Coursepak for
The Frontend: Modern JavaScript & CSS Development

DHSI
Summer 2019
Victoria, BC

Course Overview

This course is designed to teach you about modern, industrial best practices for building robust, dynamic, and powerful web applications using JavaScript and CSS.

In the last decade, the complexity that is possible to produce with JavaScript has grown immensely. Now, many dynamic web applications are written entirely in JavaScript, including both the client and the server!

Technologies such as React, Redux, Fetch in JavaScript and SASS (**S**yntactically **A**ware **S**tyle**s**heets) have greatly increased the things programmers can do within a visitor's browser. We will be talking about all of these technologies during this course.

The Week’s Plan

This week, students can expect to learn about the following topics:

1. Getting Started
    a. Node.js
    b. NPM
    c. Visual Studio Code (text editor)
    d. JavaScript, The Good Stuff
    e. ES6 – What’s New in JS?
2. JavaScript/CSS as Grown-up Languages
    a. Webpack
    b. SASS
    c. Accessing External Data
3. React
    a. Progressive Web Frameworks
    b. Our First React Project
    c. SASS/CSS in React
4. Advanced Topics / Future Directions
    a. Testing
    b. Express & Server-Side JavaScript

What's in the Coursepak?

Rather than include documentation that is available online, this coursepak is taken up with introductory and opinion pieces that will situate you in the modern JS and CSS world.

Specifically, there is material that introduces the basics of JavaScript, CSS, and SASS in addition to highlighting some of the trickier aspects of JS.

Titles marked in bold are considered background; please at least glance at them before the start of class.

I have drawn content from a number of sources, but much of the content comes from two important sources:

1. Moacir P. de Sá Pereira's *The JavaScripting English Major*, which provides basic background on the JavaScript language and also discusses Atom, the text editor I will be recommending you use in this class.

2. Two of Eric Elliott's excellent series: "Master the JavaScript Interview" and "Composing Software." The "Interview" series provides background on some of the more esoteric aspects of JavaScript, while "Composing" captures some of the *ethos* behind the recent revolution in JS.

JavaScript Documents

1. ***The JavaScripting English Major***

    a. "Introduction"

    b. "Setting Up the Environment"

    c. "JavaScript Calculator"

    d. "Programming"

    e. "Functions"

    f. "Collections of Data"

    g. "Abstraction"

2. **“A Functional Programmer’s Introduction to JavaScript (Composing Software)”**
3. “Composing Software: An Introduction”
4. **“Master the JavaScript Interview: What is a Closure?”**
5. **“Master the JavaScript Interview: What’s the Difference Between Class & Prototypal Inheritance?”**
6. “Master the JavaScript Interview: What is a Promise?”
7. “Master the JavaScript Interview: What is a Pure Function?”
8. *AirBnB JavaScript Style Guide* – An extremely popular JavaScript style guide

CSS/SASS Documents

1. **“Getting Started with CSS – Part 1”**
2. **“Getting Started with CSS – Part 2”**
3. “Getting Started with SASS”
4. “Medium’s CSS is Actually Pretty F**king Good” – Interesting overview of code standards in a large CSS project

JavaScript Documents

The following documents are a recompilation of the first 6 chapters of

The JavaScripting English Major

By
Moacir P. de Sá Pereira

The full eBook is available online at:

https://the-javascripting-english-major.org

Setting up the Environment

It's possible to start JavaScripting just by opening your browser, but for the purposes of this course, we're going to introduce two new pieces of software that work to make your programming life easier.

This establishes a programming **environment**, which is a workflow in which multiple pieces of software work together in order to make reaching your goals as easy as possible. Many programming rely on IDEs, or integrated development environments, such as Xcode for writing MacOS and iOS apps or R Studio for writing in R. These are one-stop shops that try to handle the entirety of your project within one piece of software. We'll reach a similar state with Atom, below, but not before first making a detour through Git.

A key concept to application development is **versioning**. That is, by making use of a piece of software called a **version control system**, a project can work as though it has a time machine attached to it. But not just a time machine.

Before moving forward, note that above I don't refer to a "document" or even a "file." Most of the homework you've ever turned in before for an English class has probably been a document, most likely written in Microsoft Word. In this environment, we're not creating one document or one file. Instead, we're working on one **project**, which is made up of many files, where each file contributes its own bit to the project as a whole. A website (or a webapp) is an example of a project—one that is made up, of course, of many documents. For example, on even the most basic website, the information on the front page could be saved in one file, and the information in the "About" page could be saved in another.

Thinking in terms of *projects*, not documents, is perhaps the first vital leap for you to make. What we are building in this course is a website that is a project. As such, you will have a folder for your project, and everything in that folder will be under version control.

Git

The version control system we will use in this course is called Git. Git is famously opaque, and, worse, a novice user (like us) uses about three commands all the time except when things go wrong. Luckily, when things go wrong, Git is often there to help us out. Nevertheless, Git deserves its reputation that leads to websites like "Oh Shit, Git!"

Git calls a project a **repository**, so in the context of using Git, I'll do the same. A repository is a folder with files in it, like any regular folder of files you have on your computer. However it also has, hidden from view, a history of all the files in the folder. Git knows exactly when a file was added. It knows every change ever made to each file. It even knows about files that have been deleted, and it can resurrect them, if needed.

So far, this sounds like a backup system. And Git is *also* that. But it can do much more. For example, every change you record with Git, called a **commit**, also has a **commit message** attached to it, where you can leave a note for yourself (or a collaborator!) that tells you

what you were thinking when you made a change. Why did you put this paragraph before that one? Read your old commit messages.

Additionally, Git lets you **push** your changes to a central version of the repository. This means that you can work on one computer, save your work, commit it, push it, and then, on another computer, **pull** the changes and get right back to work—with all that history still built in, even though you were working *on another computer!*

That covers all of what we'll be doing with Git in this course, and that is already a lot. But as you can see, the system of committing, pushing, and pulling means that collaboration becomes very easy. Additionally, Git allows for **branching**, where you create new branches for your project. Now Git is not only a time machine, but it's also an interdimensional portal leading you to alternate universes. In one universe, your text could be written full of jokes. In another, the writing can be very serious.

GitHub

For the purposes of this course, we will be using Git via the website GitHub. GitHub will serve as the remote point to which you push your project, and GitHub will even, down the road, host the project. At the end of this course, you'll have a web address including "github.io" that you can send to your friends and family to show off your progress.

In addition to being a user-friendlier face for the Git software, GitHub also includes features that expand what Git offers, such as built in discussion boards where users can discuss issues. We won't be using that functionality in this class, but that should stand in to remind you that *things on GitHub are public by default*. Keep this in mind when you're typing commit messages or working on your project: it's all visible to anyone who has a look at your repository. Don't let that scare you from using GitHub. Just keep it in mind.

Create yourself a GitHub user

The first step with GitHub is easy. Go to GitHub.com and create an account. Choose the free plan. You should get an email, and once you verify it, you can click on the "Start a project" button.

If you're a current student, you can request a discount for GitHub. That discount entitles you to a limited number of private repositories.

Fork the course repository

In GitHub, anyone can access anyone else's (public) repositories, or projects. And, what's more, anyone can make an identical copy of that repository for their own use. This is called **forking**, and the next step is to fork the repository I have already made for this course.

While logged into GitHub, point your browser to:

https://github.com/muziejus/javascripting-english-major-project

Once that page loads, there should be a "Fork" button in the top-right corner. Click on that. If it asks you where to fork it, click on your user, and now the repository should appear in your account under this URL:

```
https://github.com/YOURUSERNAME/javascripting-english-major-project
```

Where it reads `YOURUSERNAME`, it should read, of course, the GitHub username you chose for yourself. Make a note of this URL, as you'll need it in a few steps.

Atom

You're probably used to writing your English homework in Microsoft Word or, maybe, Pages. Or maybe you use Google Docs. They are all fine word processors, and I've used them all. Actually, I've been using Microsoft Word since well before you were born, probably. But they all make two conceptual assumptions that go against the work we are doing in this course.

First, they deal with documents, not projects. Remember, in this course you have to think in terms of a project, not just in terms of a single document. Something like Word is great for a single document, like a business letter or five-page paper. When it comes to something with lots of moving parts, like a web project (or a dissertation…), it starts to creak.

Second, the big advance Word brought to the computing world was fusing what a document says along with how it looks.[1] In Word, whatever the document looks like on your screen is a pretty good approximation of what it will look like in your printer.[2] You can change fonts, font sizes, margins, and so on, for a single document, and when you print it, it will follow your instructions as closely as possible as it prints to a US Letter sheet of paper.[3] This is, of course, because of the first assumption: you're writing a document that is going to get printed.

We're writing a website, however. And who knows what kind of device will be used to view it. Imagine if you write a page with 2-inch margins and visit it on a smartphone. You'll see nothing but margins! When you're writing for the web, you're handing off a lot of decisions regarding how the content will look to the user, who could be using a giant monitor or a teeny smartphone to view your site. Of course, you still have *some* control over how things

---

[1] Of course, whether a text still means the same thing when it looks different is an unsettled question in literary study.

[2] Microsoft Word was not, of course, the first "WYSIWYG" ("What You See Is What You Get") word processor. It appeared around the same time as MacWrite did, and both of those applications were building on ideas established years earlier at Xerox. Nevertheless, as Windows began to gain dominance on the PC market, Word became ubiquitous, leaving it as the "default" example of a WYSIWYG word processor to this day.

[3] Or A4 paper, if you're outside the United States.

look. A lot, in fact. But in order to do that, you have to rethink the basics of writing. You now have to focus on the content much more than the look.

A plain text editor

One way of "focusing on the content" is by stripping away a lot of the frippery that a program like Word provides. We won't need something that can make charts or insert clip art. We won't be changing fonts willy-nilly. And we definitely aren't creating something that lives only to be printed out. We are creating, again, a *project*, not a document.

Furthermore, we will be writing code in addition to text. In your own project, you will be programming functions and writing paragraphs. The former requires simple text files that can be read by the JavaScript interpreter (also known as the browser). The latter will be converted into HTML, which means it has to be written in plain text that can be fed to the HTML renderer (also known as the browser). Instead of one `.docx` file, we will have `.js` files for our JavaScript programming and `.html` files for our HTML webpages. Finally, Git is much friendlier with regular text files (like `.js` or `.html` files) than it is with `.docx` files. You can use Git with Word, but then you're basically just using it as a backup system, not as a version control system.

What we need, then, is a "plain text editor." Something that just writes text, more or less agnostically. Though it'd be nice if maybe it colored the text certain ways when writing JavaScript or HTML. But let's not jump ahead. You've probably used a plain text editor before, most notably the Notes application on your smartphone. Every personal computer has some kind of plain text editor installed, but we'll install one that runs on nearly any personal computer, Atom.

Enter Atom

Atom is a fully-featured text editor that can also serve as an IDE. It also has Git (and GitHub) support built in. This makes sense; it's part of the GitHub ecosystem. Atom is a serious program with lots and lots of features. I'll only be teaching a few here, but I'll also teach a few more as the course goes on. It also does not behave like Word, which will take a bit of getting used to. Still, I hope that, for project-based work, you will see that it works much better than Word or a similar word processor.

You can download the software from the Atom page, and installation should be rather straightforward[4]. Once you install it, when you open it, you will be greeted with a welcome tab and the welcome guide tab in a separate pane. Atom works on the visual metaphor of

---

[4] There are two hiccups that will come up during installation when you aim to use git with Atom. On Macs, you will be prompted to install the command-line tools. It should work automagically, but if it doesn't, see this post. On Windows, you need to install the gitbash shell and then configure git using gitbash.

panes with tabs. You can change the widths of the panes and show and hide them with your mouse. Similarly, the tabs work like the tabs do in Chrome, for good reason.[5]

Start customizing with packages

From the Welcome Guide, I recommend immediately clicking on "Install a Package." Ben Balter has come up with a list of useful Atom packages for writing prose, but I'll mention the most useful ones here. Once you click on "Install a Package," you can click on "Open Installer" and start installing packages by searching for them and then clicking the "Install" button.

- file-icons: this gives you pretty icons in the Atom sidebar and tabs. These visual cues are, in my experience, more useful than file name extensions.
- linter-jshint: adds a linter based on JSHint for JavaScript, meaning the Atom will warn you when the JavaScript you write has problems.

For these, you have to type the name into the little search bar in the Install Packages part of the Settings tab. The linter may ask you to add some other packages to fulfill dependencies. That's ok.

Continue customizing by enabling autosave

Atom ships with autosave disabled by default. That's probably not behavior you're expecting, so you should enable it! After you're done installing packages, you'll be on the Settings tab, which has a few subcategories, like Core, Editor, and so on. Choose Packages. This gives a list of all the packages you have installed, as well as giving you a chance to configure them. Type `autosave` into the filter box at the top, and then click on "Settings" once the autosave package shows up under "Core Packages."

Tick the box beside "Enabled" under settings, and now Atom will autosave any file as soon as you click away from its tab, even if you go to another application.

While the Settings tab is still open, click on "Core" and "Editor" and change things around these if you like. The defaults are pretty good, but you may want to change the "Font Family" in Editor. Type in the name of your preferred font, provided that it is installed. You can make the default text larger or smaller, and so on. One more setting to consider in Editor is "Soft Wrap." With it off, your text will keep running off the side of the window as you type. Soft Wrap, like in Word or Notes, breaks lines so that everything fits inside the window.

Finally, under Themes, you can do some basic changing of the user interface, theme, and the syntax theme for the editor. Pick a light one or a dark one, whichever you think fits your

---

[5] Atom is, basically, a very customized version of Chrome that talks to a JavaScript server that you run in the background when you launch the application.

personality. I personally use the solarized themes, based on Ethan Schoonover's Solarized precision color project.

Link Atom to GitHub

Atom is written by the people at GitHub, so it's pretty easy to link the two together. In fact, that's the main reason I encourage you to install Atom. The Atom people have written a decent introduction to most of what I describe here, so it's useful to glance over at their screenshots, etc., if this part gets too confusing.

