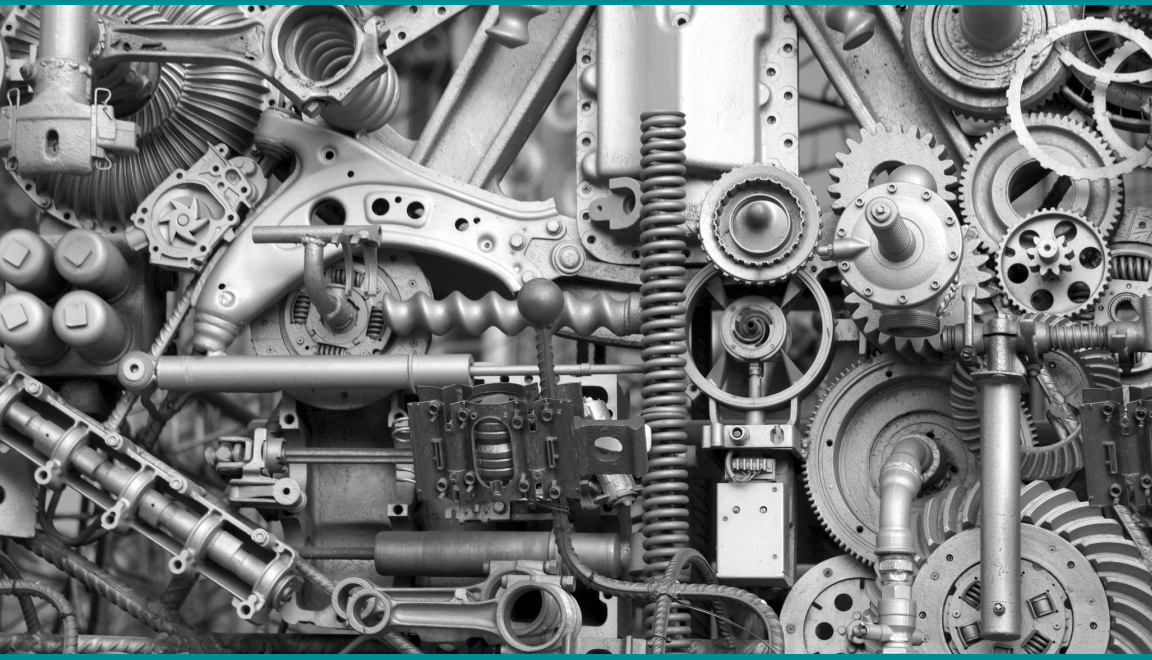
O'REILLY®

2nd Edition

JS.Next: A Manager's Guide



Aaron Frost

SECOND EDITION

JS.Next: A Manager's Guide

Aaron Frost

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

JS.Next: A Manager's Guide
by Aaron Frost

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* .

Editor: Meg Foley
Production Editor: Matthew Hacker
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

May 2013: First Edition
April 2015: Second Edition

Revision History for the Second Edition
2015-03-27: First Release

978-1-491-92019-0

[LSI]

Table of Contents

Preface

Writing this book was extremely fun and proved to be a helpful exercise. Researching and describing each topic was a process that lasted about two and a half years. When Simon (@simonstl) and Mike (@mikeloukides) approached me about the idea, I wasn't sure that I would be able to deliver what they were asking for. Their vision was to explain ECMAScript 6 in a way that non-developers would understand it. Additionally, they wanted to help everyone understand the importance of adopting the new syntax into their current projects, as opposed to waiting years for certain parts of the Web to catch up. Much like steering a donkey with a carrot on a stick, Simon and Mike helped steer my efforts. Without them, much of what was written wouldn't be. I appreciate all of their mentoring and guidance.

Once I finally understood the direction in which we needed to go, I simply needed time. A special thanks goes to my wonderfully understanding wife (Sarai) and to my four children (Naomi, Joceline, Ryan, and Owen). Family life is already a lot of work. Having a husband/dad that is busy writing a book only adds to it. Each of them helped me race to get this finished in time for FluentConf 2015. Thank you.

Everyone made a very serious effort to disguise how sleep deprived I was when finishing this. A special thanks to the inventors/makers/distributors of Diet Mountain Dew and Mio Energy Drops. While the ideas are my own, many of the words used to spell out my ideas were heavily fueled by caffeine from these sources.

To my friends and colleagues who helped out, you know who you are, thank you! Chad "the knife" (@chadmaughan) and Tom

(@tvalletta), thank you for mentoring me and helping my solidify some of the ideas expressed here. Mom (@marlli53), Neal (@Neal-Midgley), Steveo (@steveolyo), Ted "the head" (@jsbalrog), and Tayler (@taylersumms). These are my people who read the pages when they were fresh off the press. Each of them took part in ensuring the quality of the text.

And a very special thanks to each of the members of the TC39. This book is only possible because of their efforts. While the JavaScript community eagerly await the ES6 updates, the members of the TC39 remain focused as they continue their daily effort of solidifying the ES6 specification. I feel lucky that I have been able to work directly with a handful of them. While I want to thank each of them, the following are the members who have directly had a hand in helping my efforts: Dave Herman (@littlecalculist), Allen Wirfs-Brock (@awbjs), Brendan Eich (@BrendanEich), Rafael Weinstein (@rzweinstein), Rick Waldron (@rwaldron), and Alex Russell (@slightlylate). *Note to whomever is running the @FakeAlexRussell account: you're brilliant!*

CHAPTER 1

You Can't Afford to Avoid ES6

ECMAScript 6 is a big deal. ECMAScript, everyone's favorite scripting API, hasn't had an update this significant since it was initially formalized. Some people may feel overwhelmed as they browse through the impressive list of new features. Each was carefully considered, discussed at length, and selected for adoption into the official API. Ours is the task of rolling out these new features, bringing ES6 to our teams and to our projects.

But exactly how are we do that? How do we take these new features and concepts and infuse them into the brains of our developers? How can we inject this new power into our current projects? Just as important, and possibly more so, is when should we do this?

You may feel that you can't afford to implement these features in your world. Some of you may prove yourselves to be extremely talented as creating reasons why you can't afford it at this time. I am here to tell you that you can't afford not to. As you read on, consider yourself warned: the content that follows is highly controversial.

