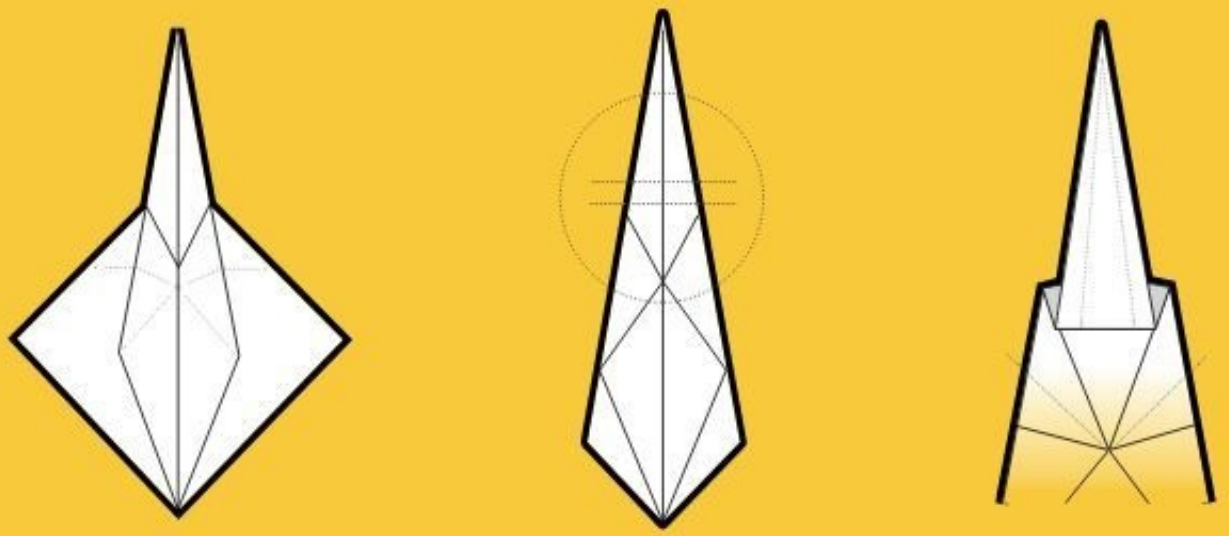
sitepoint

JAVASCRIPT: BEST PRACTICE



4. 5. 6.

CLEAN, MAINTAINABLE, PERFORMANT CODE

JavaScript: Best Practice

Cover Design: Alex Walker

Notice of Rights

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Preface

There’s no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6). Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days. We're aiming to minimize that confusion with this set of books on modern JavaScript.

This book presents modern JavaScript best practice, utilizing the features now available in the language, enabling you to write more powerful code that is clean, performant, maintainable, and resusable.

Who Should Read This Book?

This book is for all front-end developers who wish to improve their JavaScript skills. You'll need to be familiar with HTML and CSS and have a reasonable level of understanding of JavaScript in order to follow the discussion.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so: `<h1>A Perfect Summer's Day</h1> <p>It was a lovely day for a walk in the park. The birds were singing and the kids were all back at school.</p>`

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed: `function animate() { ⋮ new_variable = "Hello"; }`

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web- ➥design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1: The Anatomy of a Modern JavaScript Application

by James Kolce

There’s no doubt that the JavaScript ecosystem changes fast. Not only are new tools and frameworks introduced and developed at a rapid rate, the language itself has undergone big changes with the introduction of ES2015 (aka ES6). Understandably, many articles have been written complaining about how difficult it is to learn modern JavaScript development these days.

In this article, I’ll introduce you to modern JavaScript. We’ll take a look at recent developments in the language and get an overview of the tools and techniques currently used to write front-end web applications. If you’re just starting out with learning the language, or you’ve not touched it for a few years and are wondering what happened to the JavaScript you used to know, this article is for you.

A Note About Node.js

Node.js is a runtime that allows server-side programs to be written in JavaScript. It’s possible to have full-stack JavaScript applications, where both the front and back end of the app is written in the same language. Although this article is focused on client-side development, Node.js still plays an important role.

The arrival of Node.js had a significant impact on the JavaScript ecosystem, introducing the npm package manager and popularizing the CommonJS module format. Developers started to build more innovative tools and develop new approaches to blur the line between the browser, the server, and native applications.

JavaScript ES2015+

In 2015, the sixth version of ECMAScript — the specification that defines the JavaScript language — was released under the name of ES2015 (still often referred to as ES6). This new version included substantial additions to the language, making it easier and more feasible to build ambitious web applications. But improvements don't stop with ES2015; each year, a new version is released.

Declaring variables

JavaScript now has two additional ways to declare variables: let and const.

`let` is the successor to `var`. Although `var` is still available, `let` limits the scope of variables to the block (rather than the function) they're declared within, which reduces the room for error:

```
// ES5
for (var i = 1; i < 5; i++) {
  console.log(i);
}
// <-- logs the numbers 1 to 4
console.log(i);
// <-- 5 (variable i still exists outside the loop)

// ES2015
for (let j = 1; j < 5; j++) {
  console.log(j);
}
console.log(j);
// <-- 'Uncaught ReferenceError: j is not defined'
```

Using `const` allows you to define variables that cannot be rebound to new values. For primitive values such as strings and numbers, this results in something similar to a constant, as you cannot change the value once it has been declared:

```
const name = 'Bill';
name = 'Steve';
// <-- 'Uncaught TypeError: Assignment to constant variable.'

// Gotcha
```

```
const person = { name: 'Bill' };
person.name = 'Steve';
// person.name is now Steve.
// As we're not changing the object that person is bound to,
JavaScript doesn't complain.
```

Arrow functions

Arrow functions provide a cleaner syntax for declaring anonymous functions (lambdas), dropping the `function` keyword and the `return` keyword when the body function only has one expression. This can allow you to write functional style code in a nicer way:

```
// ES5
var add = function(a, b) {
  return a + b;
}

// ES2015
const add = (a, b) => a + b;
```

The other important feature of arrow functions is that they inherit the value of `this` from the context in which they are defined:

```
function Person(){
  this.age = 0;

  // ES5
  setInterval(function() {
    this.age++; // |this| refers to the global object
  }, 1000);

  // ES2015
  setInterval(() => {
    this.age++; // |this| properly refers to the person object
  }, 1000);
}

var p = new Person();
```

Improved Class syntax

If you're a fan of object-oriented programming, you might like the addition of classes to the language on top of the existent mechanism based on prototypes. While it's mostly just syntactic sugar, it provides a cleaner syntax for developers

trying to emulate classical object-orientation with prototypes.

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}
```

Promises / Async functions

The asynchronous nature of JavaScript has long represented a challenge; any non-trivial application ran the risk of falling into a callback hell when dealing with things like Ajax requests.

Fortunately, ES2015 added native support for promises. Promises represent values that don't exist at the moment of the computation but that may be available later, making the management of asynchronous function calls more manageable without getting into deeply nested callbacks.

ES2017 introduced async functions (sometimes referred to as async/await) that make improvements in this area, allowing you to treat asynchronous code as if it were synchronous:

```
async function doAsyncOp () {
  var val = await asynchronousOperation();
  console.log(val);
  return val;
};
```

Modules

Another prominent feature added in ES2015 is a native module format, making the definition and usage of modules a part of the language. Loading modules was previously only available in the form of third-party libraries. We'll look at modules in more depth in the next section.

There are other features we won't talk about here, but we've covered at some of the major differences you're likely to notice when looking at modern JavaScript.

You can check a complete list with examples on the Learn ES2015 page on the Babel site, which you might find useful to get up to date with the language. Some of those features include template strings, block-scoped variables and constants, iterators, generators, new data structures such as Map and Set, and more.

Code linting

Linters are tools that parse your code and compare it against a set of rules, checking for syntax errors, formatting, and good practices. Although the use of a linter is recommended to everyone, it's especially useful if you're getting started. When configured correctly for your code editor/IDE, you can get instant feedback to ensure you don't get stuck with syntax errors as you're learning new language features.

You can check out ESLint, which is one of the most popular and supports ES2015+.

Modular Code

Modern web applications can have thousands (even hundred of thousands) of lines of code. Working at that size becomes almost impossible without a mechanism to organize everything in smaller components, writing specialized and isolated pieces of code that can be reused as necessary in a controlled way. This is the job of modules.

CommonJS modules

A handful of module formats have emerged over the years, the most popular of which is CommonJS. It's the default module format in Node.js, and can be used in client-side code with the help of module bundlers, which we'll talk about shortly.

It makes use of a `module` object to export functionality from a JavaScript file and a `require()` function to import that functionality where you need it.

```
// lib/math.js
function sum(x, y) {
  return x + y;
}

const pi = 3.141593

module.exports = {
  sum: sum,
  pi: pi
};


// app.js
const math = require("lib/math");

console.log("2π = " + math.sum(math.pi, math.pi));
```

ES2015 modules

ES2015 introduces a way to define and consume components right into the language, which was previously possible only with third-party libraries. You can have separate files with the functionality you want, and export just certain parts

to make them available to your application.

Native Browser Support

At the time of writing, native browser support for ES2015 modules is still under development, so you currently need some additional tools to be able to use them.

Here's an example:

```
// lib/math.js

export function sum(x, y) {
  return x + y;
}
export const pi = 3.141593;
```

Here we have a module that *exports* a function and a variable. We can include that file in another one and use those exported functions:

```
// app.js

import * as math from "lib/math";

console.log("2π = " + math.sum(math.pi, math.pi));
```

Or we can also be specific and import only what we need:

```
// otherApp.js

import {sum, pi} from "lib/math";

console.log("2π = " + sum(pi, pi));
```

These examples have been extracted from the Babel website. For an in-depth look, check out Understanding ES6 Modules.

Package Management

Other languages have long had their own package repositories and managers to make it easier to find and install third-party libraries and components. Node.js comes with its own package manager and repository, npm. Although there are other package managers available, npm has become the de facto JavaScript package manager and is said to be the largest package registry in the world.

In the npm repository you can find third-party modules that you can easily download and use in your projects with a single `npm install <package>` command. The packages are downloaded into a local `node_modules` directory, which contains all the packages and their dependencies.

The packages that you download can be registered as dependencies of your project in a package.json file, along with information about your project or module (which can itself be published as a package on npm).

You can define separate dependencies for both development and production. While the production dependencies are needed for the package to work, the development dependencies are only necessary for the developers of the package.

Example package.json file