First, you have to get the URL of the repository you forked a few steps ago:

```
https://github.com/YOURUSERNAME/javascripting-english-major-project
```

Again, `YOURUSERNAME` should be replaced with your own GitHub username. Now add `.git` to the end of it, so you have:

```
https://github.com/YOURUSERNAME/javascripting-english-major-project.git
```

This is the URL you will need for cloning.

Second, return to Atom. Here, open up the command palette by going to the "Packages" menu, choosing "Command Palette" and then "Toggle." In the little box that opens, type "git" and choose the option labeled "GitHub: Clone." For "Clone from," paste or type in the URL above ending in `.git` with your username. Atom will automatically save it to a GitHub folder it creates in your home folder, but you can change this location if you like.

Now, if you have the file-icons package installed, in the Projects pane, you should see a small book icon with your repository name beside it, and, underneath it, you should see a folder called `.git`, a file called `README.md`, and a few other files. These are the contents of your repository, and they are now on your computer.

Third, let's make an explicit connection between GitHub and Atom. Open up the GitHub pane ("Packages" > "GitHub" > "Toggle GitHub Tab"), and you should now see a button asking you to log in or a message encouraging you to go to the GitHub Atom login page. If you don't, make sure that your project is open and loaded in Atom. Otherwise, click on "Login" and then click on the web address, which will open a new page in your browser asking you for permission to have Atom talk to your GitHub account. Click on "Authorize atom" and then copy the extremely long code that appears. Paste it back into the GitHub pane back on Atom.

You should get a message saying "No pull request could be found for the branch master," and so on. You've made the connection.

Make a change, stage a file, commit, and push

We're in the homestretch now, but this section is the most important, because it's a description of what you will be doing with Git in Atom most of the time. In the "Projects" tab, there should be a file called `README.md`. Double click on it, and it should open in a new tab. You should see some text that I've written. Go ahead and delete it all and type in

something of your own, like your name and your goals for your project (it's ok if they're vague for now).

Just clicking away from the window should autosave the document, which you do by now opening the Git tab ("Packages" > "GitHub" > "Toggle Git Tab"). You will use this tab much more often than the GitHub tab, so it might be worthwhile to remember the keyboard shortcut, control-shift-9.

The Git tab is split into three horizontal sections, **Unstaged**, **Staged**, and **Commit**. Unstaged lists all the files you have saved but have not yet committed. If the icon beside the file is a green cross, that means it is a new file, never before saved to the repository. `README.md`, on the other hand, should appear in unstaged changes with a yellow box with a dot. That means that the file is in the repository, but changes have been made to it that have not yet been committed. Finally, if you were to delete a file, it would show up there with a red minus sign. Yes, in Git, a file is never truly deleted, remember.

If you click on one of the files in the Unstaged area, a new tab will open with a lot of green (and maybe some red) text. That shows everything you have added or removed from the file since its last commit. With `README.md`, it will be a lot of red and hopefully a lot of green.

Click on the "Stage All" button at the top of the Unstaged area to move all the files to the Staged area. Or, if you like, you can only move one file at a time. I usually try to commit things thematically. So if I make some changes on one file and totally unrelated changes to another file, I will make two separate commits. But if both are related (say I change the name of something in both places), then it can be one commit. There's no right way to committing. Do what feels like a good balance between often enough to be useful yet infrequent enough so that you actually get work done.

Once files are in the Staged area, you can commit. That will take the files in the Staged area and log the changes to the files in the Git time machine. Any changes you make backwards or forwards between commits doesn't matter. Git only tracks commits, not individual saves.

Type in a useful commit message (like `Edit README`) in the Commit box and press "Commit." The changes have been recorded.

The final step is pushing the changes up to GitHub. Once you have a commit, the up arrow at the bottom right corner of the Atom window will have a little "1" appear next to it. If you make another commit, that "1" will become a "2." Once you've got enough commits and want to push, click on that arrow, and Atom will ask you for your GitHub username and password. Type them in, wait a minute, and then if you go to GitHub and refresh your repository, you will see the changes that you've made.[6]

---

[6] I, personally, overpush. I usually commit and immediately push, which is generally fine, but it can be embarrassing sometimes. Also, since the default means by which Atom pushes to GitHub asks you for your username and password, it makes sense to push only every few commits or so. Find a balance that works for you, but remember to always finish your work before a break with a commit and a push, just in case!

Parting Atom thoughts

There's a whole lot going on in this section, but it's mostly stuff you just have to do once to set up the environment. As I wrote above, Atom is a very heavy duty program that can do a whole lot more than what we will use it for. Whenever you get frustrated with Atom's menus and the like, always remember that you can launch the command palette by typing command-shift-p on a Mac or control-shift-p on Windows. That brings up a small window that lets you type in whatever command you want to execute.

Also, Atom will be frustrating the first few times. Any new piece of software is. By the end of this course, though, I hope that you'll see why I insisted on this unpleasantness at the beginning. And if you are interested in learning more about Atom, please have a look at their documentation, that includes the *Flight Manual* book, and an introductory video.

Browser

Our last step in this chapter is making sure your browser has a JavaScript console. I recommend against using Internet Explorer in this course for a number of reasons, so I'll only give directions for Firefox, Chrome, and Safari.

- In Firefox, the console is hidden the "Tools" menu, under "Web Developer" and "Web Console" on the Mac. On Windows/Linux, it is in the "Web Developer" submenu of the "Firefox" menu. Or, type control-shift-k for Windows/Linux or command-option-k for Mac.
- In Chrome, open the DevTools palette either by typing control-shift-j (Windows/Linux) or command-option-j (Mac). Or, find the tools in the Chrome menu (upper right, beside the address bar), under "More Tools" and "Developer Tools." Once the DevTools palette opens, you can click on the "Console" tab.
- In Safari, look in the "Develop" menu and choose "Show JavaScript Console." Or type command-option-c.

Newer browsers will behave similarly. Brave, for example, is based on Chrome, so the keyboard shortcut is the same. Incidentally Brave was co-founded by Brendan Eich, the man who invented JavaScript.

Whichever browser you're using, the console looks more or less the same. It's a large empty window with a > at the bottom. This > is called the "prompt." Beside the prompt, type:

```
console.log('Hello, World!');
```

Hit return. The console should respond with `Hello, World!` Throughout the rest of this course, when something needs to be typed at the prompt, I will include the >. This time I skipped it, to make things a bit clearer for those who have never used a prompt before.

By typing the above, you've written your first bit of JavaScript. In other words, you're ready for the next chapter after completing the exercises below.

Exercises

1. Create a GitHub account.
2. Fork the blank repository for this course from https://github.com/muziejus/javascripting-english-major-project.
3. Install Atom on your computer with some useful packages.
4. Link Atom to your GitHub account.
5. Use Atom to clone the repository you forked.
6. Flesh out your hopes for your personal project in the `README.md` file and commit the changes.
7. Push your commit(s) from your computer up to GitHub.

Footnotes

JavaScript Calculator

The last chapter ended with the following, in the console:

```
> console.log("Hello, World!");
//--> Hello, World!
```

Note that I have included the > prompt in this example, along with the how the console responded to your single line of JavaScript, after the `//-->`. There's a lot going on in just this one example, but, briefly, `console.log()` is a way of using JavaScript to tell the console to output something.[1] Before we get too carried away with writing JavaScript, however, I think it might be useful to learn a bit about JavaScript's history.

Hello, World! Hello, JavaScript!

I mentioned a few details about JavaScript's history in the FAQs, but I'll reiterate some of that now.

Programming languages inherit from previous programming languages. The writer(s) of a language will take parts they like from one language and mix it with parts they like from another, and then add their own, new additions that makes their language special. JavaScript is no different; in fact, JavaScript might be especially good at revealing its mixed parentage. But that means, concretely, that it helps to have a sense of the history of programming in general when understanding why JavaScript looks like it does today.

The fourth episode of the BBC show *Connections* draws a line to computer programming that begins with the Roman era Barbegal aqueduct and mill. Sadly, James Burke concludes his wry take on advances in technology with punch cards, but he was also filming *Connections* in 1978. Nevertheless, Burke sees that the key to how computers work is related to how they *organize information*. How to organize information is, not coincidentally, the first part of programming that we will learn in this chapter.

Speaking a few decades later, Douglas Crockford picks up the thread from where Burke ended it (punchcards) and moves us all the way up to JavaScript in two of his *Crockford on JavaScript* lectures, "The Early Years" and the first 20 minutes of "And Then There Was JavaScript." Although the lectures are aimed at people with some programming knowledge, meaning that they get pretty geeky, what Crockford does well is hammer home two key points in the way computing has evolved over time:

---

[1] More precisely, `console` is a JavaScript object (similar to `window` or `document`, as we'll see later) that refers to the console you have opened in your browser. `.log()` is a "method" specific to the `console` object that outputs whatever is in sent as a parameter (or, inside the `()`). In our example, we sent the console a snippet of text, `"Hello, World!"`, and it returned it to us.

- The people who should be the first to recognize the value of an innovation (like programmers) are often the last.
- Obsolete technologies fade away slowly.

The implications of both points reveal why JavaScript has its quirks and why those quirks are here to stay.

The short version of Crockford's history boils down to the fact that JavaScript was written over ten days by one man, Brendan Eich. Netscape wanted a scripting language for their web browser, Navigator, and Eich delivered. At the time, Java was positioned to be "the language for the web," as we would be running Java applets on websites, so Netscape called Eich's language "JavaScript," despite the fact that JavaScript inherited nearly nothing substantial from Java.

Once Microsoft put their *own* version of JavaScript in Internet Explorer, it became clear that in order for JavaScript to be useful across the entire web, it would have to be standardized. This took time, and, what's more, standardizing involved a process of maintaining compatibility. As a result, JavaScript, as Crockford is fond to point out, has a lot of bad parts that are here to stay. Nevertheless, he adds, it also has some *very good* parts. After all, back in 2001 Crockford called JavaScript "the world's most misunderstood programming language." Yet now programmers are focusing more on these good parts (it helps that Crockford even wrote a book about JavaScript's good parts), which is part of what makes JavaScript so popular.

As Crockford likes to point out, JavaScript is many things to many people, allowing itself to be used in three different programming paradigms. It can be a procedural language. It can be used as a functional language. Or it can be used as an object-oriented language. Or all three in the same application. These are details that go beyond the scope of this course, but they gesture towards the idea that JavaScript is very flexible and very forgiving. This is part of why it's so easy to learn. But it's also why it's so easy for programmers to cause trouble with it.[2]

Now that we've got a bit of an introduction to the language's history and context out of the way, we can go back to the JavaScript console you learned to open in the last chapter and start learning the language itself.

Basic data types

We hear the word "data" every day, but what, precisely, does it mean? If you had to draw a picture of "data," what would it look like? We can say "data is information," but doesn't that just pass the buck, because now we have to ask what "information" means?

---

[2] In this course, most of the JavaScript will be procedural, which encourages a step-by-step way of thinking through a solution. Once we move to making maps with Leaflet, the more object-oriented aspects will emerge. JavaScript's functional personality is its most powerful and appealing, but it strikes me as the most difficult to understand and teach.

It maybe helps to remember that the word "data" is the plural version of the Latin word "datum." A datum is a single piece of information, abstract as that sounds. It can be a number. It can be string of letters. It can be the answer to the question "true or false?" It can also be the *lack* of information.

Data can also be more complex. A datum can be a list or collection of other pieces of data, or it can be a process that takes some data and generates new data based on that data.

These six possibilities (and more exist) make up the six most important **data types** in JavaScript, namely **number**, **string**, **boolean**, **null** / **undefined**, **array** / **object**, and **function**. We'll only be working with the first four types, the basic types, in this chapter.

Number

Numbers are precisely what they sound like—numbers. In JavaScript, it's easy to use numbers, because you just type them as numbers. Numbers can have decimal points if they're not integers. Try typing numbers into the console and see what happens:

```
> 9;
//--> 9
> 1.5;
//--> 1.5
> console.log(9);
//--> 9
```

Your console may look a bit different in what it returns (including `undefined` after `console.log(9);`), but we can ignore that for now.

But note also that I type a `;` at the end of every command. JavaScript is not very picky with semicolons, but it's better to include them as a matter of habit. The semicolon comes at the end of a "statement," which is a single instruction given to the console. `9;` just means "the number 9." `console.log(9);` means "log the number 9 to the console."

String

Strings are a bit trickier than numbers at first glance. They are any string of characters enclosed by double quotes.[3] Have you used a string already in this course? Yes, `"Hello, World!"` is a string containing the characters `Hello, World!`. We can issue statements with strings in much the same way we did with numbers:

```
> "Hello, World!";
//--> "Hello, World!"
> console.log("Hello, World!");
//--> Hello, World!
> "World, I just wanted to say \"Hi!\" 9 times!";
```

---

[3] Yes, JavaScript permits using single quotes as well, but you should use double quotes exclusively.

```
//--> "World, I just wanted to say \"Hi!\" 9 times!"
> console.log("World, I just wanted to say \"Hi!\" 9 times!");
//--> World, I just wanted to say "Hi!" 9 times!
```

Notice how strings can have double quotes inside of them, as long as you use a `\` (backslash) beforehand. What do you type if you want a backslash in the string? Notice also that when we use `console.log()`, the result does not have surrounding double quotes or the backslashes.

Boolean

A boolean is a value that is either `true` or `false`. They are named after George Boole, an English mathematician, and they are astonishingly useful in programming, because they allow programs to make decisions based on values, much like you might make a decision on where to get dinner based on a series of truth values. For example, say you eat burritos every day for dinner except Fridays, when there is a deal on falafel. You ask yourself the question, "Is today Friday?" If the answer is "yes" (`true`), then you go to the falafel joint. If the answer is "no" (`false`), you get a burrito.

Since booleans are only ever true or false, you refer to them in JavaScript using those two words:

```
> true;
//--> true
> false;
//--> false
> 9 === 9;
//--> true
> 9 === "Hello, World!";
//--> false
```

Notice that `true` and `false` do *not* have double quotes around them. What would happen if they did? I have also introduced you to `===` in this code snippet. That is the "strict equal" **comparison operator**, and it is a way of asking the console a question.[4] `9 === 9;` is the same as "Is the number nine the same thing as the number nine?" Because the answer is "yes," the console responds `true`. In the next example, we're asking the question, "Is the number nine the same thing as the string 'Hello, World!'?" Since the answer is "no," the console responds `false`.