While the main audience of this book is composed of development management, I am sure that a handful of developers will find their way here as well. If you are a developer, welcome! If you are in management, I hope that you enjoy the ride.

The remaining sections in this chapter will cover various reasons for adopting ES6 into your current and future projects. Although not a complete list of reasons, it should help show that in the long run, it will cost more to avoid ES6 than to embrace it.

Innovation Debt

When talking about debt in software development, most people will talk about technical debt. Technical debt reflects the imperfect and sometimes dangerous state of your code and processes. As deadlines approach, optional features and maintenance time can get cut from the schedule. Without enough time to properly maintain code and processes, you will inevitably have to watch as your technical debt grows. Increased technical debt is something that all teams, except perhaps those with infinite resources, face regularly.

There is, however, another type of development debt that is constantly accruing: innovation debt. The term comes from Peter Bell, an amazing author, speaker, and innovator. Peter provides a concise definition:

> Innovation debt is the cost that companies incur when they don't invest in their developers.

Like technical debt, innovation debt can spiral out of control if left unchecked, possibly threatening the existence of the company.

If you have time, please visit Peter's blog and read his full explanation of the definition.

Imagine your CEO tells you that she needs a new and modern app (that your team doesn't know how to build), built with very modern tools (that your team doesn't know how to use), and deployed using very modern build tools (that your team doesn't know how to configure). What would you say? Given the state of your team, what are the odds of getting it done right?

Consider your current technology stack, code base, feature set, and business goals with their new target feature set. Now think of all the training and practice that your team will need before you can create those target features. Consider your competitors' feature set and

how fast they are gaining on you, or how fast you are falling behind them.

Innovation debt is the cost you have to ante up before you can begin innovating again. Many teams keep their innovation debt manageable and may be able to train up a few of their current members to help bring the team back on track. However, some teams have accrued so much innovation debt that they have to hire new employees, with a new and different skill set than their current team. They hope that these new employees can pull everyone else up to speed. In extreme cases, such teams may even plan for these new team members to replace their current team. As innovation debt increases, the ability to avoid extreme decisions decreases.

So how do you pay back innovation debt? Better yet, how can you prevent innovation debt from increasing on your teams?

The answer is simple: teach your teams what they need to know so that they can innovate, and then let them practice it in the workplace.

Make time for your team members to learn and practice these new skills. Trying to pay off large lumps all at once can be too costly in the short term. Taking multiple iterations and cycles to train your teams is difficult to sell to your customers, whereas smaller and more consistent bites can be much easier to swallow.

While the "how to pay back" may seem most important, I think that the "when to pay back" is even more important. The "when" is now. Starting today, pay back small amounts of innovation debt on a regular basis. At least once per quarter we should all be taking strides toward paying back innovation debt.

Let's bring this back to ES6 now. Dropping ES6 into your current project can seem like a tough challenge, but it may prove to be your strongest ally. The ES6 release is not a minor upgrade to the language. It is a significant upgrade and improvement. And the new constructs and syntax in ES6 will enable your teams to make more progress faster than they ever have. Here are some tips on how you can help your team to catch up on ES6:

They will need time to learn it, even those who are already skilled JavaScript developers. If you don't dedicate enough time to learning and training on ES6, your teams will struggle. Create goals around learning ES5/6 and other modern JS libraries/ frameworks. Projects

like Angular, Grunt, and IOjs are a few that I am partial to. An ambitious few may even jump into server-side JavaScript, such as IO.js and Nashorn. Make sure your teams have the resources they need to learn the latest technologies. Then ask them to implement those technologies to help reduce the technical debt. Lead from in front instead of from behind. Help lead the way by regularly scheduling team training. Even if they are simple, informal meetups, make time for the team to sit down and talk about what the next steps are.

Do what you can to create a healthy culture on your team, one that harbors innovation. For example, at a past job, we ordered 100 Angular iron-on badges. We handed those out to engineers who released an Angular app into production. At our internal monthly JavaScript meetup, we ceremoniously handed out the Angular badges to those who released their app since our last meetup. We were surprised by the results. Many of those badge winners were on teams that we never expected to adopt such modern and fun frameworks. It was encouraging to see the team members innovate and learn something new. Nowadays, you can spot these badges all over the building, each one a reminder of our goal to continually innovate.

Direction of the Industry

With zero exceptions, all of today's most popular browsers are working to provide support for ES6 (see the ES6 compatibility chart). Each of them already has partial ES6 support, with a few expecting 100% support as early as Q4 2015. Once each of the major browsers fully supports ES6, our lives will get much easier. Browsers that are considered "evergreen," meaning that they automatically update independently of the operating system, will be the first to provide full ES6 support. A few examples of evergreen browsers are Chrome, Firefox, Opera, and Chrome/Firefox for Android. Within a few weeks of a new release, most users have the newest version. After a few months, over 98% of users will have the latest version of an evergreen browser. Not only do these browsers have auto-updating built in, they also adhere to very short release cycles. This means that we don't have to wait years between releases, as the updates are only weeks apart. These browsers make our life easier. It's the non-evergreen browsers that will make us wish we didn't have to get out of bed in the morning. A few examples are Internet Explorer (all

versions), Safari (desktop and mobile), and Android's "Browser" (the worst offender). These legacy browsers have caused the death of innumerable kittens.

This begs the question: if a significant number of our users don't have an evergreen browser, what should we do? Chapter 4 explains our options for using ES6 without abandoning those users. I would like to display some information about how far some companies are going to promote the use of the Web. The following are all examples of what the industry is doing to prune support for stale browsers.

Microsoft

Beginning in August of 2014, Microsoft began implementing pieces of its strategy to revive its in-house browser, Internet Explorer. The company appears no longer impartial about how long people use outdated versions. Not only did it point out that updated browsers "decrease online risks," it also pointed out that stale browsers "fragment the Web" and decrease productivity. Along with these claims, Microsoft announced that starting on January 12, 2016, it will only support the most recent version of IE available for your operating system. This means that a consumer running Windows 7 SP1 will need to be on IE11 in order to continue receiving security updates.