```
{
  "name": "demo",
  "version": "1.0.0",
  "description": "Demo package.json",
  "main": "main.js",
  "dependencies": {
    "mkdirp": "^0.5.1",
    "underscore": "^1.8.3"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Sitepoint",
  "license": "ISC"
}
```

Build Tools

The code that we write when developing modern JavaScript web applications almost never is the same code that will go to production. We write code in a modern version of JavaScript that may not be supported by the browser, we make heavy use of third-party packages that are in a `node_modules` folder along with their own dependencies, we can have processes like static analysis tools or minifiers, etc. Build tooling exists to help transform all this into something that can be deployed efficiently and that's understood by most web browsers.

Module bundling

When writing clean, reusable code with ES2015/CommonJS modules, we need some way to load these modules (at least until browsers support ES2015 module loading natively). Including a bunch of script tags in your HTML isn't really a viable option, as it would quickly become unwieldy for any serious application, and all those separate HTTP requests would hurt performance.

We can include all the modules where we need them using the `import` statement from ES2015 (or `require`, for CommonJS) and use a module bundler to combine everything together into one or more files (bundles). It's this bundled file that we're going to upload to our server and include in our HTML. It will include all your imported modules and their necessary dependencies.

There are currently a couple of popular options for this, the most popular ones being Webpack, Browserify and Rollup.js. You can choose one or another depending on your needs.

Further Reading on Module Bundling

If you want to learn more about module bundling and how it fits into the bigger picture of app development, I recommend reading Understanding JavaScript Modules: Bundling & Transpiling.

Transpilation

While support for modern JavaScript is pretty good among newer browsers, your

target audience may include legacy browsers and devices with partial or no support.

In order to make our modern JavaScript work, we need to translate the code we write to its equivalent in an earlier version (usually ES5). The standard tool for this task is Babel — a compiler that translates your code into compatible code for most browsers. In this way, you don't have to wait for vendors to implement everything; you can just use all the modern JS features.

There are a couple of features that need more than a syntax translation. Babel includes a Polyfill that emulates some of the machinery required for some complex features such as promises.

Build systems & task runners

Module bundling and transpilation are just two of the build processes that we may need in our projects. Others include code minification (to reduce file sizes), tools for analysis, and perhaps tasks that don't have anything to do with JavaScript, like image optimization or CSS/HTML pre-processing.

The management of tasks can become a laborious thing to do, and we need a way to handle it in an automated way, being able to execute everything with simpler commands. The two most popular tools for this are Grunt.js and Gulp.js, which provide a way to organize your tasks into groups in an ordered way.

For example, you can have a command like `gulp build` which may run a code linter, the transpilation process with Babel, and module bundling with Browserify. Instead of having to remember three commands and their associated arguments in order, we just execute one that will handle the whole process automatically.

Wherever you find yourself manually organizing processing steps for your project, think if it can be automatized with a task runner.

Further Reading on Gulp.js

Further reading: An Introduction to Gulp.js.

Application Architecture

Web applications have different requirements from websites. For example, while page reloads may be acceptable for a blog, that's certainly not the case for an application like Google Docs. Your application should behave as closely as possible to a desktop one. Otherwise, the usability will be compromised.

Old-style web applications were usually done by sending multiple pages from a web server, and when a lot of dynamism was needed, content was loaded via Ajax by replacing chunks of HTML according to user actions. Although it was a big step forward to a more dynamic web, it certainly had its complications. Sending HTML fragments or even whole pages on each user action represented a waste of resources — especially of time, from the user's perspective. The usability still didn't match the responsiveness of desktop applications.

Looking to improve things, we created two new methods to build web applications — from the way we present them to the user, to the way we communicate between the client and the server. Although the amount of JavaScript required for an application also increased drastically, the result is now applications that behave very closely to native ones, without page reloading or extensive waiting periods each time we click a button.

Single Page Applications (SPAs)

The most common, high-level architecture for web applications is called SPA, which stands for *Single Page Application*. SPAs are big blobs of JavaScript that contain everything the application needs to work properly. The UI is rendered entirely client-side, so no reloading is required. The only thing that changes is the data inside the application, which is usually handled with a remote API via Ajax or another asynchronous method of communication.

One downside to this approach is that the application takes longer to load for the first time. Once it has been loaded, however, transitions between views (pages) are generally a lot quicker, since it's only pure data being sent between client and server.

Universal / Isomorphic Applications

Although SPAs provide a great user experience, depending on your needs, they might not be the optimal solution — especially if you need quicker initial response times or optimal indexing by search engines.

There’s a fairly recent approach to solving these problems, called Isomorphic (or Universal) JavaScript applications. In this type of architecture, most of the code can be executed both on the server and the client. You can choose what you want to render on the server for a faster initial page load, and after that, the client takes over the rendering while the user is interacting with the app. Because pages are initially rendered on the server, search engines can index them properly.

Deployment

With modern JavaScript applications, the code you write is not the same as the code that you deploy for production: you only deploy the result of your build process. The workflow to accomplish this can vary depending on the size of your project, the number of developers working on it, and sometimes the tools/libraries you're using.

For example, if you're working alone on a simple project, each time you're ready for deployment you can just run the build process and upload the resulting files to a web server. Keep in mind that you only need to upload the resulting files from the build process (transpilation, module bundling, minification, etc.), which can be just one `.js` file containing your entire application and dependencies.

You can have a directory structure like this:

```
├── dist
│   ├── app.js
│   └── index.html
├── node_modules
├── src
│   ├── lib
│   │   ├── login.js
│   │   └── user.js
│   ├── app.js
│   └── index.html
├── gulpfile.js
├── package.json
└── README
```

You thus have all of your application files in a `src` directory, written in ES2015+, importing packages installed with npm and your own modules from a `lib` directory.

Then you can run Gulp, which will execute the instructions from a `gulpfile.js` to build your project — bundling all modules into one file (including the ones installed with npm), transpiling ES2015+ to ES5, minifying the resulted file, etc. Then you can configure it to output the result in a convenient `dist` directory.

Files That Don't Need Processing

If you have files that don’t need any processing, you can just copy them from `src` to the `dist` directory. You can configure a task for that in your build system.

Now you can just upload the files from the `dist` directory to a web server, without having to worry about the rest of the files, which are only useful for development.

Team development

If you’re working with other developers, it’s likely you’re also using a shared code repository, like GitHub, to store the project. In this case, you can run the build process right before making commits and store the result with the other files in the Git repository, to later be downloaded onto a production server.

However, storing built files in the repository is prone to errors if several developers are working together, and you might want to keep everything clean from build artifacts. Fortunately, there’s a better way to deal with that problem: you can put a service like Jenkins, Travis CI, CircleCI, etc. in the middle of the process, so it can automatically build your project after each commit is pushed to the repository. Developers only have to worry about pushing code changes without building the project first each time. The repository is also kept clean of automatically generated files, and at the end, you still have the built files available for deployment.

Conclusion

The transition from simple web pages to modern JavaScript applications can seem daunting if you've been away from web development in recent years, but I hope this article was useful as a starting point. I've linked to more in-depth articles on each topic where possible so you can explore further.

And remember that if at some point, after looking all the options available, everything seems overwhelming and messy, just keep in mind the KISS principle, and use only what you think you need and not everything you have available. At the end of the day, solving problems is what matters, not using the latest of everything.

Chapter 2: Clean Code with ES6 Default Parameters & Property Shorthands

by Moritz Kröger

Creating a method also means writing an API — whether it's for yourself, another developer on your team, or other developers using your project. Depending on the size, complexity, and purpose of your function, you have to think of default settings and the API of your input/output.

Default function parameters and property shorthands are two handy features of ES6 that can help you write your API.

ES6 Default Parameters

Let’s freshen up our knowledge quickly and take a look at the syntax again. Default parameters allow us to initialize functions with default values. A default is used when an argument is either omitted or `undefined` — meaning `null` is a valid value. A default parameter can be anything from a number to another function.

```
// Basic syntax
function multiply (a, b = 2) {
  return a * b;
}
multiply(5); // 10

// Default parameters are also available to later default
parameters
function foo (num = 1, multi = multiply(num)) {
  return [num, multi];
}
foo(); // [1, 2]
foo(6); // [6, 12]
```

A real-world example

Let’s take a basic function and demonstrate how default parameters can speed up your development and make the code better organized.

Our example method is called `createElement()`. It takes a few configuration arguments, and returns an HTML element. The API looks like this:

```
// We want a <p> element, with some text content and two classes
attached.
// Returns <p class="very-special-text super-big">Such unique
text</p>
createElement('p', {
  content: 'Such unique text',
  classNames: ['very-special-text', 'super-big']
});

// To make this method even more useful, it should always return a
default
// element when any argument is left out or none are passed at all.
createElement(); // <div class="module-text default">Very
```

```
default</div>
```

The implementation of this won't have much logic, but can become quite large due to it's default coverage.

```
// Without default parameters it looks quite bloated and
unnecessary large.
function createElement (tag, config) {
  tag = tag || 'div';
  config = config || {};

  const element = document.createElement(tag);
  const content = config.content || 'Very default';
  const text = document.createTextNode(content);
  let classNames = config.classNames;

  if (classNames === undefined) {
    classNames = ['module-text', 'default'];
  }

  element.classList.add(...classNames);
  element.appendChild(text);

  return element;
}
```

So far, so good. What's happening here? We're doing the following:

1. setting default values for both our parameters `tag` and `config`, in case they aren't passed *(note that some linters [don't like parameter reassigning](#))*
2. creating constants with the actual content (and default values)
3. checking if `classNames` is defined, and assigning a default array if not
4. creating and modifying the element before we return it.

Now let's take this function and optimize it to be cleaner, faster to write, and so that it's more obvious what its purpose is:

```
// Default all the things
function createElement (tag = 'div', {
  content = 'Very default',
  classNames = ['module-text', 'special']
} = {}) {
  const element = document.createElement(tag);
  const text = document.createTextNode(content);

  element.classList.add(...classNames);
```

```
  element.appendChild(text);

  return element;
}
```

We didn’t touch the function’s logic, but removed all default handling from the function body. The function signature now contains all defaults.

Let me further explain one part, which might be slightly confusing:

```
// What exactly happens here?
function createElement ({
  content = 'Very default',
  classNames = ['module-text', 'special']
} = {}) {
  // function body
}
```

We not only declare a default `object` parameter, but also default object *properties*. This makes it more obvious what the default configuration is supposed to look like, rather than only declaring a default object (e.g. `config = {}`) and later setting default properties. It might take some additional time to get used to it, but in the end it improves your workflow.

Of course, we could still argue with larger configurations that it might create more overhead and it’d be simpler to just keep the default handling inside of the function body.

ES6 Property Shorthands

If a method accepts large configuration objects as an argument, your code can become quite large. It’s common to prepare some variables and add them to said object. Property shorthands are *syntactic sugar* to make this step shorter and more readable:

```
const a = 'foo', b = 42, c = function () {};



// Previously we would use these constants like this.

const alphabet = {

  a: a,

  b: b,

  c: c

};



// But with the new shorthand we can actually do this now,

// which is equivalent to the above.
```

```
const alphabet = { a, b, c };
```

Shorten Your API

Okay, back to another, more common example. The following function takes some data, mutates it and calls another method:

```
function updateSomething (data = {}) {

  const target = data.target;

  const veryLongProperty = data.veryLongProperty;

  let willChange = data.willChange;




  if (willChange === 'unwantedValue') {

    willChange = 'wayBetter';

  }




  // Do more.
```

```
  useDataSomewhereElse({

    target: target,

    property: veryLongProperty,

    willChange: willChange,

    // .. more

  });

}
```

It often happens that we name variables and object property names the same. Using the property shorthand, combined with destructuring, we actually can shorten our code quite a bit:

```
function updateSomething (data = {}) {

  // Here we use destructuring to store the constants from the data
object.

  const { target, veryLongProperty: property } = data;

  let { willChange } = data;
```

```
  if (willChange === 'unwantedValue') {

    willChange = 'wayBetter';

  }



  // Do more.



  useDataSomewhereElse({ target, property, willChange });

}
```

Again, this might take a while to get used to. In the end, it’s one of those new features in JavaScript which helped me write code faster and work with cleaner function bodies.

But wait, there’s more! Property shorthands can also be applied to method definitions inside an object:

```
// Instead of writing the function keyword everytime,

const module = {
```

```
  foo: 42,

  bar: function (value) {

    // do something

  }

};