Between numbers, strings, and booleans, we can start using JavaScript as a calculator, but one type remains.

---

[4] JavaScript has a less strict equality comparison operator, but its behavior can be unexpected. It is considered one of JavaScript's bad parts, and I won't be teaching it in this course.

Null/Undefined

Sometimes we deal with a lack of information, instead of information we understand as a datum. There are two main ways JavaScript understands that lack, through an object called `null` and a data type called `undefined`.

`null` is an object that indicates the lack of information. Say you're filling out a questionnaire online and you left the last few questions unanswered but still pressed "Submit." The answered questions would be strings or numbers, probably. But how should the computer understand the unanswered questions? The number zero doesn't seem right. Nor does a blank string. In this case, it makes sense to register that lack of answers with `null`.

`undefined`, on the other hand, is for information that we do not yet have. Using the questionnaire example again, all of the answers are `undefined` until you press "Submit" and send them to the computer, where it can then decide what to do with them, by turning them into strings, numbers, or just `null`.

The distinction is tricky, but now you're definitely ready to start add up some numbers in JavaScript.

Using JavaScript as a calculator

You learned a comparison operator in the previous section, `===`, and you'll learn a few more in this section. You'll also learn all five arithmetic operators. In fact, let's start with them:

```
> 2 + 3;
//--> 5
> 2 - 3;
//--> -1
> 2 * 3;
//--> 6
> 2 / 3;
//--> 0.6666666666666666
```

As you can see, the arithmetic operators give numbers as answers. The comparison operators, like `===`, respond with `true` or `false`:

```
> 2 + 3 * 6 === (2 + 3) * 6;
//--> false
> 2 + 3 * 6 < (2 + 3) * 6;
//--> true
> 2 > 2;
//--> false
> 2 >= 2;
//--> true
> 2 < 2;
//--> false
> 2 <= 2;
//--> true
```

Programmers have figured out nifty ways to get around the difficulty of typing ≥ and ≤!

Strings also have an operator +. It comes in very handy in web development:

```
> "I had a thought, but… " + "Oh yeah, I remember. Falafel on Fridays!";
//--> "I had a thought, but… Oh yeah, I remember. Falafel on Fridays!"
```

What happens when you add a string to a number? How about a number to a string? Why?

A JavaScript calculator doesn't sound terribly interesting, so let's add one more wrinkle to it by introducing variables.

What if the calculator understood variables?

Everything we have been doing above is fun for about three seconds. It is interesting to test edge cases (what happens when you multiply a number with a boolean?), of course, as that is a good way to understand the assumptions the language is making. But we're building websites, not calculators.

Nevertheless, getting a bit of flexibility with the console is useful. Let's expand on that, then, with the `let` statement, which lets us define variables. Type along in the console.

```
> let burrito;
> burrito = "Basically the best food around.";
> console.log(burrito);
//--> Basically the best food around.
```

In the first line, you **defined** a variable, `burrito`. In the second, you **assigned** to the variable the string, "Basically the best food around."[5]

Then you tell the console to log the variable burrito, and it logs its value, `Basically the best food around.`

Any of the data types you have learned about already you can assign to a variable, and we can define multiple variables at once. It's generally good practice to define all your variables at the top, so you know what you will be working with in the future.

```
> let magicNumber, secretNumber;
> magicNumber = 9;
> secretNumber = 10;
> secretNumber + magicNumber;
//--> 19
> secretNumber === magicNumber;
//--> false
```

---

[5] `let` is a statement that is not supported in all JavaScript consoles. If typing the above causes the console to complain `Uncaught SyntaxError: Unexpected identifier`, then you have one of those older consoles. The solution is, luckily, straightforward. For the duration of this course, where I instruct you to use `let`, you can use `var`, instead.

```
> secretNumber > magicNumber;
//--> true
> secretNumber = secretNumber + 1;
//--> 11
> console.log(secretNumber);
//--> 11
```

These variables persist only for the duration of the console. If you close the console, then you'll lose them. But the variables are also mutable, as you can see. `secretNumber` starts out assigned to the number 10, but then it becomes assigned to the number 11.

Let's play a bit more with assigning variables:

```
> let tipRate, bill, billPlusTip;
> tipRate = 0.20;
> bill = 10.00;
> billPlusTip = bill + (tipRate * bill);
> console.log(billPlusTip);
//--> 12
> let question, burritos, answer;
> question = "What is the best food around?\n";
> burritos = "Delicious burritos";
> answer = burritos + " are clearly the best!";
> console.log(question, answer);
//--> What is the best food around?
//--> Delicious burritos are clearly the best!
```

There are two new things in this example: `\n` can be used to make a new line to make a new line, and now you see that `console.log()` can take multiple values in the parentheses, separated by commas.

Finally, this discussion of variables allows me to introduce one more operator, the `typeof` operator. If you get a variable, sometimes you don't know what kind of data type it is. Yet as we have seen with `+`, it behaves differently depending on the data. Continuing with the variables in the previous example:

```
> typeof question;
//--> "string"
> typeof magicNumber;
//--> "number"
> typeof 2;
//--> "number"
> let isItTrue;
> isItTrue = 1 === 1;
> typeof isItTrue;
//--> "boolean"
> typeof badTastingBurrito;
//--> "undefined"
```

As you can see, this example makes use of variables defined in earlier exercises, but it also refers to variable that has not yet been assigned (or even defined).

Exercises

1. Get `console.log()` to log a string that includes a backslash.
2. What happens when you surround `true` or `false` with double quotes? Are they still booleans?
3. What happens when you add a string to a number? What about the reverse? Why?
4. Use the statements to find `billPlusTip` above but have the response from `console.log()` be "You should pay $12 because the service was good."

Footnotes

Programming

It's time to unlock what JavaScript can do when it's not just a calculator. We've skipped over a lot of details about data types to get to here, but it's important to start thinking *programmatically* as soon as possible. Rules and properties we can always look up. How to do things programmatically, however, is a skill that needs to be nurtured. As a result, it's a source for early frustrations as well. Just remember, programming is puzzle solving, and in this part of the process you can finally start thinking of the puzzle as a journey.

Control flow

Control flow is an idea you've probably seen before, like in flow charts. They're all over social media and often funny. In a flow chart, you start from some position and answer questions. Depending on the answers to those questions, you end up in a certain location. Other answers lead you somewhere else. But the idea is that you are interacting with information, in that you are being provided a prompt for some input, and your input directs what happens.

Let's sketch out a toy program to begin illustrating control flow in a program.

We want to write a program that asks the user what they want for dinner. If they answer "burrito," the program congratulates their choice. If they answer anything else, the program scolds them for not wanting a burrito. What might that look like in **pseudocode** (pretend programming that's not a real language)? Let's try it out while also using some JavaScript we already know, like `let` and `console.log()`.

First, the program needs to get the information from the user, so we need some kind of input. Let's save that as a variable.

```
// THIS IS PSEUDOCODE. It is for illustration only. It will crash.
//
let userInput;
userInput = prompt_the_user_for_what_they_want_for_dinner;
```

We have a variable now, `userInput`, that has whatever the user has input. Now let's test that variable, using the `===` operator you've already learned.

```
// THIS IS PSEUDOCODE. It is for illustration only. It will crash.
//
if userInput === "burrito";
  then console.log("Brilliant choice!");
```

OK. But what if the input *isn't* "burrito"?

```
// THIS IS PSEUDOCODE. It is for illustration only. It will crash.
//
```

```
if userInput !== "burrito";
  then console.log("Don't you want a burrito?");
```

Here I'm using the negation operator `!==`. It's the same as `===`, but its inverse.

And that's it. We have our program. The JavaScript, as we'll see, isn't so terribly different from this code we already have.

If statements

An if statement is an example of **conditional statement**. That means that it behaves in a certain way depending on a condition. From the example above, we can read

```
// THIS IS PSEUDOCODE. It is for illustration only. It will crash.
//
if userInput === "burrito";
  then console.log("Brilliant choice!");
```

As "If the condition that the variable `userInput` is equivalent to the string 'burrito' is true, then respond to the console the string 'Brilliant choice!'"

That's a mouthful, but it is actually three distinct steps:

1. The test of whether `userInput` is equivalent to the string "burrito."
2. If the test in 1. is `true`, then the condition in the if statement is met, so we can go on to 3.
3. log to the console the string "Brilliant choice!"

Now, JavaScript's syntax is different from the pseudocode above. First, there is no `then` statement. Instead, that `then` is replaced with braces (`{}`). Everything within the braces gets executed if the truth test (`userInput === "burrito"`) returns `true`. And the truth test itself is surrounded by parentheses. Generically, then, it looks like this:

```
// This code is for illustration only. It will crash your console.
//
if ( truth_test ) {
  do_things_if_the_test_returns_true;
}
```

Note where the semicolons are (and are not!) in this example. Now we can fill it out with some of our toy program. We know that `do_things...` is actually `console.log("Brilliant choice!")`, so we can put that in. As for `truth_test`, let's just put in `true` for now.

```
> if ( true ) {
    console.log("Brilliant choice!");
  }
//--> Brilliant choice!
```

Type this into the console (pressing the return key where appropriate). You should get "Brilliant choice!" logged to your console. Now replace `true` with `false`. What happens? Why?

Now you can add some complexity, this time using the variable `userInput`:

```
> let userInput;
> userInput = "burrito";
> if ( userInput === "burrito" ) {
    console.log("Brilliant choice!");
  }
//--> Brilliant choice!
```

Notice where you have to use = (which *assigns* a value to a variable) and where you have to use === (which *tests* whether something is true). If you type this in the console, again it should congratulate you. If you replace the second line with `userInput = "samosa";`, what happens? Why?

In our program, we had a second condition, which would scold the user if they didn't want a burrito. That's pretty straightforward to write:

```
> let userInput;
> userInput = "samosa";
> if ( userInput !== "burrito" ) {
    console.log("Don't you want a burrito?");
  }
//--> Don't you want a burrito?
```

However, we can *join* both truth tests using `else`:

```
> let userInput;
> userInput = "samosa";
> if ( userInput === "burrito" ) {
    console.log("Brilliant choice!");
  } else {
    console.log("Don't you want a burrito?");
  }
//--> Don't you want a burrito?
```

Now, we can see this as "if the truth test is true, then log 'Brilliant choice!' Otherwise, respond 'Don't you want a burrito?'" Less typing! If you type this snippet into your console, you'll see that it scolds you. What do you have to change so that it congratulates you?

So far so good, but we still have the line of the pseudocode, `userInput = prompt_the_user_for_what_they_want_for_dinner;`. Getting that part to work requires writing some HTML.

Embedding JavaScript in a webpage

Typing things into the console becomes tedious to have to retype everything every time. Fixing mistakes is difficult. And if you close the window (or reload your browser), you lose everything you've already done. There must be a better way to write and execute JavaScript, right? Of course there is.

Back in Chapter one, I had you clone a project with Atom, and that project included a file called `index.html`. Open that file up in Atom by double clicking on it in the Projects tab on the left of the Atom window. Now we need to open this file in a browser, as well. This is a bit tricky, but if you look at the title bar for Atom, it should say something like "Project — ~/github/javascripting-english-major-project." That means the file is in a folder, called `javascripting-english-major-project`. That folder, in turn, is in a folder called `github`, which is in your home folder. On Windows, it's saved somewhere else.

Open up your home folder, like you would to open up any kind of regular folder, and you should see the `github` folder. Open that up, and you should see a folder with your repository. If you open that up, you'll see the file `index.html`. Double click on that, and it should open up in your browser. It'll be boring, but it'll be there. You should be greeted with a blank white webpage with, in large text, "This is my project!"

For the rest of this course, we will be using Atom *and* a web browser, so it's time to get used to having both open at once. Back in Atom, change the text "This is my project" to something else and save your change. Switch over to the browser and press the reload button. The text should change.

Congratulations, you are now a web editor.[1] Let's make you a web developer, though. That involves adding a few lines to `index.html`, so go back to Atom.

Inside the `<body>` tags, underneath the `<h1>`, add:

```
<div id="response">This is the #response div.</div>
<script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
<script src="scripts.js"></script>
```

Type that second line carefully. When you're done, the whole of `index.html` should look like this:

```
<!doctype HTML>
<html lang="en">
  <head>
    <meta charset="utf-8">
```

---

[1] Yes, you are now writing HTML without learning how to do it. The key grammar of the markup is clear from this example, though. HTML is made up of nested tags that look like this, for example: `<h1>` to open and `</h1>` to close. Some tags, like the `<meta>` and `<!doctype>` tags don't need to be closed, but most do.

```
    <title>My JavaScripting English Major Project</title>
  </head>
  <body>
    <h1>This is my project!</h1>
    <div id="response">This is the #response div.</div>
    <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
    <script src="scripts.js"></script>
  </body>
</html>
```

Save and commit, like you learned earlier. A good commit message would be "Add jQuery and local JavaScript."

There are two new tags here, `<div>` and `<script>`. The first is a generic container tag that lets you mark off a part of a page for content. Note that it has an `id` **attribute**, and the value of that attribute is `response`. Throughout, I'll be referring to this `<div>` container as `#response`. The second tag, `<script>`, tells the webpage to look for files indicated by the `src`, or **source**, attribute. In other words, first it goes to a web server, `code.jquery.com`, and downloads a source file from there, called `jquery-3.2.1.min.js`. Then it looks for a local file, on your computer, called `scripts.js`. But that file does not yet exist. So now you can make it.

Back in Atom, right-click (or control-click) on your project icon and choose "New File" from the dropdown menu. A tiny window asking for the file's name will appear, and you should type in `scripts.js`. In the new window that opens, type in:

```
alert("scripts.js has loaded!");
```

Save the file and reload the browser. You should get an alert, telling you that "scripts.js has been loaded!" If you do, stage and commit. A good commit message now would be "Create scripts.js." In the project pane in Atom, you should see `scripts.js` alongside `index.html`.

That's obviously pretty annoying, so change `alert` in `scripts.js` to `console.log`. Now open up the console on the browser and reload the page. As you can see, instead of typing JavaScript directly in the console, we can type it in Atom, instead, and reload our page. But even writing to the console isn't terribly interesting, except when you're testing or debugging. Let's use JavaScript, instead, to write to the webpage *itself*.