On January 21, 2015, Microsoft announced that their new operating system, Windows 10, will be a free upgrade for anyone running Windows 7 or newer (you need to upgrade within the first year). Further, all subsequent updates will be free. Further, they announced that Windows 10 will include a new browser (currently called Project Spartan) that will be updated independently of the operating system. In March of 2015, Microsoft announced that IE will no longer be the default browser on Windows and that Project Spartan will take over in Windows 10. This means that the Microsoft browser of the future will be evergreen.

Microsoft is taking some aggressive (and expensive) moves toward helping users avoid an insecure and outdated Internet experience. If Microsoft is abandoning support for "oldIE," then what business do we have supporting it?

Google

The king of search, Google, has a similar support strategy. On its help and support page for sites such as Google Apps, Google spells

out its policy for supported browsers. It supports the current and previous version of all major browsers. In its documentation, the company explains its reasoning:

> At Google, we're committed to developing web applications that go beyond the limits of traditional software. Our engineering teams make use of new capabilities available in modern, up-to-date browsers. That's why we made the decision last year to support only modern browsers, which also provide improved security and performance.

Rather than spend money to help people limp along in their out-of-date browser, Google opted to spend money innovating and gaining a competitive edge by building websites that "go beyond the limits" of traditional websites.

Kogan.com

One last example of the direction of the industry demonstrates in-your-face boldness.

An Australian electronics store, Kogan, took a strong stance against stale browsers. As of June 2012, any shopper checking out in IE7 or lower will be charged a special IE tax. The IE tax rate is 6.8%, which is 0.1% for each month since Microsoft released IE7 and the date Kogan.com rolled out its IE7 tax feature. The following is Kogan's explanation to the user about the IE7 tax:

Conclusion

The industry as a whole is largely on the fence with regard to abandoning stale browsers. However, two of the biggest movers in the game (Microsoft and Google) are herding people to modern browsers. With that, they are saying: When faced with spending your money on stagnating to support "oldIE" or innovating and building the apps of tomorrow, always bet on tomorrow. It will help you retain a competitive edge and keep your teams sharp.

Additionally, you should make an informed decision when deciding to prune support for legacy browsers. If you are setup with a web analytics platform, look at the data to find out what percentage of your users are on these old browsers. You may be surprised with what you find. While management won't easily prune support for an unknown number of users, you will find that people are much more willing to move forward with a decision when you provide them with current and past browser usage statistics for your company. Once you know the statistics, everyone can make a more informed decision with regard to moving forward with the Web.

Please check your pulse. If reading this section has raised your heart rate, no worries. Recommending that you drop support for Internet Explorer can have that effect. If this is you, go ahead and skip to Chapter 4 and read "Graceful Degradation" on page 25 and "Using a Transpiler" on page 27. These offer serious solutions that will allow your team to use ES6 without completely abandoning IE users in the process.

Recruit and Retain Top Talent

Suppose that your team has an open spot. You would *really love* to fill that position with a rockstar developer. You know the kind I'm talking about. One of those developers who sleeps using a keyboard for a pillow. But where can you find this (He-Man or She-Rah)-gone-programmer? A better question would be: what can you do to make that person come to you? And an equally pertinent question would be: how do you keep that person with you?

Unfortunately, limitless sodas, snacks, and an arcade machine aren't considered perks anymore. These days, those perks are all too common and are expected. Still, employers have many opportunities to

draw in and keep a top-talent developer. One of those opportunities is your technology stack.

You can't send a ninja into a sword fight without a sword. Ninjas needs their swords. In the world of JavaScript ninjas, there are things you can do that will make them feel like you've taken their sword away. Things like telling them that their cutting-edge experience needs to be throttled back to match decade-old standards. If decade-old standards are your target, I would ask you: do you really need a top-talent developer?

Telling your JS ninja that he can't innovate is another way to make him feel like a sad, swordless ninja. On page 62 of his book *The Myths of Innovation* (O'Reilly), author Scott Berkun asserts that when we mix innovative people with frustrating situations that prevent them from innovating, those people will leave. His examples range from Michelangelo and da Vinci, to the founders of Apple, Google, Microsoft, Yahoo!, and HP. Each is an example of people, or groups of people, who were frustrated by the limited thinking of their peers or management. In each case, the frustration, combined with their need to innovate, forced them to seek out another home for their ideas. In each case, their frustration was justified. Their innovative thinking proved to be very successful.

This type of frustration will result in employees finding another home. Rockstar "bro-grammers" and "diva-elopers" need an environment that can keep up. The best way to keep them is to feed their need to learn and trailblaze and allow them to keep disrupting. Adopting ES6 as a standard will help satisfy your innovators' need to learn and practice those new findings. It will fulfill their need to become current and disrupt, without incurring unwanted risks for your organization.

Truly, ours is the job of coexisting with innovators rather than forcing them out the door to find a home for their ideas.

Efficiency

The term "efficient code" can have a few different meanings. The many new features in ES6 can each be categorized as satisfying one or both of these definitions.

The first is: does it run efficiently? If I write code using the new ES6, does it run faster than code written in ES5 and earlier? Many of the

features in ES6 are runtime optimizations. Many of these new features have been taken from other languages, where these optimizations were found and implemented.

The second is: can I write/maintain it more efficiently? I was unable to accurately attribute the following quote to any single author. However, consider the following:

> If I had more time, I would have written a shorter letter.
>
> —T.S. Eliot / Blaise Pascal / John Locke / Ben Franklin / someone else?

In programming, the same is true. Most code could be reviewed and written with fewer lines.

Does ES6 make writing code more efficient than previous versions of ES? The answer is unequivocally "Yes!" ES6 has a handful of new features that will save you dozens of lines of boilerplate inside each function. For example, writing "classes" in pre-ES6 versions of JavaScript is much more verbose than doing the same thing in ES6. And so it goes with many other features. Not only does coding with ES6 constructs help your developers make more progress, it will make their run faster than it ever has before.

The World Is Changing

In *Innovation and Entrepreneurship* (HarperBusiness), Peter Drucker said the following about management:

> Management tends to believe that anything that has lasted for a fair amount of time must be normal and go on forever. Anything that contradicts what we have come to consider a law of nature is then rejected as unsound.