// we can just omit it and have shorter declarations

const module = {

  foo: 42,

  bar (value) {

    // do something

  }

};
```

Conclusion

Default parameters and property shorthands are a great way to make your methods more organized, and in some cases even shorter. Overall, default function parameters helped me to focus more on the actual purpose of the method without the distraction of lots of default preparations and if statements.

Property shorthands are indeed more of a cosmetic feature, but I found myself being more productive and spending less time writing all the variables, configuration objects, and function keywords.

Chapter 3: JavaScript Performance Optimization Tips: An Overview

by Ivan Čurić

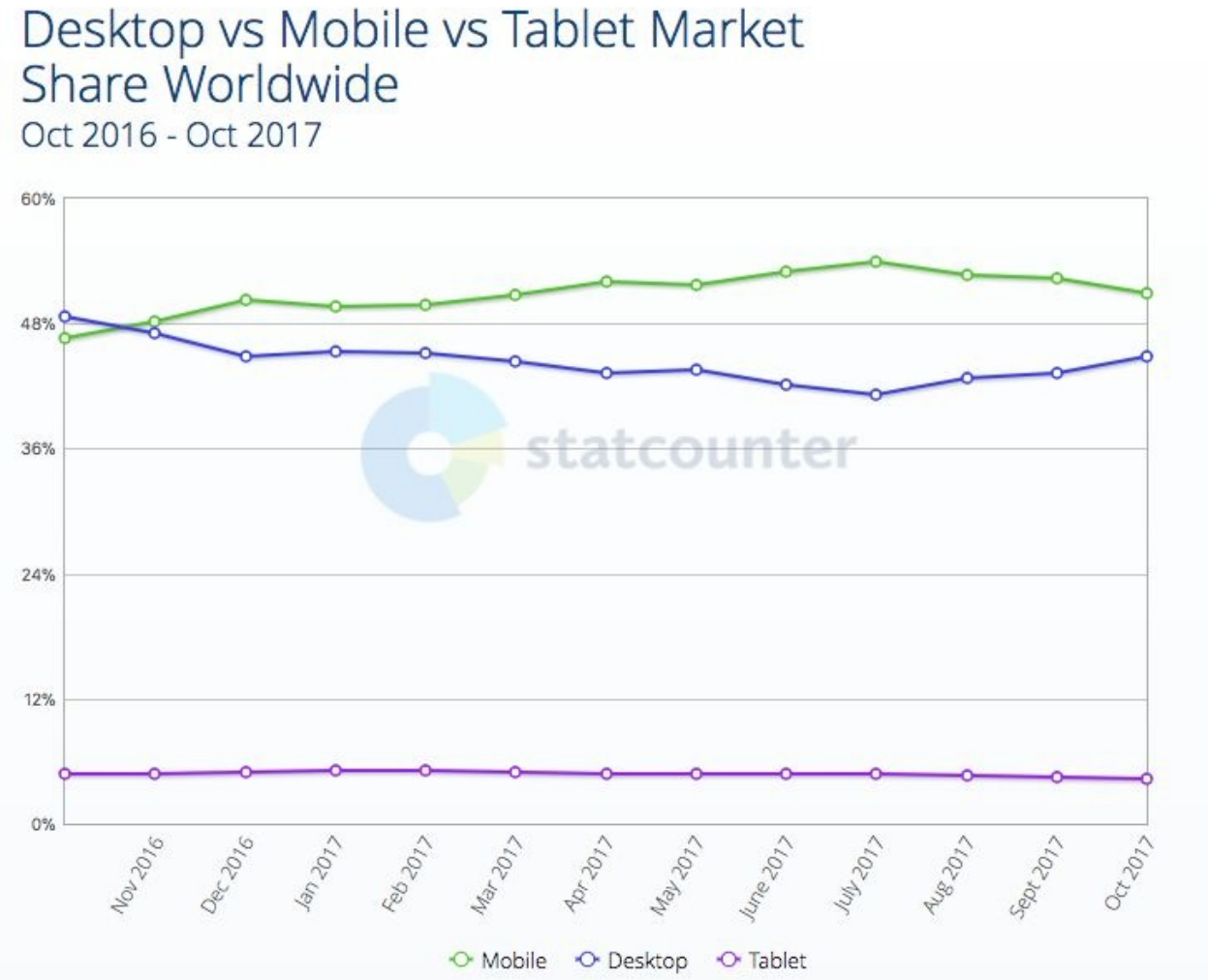
In this post, there’s lots of stuff to cover across a wide and wildly changing landscape. It’s also a topic that covers everyone’s favorite: The JS Framework of the Month™.

We’ll try to stick to the "Tools, not rules" mantra and keep the JS buzzwords to a minimum. Since we won’t be able to cover everything related to JS performance in a 2000 word article, make sure you read the references and do your own research afterwards.

But before we dive into specifics, let’s get a broader understanding of the issue by answering the following: what is considered as performant JavaScript, and how does it fit into the broader scope of web performance metrics?

Setting the Stage

First of all, let's get the following out of the way: if you're testing exclusively on your desktop device, you're excluding more than 50% of your users.

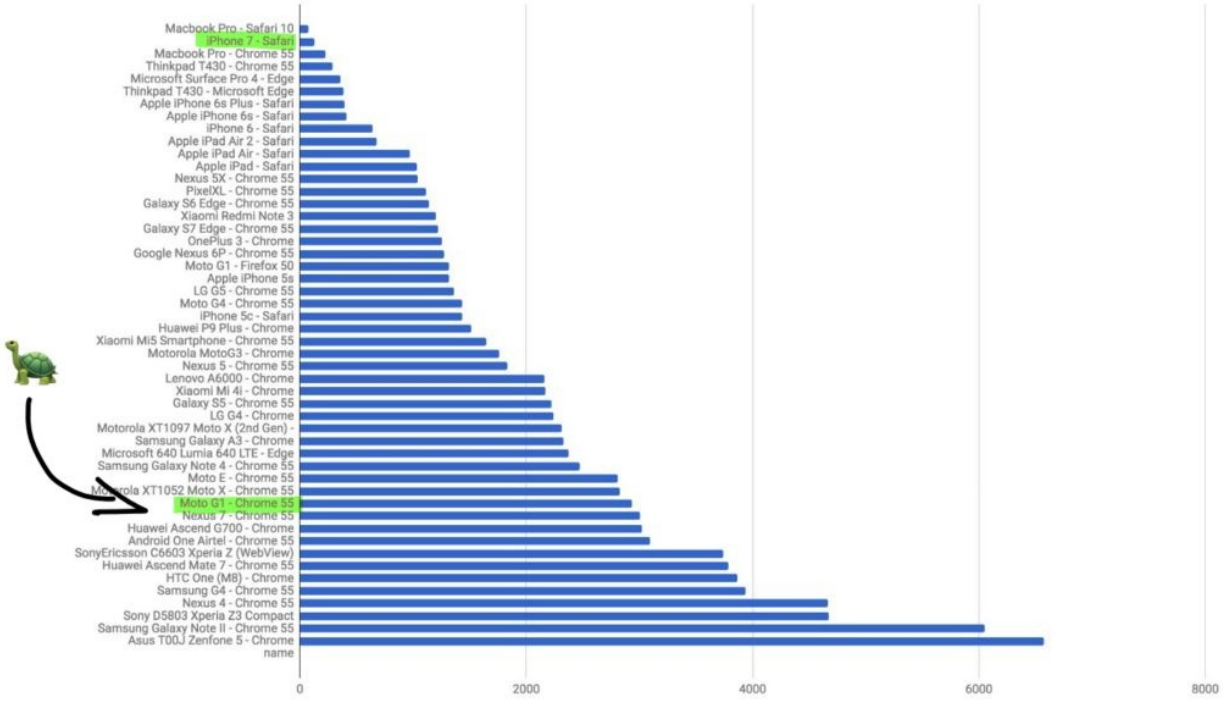


This trend will only continue to grow, as the emerging market's preferred gateway to the web is a sub-$100 Android device. The era of the desktop as the main device to access the Internet is over, and the next billion internet users will visit your sites primarily through a mobile device.

Testing in Chrome DevTools' device mode isn't a valid substitute to testing on a real device. Using CPU and network throttling helps, but it's a fundamentally different beast. Test on real devices.

Even if you *are* testing on real mobile devices, you're probably doing so on your brand spanking new $600 flagship phone. The thing is, that's not the device your users have. The median device is something along the lines of a Moto G1 --- a device with under 1GB of RAM, and a very weak CPU and GPU.

Let's see how it stacks up when parsing an average JS bundle.



Addy Osmani: Time spent in JS parse & eval for average JS.

Ouch. While this image only covers the parse and compile time of the JS (more on that later) and not general performance, it's strongly correlated and can be treated as an indicator of general JS performance.

To quote Bruce Lawson, "it's the World-Wide Web, not the Wealthy Western Web". So, your target for web performance is a device that's **~25x slower** than your MacBook or iPhone. Let that sink in for a bit. But it gets worse. Let's see what we're actually aiming for.

What Exactly is Performant JS Code?

Now that we know what our target platform is, we can answer the next question: what *is* performant JS code?

While there's no absolute classification of what defines performant code, we do have a user-centric performance model we can use as a reference: The RAIL model.

Sam Saccone: Planning for Performance: PRPL

Respond

If your app responds to a user action in under 100ms, the user perceives the response as immediate. This applies to tappable elements, but not when scrolling or dragging.

Animate

On a 60Hz monitor, we want to target a constant 60 frames per second when animating and scrolling. That results in around 16ms per frame. Out of that 16ms

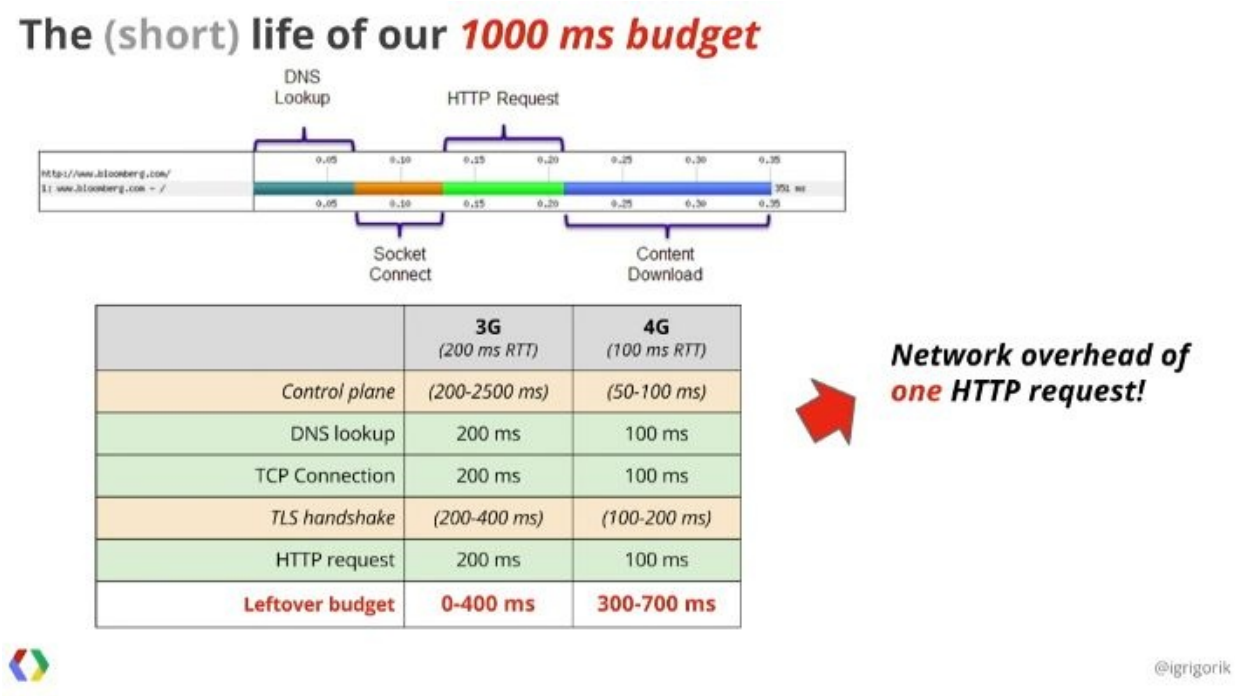
budget, you realistically have 8–10ms to do all the work, the rest taken up by the browser internals and other variances.

Idle work

If you have an expensive, continuously running task, make sure to slice it into smaller chunks to allow the main thread to react to user inputs. You shouldn't have a task that delays user input for more than 50ms.

Load

You should target a page load in under 1000ms. Anything over, and your users start getting twitchy. This is a pretty difficult goal to reach on mobile devices as it relates to the page being interactive, not just having it painted on screen and scrollable. In practice, it's even less:



Fast By Default: Modern Loading Best Practices (Chrome Dev Summit 2017)

In practice, aim for the 5s time-to-interactive mark. It's what Chrome uses in their Lighthouse audit.

Now that we know the metrics, let's have a look at some of the statistics:

- 53% of visits are abandoned if a mobile site takes more than three seconds to load
- 1 out of 2 people expect a page to load in less than 2 seconds
- 77% of mobile sites take longer than 10 seconds to load on 3G networks
- 19 seconds is the average load time for mobile sites on 3G networks.

And a bit more, courtesy of Addy Osmani:

- apps became interactive in 8 seconds on desktop (using cable) and 16 seconds on mobile (Moto G4 over 3G)
- at the median, developers shipped 410KB of gzipped JS for their pages.

Feeling sufficiently frustrated? Good. Let’s get to work and fix the web.

Context is Everything

You might have noticed that the main bottleneck is the time it takes to load up your website. Specifically, the JavaScript download, parse, compile and execution time. There's no way around it but to load less JavaScript and load smarter.

But what about the actual work that your code does aside from just booting up the website? There has to be some performance gains there, right?

Before you dive into optimizing your code, consider what you're building. Are you building a framework or a VDOM library? Does your code need to do thousands of operations per second? Are you doing a time-critical library for handling user input and/or animations? If not, you may want to shift your time and energy somewhere more impactful.

It's not that writing performant code doesn't matter, but it usually makes little to no impact in the grand scheme of things, especially when talking about microoptimizations. So, before you get into a Stack Overflow argument about `.map` vs `.forEach` vs `for` loops by comparing results from JSperf.com, make sure to see the forest and not just the trees. 50k ops/s might sound 50× better than 1k ops/s on paper, but it won't make a difference in most cases.

Parsing, Compiling and Executing

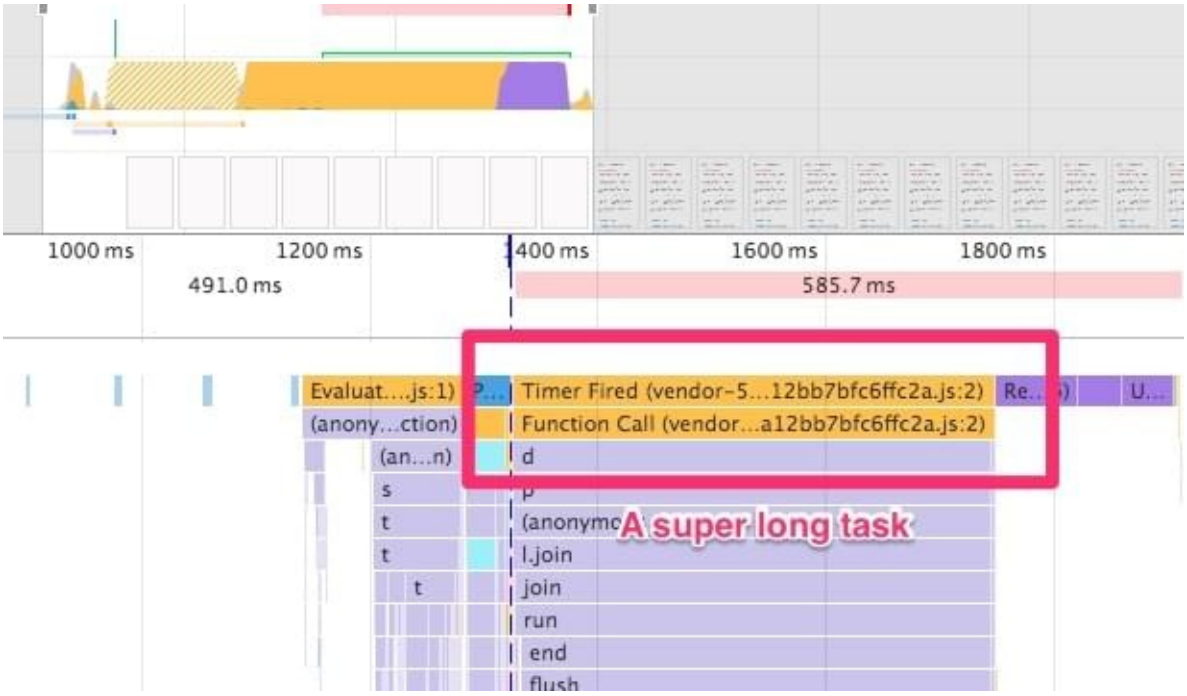
Fundamentally, the problem of most non-performant JS is not running the code itself, but all the steps that have to be taken *before* the code even starts executing.

We're talking about levels of abstraction here. The CPU in your computer runs machine code. Most of the code you're running on your computer is in the compiled binary format. (I said *code* rather than *programs*, considering all the Electron apps these days.) Meaning, all the OS-level abstractions aside, it runs natively on your hardware, no prep-work needed.

JavaScript is not pre-compiled. It arrives (via a relatively slow network) as readable code in your browser which is, for all intents and purposes, the "OS" for your JS program.

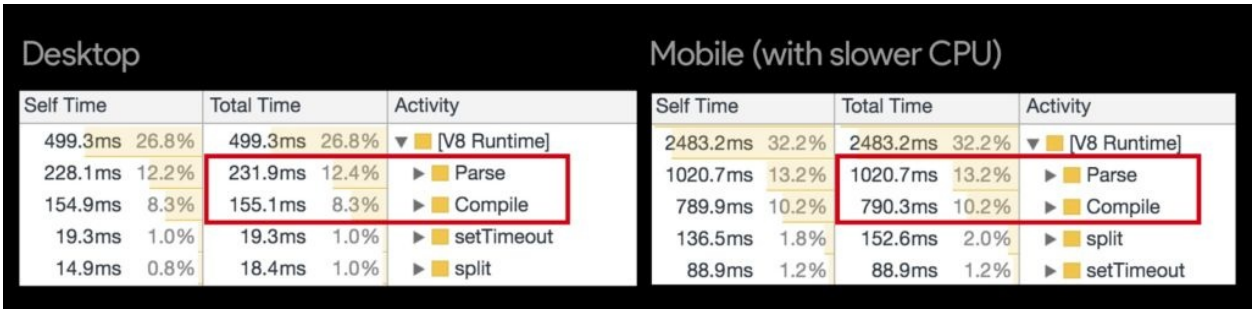
That code first needs to be parsed --- that is, read and turned into an computer-indexable structure that can be used for compiling. It then gets compiled into bytecode and finally machine code, before it can be executed by your device/browser.

Another *very* important thing to mention is that JavaScript is single-threaded, and runs on the browser's main thread. This means that only one process can run at a time. If your DevTools performance timeline is filled with yellow peaks, running your CPU at 100%, you'll have long/dropped frames, janky scrolling and all other kind of nasty stuff.



Paul Lewis: [When everything's important, nothing is!](#).

So there's all this work that needs to be done before your JS starts working. Parsing and compiling takes up to 50% of the total time of JS execution in Chrome's V8 engine.



Addy Osmani: [JavaScript Start-up Performance](#).

There are two things you should take away from this section:

1. While not necessarily linearly, JS parse time scales with the bundle size. The less JS you ship, the better.
2. Every JS framework you use (React, Vue, Angular, Preact…) is another level of abstraction (unless it's a precompiled one, like [Svelte](#)). Not only will it increase your bundle size, but also slow down your code since you're not talking directly to the browser.

There are ways to mitigate this, such as using service workers to do jobs in the background and on another thread, using asm.js to write code that's more easily compiled to machine instructions, but that's a whole 'nother topic.

What you can do, however, is avoid using JS animation frameworks for everything and read up on what triggers paints and layouts. Use the libraries only when there's absolutely no way to implement the animation using regular CSS transitions and animations.

Even though they may be using CSS transitions, composited properties and `requestAnimationFrame()`, they're still running in JS, on the main thread. They're basically just hammering your DOM with inline styles every 16ms, since there's not much else they can do. You need to make sure all your JS will be done executing in under 8ms per frame in order to keep the animations smooth.

CSS animations and transitions, on the other hand, are running off the main thread --- on the GPU, if implemented performantly, without causing relayouts/reflows.

Considering that most animations are running either during loading or user interaction, this can give your web apps the much-needed room to breathe.

The Web Animations API is an upcoming feature set that will allow you to do performant JS animations off the main thread, but for now, stick to CSS transitions and techniques like FLIP.

Bundle Sizes are Everything

Today it's all about bundles. Gone are the times of Bower and dozens of `<script>` tags before the closing `</body>` tag.

Now it's all about `npm install`-ing whatever shiny new toy you find on NPM, bundling them together with Webpack in a huge single 1MB JS file and hammering your users' browser to a crawl while capping off their data plans.

Try shipping less JS. You might not need the entire Lodash library for your project. Do you absolutely *need* to use a JS framework? If yes, have you considered using something other than React, such as Preact or HyperHTML, which are less than 1/20 the size of React? Do you need TweenMax for that scroll-to-top animation? The convenience of npm and isolated components in frameworks comes with a downside: the first response of developers to a problem has become to throw more JS at it. When all you have is a hammer, everything looks like a nail.

When you're done pruning the weeds and shipping less JS, try shipping it *smarter*. Ship what you need, when you need it.

Webpack 3 has *amazing* features called code splitting and dynamic imports. Instead of bundling all your JS modules into a monolithic `app.js` bundle, it can automatically split the code using the `import()` syntax and load it asynchronously.

You don't need to use frameworks, components and client-side routing to gain the benefit of it, either. Let's say you have a complex piece of code that powers your `.mega-widget`, which can be on any number of pages. You can simply write the following in your main JS file:

```
if (document.querySelector('.mega-widget')) {
    import('./mega-widget');
}
```

If your app finds the widget on the page, it will dynamically load the required supporting code. Otherwise, all's good.

Also, Webpack needs its own runtime to work, and it injects it into all the .js

files it generates. If you use the `commonChunks` plugin, you can use the following to extract the runtime into its own chunk:

```
new webpack.optimize.CommonsChunkPlugin({
  name: 'runtime',
}),
```

It will strip out the runtime from all your other chunks into its own file, in this case named `runtime.js`. Just make sure to load it before your main JS bundle. For example:

```
<script src="runtime.js">
<script src="main-bundle.js">
```

Then there's the topic of transpiled code and polyfills. If you're writing modern (ES6+) JavaScript, you're probably using Babel to transpile it into ES5 compatible code. Transpiling not only increases file size due to all the verbosity, but also complexity, and it often has performance regressions compared to native ES6+ code.

Along with that, you're probably using the `babel-polyfill` package and `whatwg-fetch` to patch up missing features in older browsers. Then, if you're writing code using `async/await`, you also transpile it using generators needed to include the `regenerator-runtime`...

The point is, you add almost 100 kilobytes to your JS bundle, which has not only a huge file size, but also a huge parsing and executing cost, in order to support older browsers.

There's no point in punishing people who are using modern browsers, though. An approach I use, and which Philip Walton covered in this article, is to create two separate bundles and load them conditionally. Babel makes this easy with `babel-preset-env`. For instance, you have one bundle for supporting IE 11, and the other without polyfills for the latest versions of modern browsers.

A dirty but efficient way is to place the following in an inline script:

