEFINITIONS, WHO'S WHO, AND TERMINOLOGY

To talk about what's going on with JavaScript, we need to define some names and common terminology.

Ecma? ECMAScript? TC39?

What we think of as “JavaScript” is standardized as “ECMAScript” by Ecma International,[1] a standards organization responsible for multiple computing standards. The ECMAScript standard is ECMA-262. The people in

charge of the standard are members of Ecma International Technical Committee 39 (“TC39”), charged with “Standardization of the general purpose, cross platform, vendor-neutral programming language ECMAScript. This includes the language syntax, semantics, and libraries and complementary technologies that support the language.”[2] They manage other standards as well, such as the JSON Syntax Specification (ECMA-404), and notably the ECMAScript Internationalization API Specification (ECMA-402).

In this book and in common usage, JavaScript is ECMAScript and vice versa. Sometimes, particularly during the decade of different groups doing different things, “JavaScript” was used to specifically mean the language Mozilla was developing (which had several features that either never made it into ECMAScript, or changed markedly before they did), but since Harmony that usage is increasingly outdated.

ES6? ES7? ES2015? ES2020?

Having all these various abbreviations can be confusing, not least because some have edition numbers but others have years. This section explains what they are and why there are two kinds of them.

Up through the 5th Edition, TC39 referred to versions of the specification via their edition number. The full title of the 5th Edition spec is:

> Standard ECMA-262
>
> 5th Edition / December 2009
>
> ECMAScript Language Specification

Since “ECMAScript 5th Edition” is a bit of a mouthful, saying “ES5” was the natural thing to do.

Starting with the 6th Edition in 2015, TC39 adopted a continual improvement process where the specification is a living editor's draft[3] with annual snapshots. (More about that later in this chapter.) When they did that, they added the year to the language name:

> Standard ECMA-262
>
> 6th Edition / June 2015
>
> ECMAScript® **2015** Language Specification

So the ECMAScript 6th Edition standard (“ES6”) defines ECMAScript 2015, or “ES2015” for short. Prior to publication, “ES6” became a buzzword of its own and is still in common usage. (Unfortunately, it's often used inaccurately, referring not just to features from ES2015, but also to features that came afterward in ES2016, ES2017, and so on.)

That's why there are two styles, the style using the edition (ES6, ES7, …) and the style using the year (ES2015, ES2016, …). Which you use is up to you. ES6 is ES2015 (or sometimes, incorrectly, ES2015+), ES7 is ES2016, ES8 is ES2017, and so on through (as of this book's release) ES11, which is ES2020. You'll also see “ESnext” or “ES.next,” which are sometimes used to refer to upcoming changes.

In this book, I use what I see as the emerging consensus: the old style for ES5 and earlier, and the new style for ES2015 and later.

All of that said, although I will usually call out the specific edition where a feature was introduced, the fact that `Array.prototype.includes` is from ES2016 and `Object.values` is from ES2017 doesn't really matter very much. What matters more is what's actually supported in your target environments and whether you need to either refrain from using a specific feature, or transpile and/or polyfill it. (More

on transpiling and polyfilling later in the “Using Today's Toys in Yesterday's Environments, and Tomorrow's Toys Today” section.)

JavaScript “Engines,” Browsers, and Others

In this book I'll use the term “JavaScript engine” to refer to the software component that runs JavaScript code. A JavaScript engine must know how to:

- Parse JavaScript,
- Either interpret it or compile it to machine code (or both), and
- Run the result within an environment that works as described by the specification.

JavaScript engines are also sometimes called virtual machines, or VMs for short.

One usual place you find JavaScript engines is in web browsers, of course:

- Google's Chrome browser uses their V8 engine (also used in Chromium, Opera, and Microsoft Edge v79 and later), except on iOS (more on that in a bit).
- Apple's Safari browser (for Mac OS and for iOS) uses their JavaScriptCore engine.
- Mozilla's Firefox uses their SpiderMonkey engine, except on iOS.
- Microsoft's Internet Explorer uses their JScript engine, which is increasingly out of date as it only gets security fixes.
- Microsoft Edge v44 and earlier (“Legacy Edge”) uses Microsoft's Chakra engine. In January 2020, Edge v79 was released, which is based on the Chromium project

and uses the V8 engine, except on iOS. (The version number jumped from 44 to 79 to align with Chromium.) Chakra is still used in various products that use the Microsoft WebView control, such as Microsoft Office JavaScript add-ins, though it may be replaced at some stage. (WebView2, using Chromium Edge, is in developer preview as of early 2020.)