Moving away from heavy "oldIE" support may be met with resistance. Transitioning your web architecture from server-side templating to a heavy, frontend templated JavaScript solution may also be met with resistance. You may even be the one resisting. Those who have seen success in the past tend to think erroneously that their one road traveled is the only road worth traveling. As Drucker suggests, proposing alternatives to tried methods is often "rejected as unsound." Drucker refers to this as a "myth of management," a myth that we can help overcome.

The opposite of this myth is known as "chronological snobbery", and it can cause entirely different problems. By constantly discrediting past ideas due to having been thought up before we had our present knowledge, you rob yourself of the stability that comes with making a decision once and then sticking with it for a while. If decisions like which technology to use are being re-decided every few months, you may find that you have a chronological snob among you.

I once had a conversation about implementing a newer server architecture in our organization. I was told that "old technologies with six- to seven-year proven track records are what is needed in an enterprise arena." Additionally, I was told that "these newer projects (that I was proposing) change version numbers on a daily basis." The person saying this meant that these constant commits were a sign of instability, which scared him. To these two comments, I had two responses. The first was that IE7 is six to seven years old. Was my friend suggesting that we roll back all production to IE7 standards? He shook his head. Second, if I had to choose between an architecture that has dozens of commits per day versus two or three commits per year, I'd choose the more active platform. If the community around your architecture can't manage to commit updates and fixes on a daily basis, then you are on a platform that doesn't have any demonstrable longevity. Platforms with active communities that innovate are the platforms of tomorrow.

Technologies don't need to have a record-setting trend before they can be safely adopted. Three years ago, AngularJS was a dark horse. Now its popularity has surpassed the combined popularity of all other past and current client-side JavaScript frameworks. And similarly, prior to 2014 no one had even heard of React.js. Fast-forward less than 12 months, and we see that React.js is embraced by many of the smartest JavaScript developers around.

Some of the most beautiful parts of the Web would have been rejected if we all bought into this myth. We would still use XML instead of JSON, and SOAP instead of REST. Teams need to evaluate technologies on their merits. We need to trust in our ability to innovate now and refactor later on, where needed.

CHAPTER 2

ES6 Goals

When looking at what's new in ES6, I have found it helpful to understand some of the history behind JavaScript.

History in the Making

July 2008 marked the beginning of change for ECMAScript. These changes were the result of years of hard work, deep thought, and discussion. Many brilliant minds spent years debating, and at times fighting over, what the next steps for ECMAScript should be. The Web had spent years growing organically and evolving. That growth and evolution had different meaning for many companies and individuals. Many stood to benefit significantly, if they could only make a few changes of their own to the ECMAScript specification. These biases made life difficult for the TC39, the committee in charge of steering the ECMAScript specification. None of the TC39 members could have predicted the challenges they would face as they attempted to advance ECMAScript, and by proxy, JavaScript.

As in many debates, all sides argued their honest opinions about what needed to be changed to further the language. Prior to July of 2008, many of these debates became heated, and very few saw much progress, if any. Due to these conflicts, the small group that initially endeavoured to fight for JavaScript inevitably broke down. Brendan Eich, the creator of JavaScript, compared the history of the ECMAScript standardization committee to J.R.R. Tolkien's *The Fellowship of the Ring*. It is a story about a once strong group of friends who are ultimately divided into smaller groups, taking separate journeys.

(See Douglas Crockford's "The State and Future of ECMAScript" and Brendan Eich's Lord of the Rings analogy.)

These separations had all but stagnated the progress of the language for years. Before July 2008, progress in the browser (and specifically JavaScript) had come to a halt. (See Brendan Eich's keynote at YUICONF 2009.) Something had to give if this would ever change. Not everyone could get what they wanted if ECMAScript were to move forward.

The Meeting

Opera hosted the TC39's monthly meeting in Oslo, Norway's capital. There were two different camps fighting for control of the next ECMAScript update. One camp had proposed a smaller update under the release number ES3.1. The other camp wanted a more robust release, full of features that hadn't reached a group consensus. This second, larger release was dubbed ES4. And this had been the debate for months: whether to choose the smaller ES3.1 release, or the larger, feature-rich ES4. Unlike the previous meetings, these two sides would reach a compromise. ES4 would be postponed, and ES3.1 would become the next release. However, because the group had torn down their walls and made new alliances, they changed the release number from ES3.1 to ES5 to account for those milestones.

The final ES5 specification was approved in September of 2009, with a follow-up release 5.1 landing in June of 2011. This marked a huge step forward for browser vendors. Progress and standards prevailed, and JavaScript was again moving forward with new features. This was all very cool.

Harmony

What about all the ES4 features that no one could agree upon? Where did they end up?

A super-majority of the ECMAScript proposals fell into this bucket. In August 2008, Eich addressed the TC39 and let them know that all remaining features (ES4 and beyond) would be grouped into a collection labeled "Harmony," a tribute to all the committee members that harmoneously came together to help move the language forward again. In his email, Eich outlined a list of goals for the Harmony features. A few additional goals have been standardized since

then, and they can be found on the ES Harmony wiki. The goals include:

1. Provide a better language for writing
   a. complex applications,
   b. libraries,
   c. and code generators targeting the new edition.
2. Switch to a testable specification.
3. Improve interoperation, adopting de facto standards where possible.
4. Keep versioning as simple and linear as possible.
5. Support a statically verifiable, object-capability secure subset.

These goals still guide TC39 today.

Complex Applications

As JavaScript began, the engineers pioneering its adoption had only a fraction of the ambition that we demand from JavaScript. What started as a very simple scripting language has grown into the most used language development language on the planet, appearing in the browser, on our servers, and event-powering robots. JavaScript needs features that allow for less coding while producing more functionality.

Libraries

Any given page on the Internet may have dozens of JavaScript dependencies. *As a JavaScript project becomes larger, the task of library and dependency management increases in difficulty.* Harmony has a handful of features that will provide a better experience for library builders and app developers alike.