```
(function() {
  try {
    new Function('async () => {}')();
  } catch (error) {
    // create script tag pointing to legacy-bundle.js;
```

```
    return;
  }
  // create script tag pointing to modern-bundle.js;;
})();
```

If the browser isn’t able to evaluate an `async` function, we assume that it’s an old browser and just ship the polyfilled bundle. Otherwise, the user gets the neat and modern variant.

Conclusion

What we would like you to gain from this article is that JS is expensive and should be used sparingly.

Make sure you test your website’s performance on low-end devices, under real network conditions. Your site should load fast and be interactive as soon as possible. This means shipping less JS, and shipping faster by any means necessary. Your code should always be minified, split into smaller, manageable bundles and loaded asynchronously whenever possible. On the server side, make sure it has HTTP/2 enabled for faster parallel transfers and gzip/Brotli compression to drastically reduce the transfer sizes of your JS.

Chapter 4: JavaScript Design Patterns: The Singleton

by Samier Saeed

In this chapter, we'll dig into the best way to implement a singleton in JavaScript, looking at how this has evolved with the rise of ES6.

Among languages used in widespread production, JavaScript is by far the most quickly evolving, looking less like its earliest iterations and more like Python, with every new spec put forth by ECMA International. While the changes have their fair share of detractors, the new JavaScript does succeed in making code easier to read and reason about, easier to write in a way that adheres to software engineering best practices (particularly the concepts of modularity and SOLID principles), and easier to assemble into canonical software design patterns.

Explaining ES6

ES6 (aka ES2015) was the first major update to the language since ES5 was standardized in 2009. Almost all modern browsers support ES6. However, if you need to accommodate older browsers, ES6 code can easily be transpiled into ES5 using a tool such as Babel. ES6 gives JavaScript a ton of new features, including a superior syntax for classes, and new keywords for variable declarations. You can learn more about it by perusing SitePoint articles on the subject.

What Is a Singleton

In case you're unfamiliar with the singleton pattern, it is, at its core, a design pattern that restricts the instantiation of a class to one object. Usually, the goal is to manage global application state. Some examples I've seen or written myself include using a singleton as the source of config settings for a web app, on the client side for anything initiated with an API key (you usually don't want to risk sending multiple analytics tracking calls, for example), and to store data in memory in a client-side web application (e.g. stores in Flux).

A singleton should be immutable by the consuming code, and there should be no danger of instantiating more than one of them.

Are Singletons Bad?

There are scenarios when singletons might be bad, and arguments that they are, in fact, always bad. For that discussion, you can check out this helpful article on the subject.

The Old Way of Creating a Singleton in JavaScript

The old way of writing a singleton in JavaScript involves leveraging closures and immediately invoked function expressions . Here's how we might write a (very simple) store for a hypothetical Flux implementation the old way: `var UserStore = (function(){ var _data = []; function add(item){ _data.push(item); } function get(id){ return _data.find((d) => { return d.id === id; }); } return { add: add, get: get }; }());`

When that code is interpreted, `UserStore` will be set to the result of that immediately invoked function — an object that exposes two functions, but that does not grant direct access to the collection of data.

However, this code is more verbose than it needs to be, and also doesn't give us the immutability we desire when making use of singletons. Code executed later could modify either one of the exposed functions, or even redefine `UserStore` altogether. Moreover, the modifying/offending code could be anywhere! If we got bugs as a result of unexpected modification of `UsersStore`, tracking them down in a larger project could prove very frustrating.

There are more advanced moves you could pull to mitigate some of these downsides, as specified in this article by Ben Cherry. (His goal is to create modules, which just happen to be singletons, but the pattern is the same.) But those add unneeded complexity to the code, while still failing to get us exactly what we want.

The New Way(s)

By leveraging ES6 features, mainly modules and the new `const` variable declaration, we can write singletons in ways that are not only more concise, but which better meet our requirements.

Let's start with the most basic implementation. Here's (a cleaner and more powerful) modern interpretation of the above example:

```
const _data = [];

const UserStore = {
  add: item => _data.push(item),
  get: id => _data.find(d => d.id === id)
}

Object.freeze(UserStore);
export default UserStore;
```

As you can see, this way offers an improvement in readability. But where it really shines is in the constraint imposed upon code that consumes our little singleton module here: the consuming code cannot reassign `UserStore` because of the `const` keyword. And as a result of our use of Object.freeze, its methods cannot be changed, nor can new methods or properties be added to it. Furthermore, because we're taking advantage of ES6 modules, we know exactly where `UserStore` is used.

Now, here we've made `UserStore` an object literal. Most of the time, going with an object literal is the most readable and concise option. However, there are times when you might want to exploit the benefits of going with a traditional class. For example, stores in Flux will all have a lot of the same base functionality. Leveraging traditional object-oriented inheritance is one way to get that repetitive functionality while keeping your code DRY.

Here's how the implementation would look if we wanted to utilize ES6 classes:

```
class UserStore {
  constructor(){
    this._data = [];
  }
```

```
  add(item){
    this._data.push(item);
  }

  get(id){
    return this._data.find(d => d.id === id);
  }
}

const instance = new UserStore();
Object.freeze(instance);

export default instance;
```

This way is slightly more verbose than using an object literal, and our example is so simple that we don't really see any benefits from using a class (though it will come in handy in the final example).

One benefit to the class route that might not be obvious is that, if this is your front-end code, and your back end is written in C# or Java, you can employ a lot of the same design patterns in your client-side application as you do on the back end, and increase your team's efficiency (if you're small and people are working full-stack). Sounds soft and hard to measure, but I've experienced it firsthand working on a C# application with a React front end, and the benefit is real.

It should be noted that, technically, the immutability and non-overridability of the singleton using both of these patterns can be subverted by the motivated provocateur. An object literal can be copied, even if it itself is `const`, by using Object.assign. And when we export an instance of a class, though we aren't directly exposing the class itself to the consuming code, the constructor of any instance is available in JavaScript and can be invoked to create new instances. Obviously, though, that all takes at least a little bit of effort, and hopefully your fellow devs aren't so insistent on violating the singleton pattern.

But let's say you wanted to be extra sure that nobody messed with the singleness of your singleton, and you also wanted it to match the implementation of singletons in the object-oriented world even more closely. Here's something you could do:

```
class UserStore {
  constructor(){
   if(! UserStore.instance){
     this._data = [];
```

```
    UserStore.instance = this;
   }

   return UserStore.instance;
  }

 //rest is the same code as preceding example

}

const instance = new UserStore();
Object.freeze(instance);

export default instance;
```

By adding the extra step of holding a reference to the instance, we can check whether or not we've already instantiated a `UserStore`, and if we have, we won't create a new one. As you can see, this also makes good use of the fact that we've made `UserStore` a class.

Conclusion

There are no doubt plenty of developers who have been using the old singleton/module pattern in JavaScript for a number of years, and who find it works quite well for them. Nevertheless, because finding better ways to do things is so central to the ethos of being a developer, hopefully we see cleaner and easier-to-reason-about patterns like this one gaining more and more traction. Especially once it becomes easier and more commonplace to utilize ES6+ features.

Chapter 5: JavaScript Object Creation: Patterns and Best Practices

by Jeff Mott

In this chapter, I'm going to take you on a tour of the various styles of JavaScript object creation, and how each builds on the others in incremental steps.

JavaScript has a multitude of styles for creating objects, and newcomers and veterans alike can feel overwhelmed by the choices and unsure which they should use. But despite the variety and how different the syntax for each may look, they're more similar than you probably realize.

Object Literals

The first stop on our tour is the absolute simplest method of JavaScript object creation — the object literal. JavaScript touts that objects can be created “ex nilo”, out of nothing — no class, no template, no prototype — just *poof!*, an object with methods and data:

```
var o = {
  x: 42,
  y: 3.14,
  f: function() {},
  g: function() {}
};
```

But there’s a drawback. If we need to create the same type of object in other places, then we’ll end up copy-pasting the object’s methods, data, and initialization. We need a way to create not just the one object, but a family of objects.

Factory Functions

The next stop on our JavaScript object creation tour is the factory function. This is the absolute simplest way to create a family of objects that share the same structure, interface, and implementation. Rather than creating an object literal directly, instead we return an object literal from a function. This way, if we need to create the same type of object multiple times or in multiple places, we only need to invoke a function:

```
function thing() {
  return {
    x: 42,
    y: 3.14,
    f: function() {},
    g: function() {}
  };
}

var o = thing();
```

But there's a drawback. This approach of JavaScript object creation can cause memory bloat, because each object contains its own unique copy of each function. Ideally, we want every object to *share* just one copy of its functions.

Prototype Chains

JavaScript gives us a built-in mechanism to share data across objects, called the **prototype chain**. When we access a property on an object, it can fulfill that request by delegating to some other object. We can use that and change our factory function so that each object it creates contains only the data unique to that particular object, and delegate all other property requests to a single, shared object:

```
var thingPrototype = {

  f: function() {},

  g: function() {}

};




function thing() {

  var o = Object.create(thingPrototype);




  o.x = 42;

  o.y = 3.14;
```

```
  return o;

}



var o = thing();
```

In fact, this is such a common pattern that the language has built-in support for it. We don’t need to create our own shared object (the prototype object). Instead, a prototype object is created for us automatically alongside every function, and we can put our shared data there:

```
thing.prototype.f = function() {};

thing.prototype.g = function() {};




function thing() {

  var o = Object.create(thing.prototype);
```

```
  o.x = 42;

  o.y = 3.14;




  return o;

}




var o = thing();
```

But there’s a drawback. This is going to result in some repetition. The first and last lines of the `thing` function are going to be repeated almost verbatim in every such delegating-to-prototype factory function.

ES5 Classes

We can isolate the repetitive lines by moving them into their own function. This function would create an object that delegates to some other arbitrary function's prototype, then invoke that function with the newly created object as an argument, and finally return the object:

```
function create(fn) {
  var o = Object.create(fn.prototype);

  fn.call(o);

  return o;
}

// ...