There are a lot of ways to do that, but that `<script>` line about "jquery" above will make things a bit easier. jQuery is a powerful JavaScript **library**, meaning it is a set of tools that make programming easier. It lets us developers do many tasks succinctly and clearly. Replace your single line in `scripts.js` with this, then:

```
$("#response").html("scripts.js has loaded!");
```

Now reload the page in the browser. That earlier text, that read "This is the #reponse div." is now replaced. We'll learn jQuery in greater detail as we move along, but that one line of JavaScript does this:

- `$(`: Select something in the webpage with jQuery.

- `"#response"`): In fact, select the HTML element with the `id` of `response`. (The # means are looking for something with a specific `id`.)
- `.html(`: Do something with the HTML that is inside the element we selected.
- `"scripts.js has loaded!");`: Change the HTML with this new HTML.

The jQuery selector, `$("")`, is one of the most important bits of code on the web.[2]

Catch your breath. A whole lot has happened in this section. Let's return to the toy program from the previous section, though, and update the `console.log()` parts with the jQuery. In other words, make your `scripts.js` file look like this:

```
let userInput;
userInput = "samosa";
if ( userInput === "burrito" ) {
  $("#response").html("Brilliant choice!");
} else {
  $("#response").html("Don't you want a burrito?");
}
```

Save and reload in the browser. What happens now? What if you change "samosa" to "burrito" in `scripts.js`, save, and reload?

We're back to where we were at the start of this section, but now instead of logging information to the console, you're changing text on the webpage. Yet there's still that little detail about how to ask the user what they want for dinner.

In other words, we want to change `userInput = "samosa";` in `scripts.js` to `userInput = prompt_the_user_for_what_they_want_for_dinner;`, but in JavaScript, not pseudocode.

There are a lot of ways to get information from the user, but for now we can use JavaScript's `prompt()` function, which asks the user to type something in. This works on most browsers, but some (like Brave) might have it disabled. Change the second line of `scripts.js` to:

```
userInput = prompt("What do you want to have for dinner?", "Type your answer here.");
```

Save and reload. If all goes well, when you reload the page in the browser, you should immediately have a teeny window asking you what you want for dinner. No matter what you type in, unless it's "burrito," the webpage will scold you. If that works, commit.

You've written a full program. It takes in input, feeds it through control flow (in this case, a conditional statement), and delivers output appropriate to the input. Of course, it's still pretty basic, but baby steps. Baby steps.

---

[2] To give a sense of jQuery's pithiness, the same line in vanilla JavaScript would be: `document.getElementById("response").innerHTML = "scripts.js has loaded!";` To my eyes, this is unnecessarily verbose.

While and for loops

If you've got conditionals down, we can now move to looping. Remember, in JavaScript, a conditional statement takes the form of:

```
// This code is for illustration only. It will crash your console.
//
if ( some_condition_is_true ) {
  do_things;
}
```

Loops use the same syntax:

```
// This code is for illustration only. It will crash your console.
//
while ( some_condition_is_true ) {
  do_things;
}

for ( complicated_stuff_we'll_get_to ) {
  do_things;
}
```

While loops

While loops keep executing over and over until the truth test becomes false. As a result, something like:

```
while ( 0 < 1 ) {
  console.log("Zero is less than one.");
}
```

will keep running forever or until your browser crashes. Whichever comes first. As you can see, the while loops can be dangerous if the truth test never stops being true. An if statement gets executed once and moves on. Loops, though. Well, it's in the name!

Yet it's possible to use while loops with a bit of discretion. For example, replace the contents of `scripts.js` with this:

```
let i;
i = 1;
while ( i < 4 ) {
  $("#response").append("<br />" + i);
  i = i + 1;
}
```

Before saving and reloading the webpage, what do you think this does? Now when you reload the page and see what it does, can you figure out why? If yes, then looping already makes sense to you, and you're a step ahead. If no, I'll walk through what's going on here step by step:

```
let i;
i = 1;
```

First we define a variable `i` and assign it the number 1. It's tradition in programming to call looping variables `i`, so I'm continuing that tradition.

```
while ( i < 4 ) {
```

Read that out in English: "while the variable i is less than the number four." That means that as long as `i` is less than 4, the loop will be true, and the program will loop.

```
  $("#response").append("<br />" + i);
```

This line is a bit sneaky. First, it uses `.append()` instead of `.html()`. That just means that it adds the HTML to the end the already existing HTML instead of replacing all of it. Next, it uses `<br />`, which is the HTML tag for making a line break. But note that little `i` at the end. We'll come back to it in a second, but for now it should be clear that this line means "append the HTML line break tag and the value of the variable `i` to the HTML element `#response`."

```
  i = i + 1;
}
```

Finally, and this is what gives the loop its magic, we see this expression. In plain English, this means, "assign to the variable i the value of the variable i plus 1." This may seem like nonsense. After all, you can't have "$x = x + 1$" in algebra class, which is probably the last time you dealt with variables. But in programming, this is allowed. And, in fact, it's super useful.

Let's look at the program in its entirety again:

```
let i;
i = 1;
while ( i < 4 ) {
  $("#response").append("<br />" + i);
  i = i + 1;
}
```

Now we can see that the third line can be simplified, in English, to "append the HTML line break tag and '1' to the `#response` HTML element." Then the fourth line, in English, is "assign to the variable `i` the result of 1 + 1."

At the end of the first time through the loop, then, `i` is now 2, not 1. Now we can see it go back to the beginning of the loop. First, truth test: is `2 < 4` true? Yes. Next, append a line break and "2." Next, reset `i` to be equal to 2 + 1, or 3. Once back through the loop... Now reset `i` to be equal to 3 + 1, or 4 and…

Break. Because `4 < 4` is not true, the loop stops executing. More colloquially, we **break out of the loop**. And that's why your webpage should read:

This is the #response div. 1 2 3

What are some changes we could make if we wanted it to go through the loop four times? Two should be immediately obvious, and both involve changing the truth test. We can change it either to `i < 5` or `i <= 4`. Remember, the <= operator is the same as ≤, or less-than-or-equal-to.

It's important to understand why this looping works the way it does, so it's useful to loop over (as it were) this section until it's clear.

For loops

For loops are just like while loops, except the truth test is replaced by a three part expression relating to a **control variable**. The three parts are:

1. **Initialization**. What is the initial state of the control variable before the loop begins?
2. **Condition**. What is the truth test that the control variable has to pass?
3. **Afterthought**. What is the change made to the control variable each time through the loop?

If we look back at our while loop and consider `i` to be the control variable, we can see that a for loop is just a fancier version of a while loop. After all, `i = 1;` sets the **initial** state of the control variable before the loop begins. Next, `i < 4` is the **condition** that the control variable has to pass in order for the loop to continue. Finally, `i = i + 1;` is the **afterthought**, or the change the control variable undergoes each time through the loop. In other words, you can collapse a five-line program into just three:

```
for (let i = 1; i < 4; i = i + 1 ) {
  $("#response").append("<br />" + i);
}
```

Note where the semicolons are placed, and also note that the two initialization lines

```
let i;
i = 1;
```

are collapsed into one statement, `let i = 1;`. This works fine even outside the while loop, but it's tidier to define your variables and assign them separately.

In addition to being terser, this syntax limits the control variable (`i`) to the loop itself, instead of defining it outside of the loop, like in a while loop.

Loops are tricky, but they're vital to understanding how programming works. Try out the exercises to see how well you understand them.

Exercises

(These are all to be written and tested using your project and the web browser.)

1. Write a program so that when you reload your page, it asks for a number and prints all the numbers from 1 to it, including it.

2. Building on that program, create it so that instead of printing the number, it prints something like:

This is the #response div. 1 is odd. 2 is even. 3 is odd. 1. Building on the previous program, have it print the same, except without that ugly “This is the #response div.” line. 1. Write down what seem to be common mistakes you are making. Are you forgetting to add some aspect of the JavaScript syntax?

Footnotes

Functions

The previous chapter was very conceptual. Loops are tricky to get a hang of, but once you visualize how looping can solve problems, you're definitely on your way to thinking algorithimically, which is to say, programmatically. This chapter is a bit more focused, but it builds on the idea of a **block** of code, like the loop or the if statement. Blocks in JavaScript are always surrounded by braces ({}), and that is true for **functions**, as well.

Function, function, what's your...

Just as looping is useful because it automates repetitive tasks, functions break up your code into smaller pieces. This means that it's easier to reason about your work abstractly and find problems. Additionally, functions also automate repetitive tasks, and that's the way I'll introduce them to you.

Imagine if you want your computer to make burritos. Wouldn't it be great to just tell it "make a burrito" every time you wanted one, instead of saying "get out the tortillas, take one out, place it on the griddle," and so on? We make little tasks in our daily life abstract using the huge figurative power of language as well as our own memories. Computers can't think as abstractly, however. And though they do have memories, we still need to spell things out in detail. Yet once we do it, we're set. We can ask the computer to make us a burrito from then on.

Of course, I don't think computers can make good burritos, but stick with me.

All those burrito-making instructions can be collapsed into a function. That might look something like:

```
// This code is for illustration only. It will crash your console.
//
let makeABurrito;
makeABurrito = function(){
  prepareTortilla();
  addBeans();
  addOnionsAndCilantro();
  // etc.
  rollUpTortilla();
}
```

As you can see, there are syntactic similarities between functions and loops. Both use blocks with braces, and both have parentheses. In fact, this snippet of code is perfectly fine JavaScript. That's because the `makeABurrito()` function does nothing but **call** other functions. We can imagine that a function like `prepareTortilla()` has even more specific steps inside it. But with the function in place, you just have to execute `makeABurrito()` once and be done with it. The internals of the function take care of everything else.

Quickly before moving on, that line `// etc.` is a **comment**. Comments are very useful in programming because they can serve as little messages to yourself (or to other programmers) about what is going on in your program. In JavaScript, everything after two slashes (`//`) to the end of the line is **commented out**. If you want to comment out a whole section of multiple lines, begin it with `/*` and close it with `*/`. Or you can put a `//` in front of every line. I'll start commenting the code I provide, where necessary.

Parameters

Back to the function. Did you notice the parentheses that follow the function name? Where have you seen this kind of of syntax before? We've already gone over `prompt()`, for example, in the previous chapter, and that is, of course, a function.[1] But recall how we typed it:

```
let userInput;
userInput = prompt("What do you want to have for dinner?", "Type your answer
here.");
```

The parentheses aren't empty. Instead, they have two strings in them. From experience, we know that that the first string is what appears at the top of the prompt box, and the second string is what appears where we type our answer. We can abstract out the function, then, as `prompt(promptText, defaultText)`, where `promptText` and `defaultText` are two variables. And, in fact, if we were to rewrite the above as:

```
let promptText, defaultText, userInput;
promptText = "What do you want to have for dinner?";
defaultText = "Type your answer here.";
userInput = prompt(promptText, defaultText);
```

It would work in exactly the same way. These two variables, `promptText` and `defaultText` are **parameters** that we **send** to the function.[2] Giving functions parameters lets us change the internals of the function to let it react to specific instances. Now sometimes I want black beans in my burrito, and sometimes I want pinto beans. Let's add a parameter to `makeABurrito()` to let us specify which beans to use on the fly:

```
// This code is for illustration only. It will crash your console.
//
let makeABurrito;
makeABurrito = function(beansVariable){
  let beansResponse;
  prepareTortilla();
  addBeans();
  addOnionsAndCilantro();
```

---

[1] `console.log()` also looks like a function, but as I mentioned back in chapter 2, `.log()` is a method belonging to the `console` object.

[2] Parameters are also often called **arguments**, but to my ears, that term is more opaque.

```
  beansResponse = "You ordered " + beansVariable + " beans. Good choice!";
  $("#response").html(beansResponse);
  // etc.
  rollUpTortilla();
}
```

If we were to execute:

```
// This code is for illustration only. It will crash your console.
//
let blackBeans;
blackBeans = "black";
makeABurrito(blackBeans);
```

We would see that the webpage would now read "You ordered black beans. Good choice!" Don't actually try this, yet, though. Can you see why that is the case? We define a variable, `blackBeans` and assign it to the string `"black"`. Next, we send (or **pass**) that variable as a parameter to `makeABurrito()`. Now, inside the function, we see that it makes reference to a `beansVariable`, that has the value "black," which it then prints in `#response`, like we did last chapter.

But where did `beansVariable` come from? And how did it get set to "black"? There's no `beansVariable = "black"`, after all. The answer is that the variable is defined at the same time as the function is. `makeABurrito = function(beansVariable){}` defines both the function, `makeABurrito()`, and the parameter, `beansVariable`, which can be used as a variable inside the function.

The number of parameters you can define is up to you. Imagine you had different kinds of tortillas, like whole wheat and regular wheat. You can redefine the function as:

```
// This code is for illustration only. It will crash your console.
//
let makeABurrito;
makeABurrito = function(beansVariable, tortillaVariable){
  prepareTortilla(tortillaVariable);
  addBeans(beansVariable);
  addOnionsAndCilantro();
  // etc.
  rollUpTortilla();
}
```

What would happen if you executed `makeABurrito("black", "whole wheat")`? Notice how this looks rather similar to `prompt(promptText, defaultText);`?

Back to numbers

`makeABurrito()` is a great function, and it's making me hungry, so let's abandon it for a bit and go back to using numbers. Back in Chapter 2, we made a tipping calculator. We can build on that example with real, usable code.

First, what information do you need in order to know how much to tip? You need to know the `total` and the `tipRate`, which is a percentage, like 15 or 20%. The barebones function looks like this, then:

```
let tipCalculator;
tipCalculator = function(total, tipRate){
  // 1. Calculate the percentage of the total
  // as a variable "tipAmount"
  //
  // 2. Change #response to tell us the tip
  // amount.
}
```

In fact, type this into your `scripts.js` file. It can go after the earlier code you've written, or it can replace it. Note what's in the comments; it's a sketch of what the function will do. The first step is that it'll do some math—an easy calculation. In the second step, it will tell us what the result of the calculation is.

```
let tipCalculator;
tipCalculator = function(total, tipRate){
  // step 1:
  let tipAmount;
  tipAmount = tipRate * total;
  // and step 2:
  $("#response").html("Your tip is $" + tipAmount);
};

// Now call (or "execute") the function, passing a
// total of $50.00 and a tipRate of 20%:

tipCalculator(50.00, 0.2);
```

Save and reload, and the webpage should now inform you that you owe $10. If it does, commit. In the exercises, we'll expand on this function.