Adopt De Facto Standards

JavaScript is only one of the programming languages involved in building modern web applications. Many of today's best developers have other languages they love in addition to JavaScript. The silver lining in being late to the web standards game (don't forget that we spent years fighting, not updating) is that you get to see what everyone else is doing, and you can aggressively reject the bad and assimi-

late the good. The ECMAScript specification is adopting many of the most popular features from some of today's best languages, including Python, Ruby, Java, and CoffeeScript. This will make JavaScript appear to be more familiar and friendly to developers that have a grasp on those other languages.

ES6: Subsetting Harmony

Immediately after ES5.1 was formally released in June 2011, the TC39 began discussing and planning the next update to the ECMAScript specification. Many committee members already had favorite features in mind. Because of that, many features were quickly sorted to the front of the line.

A handful of these features have been part of Mozilla's Firefox for the past six to seven years (yes, they predate the ES5 specification approval). I found several instances of ES6 features existing in production browsers, including a seven-year-old code commit to the Firefox source code by Brendan Eich himself, in which he implemented destructuring. This means that the browser vendors implemented many features that had yet to be officially approved by the standards committee. That may sound bad. It is more likely that this accelerated the process of approving features for the ES6 specification. Because the features were already in production browsers, it was easier for the committee to see how they would affect the language.

Now, four years after the ES5.1 specification was officially approved, the TC39 is busy preparing to approve ES6 in June of 2015. As an outsider, I have watched the TC39 make make of their decisions. I have read their thoughts and meetings notes. I am excited for this latest release.

Imagine swapping a car's motor while it is driving down the freeway. This is what the TC39 is trying to do with this latest version of the ECMAScript specification. Their goal is to have a final approval from the ECMA General Assembly by June of 2015. Yet, just a few months ago in January 2015, they continued to modify their proposals and make changes to the spec. Further, while the JavaScript community at large has come to embrace this upcoming release by the name of ES6, the TC39 has opted to rename the release to ES2015. However, like most people, I will continue to refer to this release as ES6.

Because the Web depends on JavaScript, and JavaScript depends on the ECMAScript specification, progress in the Web is highly coupled to the success of the TC39 members. Given their successes in the past few years, the Web, and all other things JavaScript are ready to blaze forward faster than they ever have.

CHAPTER 3

Features Explained

ECMAScript 6 is a large collection of new features that programmers need to learn and explore and ultimately bring to their projects. The following is a list of some of the features, with a brief description of each one.

Arrow Functions

When JavaScript programmers talk about what `this` means in JavaScript, you may feel like you're watching "Who's on First?" For those who are new to JavaScript, understanding what `this` is can often prove difficult.

What better way to provide clarity than to add yet another meaning for `this`? Arrow functions are a new syntax that allow developers to manage their scope differently than before, providing yet another value for `this`.

While that may sound confusing, arrow functions will help make JavaScript code more readable. When using arrow functions, your code reads the same way that it will execute. The same cannot be said for all usages of `this`. Such functionality will help make JavaScript code more readable and predictable for developers, which translates into easier maintainability.

In addition to helping developers understand what `this` is, arrow functions have some syntactic sugar that allows you to opt-out of using the `function` and `return` keywords throughout your code.

Let, Const, and Block Functions

Prior to the ES6 release, each time you declared a new variable, you used the keyword `var`. There were no alternate keywords to define a variable. Starting with the ES6 release, you will now have two additional constructs for defining new variables: `const` and `let`.

Using `const` makes your variables a constant value. Variables defined using the keyword `const` will never be changeable. Other languages call these final variables, as their value cannot be changed once it is set.

`let`, on the other hand, is more like `var`, in the sense that you can change the value repeatedly. Where `let` and `var` differ is in relation to how they scope themselves. To the chagrin of many JavaScript developers, using the keyword `var` to define your variables can leave you with code that looks one way but acts differently when it comes time to be executed. Using `let` instead of `var` produces clearer code, which is the linchpin of maintainability. `let` allows the code to execute the way it was written. This improved predictability will make life easier for everyone involved with your project.

Destructuring

As you pass data around inside your apps, you will invariably need to pull pieces of the data apart to examine them individually. The task of pulling data apart is also referred to as destructuring, because you are taking the data's structure apart. Prior to ES6, programmers were required to be fairly verbose while destructuring their data. ES6 destructuring allows you to perform the same task of taking your data apart with a lot less code.

Default Values

Toward the top of many JavaScript functions there is code that verifies that all incoming parameters have a value and are not left undefined. Unwanted undefined values can create hard-to-find bugs in code. ES6 default values allow you to elegantly verify and provide default values for each of your parameters. It has always been possible to catch these undefined parameters and trap them before they make a mess; however, the new default values syntax simply makes it much easier. Many other modern languages have this feature.

Modules

Prior to ES6, JavaScript had two main competing module systems: AMD and CommonJS. Neither is native to the browser: they are monkey-patched into the browser at runtime. Both of these module systems have different strengths. When it came time for the TC39 to decide between the two, the committee decided to come up with yet another approach for modules. The new approach is elegant and covers an even wider spread of functionality then either of the predecessors.

Modules, in general, solve several problems for developers. First, they allow the developer to separate code into smaller pieces, called modules. Second, they make it easy for developers to load (inject) those modules into other sections of code. Having modules injected like this helps keep project code uncoupled from the module (read: improved testability). And third, modules can load scripts asynchronously. This means that apps can begin loading faster, as they don't require all scripts to be loaded prior to executing code.

Under the current proposal, modules will provide features from both AMD and CommonJS. In addition, ES6 modules have additional pieces that don't exist in either of the others.

Classes

One of the easiest ways to reuse code is by allowing one piece of code to inherit the functionality from another piece. This way, both pieces of code have the same functionality, but the code only gets written once. By using an inheritance chain known as prototypes, JavaScript has always been able to share code this way.

While prototype chaining works, most other object-oriented programming languages use a variation of class-based inheritance to share functionality from one piece of code with another. Starting in ES6, you will be able to define and inherit functionality by using classes as well as prototypes. The syntax to share code using classes is much cleaner than its equivalent in prototypical code sharing.