Thing.prototype.f = function() {};
Thing.prototype.g = function() {};

function Thing() {
  this.x = 42;
  this.y = 3.14;
}

var o = create(Thing);
```

In fact, this too is such a common pattern that the language has some built-in support for it. The `create` function we defined is actually a rudimentary version of the `new` keyword, and we can drop-in replace `create` with `new`:

```
Thing.prototype.f = function() {};
Thing.prototype.g = function() {};

function Thing() {
  this.x = 42;
  this.y = 3.14;
}

var o = new Thing();
```

We've now arrived at what we commonly call "ES5 classes". They're object creation functions that delegate shared data to a prototype object and rely on the `new` keyword to handle repetitive logic.

But there’s a drawback. It’s verbose and ugly, and implementing inheritance is even more verbose and ugly.

ES6 Classes

A relatively recent addition to JavaScript is ES6 classes, which offer a significantly cleaner syntax for doing the same thing:

```
class Thing {
  constructor() {
    this.x = 42;
    this.y = 3.14;
  }

  f() {}
  g() {}
}

const o = new Thing();
```

Comparison

Over the years, we JavaScripters have had an on-and-off relationship with the prototype chain, and today the two most common styles you're likely to encounter are the class syntax, which relies heavily on the prototype chain, and the factory function syntax, which typically doesn't rely on the prototype chain at all. The two styles differ — but only slightly — in performance and features.

Performance

JavaScript engines are so heavily optimized today that it's nearly impossible to look at our code and reason about what will be faster. Measurement is crucial. Yet sometimes even measurement can fail us. Typically, an updated JavaScript engine is released every six weeks, sometimes with significant changes in performance, and any measurements we had previously taken, and any decisions we made based on those measurements, go right out the window. So, my rule of thumb has been to favor the most official and most widely used syntax, under the presumption that it will receive the most scrutiny and be the most performant *most of the time*. Right now, that's the class syntax, and as I write this, the class syntax is roughly 3x faster than a factory function returning a literal.

Features

What few feature differences there were between classes and factory functions evaporated with ES6. Today, both factory functions and classes can enforce truly private data—factory functions with closures and classes with weak maps. Both can achieve multiple-inheritance factory functions by mixing other properties into their own object, and classes also by mixing other properties into their prototype, or with class factories, or with proxies. Both factory functions and classes can return any arbitrary object if need be. And both offer a simple syntax.

Conclusion

All things considered, my preference for JavaScript object creation is to use the class syntax. It’s standard, it’s simple and clean, it’s fast, and it provides every feature that once upon a time only factories could deliver.

Chapter 6: Best Practices for Using Modern JavaScript Syntax

by M. David Green

Modern JavaScript is evolving quickly to meet the changing needs of new frameworks and environments. Understanding how to take advantage of those changes can save you time, improve your skill set, and mark the difference between good code and great code.

Knowing what modern JavaScript is trying to do can help you decide when to use the new syntax to your best advantage, and when it still makes sense to use traditional techniques.

Something Solid to Cling To

I don’t know anybody who isn’t confused at the state of JavaScript these days, whether you’re new to JavaScript, or you’ve been coding with it for a while. So many new frameworks, so many changes to the language, and so many contexts to consider. It’s a wonder that anybody gets any work done, with all of the new things that we have to learn every month.

I believe that the secret to success with any programming language, no matter how complex the application, is getting back to the basics. If you want to understand Rails, start by working on your Ruby skills, and if you want to use immutables and unidirectional data flow in isomorphic React with webpack (or whatever the cool nerds are doing these days) start by knowing your core JavaScript.

Understanding how the language itself works is much more practical than familiarizing yourself with the latest frameworks and environments. Those change faster than the weather. And with JavaScript, we have a long history of thoughtful information online about how JavaScript was created and how to use it effectively.

The problem is that some of the new techniques that have come around with the latest versions of JavaScript make some of the old rules obsolete. But not all of them! Sometimes a new syntax may replace a clunkier one to accomplish the same task. Other times the new approach may seem like a simpler drop-in replacement for the way we used to do things, but there are subtle differences, and it’s important to be aware of what those are.

A Spoonful of Syntactic Sugar

A lot of the changes in JavaScript in recent years have been described as syntactic sugar for existing syntax. In many cases, the syntactic sugar can help the medicine go down for Java programmers learning how to work with JavaScript, or for the rest of us we just want a cleaner, simpler way to accomplish something we already knew how to do. Other changes seem to introduce magical new capabilities.

But if you try to use modern syntax to recreate a familiar old technique, or stick it in without understanding how it actually behaves, you run the risk of:

- having to debug code that worked perfectly before
- introducing subtle mistakes that may catch you at runtime
- creating code that fails silently when you least expect it.

In fact, several of the changes that appear to be drop-in replacements for existing techniques actually behave differently from the code that they supposedly replace. In many cases, it can make more sense to use the original, older style to accomplish what you're trying to do. Recognizing when that's happening, and knowing how to make the choice, is critical to writing effective modern JavaScript.

When Your `const` Isn't Consistent

Modern JavaScript introduced two new keywords, `let` and `const`, which effectively replace the need for `var` when declaring variables in most cases. But they don't behave exactly the same way that `var` does.

In traditional JavaScript, it was always a clean coding practice to declare your variables with the `var` keyword before using them. Failure to do that meant that the variables you declared could be accessed in the global scope by any scripts that happened to run in the same context. And because traditional JavaScript was frequently run on webpages where multiple scripts might be loaded simultaneously, that meant that it was possible for variables declared in one script to leak into another.

The cleanest drop-in replacement for `var` in modern JavaScript is `let`. But `let` has a few idiosyncrasies that distinguish it from `var`. Variable declarations with `var` were always hoisted to the top of their containing scope by default, regardless of where they were placed inside of that scope. That meant that even a deeply nested variable could be considered declared and available right from the beginning of its containing scope. The same is not true of `let` or `const`.

```
console.log(usingVar); // undefined
var usingVar = "defined";
console.log(usingVar); // "defined"

console.log(usingLet); // error
let usingLet = "defined"; // never gets executed
console.log(usingLet); // never gets executed
```

When you declare a variable using `let` or `const`, the scope for that variable is limited to the local block where it's declared. A block in JavaScript is distinguished by a set of curly braces `{}`, such as the body of a function or the executable code within a loop.

This is a great convenience for block-scoped uses of variables such as iterators and loops. Previously, variables declared within loops would be available to the containing scope, leading to potential confusion when multiple counters might use the same variable name. However `let` can catch you by surprise if you expect your variable declared somewhere inside of one block of your script to be

available elsewhere.

```
for (var count = 0; count < 5; count++) {
  console.log(count);
} // outputs the numbers 0 - 4 to the console
console.log(count); // 5


for (let otherCount = 0; otherCount < 5; otherCount++) {
  console.log(otherCount);
} // outputs the numbers 0 - 4 to the console
console.log(otherCount); // error, otherCount is undefined
```

The other alternative declaration is `const`, which is supposed to represent a constant. But it's not completely constant.

A `const` can't be declared without a value, unlike a `var` or `let` variable.

```
var x; // valid
let y; //valid
const z; // error
```

A `const` will also throw an error if you try to set it to a new value after it's been declared:

```
const z = 3; // valid
z = 4; // error
```

But if you expect your `const` to be immutable in all cases, you may be in for a surprise when an object or an array declared as a `const` lets you alter its content.

```
const z = []; // valid
z.push(1); // valid, and z is now [1]
z = [2] // error
```

For this reason, I remain skeptical when people recommend using `const` constantly in place of `var` for all variable declarations, even when you have every intention of never altering them after they've been declared.

While it's a good practice to treat your variables as immutable, JavaScript won't enforce that for the contents of a reference variables like arrays and objects declared with `const` without some help from external scripts. So the `const` keyword may make casual readers and newcomers to JavaScript expect more protection than it actually provides.

I’m inclined to use `const` for simple number or string variables I want to initialize and never alter, or for named functions and classes that I expect to define once and then leave closed for modification. Otherwise, I use `let` for most variable declarations — especially those I want to be bounded by the scope in which they were defined. I haven’t found the need to use `var` lately, but if I wanted a declaration to break scope and get hoisted to the top of my script, that’s how I would do it.

Limiting the Scope of the Function

Traditional functions, defined using the `function` keyword, can be called to execute a series of statements defined within a block on any parameters passed in, and optionally return a value: `function doSomething(param) {`

`return(`Did it: ${param}`); }`

`console.log(doSomething("Hello")); // "Did it: Hello"`

They can also be used with the `new` keyword to construct objects with prototypal inheritance, and that definition can be placed anywhere in the scope where they might be called: `function Animal(name) {`

`this.name = name;`

`}`

`let cat = new Animal("Fluffy"); console.log(`My cat's name is ${cat.name}.`); // "My cat's name is Fluffy."`

Functions used in either of these ways can be defined before or after they're called. It doesn't matter to JavaScript.

```
console.log(doSomething("Hello")); // "Did it: Hello"




let cat = new Animal("Fluffy"); console.log(`My cat's name is
${cat.name}.`); // "My cat's name is Fluffy."




function doSomething(param) {


  return(`Did it: ${param}`); }
```

```
function Animal(name) {

  this.name = name;

}
```

A traditional function also creates its own context, defining a value for `this` that exists only within the scope of the statement body. Any statements or sub-functions defined within it are executing, and optionally allowing us to bind a value for `this` when calling the function.

That's a lot for the keyword to do, and it's usually more than a programmer needs in any one place. So modern JavaScript split out the behavior of the traditional function into arrow functions and classes.

Getting to Class on Time

One part of the traditional `function` has been taken over by the `class` keyword. This allows programmers to choose whether they would prefer to follow a more functional programming paradigm with callable arrow functions, or use a more object-oriented approach with classes to substitute for the prototypal inheritance of traditional functions.

Classes in JavaScript look and act a lot like simple classes in other object-oriented languages, and may be an easy stepping stone for Java and C++ developers looking to expand into JavaScript as JavaScript expands out to the server.

One difference between functions and classes when doing object-oriented programming in JavaScript is that classes in JavaScript require forward declaration, the way they do in C++ (although not in Java). That is, a `class` needs to be declared in the script before it is instantiated with a `new` keyword. Prototypal inheritance using the `function` keyword works in JavaScript even if

it’s defined later in the script, since a `function` declaration is automatically hoisted to the top, unlike a `class`.

```
// Using a function to declare and instantiate an object (hoisted)
let aProto = new Proto("Myra"); aProto.greet(); // "Hi Myra"




function Proto(name) {


  this.name = name;


  this.greet = function() {


    console.log(`Hi ${this.name}`); };


};




// Using a class to declare and instantiate an object (not hoisted)
class Classy {


  constructor(name) {


    this.name = name;


  }
```

```
  greet() {

    console.log(`Hi ${this.name}`); }

};




let aClassy = new Classy("Sonja"); aClassy.greet(); // "Hi Sonja"
```

Pointed Differences with Arrow Functions

The other aspect of traditional functions can now be accessed using arrow functions, a new syntax that allows you to write a callable function more concisely, to fit more neatly inside a callback. In fact, the simplest syntax for an arrow function is a single line that leaves off the curly braces entirely, and automatically returns the result of the statement executed: `const traditional = function(data) {`

`return (`${data} from a traditional function`); }`

`const arrow = data => `${data} from an arrow function`;`

`console.log(traditional("Hi")); // "Hi from a traditional function"`

`console.log(arrow("Hi")); // "Hi from an arrow function"`

Arrow functions encapsulate several qualities that can make calling them more convenient, and leave out other behavior that isn't as useful when calling a function. They are not drop-in replacements for the more versatile traditional `function` keyword.

For example, an arrow function inherits both `this` and `arguments` from the contexts in which it's called. That's great for situations like event handling or

`setTimeout` when a programmer frequently wants the behavior being called to apply to the context in which is it was requested. Traditional functions have forced programmers to write convoluted code that binds a function to an existing `this` by using `.bind(this)`. None of that is necessary with arrow functions.