Scope

Alongside the idea of a function block, that is, the set of curly braces, we also have the idea of **scope**. Beginner programmers often get tripped up by scope, but that's ok, so do veteran programmers. Note that though conditionals and loops also use blocks, they don't affect scope in the same way.

Just like a "scope" is used to see things that are far away (tele*scope*) or are really tiny (micro*scope*), in JavaScript, scope also has to do with visbility. Variables defined with the `let` keyword have block-level scope.[3] That means:

```
> let global;
> global = " ";
> if (true) { console.log(global); };
//-->
> let globalFunction;
> globalFunction = function(){ console.log(global); };
> globalFunction();
//-->
```

When we define `global`, it's *visible* to us inside subsequent if statements and functions. But notice this:

```
> let global;
> global = " ";
> if (true) {
    let blocky;
    blocky = " ";
    console.log("global is " + global);
    console.log("blocky is " + blocky);
  }
//--> global is
//--> blocky is
> console.log("Wait, the value of blocky is really " + blocky + "?");
```

This last line will crash with a `ReferenceError`. `blocky`, it turns out, is only visible *within* the if statement block. Once the code leaves the `{}`, `blocky` is no longer available. The same works with functions:

```
> let global;
> global = " ";
> let showMeABurrito = function(){
    let burrito;
    burrito = " ";
    global = "I'm global!";
    console.log("global is " + global);
    console.log("burrito is " + burrito);
  }
> showMeABurrito();
//--> global is I'm global!
//--> burrito is
```

---

[3] This is one difference between `var` and `let`, but I'm not teaching `var` except for those of you with old browsers. Other than the slightly contrived examples in this section, my examples won't be making a lot of use of block-scoping.

```
> console.log(global);
//--> I'm global!
> console.log("Wait, the value of burrito is really " + burrito + "?");
```

Crash. Again. Sadly, `burrito is not defined`. But notice that `global` was changed inside the function, and that change persisted outside the function. `global` is visible—and, as a result, changeable (or **mutable**)—everywhere. But `burrito` is not.

As Molly Bloom asks, "who's he when he's at home?" and we may, also, ask, "where are we when we're in the console?" That is, if `burrito` is defined in the function block, where on earth is `global` defined? Even though we talk about typing "in" the console, we're actually always within a special object called `window`. It is the window of the browser, and when we open up the console, we're getting closer access to that window. In fact, `prompt()` and `alert()`, two functions you've already seen, are actually **methods** that belong to `window`; `window.alert()` and `alert()` will do the same thing. But more on methods next chapter. The `window` is typically ignored, as it's the frame that is unavoidable. Its ubiquity gives it the privilege of being silent.

Back to the purpose of this section. Scope helps you keep your code tidy, because there is less risk of variables' being accessed where they shouldn't be. Just remember, whenever you type `let` to define a variable inside a function, that variable is only available inside that function.

In closing, functions are powerful things, as we can see. And though, ultimately, the goals of this course are not to write code that is as modular as the use of functions would make possible, you will still be typing the word `function` a lot.

Exercises

1. Add functionality to the tip calculator so that you can enter "20" *or* ".2" for 20%, and the calculator understands the difference.

Footnotes

Collections of Data

"Data," you may recall, is a plural. Just like "bacteria" is a collection of many instances of a single "bacterium," so it is with data. Data are a collection of single "datums." So far, we've been working for the most part with just single pieces of information, like a single number. We've been using that number with other numbers, of course, but not as a collection of information.

A collection of information combines to form something larger than it itself. Perhaps the easiest collection to think of in programming terms is a list, like, say, a to-do list. The items on the list don't interact with each other, necessarily (what does "schedule dentist appointment" have to do with "buy kitty litter"?), but, in a list, they can be ordered and shuffled around. Furthermore, by being in a list, each chore is exposed to **iterability**, the ability to loop over them.

Iterability it a crucial concept in programming. We've already seen it in action with while and for loops, but with collections of data, it becomes even more powerful. Let's take another list, a list of friends. It's your birthday, and you want to invite them to a party. But you also want the invitations to be "personalized." You could iterate over (loop over) your list, get your friends' first names, and then use that name in an email that opens "Dear FIRST_NAME." They each get the same email, but the first name matches their own.

Arrays

In JavaScript, the simplest list data structure is called an **array**. Arrays are common in programming languages, and they are typically designed to be extremely fast at sorting and iterating. In JavaScript, we're not so lucky; arrays don't offer the same kind of speed benefits. Still, they are crucial, and we'll be seeing them a lot from now on.

Simply put, an array is a set of pieces of data surrounded by brackets (`[]`). The following are all valid arrays, and type them into your `scripts.js` (you can delete everything from the previous chapter):

```
let arrayOfStrings, arrayOfNumbers, arrayMixed;
arrayOfStrings = ["a", "b", "c"];
arrayOfNumbers = [1, 2, 3];
arrayMixed = ["a", 1, null, true, arrayOfNumbers, [4.5, 5.6]];
```

Notice that you are not limited to a single data type in an array. Strings, numbers, `null`, `true`, variables, and even other arrays can be used as the contents of arrays. Later on, when we start building maps, we will have at least one array made up of geographical points.

Each item in an array can be accessed by its **index**, which is an integer unique to that item. The indices begin with 0, which is confusing for beginners. So if you add to the above:

```
$("#response").html(arrayOfStrings[2]);
```

`#response` will read "c." The third value of the array is `"c"`, but its index is 2, because the index begins with 0. So to get "a," we would call `$("#response").html(arrayOfStrings[0]);`. We are interested in the zeroth value. Again, I know this is confusing, but you will get the hang of it with practice, and then you can join that exclusive club of people who make jokes about zero-based numbering.

Objects

Above, when I mentioned that arrays are a bit peculiar in JavaScript, that is because arrays are a simplified version of the JavaScript **object**. Because "object" is such a common word in English, in this text, from now on, when you see `Object`, you know that I mean, specifically, this generic JavaScript data type. Where arrays are pieces of data surrounded by brackets, `Objects` are surrounded by braces ({}). Similarly, while arrays have indices, `Objects` have **properties**. Let's define an `Object`.

```
let myBurritoObject;
myBurritoObject = {
  tortilla: "wheat",
  guacamole: true,
  beans: "pinto",
  habaneroSauceSquirts: 3
  };
// and let's access a property
$("#response").html(myBurritoObject["tortilla"]);
```

With an array, we call it using the syntax `arrayName[indexNumber]`. With an `Object`, we replace the index with a property. But we can do even better:

```
$("#response").html(myBurritoObject.tortilla);
```

It's much more succinct to use **dot-notation** to access properties.[1] In fact, for the rest of this text, whenever I refer to a property, I'll refer to it as a `.property`. Properties are especially useful because arrays have them as well. For example, every array has a `.length` property:

```
let arrayOfStrings;
arrayOfStrings = ["a", "b", "c"];
$("#response").html(arrayOfStrings.length);
```

This will print "3," because the value of that array's `.length` property, or its length, is three. So even though the largest *index* value in the array is 2, its length is 3.

`Objects` can contain other `Objects`, of course, but we really start cooking when we build arrays of `Objects`. Those points on a map I mentioned before? They will be an array of `Objects`, where each `Object` has properties that give its place name and its coordinates.

---

[1] Dot-notation does not work, however, for index values. `arrayOfStrings.1` will cause an error.

Methods

JavaScript, like Ruby, is famous because in both languages, *everything* is an `Object`. Objects are `Objects`, arrays are `Objects`, strings are `Objects` (in that they have properties, as we'll see below), `null` is an `Object`, and even functions are `Objects`. Since a property of an `Object` can be any other kind of `Object`, that means that a property can even be a function. For example, to return to `myBurritoObject`, you can add a new property:

```
let myHabaneroSauceSquirts, myBurritoObject;
// First, define and assign a variable for how
// spicy the burrito is.
myHabaneroSauceSquirts = 3;
// Now assign the burrito object.
myBurritoObject = {
  tortilla: "wheat",
  guacamole: true,
  beans: "pinto",
  // Make use of the variable above.
  habaneroSauceSquirts: myHabaneroSauceSquirts,
  // Use the variable again in a function.
  spiciness: function(){
    if (myHabaneroSauceSquirts > 0 ){
      alert("This is a spicy burrito!");
    } else {
      alert("This is a mild burrito.");
    }
  }
};
$("#response").html("Your burrito has " +
  myBurritoObject.habaneroSauceSquirts +
  " squirts of habanero.");
myBurritoObject.spiciness();
```

Save, and reload, and see what happens. If you're told the burrito is spicy, commit. Now let's have a look at the two new things I'm presenting here. The property here, `.spiciness` is actually a function, and it is defined in a way similar to how we have been defining functions all along. That is, all along we have been writing **anonymous functions**. They are anonymous in that they are ephemeral. They exist and then they're gone. When we create a function like:

```
let makeABurrito = function(){
  // Do stuff.
};
```

The `function(){}` part of it disappears into the variable `makeABurrito`. We can then resurrect it using `makeABurrito()`. In `.spiciness`, however, we are assigning the function to a *property*, not even a variable. Later, we will make even more ephemeral anonymous functions, where the function gets called, executed, and then disappears, without even a

variable or property to resurrect it. But anonymous functions pop up all over the place in JavaScript, which is why I promised last chapter that you would be typing `function` a lot.

When properties are functions, they are called **methods**. Methods are built into the `Object`. To use an example we've already seen, every `console` `Object` has the `.log()` method built in. Arrays also have a series of useful methods:

```
let turtles, sortedTurtles, reversedTurtles, turtleNames;
turtles = ["Leonardo", "Donatello", "Raphael", "Michelangelo"];
sortedTurtles = turtles.sort();
// sortedTurtles is:
// ["Donatello", "Leonardo", "Michelangelo", "Raphael"]
reversedTurtles = turtles.reverse();
// reversedTurtles is:
// ["Raphael", "Michelangelo", "Donatello", "Leonardo"]
turtleNames = turtles.join(" ");
// turtleNames is "Leonardo Donatello Raphael Michelangelo"
turtles.push("Splinter");
// turtles is now:
// ["Leonardo", "Donatello", "Raphael", "Michelangelo", "Splinter"]
turtles.pop();
// back to ["Leonardo", "Donatello", "Raphael", "Michelangelo"]
```

Note that `.sort()`, `.reverse()`, and `.join()` do not change the value of `turtles`. Instead, we define new variables, `sortedTurtles`, `reversedTurtles`, and `turtleNames`. Then we assign to those variables two new arrays and a string. `.pop()` and `.push()`, however, *do* change `turtles`.

Strings as arraylike things

Because everything is an `Object`, that includes strings. Strings can behave a bit like arrays, but they also, as `Objects`, have properties and methods. I'll mention a few here, because manipulating strings (or "text") is a vital feature of writing web pages.

```
let string, firstLetter, stringLength;
string = "This is a string.";
// Strings have indices and lengths, just like arrays:
firstLetter = string[0];
// firstLetter is "T"
stringLength = string.length;
// stringLength is 17
//
// Strings also have methods, just like arrays:
let upperCaseString, replacedString;
upperCaseString = string.toUpperCase();
// upperCaseString is "THIS IS A STRING."
replacedString = string.replace("string", "pipe");
// replacedString is "This is a pipe."
```

Exercises

1. Write a function that always returns the last item in whatever array you pass it.
2. Numbers also have methods and properties. Look them up at MDN and change your webpage so that it asks for a number and tells you if it is an integer or not.

Footnotes

Abstraction

Between strings, numbers, booleans, functions, arrays, and `Objects`, we have the fundamentals of JavaScript down. These are the pieces that build together to make all JavaScript web projects. But knowing how to put the pieces together is where programming stops being a series of clunky, step-by-step instructions and becomes creative expression.

Recall the example from chapter 4 of teaching the computer to make a burrito. One way to do it could be to list out every command, step by excruciatingly small step. Think about step 1: "get a tortilla." What is a tortilla? What does it mean "to get"? Well, it seems that's not step 1. Because there are steps that come before, that define both of those things. Abstraction is when you start combining these little chunks of code into something more elegant, something that reads more like English.

Furthermore, abstracting also leads to reusable code. Why not define a function `get()` in getting a tortilla, that you can then reuse for getting beans, for getting cheese, for getting guacamole? Define it once, reuse it forever.

Another way of thinking about abstraction is through a programming philosophy called **DRY**, for "Don't Repeat Yourself." If you find yourself writing the same kind of code over and over, it means you haven't thought about the problem abstractly enough to realize moments where your code could benefit from abstraction.

If this all sounds abstract (as it were), we'll get to some details in a bit. However, first we need to look back at functions and methods and learn an important detail I left out.

Returning

In the previous chapter, when working with array methods, I asked you to think about why `.push()` and `.pop()` change the array, while `.sort()` and `.reverse()` do not. The why of the question is for discussion (and hopefully was already discussed in class), but now we have to think about the *how*.

Open up the JavaScript console, and type in `let one; one = 1;`. When you hit return, the console should read `undefined`. But now just type `one;` and hit return. Now the console responds with `1`, which is what we assigned to the variable `one`.