This is a polarizing feature among JavaScript developers. You won't find many developers who are on the fence with regard to classes in JavaScript. They will either like it or dislike it.

As I see it, classes have two main advantages. First, classes add some syntactic sugar to inheritance in JavaScript. This means that inheritance in ES6 will be more straightforward, and the code will be easier to follow. Further, languages that compile to JavaScript will be able to do so more easily and exactly. Second, once classes have completely replaced traditional prototype-inheritance, regular functions will potentially have a smaller footprint in memory.

Rest Parameters

The term "rest" frequently refers to REST, a common pattern for writing web services. In the context of the ES6 release, "rest" has nothing to do with a web service.

Many other modern languages allow you to make a method that takes a dynamic number of parameters. Consider a function called `add`. It may need to add 2 numbers together; it may need to add 100 numbers together. Suppose that at times you need to add two numbers together, and at time you need to sum three numbers. You wouldn't want to write two different `add` functions to accomplish this. ES6 rest parameters now give JavaScript the ability to have functions that accept a dynamic number of parameters.

Previously in JavaScript, if you wanted to pass around a dynamic amount of numbers, you had to put them inside a container, like an array. Rest parameters allow you to just pass the numbers, without jumping through the hoop of manually wrapping them in a container.

Spreading

Arrays are just linear collections of objects. They could be a collection of numbers, words, or even other arrays, for example. How can you get things out of the array once they are there? Looking backward, you always had to reference each of the items one by one. This always worked; however, it tends to be a bit verbose, not to mention that the code can become cryptic. ES6 spreading allows you to easily spread out the array, without all the mess.

Proper Tail Calls

Proper tail calls leverage a low-level enhancement inside the JavaScript engines. In spite of the many things that JavaScript is good at, it has always been notably horrific at recursion. A recursive method is one that will call itself over and over again. These methods may only need to do this a few times, or they may need to recurse tens of thousands of times. Each time the method calls itself, it consumes more memory. There is no limit to the amount of memory that recursion may use. Within milliseconds, it is capable of crashing your entire website due to memory consumption.

ES6 proper tail calls allow JavaScript to run a properly written recursive function that can call itself tens of thousands of times without using additional memory for each call. Fibonacci sequence, here we come!

Sets

JavaScript has has two very powerful containers: arrays and objects. Arrays are lightweight containers. You can treat them like a queue or a list, popping an object off the end, or shifting an object off the front. However, there were some things that an array couldn't easily do.

Prior to ES6, if you wanted to have an array without any duplicates, you had to write your own code to prevent duplicates from being added to the array. This duplicate-preventing functionality now comes built in with sets. Sets are very much like arrays in JavaScript. However, if you try to add the same value twice to a set, the set will simply ignore the second attempt to add that value. With sets there is now an easy way to have collections of unique values.

Maps

In other languages, we refer to maps as hashmaps or dictionaries. A hashmap maps a key with a value. An example would be mapping the key "name" with the value "Freddie Mercury" and the key "band" with the value "Queen."

JavaScript has had these hashmaps for a long time. In fact, JSON is an entire data structure based on keys that are mapped to values.

Maps add functionality to JavaScript's JSON and Object Literal syntax. Historically, you could only use strings and numbers as keys in object literals. With the new `Map` object, however, you can use any object as the key. We aren't restricted to strings and numbers. You could have a key that is a function or an HTML element. Maps are more dictionary-like than anything JavaScript has ever had.

Weak Maps

Weak maps are exactly like maps, with a few exceptions. The primary difference is: weak maps can make cleanup easy.

Suppose you have a map and add a key that represents some HTML element from your page. If that HTML element is ever removed from your DOM, having a reference to that element in your map prevent the browser from completely removing the HTML element from memory. This is known as a memory leak. Because the map still points to the element, the garbage collector will not remove it and free up the memory that it was occupying.

Unlike a map, weak maps will let the garbage collector remove the element from memory. If the only reference to the HTML element is the key in the weak map, the weak map will release its reference as well. Once all references have been dropped, you can then say goodbye to the HTML element forever.

Generators

Generators are functions that can be paused. By adding the keyword `yield` inside our generator functions, the system will pause the execution of code, return the value to the right of the keyword `yield`, and wait until we tell it to continue. (Generators have been in Firefox for quite a while now.)

Generators have many practical uses. One would be generating unique IDs. Each time you need a new ID, you could call `generator.next()`. The generator would return the next ID and then pause again. You could write your own `for` loops with generators, along with many other features. I predict that generators will help library and framework authors to produce more efficient code.

Iterators

In conjunction with generators, iterators offer enhanced support for looping. If you have an object that is iterable, you may now use a new `for` loop to iterate through that object's properties. The new `for…in` loops make it easier to go through an object's values. Prior to iterators, you needed to reference each value by first referencing the key and then using it to retrieve its values. `For…in` loops give us direct access to the object's values.

Direct Proxies (and Supporting Features)

Direct proxies allow developers to intercept method calls on an object and have a separate object perform those instructions instead. While this sounds complicated and entirely convoluted, they will provide increased functionality for library builders. Direct proxies help developers abstract functionality out of individual objects and into more powerful APIs.

To drive the point home, one could use a proxy when saving an object in JavaScript. When you tell the object to save itself, you may have a proxy that intercepts and carries out the save instruction. As of today, the proxy might save the object to your server. At a later date, you may need to implement a new proxy that saves the object to your browser's `localStorage`. Simply switching the proxy can change where your objects save. This can be done without ever modifying the object code.

Along with the ability to proxy method calls, several other changes were needed in order to completely enable proxies to work, including prototype climbing refactoring, a new reflect API, virtual objects API, and some extensions to the object API.

String Literals

ES6 string literals fill multiple roles. One role is templating; another is string interpolation. In the browser, JavaScript is often used to glue together HTML, CSS, JSON, and XML. While there are many client-side templating libraries out there, they will all enjoy what ES6 quasis have to offer.

String literals (referred to as quasis by some) will perform much faster than any templating library could ever hope to. Additionally,

they will come packed with all sorts of data-scrubbing goodies to help developers prevent security issues like cross-site scripting, SQL injection, and more. And finally, string literals will allow developers to use multiline strings. Any developer that denies lusting after multiline strings is lying.