Chrome, Firefox, Edge, and other browsers running on Apple's iOS operating system for iPad and iPhone can't currently use their own JavaScript engines, because to compile and run JavaScript (rather than just interpreting it), they have to allocate executable memory, and only Apple's own iOS apps are allowed to do that, not ones from other vendors. So Chrome and Firefox (and others) have to use Apple's JavaScriptCore instead on iOS even though they use their own engine on desktop and Android. (At least, that's true for now; the V8 team added an "interpreter only" mode to V8 in 2019, which means Chrome and others using V8 could use that mode on iOS, since it doesn't have to use executable memory.) In this book, if I say something is "supported by Chrome" or "supported by Firefox," I'm referring to the non-iOS versions using V8 or SpiderMonkey, respectively.

JavaScript engines are also used in desktop applications (Electron,[4] React Native,[5] and others), web servers and other kinds of servers (often using Node.js[6]), non-web applications, embedded apps—just about everywhere.

WHAT ARE THE "NEW TOYS"?

For this book, the "new toys" are the new features added to JavaScript in ES2015 through ES2020 (and previewing some that are coming soon). The language has come a *long* way in those six updates. The following is a general

overview. (Appendix A has a more complete list of changes.) Some of the terms on the list may be unfamiliar; don't worry about that, you'll learn them across the course of the book.

- *Opt-in block scope* (`let`, `const`): Narrower scoping for variables, clever handling of `for` loop body scope, “variables” whose value cannot change (`const`)
- *“Arrow” functions*: Lightweight, concise functions that are particularly useful for callbacks since they close over `this` rather than having their own `this` value that's set when they're called
- *Improvements to function parameters*: Default values; parameter destructuring, “rest” parameters, trailing commas
- *Iterable objects*: Well-defined semantics for creating and consuming iterable objects (such as arrays and strings), in-language iteration constructs (`for-of`, `for-await-of`); generator functions for generating sequences that can be iterated (including asynchronous sequences)
- *“Spread” syntax*: Spreading out array (or other iterable) entries into new arrays, spreading out object properties into new objects, and spreading out iterable entries into discrete function arguments; particularly useful for functional programming, or anywhere immutable structures are used
- *“Rest” syntax*: Gathering together the “rest” of an object's properties, an iterable's values, or a function's arguments into an object or array
- *Other syntax improvements*: Allowing a trailing comma in the argument list when calling a function; omitting unused identifiers in `catch` clauses; new-style octal

literals; binary literals; separator characters in numeric literals; and more

- *Destructuring*: Picking out the values from arrays/objects in a concise way that mirrors object and array literal syntax
- `class`: Markedly simpler, declarative syntax for creating constructor functions and associated prototype objects, while preserving JavaScript's inherent prototypical nature
- *Asynchronous programming improvements*: Promises, `async` functions and `await`; these markedly decrease “callback hell”
- *Object literal improvements*: Computed property names, shorthand properties, method syntax, trailing commas after property definitions
- *Template literals*: A simple, declarative way to create strings with dynamic content, and go beyond strings with tagged template functions
- *Typed arrays*: Low-level true arrays for using native APIs (and more)
- *Shared memory*: The ability to genuinely share memory between JavaScript threads (including inter-thread coordination primitives)
- *Unicode string improvements*: Unicode code point escape sequences; support for accessing code points instead of code units
- *Regular Expression improvements*: Lookbehind assertions; named capture groups; capture indices; Unicode property escapes; Unicode case insensitivity
- *Maps*: Key/value collections where the keys don't have to be strings

- *Sets*: Collections of unique values with well-defined semantics
- *WeakMap, WeakSet, and WeakRef*: Built-ins for holding only weak references to objects (allowing them to be garbage collected)
- *Standard library additions*: New methods on `Object`, `Array`, `Array.prototype`, `String`, `String.prototype`, `Math`, and others
- *Support for dynamic metaprogramming*: `Proxy` and `Reflect`
- *Symbols*: Guaranteed-unique values (particularly useful for unique property names)
- *BigInt*: Arbitrary precision integers
- Many, many others

All these new features, in particular the new syntax, could be overwhelming. Don't be concerned! There's no need to adopt new features until/unless you're ready to and have need of them. One of the key principles TC39 adheres to is "Don't break the web." That means that JavaScript must remain "web compatible"—that is, compatible with the huge body of code that already exists in the world today.[7] If you don't need a new feature, or you don't like a new feature, you don't have to use it. Your old way of doing whatever that feature does will always continue to work. But in many cases, you're likely to find there's a compelling reason for the new feature, in particular new syntax features: they make something simpler and less error-prone to write and understand, or—in the case of `Proxy` and `WeakMap`/ `WeakSet`, shared memory, and others—they enable things that mostly couldn't be done without them.

For space reasons, this book only covers the new toys in the JavaScript specification itself, ECMA-262. But there are