In these two examples, `undefined` and `1` are the **return values** of the two commands you send the console. Every statement in JavaScript has a return value. The default is `undefined`, but with functions, you can set your own return value using the `return` keyword. For example, we have been using `console.log()` to write to the console so far, but try this in the console:

```
> let f;
> f = function(){ return "I am a return value." };
> f();
```

First, note that I collapsed the entire function definition to one line. But, second, note that the function prints text to the console. Let's build on this by writing a function that combines someone's first and last names into a full name with a space between them:

```
> let makeFullName, hughessFullName;
> makeFullName = function(firstName, lastName){
    firstName + " " + lastName;
  }
> hughessFullName = makeFullName("Langston", "Hughes");
> console.log("Is your name " + hughessFullName + "?");
//--> Is your name undefined?
```

The function is doing stuff, but we don't get the result we want. We want the console to ask, "Is your name Langston Hughes?" not "Is your name undefined?" The problem is that the variable, `hughessFullName`, is assigned to `undefined`. But we want, instead, for that variable to hold the value `"Langston Hughes"`. We do this by telling the function to **return** the string.

```
> let makeFullName, hughessFullName;
> makeFullName = function(firstName, lastName){
    return firstName + " " + lastName;
  }
> hughessFullName = makeFullName("Langston" "Hughes");
> console.log("Is your name " + hughessFullName + "?");
//--> Is your name Langston Hughes?
```

Knowing the return value of a function helps you manipulate it with confidence. For example, we know that `.sort()` and `.reverse()` *return* new arrays, leaving the original array unchanged. Since we know this, we can even **chain** the two methods:

```
> let turtles, sortedReversedTurtles;
> turtles = ["Leonardo", "Donatello", "Raphael", "Michelangelo"];
> sortedReversedTurtles = turtles.sort().reverse();
//--> ["Raphael", "Michelangelo", "Leonardo", "Donatello"]
```

Say we accidentally included Splinter in the list of turtles, and decided to `.pop()` him off before reversing:

```
> let turtlesWithSplinter, reversedTurtlesWithoutSplinter;
> turtlesWithSplinter = ["Leonardo", "Donatello", "Raphael", "Michelangelo",
"Splinter"];
> // oops. let's pop() Splinter off before reversing…
> reversedTurtlesWithoutSplinter = turtlesWithSplinter.pop().reverse();
```

Uh-oh. This causes an error. Why does this happen? After all, we're just telling JavaScript to pop off the last value of the array, then sort it, and then reverse it. The answer is that, although `.sort()` and `.reverse()` return *arrays*, `.pop()` does not. It returns the *value* of

the popped off element in the array. In other words, we're asking JavaScript to run `.reverse()` on `"Splinter"`, which is a string. And strings have no `.reverse()` method. Error ensues.

Return values encourage programmers to think in terms of the effect of the way their functions manipulate data. This is valuable when we start to talk about iteration.

Iterating

We know that strings can behave a bit like arrays, in that they have a `.length` property and index values. They also have the method `.toUpperCase()`, which makes a string all caps. What if we wanted to write a function that made every letter upper case *except* "e"? Let's close the console for now and work this out in Atom. Erase your `scripts.js` and type in this:

```
let userString, upperCaseMinusE, upperCasedString;
// First, we need a string from the user.
userString = prompt("What do you want to UPPeRCASe?");
// Second, we need to create our function.
upperCaseMinusE = function(string){
  // Something will happen here…
};
// Third, we need to pass the user's string to the
// function and assign the return value to a
// variable.
upperCasedString = upperCaseMinusE(userString);
// And we can then print the string to the webpage.
$("#response").html(upperCasedString);
```

Perhaps the easiest way to write the function would be to have it upper case the whole string at once and then just replace every "E" with "e." But say the rules were that if a user *enters* an "E," like "uppercase everything but the little 'e's and leave this 'E' alone," then it wouldn't work. Instead, let's go over the string, letter by letter, and uppercase each letter on its own, while skipping the letter whenever it is "e." That should be easy enough to write:

```
upperCaseMinusE = function(string){
  if ( letter === "e" ) {
    result = letter;
  } else {
    result = letter.toUpperCase();
  }
};
```

This code won't yet work, mostly because it references two variables, `letter`, and `result` that are undefined. But the mechanics should be clear. Given a variable `letter`, if it's equal to "e," let it be and set `result` to it. If it's not, make it upper case. Now, how do we iterate over it? We use a for loop. Remember, a for loop takes three parameters: the initialization, the condition, and the afterthought. So let's add a for loop to our function:

```
upperCaseMinusE = function(string){
  for ( let i = 0; i < string.length ; i = i + 1 ) {
    if ( letter === "e" ) {
      result = letter;
    } else {
      result = letter.toUpperCase();
    }
  }
};
```

`letter` still isn't defined, but we're at least iterating over the string. Notice that the condition is that our counter variable, `i`, be less than the length of the string, which we get by asking for the `string.length`. But why not set `i` to 1, and make the condition `i <= string.length`? The answer has to do with zero indexing. Recall that to get the first letter of a string, we need to ask it for `string[0]`. So the first time through the loop, we want `i` to be 0, so that we can ask for `string[i]` and get the zeroth letter in the string. And that's what we'll assign to `letter`!

```
upperCaseMinusE = function(string){
  for ( let i = 0; i < string.length ; i = i + 1 ) {
    let letter;
    letter = string[i];
    if ( letter === "e" ) {
      result = letter;
    } else {
      result = letter.toUpperCase();
    }
  }
};
```

`i`, of course, is defined for the course of the loop, and it increases every time through it. As a result, we get access to each letter in our `string` variable by calling `string[i]` on it every time through the loop, when `i` has a different value. But that `result` variable is still undefined, and it's still not doing anything. `result` will be what we return from the function. And we know that what we return will be a string, so let's define `result` at the beginning of the function and assign it to a blank string. `scripts.js` should now look like this:

```
let userString, upperCaseMinusE, upperCasedString;
userString = prompt("What do you want to UPPeRCASe?");
upperCaseMinusE = function(string){
  let result;
  result = "";
  for ( let i = 0; i < string.length ; i = i + 1 ) {
    let letter;
    letter = string[i];
    if ( letter === "e" ) {
      result = letter;
    } else {
```

```
      result = letter.toUpperCase();
    }
  }
  return result;
};
upperCasedString = upperCaseMinusE(userString);
$("#response").html(upperCasedString);
```

At last, the code isn't broken any longer, so you can save, commit, and reload the webpage. But if you try it, you'll see that it only prints one letter to the webpage… the last one. Can you see why? We want to be *adding* each letter to the `result` variable every time we step through the loop, so we simply have to change two lines:

```
let userString, upperCaseMinusE, upperCasedString;
userString = prompt("What do you want to UPPeRCASe?");
upperCaseMinusE = function(string){
  let result;
  result = "";
  for ( let i = 0; i < string.length ; i = i + 1 ) {
    let letter;
    letter = string[i];
    if ( letter === "e" ) {
      // Change here.
      result = result + letter;
    } else {
      // And change here.
      result = result + letter.toUpperCase();
    }
  }
  return result;
};
upperCasedString = upperCaseMinusE(userString);
$("#response").html(upperCasedString);
```

Save and reload, and you'll see that it works now just as we would have hoped. If that's the case, go ahead and commit.

Take a break. We've just done a lot. Have a look over the code and make certain you understand what is going on in every line. For the for loop, try writing out the values of `result`, `letter`, `i`, and `string` for every step through with a made up value for string, like "uppErcase me!"

Arrays of `Objects`

I hope you enjoyed your break. Iterating over arrays is a vitally important aspect of programming. In fact, it's so common that JavaScript has a special method, `.forEach()`, for iterating over arrays. We could rewrite the function in the previous section this way:

```
upperCaseMinusE = function(string){
  let result, stringArray;
```

```
  result = "";
  // Since forEach() only works on arrays, we have
  // to convert the string to an array:
  stringArray = string.split("");
  // Now we call forEach() on stringArray:
  stringArray.forEach(function(letter){
    if ( letter === "e" ) {
      result = result + letter;
    } else {
      result = result + letter.toUpperCase();
    }
  }) // Note the parenthesis!
  return result;
};
```

The savings in terms of typing aren't that great, but `.forEach()` becomes far more valuable with more complicated arrays.

Let's imagine that our turtles have cards that tell you about them. On each card, we see the turtle's name, his favorite color, and his weapon of choice. We can create these cards as JavaScript `Objects`:

```
let leonardo, donatello, raphael, michelangelo, turtles;
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};
donatello = {name: "Donatello", color: "purple", weapon: "bo"};
raphael = {name: "Raphael", color: "red", weapon: "sai"};
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};
turtles = [leonardo, donatello, raphael, michelangelo];
```

Each turtle has three properties, `.name`, `.color`, and `.weapon`. And then we put all four turtles into an array, `turtles`. Now let's say we want a list of their weapons on the webpage. In `scripts.js`, type out:

```
let leonardo, donatello, raphael, michelangelo, turtles, weapons;
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};
donatello = {name: "Donatello", color: "purple", weapon: "bo"};
raphael = {name: "Raphael", color: "red", weapon: "sai"};
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};
turtles = [leonardo, donatello, raphael, michelangelo];
weapons = ""; // a list of weapons.
$("#response").html(weapons);
```

Of course, `weapons` is blank for the time being, so `#response` on the webpage will just be blank. How can we get the list of weapons, though? We need to iterate over the list of turtles and get each turtle's individual weapon. Then we can put those together into the `weapons` string and be on our way. Under `weapons = "";`, we can add:

```
turtles.forEach(function(turtle){
  weapons = weapons + turtle.weapon + " ";
})
```

Weapons starts out blank, but then every time through the `.forEach()` loop, it gets its previous value, plus the value of the turtle's `.weapon` property, plus a space. If you save and reload the browser, you should now get a list of all the turtles' weapons. This is great, but we can do better.

Mapping and filtering

Let's say we wanted not only the list of weapons, but we also wanted it in alphabetical order. How could we do that? Well, we know that arrays have the `.sort()` method, but in order to sort the weapons, we need an array of just the weapons' names. Currently, `turtles` is an array of `Objects` (one for each turtle) and `weapons` is a string. Instead of `.forEach()`, we can make use of the `.map()` method. You use `.map()` whenever you want to build an array out of another array. It's a more specific version of `.forEach()`, but you use it the same way. You write an anonymous function that takes as its first parameter the current array item over which you're iterating.

```
let leonardo, donatello, raphael, michelangelo, turtles, weapons;
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};
donatello = {name: "Donatello", color: "purple", weapon: "bo"};
raphael = {name: "Raphael", color: "red", weapon: "sai"};
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};
turtles = [leonardo, donatello, raphael, michelangelo];
weapons = turtles.map(function(turtle){
  return turtle.weapon;
});
// weapons is now ["katana", "bo", "sai", "nunchaku"]
$("#response").html(weapons);
```

Now that `weapons` is an array instead of a string, that means we can also run `.sort()` on it:

```
let leonardo, donatello, raphael, michelangelo, turtles, weapons;
leonardo = {name: "Leonardo", color: "blue", weapon: "katana"};
donatello = {name: "Donatello", color: "purple", weapon: "bo"};
raphael = {name: "Raphael", color: "red", weapon: "sai"};
michelangelo = {name: "Michelangelo", color: "blue", weapon: "nunchaku"};
turtles = [leonardo, donatello, raphael, michelangelo];
weapons = turtles.map(function(turtle){
  return turtle.weapon;
}).sort();
// weapons is now ["bo", "katana", "nunchaku", "sai"]. Sorted!
$("#response").html(weapons);
```

And the webpage is printing "bo,katana,nunchaku,sai", which isn't bad, but it looks a bit weird. Let's replace those commas with commas and spaces:

```
weapons = turtles.map(function(turtle){
  return turtle.weapon;
}).sort().join(", ");
// weapons is now "bo, katana, nunchaku, sai". Sorted, with commas.
$("#response").html(weapons);
```

Notice that `weapons` is no longer an array. It is now a string. That is because the array's `.join()` method creates a string out of an array, where it glues the array's pieces together using the parameter sent to it, in this case `,`. `.join()` is the opposite of `.split()`, which turns a string into an array, splitting on the parameter sent to it.

The `.map()` method opens up possibilities for manipulating data, because as it gives us a new array, we can use other methods inherent to arrays to work on the new data. One such method is `.filter()`.

Say we want a list of the turtles' names, but only if their names have the letter "o." Getting the names is easy; it's no different than getting the weapons:

```
let names;
names = turtles.map(function(turtle){
  return turtle.name;
}).sort().join(", ");
$("#response").html(names);
```

This gets us most of the way there, but Raphael is in the list, and we want him gone. `.filter()` works just like `.map()`, but instead of returning a value, it returns the value over which it is iterating only if it meets a conditional, like an if statement. Now, strings have a method, `.includes()`, that returns `true` if the string includes whatever the parameter is. Let's add some code, then.

```
let names, namesWithO;
names = turtles.map(function(turtle){
  return turtle.name;
}).sort();
namesWithO = names.filter(function(name){
  return name.includes("o");
}).join(", ");
$("#response").html(namesWithO);
```

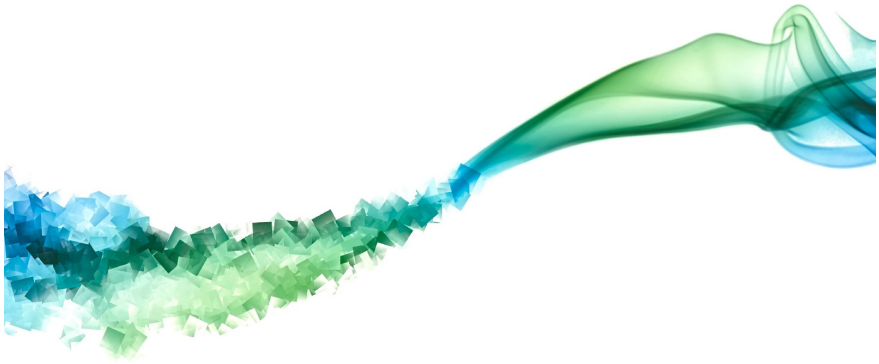
Because `"Leonardo".includes("o")` returns `true`, that name is included in the list. Because `"Raphael".includes("o")` returns `false`, it is not included.

Exercises

1. Write a function that takes an array of integers and, using `.map()`, returns an array of those integers, doubled. So if we give it `[1, 2, 3]`, we receive, in turn, `[2, 4, 6]`.
2. Add functionality to the weapons examples above so that the final result is "bo, katana, nunchaku, and sai."

Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
Feb 24, 2017 · 13 min read

A Functional Programmer's Introduction to JavaScript (Composing Software)



Smoke Art Cubes to Smoke — MattysFlicks — (CC BY 2.0)

> *Note: This is part of the "Composing Software" series on learning functional programming and compositional software techniques in JavaScript ES6+ from the ground up. Stay tuned. There's a lot more of this to come!*
> *< Previous* | *<< Start over at Part 1* | *Next >*

For those unfamiliar with JavaScript or ES6+, this is intended as a brief introduction. Whether you're a beginner or experienced JavaScript developer, you may learn something new. The following is only meant to scratch the surface and get you excited. If you want to know more, you'll just have to explore deeper. There's a lot more ahead.

The best way to learn to code is to code. I recommend that you follow along using an interactive JavaScript programming environment such as CodePen or the Babel REPL.

Alternatively, you can get away with using the Node or browser console REPLs.

Expressions and Values

An expression is a chunk of code that evaluates to a value.

The following are all valid expressions in JavaScript:

```
7;

7 + 1; // 8

7 * 2; // 14

'Hello'; // Hello
```

The value of an expression can be given a name. When you do so, the expression is evaluated first, and the resulting value is assigned to the name. For this, we'll use the `const` keyword. It's not the only way, but it's the one you'll use most, so we'll stick with `const` for now:

```
const hello = 'Hello';
hello; // Hello
```

var, let, and const

JavaScript supports two more variable declaration keywords: `var`, and `let`. I like to think of them in terms of order of selection. By default, I select the strictest declaration: `const`. A variable declared with the `const` keyword can't be reassigned. The final value must be assigned at declaration time. This may sound rigid, but the restriction is a good thing. It's a signal that tells you, "the value assigned to this name is not going to change". It helps you fully understand what the name means right away, without needing to read the whole function or block scope.

Sometimes it's useful to reassign variables. For example, if you're using manual, imperative iteration rather than a more functional approach, you can iterate a counter assigned with `let`.

Because `var` tells you the least about the variable, it is the weakest signal. Since I started using ES6, I have never intentionally declared a `var` in a real software project.

Be aware that once a variable is declared with `let` or `const`, any attempt to declare it again will result in an error. If you prefer more experimental flexibility in the REPL (Read, Eval, Print Loop) environment, you may use `var` instead of `const` to declare variables. Redeclaring `var` is allowed.

This text will use `const` in order to get you in the habit of defaulting to `const` for actual programs, but feel free to substitute `var` for the purpose of interactive experimentation.

Types

So far we've seen two types: numbers and strings. JavaScript also has booleans (`true` or `false`), arrays, objects, and more. We'll get to other types later.

An array is an ordered list of values. Think of it as a box that can hold many items. Here's the array literal notation:

```
[1, 2, 3];
```

Of course, that's an expression which can be given a name:

```
const arr = [1, 2, 3];
```

An object in JavaScript is a collection of key: value pairs. It also has a literal notation:

```
{
  key: 'value'
}
```

And of course, you can assign an object to a name:

```
const foo = {
  bar: 'bar'
}
```

If you want to assign existing variables to object property keys of the same name, there's a shortcut for that. You can just type the variable name instead of providing both a key and a value:

```
const a = 'a';
const oldA = { a: a }; // long, redundant way
const oA = { a }; // short an sweet!
```

Just for fun, let's do that again:

```
const b = 'b';
const oB = { b };
```

Objects can be easily composed together into new objects:

```
const c = {...oA, ...oB}; // { a: 'a', b: 'b' }
```

Those dots are the object spread operator. It iterates over the properties in `oA` and assigns them to the new object, then does the same for `oB`, overriding any keys that already exist on the new object. As of this writing, object spread is a new, experimental feature that may not be available in all the popular browsers yet, but if it's not working for you, there is a substitute: `Object.assign()`:

```
const d = Object.assign({}, oA, oB); // { a: 'a', b: 'b' }
```

Only a little more typing in the `Object.assign()` example, and if you're composing lots of objects, it may even save you some typing. Note that when you use `Object.assign()`, you must pass a destination object as the first parameter. It is the object that properties will be copied to. If you forget, and omit the destination object, the object you pass in the first argument will be mutated.

In my experience, mutating an existing object rather than creating a new object is usually a bug. At the very least, it is error-prone. Be careful with `Object.assign()`.

Destructuring

Both objects and arrays support destructuring, meaning that you can extract values from them and assign them to named variables:

```
const [t, u] = ['a', 'b'];
t; // 'a'
u; // 'b'
```

```
const blep = {
  blop: 'blop'
};

// The following is equivalent to:
// const blop = blep.blop;
const { blop } = blep;
blop; // 'blop'
```

As with the array example above, you can destructure to multiple assignments at once. Here's a line you'll see in lots of Redux projects:

```
const { type, payload } = action;
```

Here's how it's used in the context of a reducer (much more on that topic coming later):

```
const myReducer = (state = {}, action = {}) => {
  const { type, payload } = action;
  switch (type) {
    case 'FOO': return Object.assign({}, state, payload);
    default: return state;
  }
};
```

If you don't want to use a different name for the new binding, you can assign a new name:

```
const { blop: bloop } = blep;
bloop; // 'blop'
```

Read: Assign `blep.blop` as `bloop`.

Comparisons and Ternaries

You can compare values with the strict equality operator (sometimes called "triple equals"):

```
3 + 1 === 4; // true
```

There's also a sloppy equality operator. It's formally known as the "Equal" operator. Informally, "double equals". Double equals has a valid use-case or two, but it's almost always better to default to the `===` operator, instead.

Other comparison operators include:

- `>` Greater than
- `<` Less than
- `>=` Greater than or equal to
- `<=` Less than or equal to

- `!=` Not equal
- `!==` Not strict equal
- `&&` Logical and
- `||` Logical or

A ternary expression is an expression that lets you ask a question using a comparator, and evaluates to a different answer depending on whether or not the expression is truthy:

```
14 - 7 === 7 ? 'Yep!' : 'Nope.'; // Yep!
```

Functions

JavaScript has function expressions, which can be assigned to names:

```
const double = x => x * 2;
```

This means the same thing as the mathematical function `f(x) = 2x`. Spoken out loud, that function reads `f` of `x` equals `2x`. This function is only interesting when you apply it to a specific value of `x`. To use the function in other equations, you'd write `f(2)`, which has the same meaning as `4`.

In other words, `f(2) = 4`. You can think of a math function as a mapping from inputs to outputs. `f(x)` in this case is a mapping of input values for `x` to corresponding output values equal to the product of the input value and `2`.

In JavaScript, the value of a function expression is the function itself:

```
double; // [Function: double]
```

You can see the function definition using the `.toString()` method:

```
double.toString(); // 'x => x * 2'
```

If you want to apply a function to some arguments, you must invoke it with a function call. A function call applies a function to its arguments and evaluates to a return value.

You can invoke a function using `<functionName>(argument1, argument2, ...rest)`. For example, to invoke our double function, just add the parentheses and pass in a value to double:

```
double(2); // 4
```

Unlike some functional languages, those parentheses are meaningful. Without them, the function won't be called:

```
double 4; // SyntaxError: Unexpected number
```

Signatures

Functions have signatures, which consist of:

1. An *optional* function name.
2. A list of parameter types, in parentheses. The parameters may optionally be named.
3. The type of the return value.

Type signatures don't need to be specified in JavaScript. The JavaScript engine will figure out the types at runtime. If you provide enough clues, the signature can also be inferred by developer tools such as IDEs

(Integrated Development Environment) and Tern.js using data flow analysis.

JavaScript lacks its own function signature notation, so there are a few competing standards: JSDoc has been very popular historically, but it's awkwardly verbose, and nobody bothers to keep the doc comments up-to-date with the code, so many JS developers have stopped using it.

TypeScript and Flow are currently the big contenders. I'm not sure how to express everything I need in either of those, so I use Rtype, for documentation purposes only. Some people fall back on Haskell's curry-only Hindley–Milner types. I'd love to see a good notation system standardized for JavaScript, if only for documentation purposes, but I don't think any of the current solutions are up to the task, at present. For now, squint and do your best to keep up with the weird type signatures which probably look slightly different from whatever you're using.

```
functionName(param1: Type, param2: Type) => Type
```

The signature for double is:

```
double(x: n) => Number
```

In spite of the fact that JavaScript doesn't require signatures to be annotated, knowing what signatures *are* and what they *mean* will still be important in order to communicate efficiently about how functions are used, and how functions are composed. Most reusable function composition utilities require you to pass functions which share the same type signature.

Default Parameter Values

JavaScript supports default parameter values. The following function works like an identity function (a function which returns the same

value you pass in), unless you call it with `undefined`, or simply pass no argument at all -- then it returns zero, instead:

```
const orZero = (n = 0) => n;
```

To set a default, simply assign it to the parameter with the `=` operator in the function signature, as in `n = 0`, above. When you assign default values in this way, type inference tools such as Tern.js, Flow, or TypeScript can infer the type signature of your function automatically, even if you don't explicitly declare type annotations.

The result is that, with the right plugins installed in your editor or IDE, you'll be able to see function signatures displayed inline as you're typing function calls. You'll also be able to understand how to use a function at a glance based on its call signature. Using default assignments wherever it makes sense can help you write more self-documenting code.

> *Note: Parameters with defaults don't count toward the function's* `.length` *property, which will throw off utilities such as autocurry which depend on the* `.length` *value. Some curry utilities (such as* `lodash/curry`*) allow you to pass a custom arity to work around this limitation if you bump into it.*

Named Arguments

JavaScript functions can take object literals as arguments and use destructuring assignment in the parameter signature in order to achieve the equivalent of named arguments. Notice, you can also assign default values to parameters using the default parameter feature:

```
const createUser = ({
  name = 'Anonymous',
  avatarThumbnail = '/avatars/anonymous.png'
}) => ({
  name,
  avatarThumbnail
});
```

```
const george = createUser({
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
});
```

```
george;
/*
{
  name: 'George',
  avatarThumbnail: 'avatars/shades-emoji.png'
}
*/
```

Rest and Spread

A common feature of functions in JavaScript is the ability to gather together a group of remaining arguments in the functions signature using the rest operator: `...`

For example, the following function simply discards the first argument and returns the rest as an array:

```
const aTail = (head, ...tail) => tail;
aTail(1, 2, 3); // [2, 3]
```

Rest gathers individual elements together into an array. Spread does the opposite: it spreads the elements from an array to individual elements. Consider this:

```
const shiftToLast = (head, ...tail) => [...tail, head];
shiftToLast(1, 2, 3); // [2, 3, 1]
```

Arrays in JavaScript have an iterator that gets invoked when the spread operator is used. For each item in the array, the iterator delivers a value. In the expression, `[...tail, head]`, the iterator copies each element in order from the `tail` array into the new array created by the surrounding literal notation. Since the head is already an individual element, we just plop it onto the end of the array and we're done.

Currying

A curried function is a function that takes multiple parameters one at a time: It takes a parameter, and returns a function that takes the next parameter, and so on until all parameters have been supplied, at which point, the application is completed and the final value is returned.

Curry and partial application can be enabled by returning another function:

```
const highpass = cutoff => n => n >= cutoff;
const gt4 = highpass(4); // highpass() returns a new
function
```

You don’t have to use arrow functions. JavaScript also has a `function` keyword. We're using arrow functions because the `function` keyword is a lot more typing. This is equivalent to the `highPass()` definition, above:

```
const highpass = function highpass(cutoff) {
  return function (n) {
    return n >= cutoff;
  };
};
```

The arrow in JavaScript roughly means “function”. There are some important differences in function behavior depending on which kind of function you use (`=>` lacks its own `this` , and can't be used as a constructor), but we'll get to those differences when we get there. For now, when you see `x => x` , think "a function that takes `x` and returns `x` ". So you can read `const highpass = cutoff => n => n >= cutoff;` as:

`“highpass` is a function which takes `cutoff` and returns a function which takes `n` and returns the result of `n >= cutoff` ".

Since `highpass()` returns a function, you can use it to create a more specialized function:

```
const gt4 = highpass(4);

gt4(6); // true
gt4(3); // false
```

Autocurry lets you curry functions automatically, for maximal flexibility. Say you have a function `add3()`:

```
const add3 = curry((a, b, c) => a + b + c);
```

With autocurry, you can use it in several different ways, and it will return the right thing depending on how many arguments you pass in:

```
add3(1, 2, 3); // 6
add3(1, 2)(3); // 6
add3(1)(2, 3); // 6
add3(1)(2)(3); // 6
```

Sorry Haskell fans, JavaScript lacks a built-in autocurry mechanism, but you can import one from Lodash:

```
$ npm install --save lodash
```

Then, in your modules:

```
import curry from 'lodash/curry';
```

Or, you can use the following magic spell:

```
// Tiny, recursive autocurry
const curry = (
  f, arr = []
) => (...args) => (
  a => a.length === f.length ?
    f(...a) :
    curry(f, a)
)([...arr, ...args]);
```

Function Composition

Of course you can compose functions. Function composition is the process of passing the return value of one function as an argument to another function. In mathematical notation:

```
f . g
```

Which translates to this in JavaScript:

```
f(g(x))
```

It's evaluated from the inside out:

1. `x` is evaluated
2. `g()` is applied to `x`
3. `f()` is applied to the return value of `g(x)`

For example:

```
const inc = n => n + 1;
inc(double(2)); // 5
```

The value `2` is passed into `double()`, which produces `4`. `4` is passed into `inc()` which evaluates to `5`.

You can pass any expression as an argument to a function. The expression will be evaluated before the function is applied:

```
inc(double(2) * double(2)); // 17
```

Since `double(2)` evaluates to `4`, you can read that as `inc(4 * 4)` which evaluates to `inc(16)` which then evaluates to `17`.

Function composition is central to functional programming. We'll have a lot more on it later.

Arrays

Arrays have some built-in methods. A method is a function associated with an object: usually a property of the associated object:

```
const arr = [1, 2, 3];
arr.map(double); // [2, 4, 6]
```

In this case, `arr` is the object, `.map()` is a property of the object with a function for a value. When you invoke it, the function gets applied to the arguments, as well as a special parameter called `this`, which gets automatically set when the method is invoked. The `this` value is how `.map()` gets access to the contents of the array.

Note that we're passing the `double` function as a value into `map` rather than calling it. That's because `map` takes a function as an argument and applies it to each item in the array. It returns a new array containing the values returned by `double()`.