While the name "quasis" is unfortunate, I am confident that they will quickly become one of the more widely used pieces of the ES6 release.

Binary Data

The binary data enhancements in ES6 aim to provide some performance gains when working with binary data in JavaScript. APIs such as Canvas or FileReader return binary data. By improving performance behind the scenes, these APIs will perform much faster at runtime than they ever have before.

API Improvements

The `Number`, `Regex`, `String`, and `Math` global objects each has a handful of improvements in the newest version of JavaScript. None of the enhancements are major. Most of the methods that are new are utility methods that many people have built on their own.

Unicode

Beginning in ES6, JavaScript will begin supporting UTF-16. These changes will add better support for a wider range of less frequently encountered characters.

CHAPTER 4

Where to Start

It's settled then. You are going to begin integrating ES6 into your projects. Still, you may have some questions about where to start. Or maybe you would like some best practices for your organization. The following suggestions may make this transition easier.

Incidental Functionality First

Your team will need to gain some momentum when using these new constructs, especially if you're catching up on your innovation debt. I suggest you pick a portion of your app that isn't considered "core functionality." Let your team train up on areas of your app that add value and provide incidental functionality. This will allow you to leverage your team's training process while minimizing any risk involved.

Graceful Degradation

Some browsers have more capabilities than other browsers. Maybe you've decided that you want to use some functionality that doesn't exist in all browsers. In other words, your users will have a different experience on your site, depending on their web browser. When implementing different features for different browsers, there are two competing ideologies:

Progressive enhancement
: Progressive enhancement suggests that you begin by building your site for the worst-case and most basic scenario. Once that exists, you add functionality for the more capable browsers.

Graceful degradation
: Graceful degradation suggests that you build your app for the most capable browser. Once that exists, you find alternate functionality or turn off the functionality for less-capable browsers.

I recommend that you start out by implementing graceful degradation. Allowing your team to focus on the best possible scenario will produce the best possible experience for your customers. You may then combine your best-case scenario using a transpiler (see "Using a Transpiler" on page 27), pushing Microsoft's *Enterprise Mode* (see "Microsoft's Enterprise Mode" on page 28), and/or traditional monkey-patching. Combining these technologies may even enable the worst of browsers to run the latest ES6 code.

Monkey-patching is the term used to describe the act of modifying JavaScript's default behavior. An example of monkey-patching could be adding JSON.parse and JSON.stringify functionality into IE7. Because IE7 was released prior to the explosion of JSON, it doesn't have support for the JSON API. However, a handful of libraries out there monkey-patch IE7 so that it has JSON.parse and JSON.stringify functionality. Monkey-patching is also known as *duck-punching* and *polyfilling*.

However, please do not shy away from shutting off different pieces of your site to users who are on older browsers. As discussed previously, herding your users to an evergreen (auto-updating) browser will reduce your development and maintenance costs. Further, as a member of the web community, we all have a duty to protect our users. While modern browsers are much faster and more capable than older browsers, that isn't why you should help your users upgrade to a better browser. You should help them upgrade so that they can use a browser that has the latest security patches and updates. You should help them upgrade so that they can be safe.

Evergreen browsers automatically update themselves. Additionally, they remain up-to-date, regardless of your operating system. This ensures that you have the freshest speed, security, and functionality enhancements available.

Train Your Teams

Once you've seriously committed to training your team, those commitments should be made apparent. Your commitment should be more than simply giving them goals on their annual performance review or buying each member a book to read. Create a culture based on the results of the training. Recognize, reward, and promote individuals who are leading the way and helping others to learn.

The number of pure JavaScript conferences is increasing. Both locally and nationally, these conferences are becoming viral. Find two or three of them and send your teams. If the conference is during the work week, pay your employees to attend the conference. Don't require your team to use vacation days to go to a conference. If the conference is on a weekend, make strong recommendations that they attend. Have team members return and present their findings to the rest of the team. If you have to, offer comp time for those who use weekends to attend conferences.

Do everything in your control to get your teams trained and talking to each other about JavaScript.

Using a Transpiler

One of the quickest ways to get your team into ES6 is by using a transpiler. With a transpiler, you write ES6 code, and it is then turned into something that older browsers can understand. This means that your teams can start using tomorrow's syntax today.

This means that you can have your teams write this:

```
let add = (x, y) =>  x + y ;
```

and it will convert that into code that your users' browser can understand:

```
var add = function(x, y){
    return x + y;
};
```

If the idea of a transpiler interests you, you may be pleased to know that you have more than one option. As I write this chapter, there are two transpilers that are more widely used than others. Babel is the most widely used, and Traceur-Compiler is the second most widely used.

Babel (formerly known as 6to5 but changed its name when ES6 was renamed to ES2015) is the most popular, and currently supports 76% of the latest ES6 functionality. Babel is used and recommended by many of today's greatest JavaScript developers. While Babel supports more features from ES6, Traceur-Compiler has a more semantically-correct implementation of those features.

As I compared Babel and Traceur side by side, I realized that some of the expectations that I had for runtime ended up being less important to the Babel engine than the Traceur engine. However, when Babel transforms your code, the code is more readable than it otherwise is when transpiled with Traceur. In other words, each library has its own benefits. Yours is the job of finding those differences and helping your team make an educated decision on what you should use.

In addition to Babel and Traceur, there are other transpilers that you should listen for. JSX w/ ES6, Closure, ES6-shim, CoffeeScript, and TypeScript are the ones that you should keep your listening for. Particularly, you may want to listen for TypeScript. In March of 2015, the Angular core team announced that their plans for Angular 2.0 include the adoption of TypeScript as an official part of their recommendation. Because of this, you will begin to hear about more and more developers integrating TypeScript with their projects. Additionally, TypeScript carries with it many of the features from ES7 (ES2016), which means that it will continue to be more and more relevant. Further, the TypeScript team has more developers on it than both Babel and Traceur combined, making it a mean competitor.