```
class GreeterTraditional {


  constructor() {


    this.name = "Joe";


  }


  greet() {


    setTimeout(function () {


      console.log(`Hello ${this.name}`); }, 1000); // inner
function has its own this with no name }


}


let greeterTraditional = new GreeterTraditional();
greeterTraditional.greet(); // "Hello "







class GreeterBound {
```

```
  constructor() {

    this.name = "Steven";

  }

  greet() {

    setTimeout(function () {

      console.log(`Hello ${this.name}`); }.bind(this), 1000); //
passing this from the outside context }

}

let greeterBound = new GreeterBound(); // "Hello Steven"

greeterBound.greet();




class GreeterArrow {

  constructor() {

    this.name = "Ravi";
```

```
  }


  greet() {


    setTimeout(() => {


      console.log(`Hello ${this.name}`); }, 1000); // arrow
function inherits this by default }


}


let greeterArrow = new GreeterArrow(); greeterArrow.greet(); //
"Hello Ravi"
```

Understand What You’re Getting

It’s not all just syntactic sugar. A lot of the new changes in JavaScript have been introduced because new functionality was needed. But that doesn’t mean that the old reasons for JavaScript’s traditional syntax have gone away. Often it makes sense to continue using the traditional JavaScript syntax, and sometimes using the new syntax can make your code much faster to write and easier to understand.

Check out those online tutorials you’re following. If the writer is using `var` to initialize all of the variables, ignoring classes in favor of prototypal inheritance, or relying on `function` statements in callbacks, you can expect the rest of the syntax to be based on older, traditional JavaScript. And that’s fine. There’s still a lot that we can learn and apply today from the traditional ways the JavaScript has always been taught and used. But if you see `let` and `const` in the intializations, arrow functions in callbacks, and classes as the basis for object-oriented patterns, you’ll probably also see other modern JavaScript code in the examples.

The best practice in modern JavaScript is paying attention to what the language is actually doing. Depending on what you’re used to, it may not always be obvious. But think about what the code you’re writing is trying to accomplish, where you’re going to need to deploy it, and who will be modifying it next. Then decide for yourself what the best approach would be.

Chapter 7: Flow Control in Modern JS: Callbacks to Promises to Async/Await

by Craig Buckler

JavaScript is regularly claimed to be *asynchronous*. What does that mean? How does it affect development? How has the approach changed in recent years?

Consider the following code:

```
result1 = doSomething1();
result2 = doSomething2(result1);
```

Most languages process each line *synchronously*. The first line runs and returns a result. The second line runs once the first has finished *regardless of how long it takes*.

Single-thread Processing

JavaScript runs on a single processing thread. When executing in a browser tab, everything else stops. This is necessary because changes to the page DOM can't occur on parallel threads; it would be dangerous to have one thread redirecting to a different URL while another attempts to append child nodes.

This is rarely evident to the user, because processing occurs quickly in small chunks. For example, JavaScript detects a button click, runs a calculation, and updates the DOM. Once complete, the browser is free to process the next item on the queue.

Other Languages

Other languages such as PHP also use a single thread but may be managed by by a multi-threaded server such as Apache. Two requests to the same PHP page at the same time can initiate two threads running isolated instances of the PHP runtime.

Going Asynchronous with Callbacks

Single threads raise a problem. What happens when JavaScript calls a “slow” process such as an Ajax request in the browser or a database operation on the server? That operation could take several seconds — *even minutes*. A browser would become locked while it waited for a response. On the server, a Node.js application would not be able to process further user requests.

The solution is asynchronous processing. Rather than wait for completion, a process is told to call another function when the result is ready. This is known as a *callback*, and it’s passed as an argument to any asynchronous function. For example: `doSomethingAsync(callback1); console.log('finished'); // call when doSomethingAsync completes function callback1(error) { if (!error) console.log('doSomethingAsync complete'); }`

`doSomethingAsync()` accepts a callback function as a parameter (only a reference to that function is passed so there’s little overhead). It doesn’t matter how long `doSomethingAsync()` takes; all we know is that `callback1()` will be executed at some point in the future. The console will show: `finished doSomethingAsync complete`

Callback Hell

Often, a callback is only ever called by one asynchronous function. It’s therefore possible to use concise, anonymous inline functions:

```
doSomethingAsync(error => {
  if (!error) console.log('doSomethingAsync complete');
});
```

A series of two or more asynchronous calls can be completed in series by nesting callback functions. For example:

```
async1((err, res) => {
  if (!err) async2(res, (err, res) => {
    if (!err) async3(res, (err, res) => {
      console.log('async1, async2, async3 complete.');
    });
  });
});
```

Unfortunately, this introduces **callback hell** — a notorious concept that even has its own web page! The code is difficult to read, and will become worse when error-handling logic is added.

Callback hell is relatively rare in client-side coding. It can go two or three levels deep if you're making an Ajax call, updating the DOM and waiting for an animation to complete, but it normally remains manageable.

The situation is different on OS or server processes. A Node.js API call could receive file uploads, update multiple database tables, write to logs, and make further API calls before a response can be sent.

Promises

ES2015 (ES6) introduced Promises. Callbacks are still used below the surface, but Promises provide a clearer syntax that *chains* asynchronous commands so they run in series (more about that in the next section).

To enable Promise-based execution, asynchronous callback-based functions must be changed so they immediately return a Promise object. That object *promises* to run one of two functions (passed as arguments) at some point in the future:

- **`resolve`**: a callback function run when processing successfully completes, and
- **`reject`**: an optional callback function run when a failure occurs.

In the example below, a database API provides a `connect()` method which accepts a callback function. The outer `asyncDBconnect()` function immediately returns a new Promise and runs either `resolve()` or `reject()` once a connection is established or fails:

```
const db = require('database');

// connect to database
function asyncDBconnect(param) {

  return new Promise((resolve, reject) => {

    db.connect(param, (err, connection) => {
      if (err) reject(err);
      else resolve(connection);
    });

  });

}
```

Node.js 8.0+ provides a util.promisify() utility to convert a callback-based function into a Promise-based alternative. There are a couple of conditions:

1. the callback must be passed as the last parameter to an asynchronous function, and

2. the callback function must expect an error followed by a value parameter.

Example:

```
// Node.js: promisify fs.readFile
const
  util = require('util'),
  fs = require('fs'),
  readFileAsync = util.promisify(fs.readFile);

readFileAsync('file.txt');
```

Various client-side libraries also provide promisify options, but you can create one yourself in a few lines:

```
// promisify a callback function passed as the last parameter
// the callback function must accept (err, data) parameters
function promisify(fn) {
  return function() {
      return new Promise(
        (resolve, reject) => fn(
          ...Array.from(arguments),
        (err, data) => err ? reject(err) : resolve(data)
      )
    );
  }
}

// example
function wait(time, callback) {
  setTimeout(() => { callback(null, 'done'); }, time);
}

const asyncWait = promisify(wait);

ayscWait(1000);
```

Asynchronous Chaining

Anything that returns a Promise can start a series of asynchronous function calls defined in `.then()` methods. Each is passed the result from the previous `resolve`:

```
asyncDBconnect('http://localhost:1234')
  .then(asyncGetSession)      // passed result of asyncDBconnect
  .then(asyncGetUser)         // passed result of asyncGetSession
```

```
  .then(asyncLogAccess)       // passed result of asyncGetUser
  .then(result => {           // non-asynchronous function
    console.log('complete');  //   (passed result of
asyncLogAccess)
    return result;            //   (result passed to next .then())
  })
  .catch(err => {             // called on any reject
    console.log('error', err);
  });
```

Synchronous functions can also be executed in `.then()` blocks. The returned value is passed to the next `.then()` (if any).

The `.catch()` method defines a function that's called when any previous `reject` is fired. At that point, no further `.then()` methods will be run. You can have multiple `.catch()` methods throughout the chain to capture different errors.

ES2018 introduces a `.finally()` method, which runs any final logic regardless of the outcome — for example, to clean up, close a database connection etc. It's currently supported in Chrome and Firefox only, but Technical Committee 39 has released a .finally() polyfill.

```
function doSomething() {
  doSomething1()
  .then(doSomething2)
  .then(doSomething3)
  .catch(err => {
    console.log(err);
  })
  .finally(() => {
    // tidy-up here!
  });
}
```

Multiple Asynchronous Calls with Promise.all()

Promise `.then()` methods run asynchronous functions one after the other. If the order doesn't matter — for example, initialising unrelated components — it's faster to launch all asynchronous functions at the same time and finish when the last (slowest) function runs `resolve`.

This can be achieved with `Promise.all()`. It accepts an array of functions and returns another Promise. For example:

```
Promise.all([ async1, async2, async3 ])
  .then(values => {           // array of resolved values
    console.log(values);      // (in same order as function array)
    return values;
  })
  .catch(err => {             // called on any reject
    console.log('error', err);
  });
```

`Promise.all()` terminates immediately if any one of the asynchronous functions calls `reject`.

Multiple Asynchronous Calls with Promise.race()

`Promise.race()` is similar to `Promise.all()`, except that it will resolve or reject as soon as the *first* Promise resolves or rejects. Only the fastest Promise-based asynchronous function will ever complete:

```
Promise.race([ async1, async2, async3 ])
  .then(value => {            // single value
    console.log(value);
    return value;
  })
  .catch(err => {             // called on any reject
    console.log('error', err);
  });
```

A Promising Future?

Promises reduce callback hell but introduce their own problems.

Tutorials often fail to mention that *the whole Promise chain is asynchronous*. Any function using a series of promises should either return its own Promise or run callback functions in the final `.then()`, `.catch()` or `.finally()` methods.

I also have a confession: *Promises confused me for a long time*. The syntax often seems more complicated than callbacks, there's a lot to get wrong, and debugging can be problematic. However, it's essential to learn the basics.

Further Promise resources:

- MDN Promise documentation
- JavaScript Promises: an Introduction

- JavaScript Promises … In Wicked Detail
- Promises for asynchronous programming

Async/Await

Promises can be daunting, so ES2017 introduced `async` and `await`. While it may only be syntactical sugar, it makes Promises far sweeter, and you can avoid `.then()` chains altogether. Consider the Promise-based example below:

```
function connect() {

  return new Promise((resolve, reject) => {

    asyncDBconnect('http://localhost:1234')
      .then(asyncGetSession)
      .then(asyncGetUser)
      .then(asyncLogAccess)
      .then(result => resolve(result))
      .catch(err => reject(err))

  });
}

// run connect (self-executing function)
(() => {
  connect();
    .then(result => console.log(result))
    .catch(err => console.log(err))
})();
```

To rewrite this using `async`/`await`:

1. the outer function must be preceded by an `async` statement, and
2. calls to asynchronous Promise-based functions must be preceded by `await` to ensure processing completes before the next command executes.