Microsoft's Enterprise Mode

In the past few years, Microsoft has taken several steps toward helping the Web gain momentum. And more than one of those changes may prove helpful as you attempt to fight free of the bonds of oldIE. I am not asserting that we need to forget everything that Microsoft has done along the way to sandbag the progress of the Web. How-

ever, I do feel that we should each let our biases go and accept progress as such. Especially when it helps us move our organizations forward.

One of the reasons that many corporate environments are stuck using an old version of IE is because they have a few sites that don't run well in newer versions of any browser. In almost all cases, these are sites that they depend on. And rather than rewrite those sites to work in modern browsers, they are more than happy to force all of their network users to continue using IE8 or IE9. If this sounds like your organization, or your customers' organizations, KEEP READING! There is hope.

One of Microsoft's recent features that I think you should care about is Enterprise Mode. Enabling Enterprise Mode on a Windows Network means that you can set up a list of certain sites that need and depend on oldIE to function appropriately. Then, any network user can open Internet Explorer 11 (or whatever the latest version is when you are reading this) and IE11 will load those sites with an older version of its JavaScript and HTML/CSS rendering engines. A version that acts, thinks, and performs like oldIE performs. All other sites will get to experience the Web through IE11.

Perhaps educating your network folks (or your customers' network folks) about the benefits and ease of use of Enterprise Mode may afford you the freedom to upgrade your development process to only include more modern browsers. If are like me, you know the exact number of users that are still using oldIE. Fingers crossed that Enterprise Mode, along with Microsoft's support changes coming in January 2016, will help us all move our projects into the future.

Summary

Every organization is different. Each team will need to set its own pace. With innovation in general, constant progress is key. Aggressively seek out new opportunities to help your projects stay on track with your innovation goals. The total reduction in costs for standardizing on modern technologies will make your efforts worth it. The age of "oldIE" isolation has already begun.

CHAPTER 5
Watching for ES7

As of January 2016, the list of possible features for ES6 has been frozen. Only the proposals that are already on the list may be included in ES6. Any additional proposed features to ECMAScript will have to wait in hopes of being included in ES7. Potential features for ES7 have already begun to appear.

JS.Next refers to the next version of the JS API. Currently JS.Next refers to ES6. Once the ES6 release is live, JS.Next will refer to the ES7 release. JS.Next will always refer to the next update to the language.

Several of the potential features have the possibility of really changing the way we think about app development in JavaScript. Taking an initial look at those features now can give us some time to digest them while the ES7 release is being prepared over the next few years.

Everything in this section is partially speculation. None of the following proposed features has been officially accepted into the final ES7 release. Additionally, only `Object.observe` has been officially approved as a Harmony proposal.

Object.observe

A popular feature in modern frontend JavaScript frameworks is two-way databinding. Having two-way databinding makes life easier on your developers. It allows them to focus on writing new code rather than worry about keeping all the data in sync. `Object.observe` makes two-way databinding easier than ever before.

On any given JavaScript-assembled web page, the values on the page represent some JavaScript variable in your code. Keeping those two values (the page value and your variable's value) in sync is important. Any time your users update the values on the page, your variables are updated as well. Additionally, if you change the value of your variables, you want the page to be updated with your changes. That is two-way databinding in a nutshell.

We have already seen a handful of frameworks implement `Object.observe`, each reporting huge performance increases. In late 2012, Chromium, the open source browser project on which Google Chrome is based, released `Object.observe` functionality in its Canary channel. A few of the modern frameworks then updated their two-way databinding to use `Object.observe`. Those frameworks saw their performance increase. The AngularJS team reported that their code ran 20-40 times faster than it had previously. This means that once `Object.observe` is released into the Web, most libraries and frameworks that switch to it will see increased efficiency and optimization.

Multithreading

One of the best things about JavaScript is that it runs in a single-threaded nature. Because you can write code without worrying about multiple threads, application development tends to be easier. While this is a huge plus, it also present limitations. For example, if any of your JavaScript code runs too long, your entire app will appear to be frozen. Things like intense graphics can easily bring your code to a halt. If JavaScript apps had a good option for running code in parallel, on different threads, it would be a much more powerful language.

Intel Labs has a research project called RiverTrail. Together with Mozilla, Intel aims to allow JavaScript applications to run using multiple threads. This means that your browser would be able to utilize all your computer's processors. As shown in their demos, RiverTrail allows things like intense graphics processing to happen right inside the browser.

With any luck, the RiverTrail project will yield some conclusions of sorts that can be integrated into the ECMAScript specification and then implemented in JavaScript. Having the ability to easily program in parallel would be a great addition to the language. Maybe your average JavaScript developer wouldn't use this type of low-level functionality, but your framework developers would. This means that once you implement the frameworks, they will take care of the lower-level stuff like improved optimization.

Traits

In many object-oriented languages, your objects share functionality by inheriting it from parent classes. One of the alternatives to class-based inheritance is traits. By using traits, you define the functionality to be shared as a trait and then add the desired traits onto your object. By adding a trait to your object, your object will absorb the functionality of your defined trait.

Implementing traits provides another powerful way of sharing functionality. It can also reduce the risk of over-architecting traditional class-based inheritance models.

Additional Potential Proposals

Here is a list of additional items to watch for in ES7. I am sure this list will grow before the ES7 approval process is over:

- Precise math
- Improved garbage collection
- Cross-site security enhancements
- Typed, low-level JavaScript
- Internationalization support
- Digital electronic rights
- Additional data structures

- Array comprehensions
- Async functions
- Typed objects

Invite your teams to keep their ears to the ground, listening for more cutting-edge features that may appear in JavaScript.

About the Author

Aaron Frost spent the last several years swimming (at times sinking) in the Open Web waters. Finding JS and CSS/HTML was the best thing that could have happened to him. By day he is a part of the Domo frontend team, building an app that makes everyone a CEO. By night he is working with O'Reilly Media and is writing the book *JS.Next: ES6*. The final release may be much later, as the final spec will take a while to solidify. He is also a Google Developer Expert, nominated by Google for his work with AngularJS and its community. He is also an Egghead.io author. Additionally, he works on several small projects for himself, and one with his identical twin brother. Peppered in between working hours, he enjoys being married to a wonderful wife and being the dad of four amazing monsters. When the world is white and frozen, you will find him atop the mountain, ice fishing.