```
async function connect() {

  try {
    const
      connection = await asyncDBconnect('http://localhost:1234'),
      session = await asyncGetSession(connection),
      user = await asyncGetUser(session),
      log = await asyncLogAccess(user);

    return log;
  }
  catch (e) {
```

```
    console.log('error', err);
    return null;
  }

}

// run connect (self-executing async function)
(async () => { await connect(); })();
```

`await` effectively makes each call appear as though it's synchronous, while not holding up JavaScript's single processing thread. In addition, `async` functions always return a Promise so they, in turn, can be called by other `async` functions.

`async`/`await` code may not be shorter, but there are considerable benefits:

1. The syntax is cleaner. There are fewer brackets and less to get wrong.
2. Debugging is easier. Breakpoints can be set on any `await` statement.
3. Error handling is better. `try`/`catch` blocks can be used in the same way as synchronous code.
4. Support is good. It's implemented in all browsers (except IE and Opera Mini) and Node 7.6+.

That said, not all is perfect …

Promises, Promises

`async`/`await` still relies on Promises, which ultimately rely on callbacks. You'll need to understand how Promises work, and there's no direct equivalent of `Promise.all()` and `Promise.race()`. It's easy to forget about `Promise.all()`, which is more efficient than using a series of unrelated `await` commands.

Asynchronous Awaits in Synchronous Loops

At some point you'll try calling an asynchronous function *inside* a synchronous loop. For example:

```
async function process(array) {
  for (let i of array) {
    await doSomething(i);
  }
}
```

It won’t work. Neither will this:

```
async function process(array) {
  array.forEach(async i => {
    await doSomething(i);
  });
}
```

The loops themselves remain synchronous and will always complete before their inner asynchronous operations.

ES2018 introduces asynchronous iterators, which are just like regular iterators except the `next()` method returns a Promise. Therefore, the `await` keyword can be used with `for … of` loops to run asynchronous operations in series. for example:

```
async function process(array) {
  for await (let i of array) {
    doSomething(i);
  }
}
```

However, until asynchronous iterators are implemented, it’s possibly best to `map` array items to an `async` function and run them with `Promise.all()`. For example:

```
const
  todo = ['a', 'b', 'c'],
  alltodo = todo.map(async (v, i) => {
    console.log('iteration', i);
    await processSomething(v);
});

await Promise.all(alltodo);
```

This has the benefit of running tasks in parallel, but it’s not possible to pass the result of one iteration to another, and mapping large arrays could be computationally expensive.

try/catch Ugliness

`async` functions will silently exit if you omit a `try`/`catch` around any `await` which fails. If you have a long set of asynchronous `await` commands, you may

need multiple `try`/`catch` blocks.

One alternative is a higher-order function, which catches errors so `try`/`catch` blocks become unnecessary (thanks to @wesbos for the suggestion):

```
async function connect() {

  const
    connection = await asyncDBconnect('http://localhost:1234'),
    session = await asyncGetSession(connection),
    user = await asyncGetUser(session),
    log = await asyncLogAccess(user);

  return true;
}

// higher-order function to catch errors
function catchErrors(fn) {
  return function (...args) {
    return fn(...args).catch(err => {
      console.log('ERROR', err);
    });
  }
}

(async () => {
  await catchErrors(connect)();
})();
```

However, this option may not be practical in situations where an application must react to some errors in a different way from others.

Despite some pitfalls, `async`/`await` is an elegant addition to JavaScript. Further resources:

- MDN async and await
- Async functions - making promises friendly
- TC39 Async Functions specification
- Simplifying Asynchronous Coding with Async Functions

JavaScript Journey

Asynchronous programming is a challenge that's impossible to avoid in JavaScript. Callbacks are essential in most applications, but it's easy to become entangled in deeply nested functions.

Promises abstract callbacks, but there are many syntactical traps. Converting existing functions can be a chore and `.then()` chains still look messy.

Fortunately, `async`/`await` delivers clarity. Code looks synchronous, but it can't monopolize the single processing thread. It will change the way you write JavaScript and could even make you appreciate Promises — if you didn't before!

Chapter 8: JavaScript's New Private Class Fields, and How to Use Them

by Craig Buckler

ES6 introduced classes to JavaScript, but they're too simplistic for complex applications. Class fields (also referred to as *class properties*) aim to deliver simpler constructors with private and static members. The proposal is currently at TC39 stage 3: candidate and could appear in ES2019 (ES10).

A quick recap of ES6 classes is useful before we examine class fields in more detail.

ES6 Class Basics

JavaScript's prototypal inheritance model can appear confusing to developers with an understanding of the classical inheritance used in languages such as C++, C#, Java and PHP. JavaScript classes are primarily syntactical sugar, but they offer more familiar object-oriented programming concepts.

A class is a *template* which defines how objects of that type behave. The following `Animal` class defines generic animals (classes are normally denoted with an initial capital to distinguish them from objects and other types):

```
class Animal {

  constructor(name = 'anonymous', legs = 4, noise = 'nothing') {

    this.type = 'animal';
    this.name = name;
    this.legs = legs;
    this.noise = noise;

  }

  speak() {
    console.log(`${this.name} says "${this.noise}"`);
  }

  walk() {
    console.log(`${this.name} walks on ${this.legs} legs`);
  }

}
```

Class declarations execute in strict mode; there's no need to add `'use strict'`.

A `constructor` method is run when an object of this type is created, and it typically defines initial properties. `speak()` and `walk()` are methods which add further functionality.

An object can now be created from this class with the `new` keyword:

```
const rex = new Animal('Rex', 4, 'woof');
rex.speak();          // Rex says "woof"
rex.noise = 'growl';
```

```
rex.speak();           // Rex says "growl"
```

Getters and Setters

Setters are special methods used to define values only. Similarly, *Getters* are special methods used to return a value only. For example: `class Animal {`

```
constructor(name = 'anonymous', legs = 4, noise = 'nothing') {


this.type = 'animal'; this.name = name; this.legs = legs;
this.noise = noise;

}


speak() {

console.log(`${this.name} says "${this.noise}"`); }


walk() {

console.log(`${this.name} walks on ${this.legs} legs`); }


// setter

set eats(food) {

this.food = food; }


// getter

get dinner() {

return `${this.name} eats ${this.food || 'nothing'} for dinner.`; }


}
```

```
const rex = new Animal('Rex', 4, 'woof'); rex.eats = 'anything';
console.log( rex.dinner ); // Rex eats anything for dinner.
```

Child or Sub-Classes

It’s often practical to use one class as the base for another. If we’re mostly creating dog objects, `Animal` is too generic, and we must specify the same 4-leg and “woof” noise defaults every time.

A `Dog` class can inherit all the properties and methods from the `Animal` class using `extends`. Dog-specific properties and methods can be added or removed as necessary: `class Dog extends Animal { constructor(name) { // call the Animal constructor super(name, 4, 'woof'); this.type = 'dog'; } // override Animal.speak speak(to) { super.speak(); if (to) console.log(`to ${to}`); } }`

`super` refers to the parent class and is usually called in the `constructor`. In this example, the Dog `speak()` method overrides the one defined in `Animal`.

Object instances of `Dog` can now be created: `const rex = new Dog('Rex'); rex.speak('everyone'); // Rex says "woof" to everyone rex.eats = 'anything'; console.log( rex.dinner ); // Rex eats anything for dinner.`

Static Methods and Properties

Defining a method with the `static` keyword allows it to be called on a class without creating an object instance. JavaScript doesn't support static properties in the same way as other languages, but it is possible to add properties to the class definition (a `class` is a JavaScript object in itself!).

The `Dog` class can be adapted to retain a count of how many dog objects have been created: `class Dog extends Animal {`

```
constructor(name) {

// call the Animal constructor super(name, 4, 'woof'); this.type =
'dog';
// update count of Dog objects Dog.count++;

}

// override Animal.speak speak(to) {

super.speak();
if (to) console.log(`to ${to}`);
}

// return number of dog objects static get COUNT() {
return Dog.count; }

}
```

```
// static property (added after class is defined) Dog.count = 0;
```

The class's static `COUNT` getter returns the number of dogs created:

```
console.log(`Dogs defined: ${Dog.COUNT}`); // Dogs defined: 0


const don = new Dog('Don');

console.log(`Dogs defined: ${Dog.COUNT}`); // Dogs defined: 1


const kim = new Dog('Kim');

console.log(`Dogs defined: ${Dog.COUNT}`); // Dogs defined: 2
```

For more information, refer to Object-oriented JavaScript: A Deep Dive into ES6 Classes.

ESnext Class Fields

The class fields proposal allows properties to be initialized at the top of a class:

```
class MyClass {




  a = 1;


  b = 2;


  c = 3;




}
```

This is equivalent to:

```
class MyClass {




  constructor() {


    this.a = 1;
```

```
    this.b = 2;

    this.c = 3;

  }



}
```

Initializers are executed before any constructor runs (*presuming a constructor is still necessary*).

Static Class Fields

Class fields permit static properties to be declared within the `class`. For example:

```
class MyClass {



  x = 1;

  y = 2;

  static z = 3;
```

```
}



console.log( MyClass.z ); // 3
```

The inelegant ES6 equivalent:

```
class MyClass {



  constructor() {

    this.x = 1;

    this.y = 2;

  }



}
```

```
MyClass.z = 3;



console.log( MyClass.z ); // 3
```

Private Class Fields

All properties in ES6 classes are public by default and can be examined or modified outside the class. In the `Animal` example above, there's nothing to prevent the `food` property being changed without calling the `eats` setter:

```
class Animal {



  constructor(name = 'anonymous', legs = 4, noise = 'nothing') {



    this.type = 'animal';

    this.name = name;

    this.legs = legs;
```

```
    this.noise = noise;




  }




  set eats(food) {

    this.food = food;

  }




  get dinner() {

    return `${this.name} eats ${this.food || 'nothing'} for
dinner.`;

  }




}
```

```
const rex = new Animal('Rex', 4, 'woof');

rex.eats = 'anything';      // standard setter

rex.food = 'tofu';          // bypass the eats setter altogether

console.log( rex.dinner );  // Rex eats tofu for dinner.
```

Other languages permit `private` properties to be declared. That's not possible in ES6, although developers can work around it using an underscore convention (`_propertyName`), closures, symbols, or WeakMaps.

In ESnext, private class fields are defined using a hash `#` prefix:

```
class MyClass {

  a = 1;          // .a is public

  #b = 2;         // .#b is private

  static #c = 3;  // .#c is private and static
```

```
  incB() {

    this.#b++;

  }




}




const m = new MyClass();




m.incB(); // runs OK


m.#b = 0; // error - private property cannot be modified outside
class
```

Note that there’s no way to define private methods, getters and setters, although a TC39 stage 2: draft proposal suggests using a hash # prefix on names. For example:

```
class MyClass {
```

```
// private property

#x = 0;



// private method (can only be called within the class)

#incX() {

  this.#x++;

}



// private setter (can only be called within the class)

set #setX(x) {

  this.#x = x;

}
```

```
  // private getter (can only be called within the class)

  get #getX() {

    return this.$x;

  }



}
```

Immediate Benefit: Cleaner React Code!

React components often have methods tied to DOM events. To ensure `this` resolves to the component, it's necessary to `bind` every method accordingly. For example:

```
class App extends Component {




  constructor() {




    super();




    state = { count: 0 };




    // bind all methods

    this.incCount = this.incCount.bind(this);

  }
```

```
incCount() {

  this.setState(ps => ({ count: ps.count + 1 }));

}



render() {



  return (

    <div>

      <p>{ this.state.count }</p>

      <button onClick={this.incCount}>add one</button>

    </div>

  );
```

```
  }

}
```

If `incCount` is defined as a class field, it can be set to a function using an ES6 `=>` fat arrow, which automatically binds it to the defining object. The `state` can also be declared as a class field so no constructor is required:

```
class App extends Component {

  state = { count: 0 };

  incCount = () => {
    this.setState(ps => ({ count: ps.count + 1 }));
  };

  render() {
```

```
    return (
      <div>
        <p>{ this.state.count }</p>
        <button onClick={this.incCount}>add one</button>
      </div>
    );

  }
}
```

Using Class Fields Today

Class fields are not currently supported in browsers or Node.js. However, it's possible to transpile the syntax using Babel, which is enabled by default when using Create React App. Alternatively, Babel can be installed and configured using the following terminal commands: `mkdir class-properties cd class-properties npm init -y npm install --save-dev babel-cli babel-plugin-transform-class-properties echo '{ "plugins": ["transform-class-properties"] }' > .babelrc`

Add a `build` command to the `scripts` section of `package.json`: `"scripts": { "build": "babel in.js -o out.js" },`

Then run with `npm run build` to transpile the ESnext file `in.js` to a cross-browser-compatible `out.js`.

Babel support for private methods, getters and setters has been proposed.

Class Fields: an Improvement?

ES6 class definitions were simplistic. Class fields should aid readability and enable some interesting options. I don't particularly like using a hash # to denote private members, but unexpected behaviors and performance issues would be incurred without it (refer to JavaScript's new #private class fields for a detailed explanation).

Perhaps it goes without saying, but I'm going to say it anyway: *the concepts discussed in this article are subject to change and may never be implemented!* That said, JavaScript class fields have practical benefits and interest is rising. It's a safe bet.