Note that the original `arr` value is unchanged:

```
arr; // [1, 2, 3]
```

Method Chaining

You can also chain method calls. Method chaining is the process of directly calling a method on the return value of a function, without needing to refer to the return value by name:

```
const arr = [1, 2, 3];
arr.map(double).map(double); // [4, 8, 12]
```

A **predicate** is a function that returns a boolean value (`true` or `false`). The `.filter()` method takes a predicate and returns a new list, selecting only the items that pass the predicate (return `true`) to be included in the new list:

```
[2, 4, 6].filter(gt4); // [4, 6]
```

Frequently, you'll want to select items from a list, and then map those items to a new list:

```
[2, 4, 6].filter(gt4).map(double); [8, 12]
```

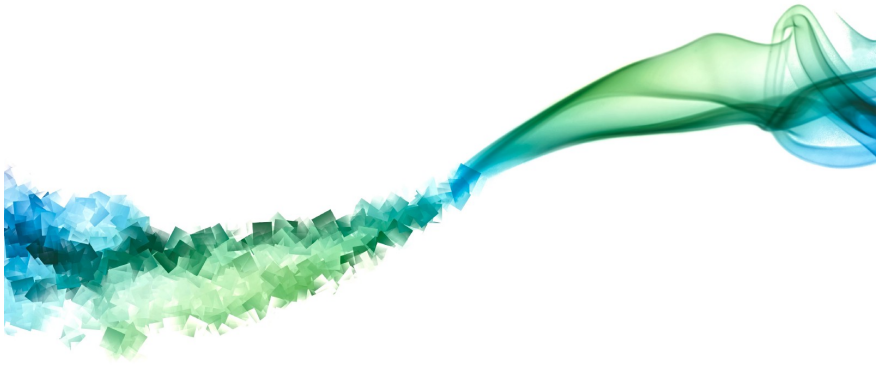
Note: Later in this text, you'll see a more efficient way to select and map at the same time using something called a *transducer*, but there are other things to explore first.

Conclusion

If your head is spinning right now, don't worry. We barely scratched the surface of a lot of things that deserve a lot more exploration and consideration. We'll circle back and explore some of these topics in much more depth, soon.

Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
May 17, 2017 · 11 min read

Composing Software: An Introduction



Smoke Art Cubes to Smoke — MattysFlicks — (CC BY 2.0)

> *Note: This is the introduction to the "Composing Software" series on learning functional programming and compositional software techniques in JavaScript ES6+ from the ground up. Stay tuned. There's a lot more of this to come!*
> Next >

> *Composition: "The act of combining parts or elements to form a whole." ~ Dictionary.com*

In my first high school programming class, I was told that software development is "the act of breaking a complex problem down into smaller problems, and composing simple solutions to form a complete solution to the complex problem."

One of my biggest regrets in life is that I failed to understand the significance of that lesson early on. I learned the essence of software design far too late in life.

I have interviewed hundreds of developers. What I've learned from those sessions is that I'm not alone. Very few working software developers have a good grasp on the essence of software development.

They aren't aware of the most important tools we have at our disposal, or how to put them to good use. 100% have struggled to answer one or both of the most important questions in the field of software development:

- What is function composition?
- What is object composition?

The problem is that you can't avoid composition just because you're not aware of it. You still do it—but you do it badly. You write code with more bugs, and make it harder for other developers to understand. This is a big problem. The effects are very costly. We spend more time maintaining software than we do creating it from scratch, and our bugs impact billions of people all over the world.

The entire world runs on software today. Every new car is a mini super-computer on wheels, and problems with software design cause real accidents and cost real human lives. In 2013, a jury found Toyota's software development team guilty of "reckless disregard" after an accident investigation revealed spaghetti code with 10,000 global variables.

Hackers and governments stockpile bugs in order to spy on people, steal credit cards, harness computing resources to launch Distributed Denial of Service (DDoS) attacks, crack passwords, and even manipulate elections.

We must do better.

You Compose Software Every Day

If you're a software developer, you compose functions and data structures every day, whether you know it or not. You can do it consciously (and better), or you can do it accidentally, with duct-tape and crazy glue.

The process of software development is breaking down large problems into smaller problems, building components that solve those smaller problems, then composing those components together to form a complete application.

Composing Functions

Function composition is the process of applying a function to the output of another function. In algebra, given two functions, `f` and `g`, `(f ∘ g)(x) = f(g(x))`. The circle is the composition operator. It's commonly pronounced "composed with" or "after". You can say that out-loud as "`f` *composed with* `g` equals `f` of `g` of `x`", or "`f` *after* `g` equals `f` of `g` of `x`". We say `f` *after* `g` because `g` is evaluated first, then its output is passed as an argument to `f`.

Every time you write code like this, you're composing functions:

```
const g = n => n + 1;
const f = n => n * 2;
```

```
const doStuff = x => {
  const afterG = g(x);
  const afterF = f(afterG);
  return afterF;
};
```

```
doStuff(20); // 42
```

Every time you write a promise chain, you're composing functions:

```
const g = n => n + 1;
const f = n => n * 2;
```

```
const wait = time => new Promise(
  (resolve, reject) => setTimeout(
    resolve,
    time
  )
);
```

```
wait(300)
  .then(() => 20)
  .then(g)
  .then(f)
  .then(value => console.log(value)) // 42
;
```

Likewise, every time you chain array method calls, lodash methods, observables (RxJS, etc…) you're composing functions. If you're chaining, you're composing. If you're passing return values into other functions, you're composing. If you call two methods in a sequence, you're composing using `this` as input data.

> *If you're chaining, you're composing.*

When you compose functions intentionally, you'll do it better.

Composing functions intentionally, we can improve our `doStuff()` function to a simple one-liner:

```
const g = n => n + 1;
const f = n => n * 2;

const doStuffBetter = x => f(g(x));

doStuffBetter(20); // 42
```

A common objection to this form is that it's harder to debug. For example, how would we write this using function composition?

```
const doStuff = x => {
  const afterG = g(x);
  console.log(`after g: ${ afterG }`);
  const afterF = f(afterG);
  console.log(`after f: ${ afterF }`);
  return afterF;
};

doStuff(20); // =>
/*
"after g: 21"
"after f: 42"
*/
```

First, let's abstract that "after f", "after g" logging into a little utility called `trace()`:

```
const trace = label => value => {
  console.log(`${ label }: ${ value }`);
  return value;
};
```

Now we can use it like this:

```
const doStuff = x => {
  const afterG = g(x);
  trace('after g')(afterG);
  const afterF = f(afterG);
  trace('after f')(afterF);
  return afterF;
};
```

```
doStuff(20); // =>
/*
"after g: 21"
"after f: 42"
*/
```

Popular functional programming libraries like Lodash and Ramda include utilities to make function composition easier. You can rewrite the above function like this:

```
import pipe from 'lodash/fp/flow';
```

```
const doStuffBetter = pipe(
  g,
  trace('after g'),
  f,
  trace('after f')
);
```

```
doStuffBetter(20); // =>
/*
"after g: 21"
"after f: 42"
*/
```

If you want to try this code without importing something, you can define pipe like this:

```
// pipe(...fns: [...Function]) => x => y
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);
```

Don’t worry if you’re not following how that works, yet. Later on we’ll explore function composition in a lot more detail. In fact, it’s so essential, you’ll see it defined and demonstrated many times throughout this text. The point is to help you become so familiar with it that its definition and usage becomes automatic. Be one with the composition.

`pipe()` creates a pipeline of functions, passing the output of one function to the input of another. When you use `pipe()` (and its twin, `compose()`) You don't need intermediary variables. Writing functions without mention of the arguments is called **point-free style.** To do it, you'll call a function that returns the new function, rather than declaring the function explicitly. That means you won't need the `function` keyword or the arrow syntax (`=>`).

Point-free style can be taken too far, but a little bit here and there is great because those intermediary variables add unnecessary complexity to your functions.

There are several benefits to reduced complexity:

Working Memory

The average human brain has only a few shared resources for discrete quanta in working memory, and each variable potentially consumes one of those quanta. As you add more variables, our ability to accurately recall the meaning of each variable is diminished. Working memory models typically involve 4–7 discrete quanta. Above those numbers, error rates dramatically increase.

Using the pipe form, we eliminated 3 variables—freeing up almost half of our available working memory for other things. That reduces our cognitive load significantly. Software developers tend to be better at chunking data into working memory than the average person, but not so much more as to weaken the importance of conservation.

Signal to Noise Ratio

Concise code also improves the signal-to-noise ratio of your code. It's like listening to a radio—when the radio is not tuned properly to the station, you get a lot of interfering noise, and it's harder to hear the music. When you tune it to the correct station, the noise goes away, and you get a stronger musical signal.

Code is the same way. More concise code expression leads to enhanced comprehension. Some code gives us useful information, and some code just takes up space. If you can reduce the amount of code you use without reducing the meaning that gets transmitted, you'll make the code easier to parse and understand for other people who need to read it.

Surface Area for Bugs

Take a look at the before and after functions. It looks like the function went on a diet and lost a ton of weight. That's important because extra code means extra surface area for bugs to hide in, which means more bugs will hide in it.

> *Less code = less surface area for bugs = fewer bugs.*

Composing Objects

> *"Favor object composition over class inheritance" the Gang of Four, "Design Patterns: Elements of Reusable Object Oriented Software"*

> *"In computer science, a composite data type or compound data type is any data type which can be constructed in a program using the programming language's primitive data types and other composite types. […] The act of constructing a composite type is known as composition." ~ Wikipedia*

These are primitives:

```
const firstName = 'Claude';
const lastName = 'Debussy';
```

And this is a composite:

```
const fullName = {
  firstName,
  lastName
};
```

Likewise, all Arrays, Sets, Maps, WeakMaps, TypedArrays, etc… are composite datatypes. Any time you build any non-primitive data structure, you're performing some kind of object composition.

Note that the Gang of Four defines a pattern called the **composite pattern** which is a specific type of recursive object composition which allows you to treat individual components and aggregated composites identically. Some developers get confused, thinking that the composite pattern is *the only form of object composition.* Don't get confused. There are many different kinds of object composition.

The Gang of Four continues, "you'll see object composition applied again and again in design patterns", and then they catalog three kinds of object compositional relationships, including **delegation** (as used in the state, strategy, and visitor patterns), **acquaintance** (when an object knows about another object by reference, usually passed as a parameter: a uses-a relationship, e.g., a network request handler might be passed a reference to a logger to log the request—the request handler *uses* a logger), and **aggregation** (when child objects form part of a parent object: a has-a relationship, e.g., DOM children are component elements in a DOM node—A DOM node *has* children).

Class inheritance can be used to construct composite objects, but it's a restrictive and brittle way to do it. When the Gang of Four says "favor object composition over class inheritance", they're advising you to use flexible approaches to composite object building, rather than the rigid, tightly-coupled approach of class inheritance.

We'll use a more general definition of object composition from "Categorical Methods in Computer Science: With Aspects from Topology" (1989):

> *"Composite objects are formed by putting objects together such that each of the latter is 'part of' the former."*

Another good reference is “Reliable Software Through Composite Design”, Glenford J Myers, 1975. Both books are long out of print, but you can still find sellers on Amazon or eBay if you’d like to explore the subject of object composition in more technical depth.

Class inheritance is just one kind of composite object construction. All classes produce composite objects, but not all composite objects are produced by classes or class inheritance. “Favor object composition over class inheritance” means that you should form composite objects from small component parts, rather than inheriting all properties from an ancestor in a class hierarchy. The latter causes a large variety of well-known problems in object oriented design:

- **The tight coupling problem:** Because child classes are dependent on the implementation of the parent class, class inheritance is the tightest coupling available in object oriented design.
- **The fragile base class problem:** Due to tight coupling, changes to the base class can potentially break a large number of descendant classes—potentially in code managed by third parties. The author could break code they’re not aware of.
- **The inflexible hierarchy problem:** With single ancestor taxonomies, given enough time and evolution, all class taxonomies are eventually wrong for new use-cases.
- **The duplication by necessity problem:** Due to inflexible hierarchies, new use cases are often implemented by duplication, rather than extension, leading to similar classes which are unexpectedly divergent. Once duplication sets in, it’s not obvious which class new classes should descend from, or why.
- **The gorilla/banana problem:** “…the problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.” ~ Joe Armstrong, “Coders at Work”

The most common form of object composition in JavaScript is known as **object concatenation** (aka mixin composition). It works like ice-cream. You start with an object (like vanilla ice-cream), and then mix in the features you want. Add some nuts, caramel, chocolate swirl, and you wind up with nutty caramel chocolate swirl ice cream.

Building composites with class inheritance:

```
class Foo {
  constructor () {
    this.a = 'a'
  }
}

class Bar extends Foo {
  constructor (options) {
    super(options);
    this.b = 'b'
  }
}

const myBar = new Bar(); // {a: 'a', b: 'b'}
```

Building composites with mixin composition:

```
const a = {
  a: 'a'
};

const b = {
  b: 'b'
};

const c = {...a, ...b}; // {a: 'a', b: 'b'}
```

We'll explore other styles of object composition in more depth later. For now, your understanding should be:

1. There's more than one way to do it.
2. Some ways are better than others.
3. You want to select the simplest, most flexible solution for the task at hand.

Conclusion

This isn't about functional programming (FP) vs object-oriented programming (OOP), or one language vs another. Components can take the form of functions, data structures, classes, etc… Different programming languages tend to afford different atomic elements for components. Java affords classes, Haskell affords functions, etc… But no matter what language and what paradigm you favor, you can't get away from composing functions and data structures. In the end, that's what it all boils down to.

We'll talk a lot about functional programming, because functions are the simplest things to compose in JavaScript, and the functional programming community has invested a lot of time and effort formalizing function composition techniques.

What we won't do is say that functional programming is better than object-oriented programming, or that you must choose one over the other. OOP vs FP is a false dichotomy. Every real Javascript application I've seen in recent years mixes FP and OOP extensively.

We'll use object composition to produce datatypes for functional programming, and functional programming to produce objects for OOP.

No matter how you write software, you should compose it well.

> *The essence of software development is composition.*

A software developer who doesn't understand composition is like a home builder who doesn't know about bolts or nails. Building software without awareness of composition is like a home builder putting walls together with duct tape and crazy glue.

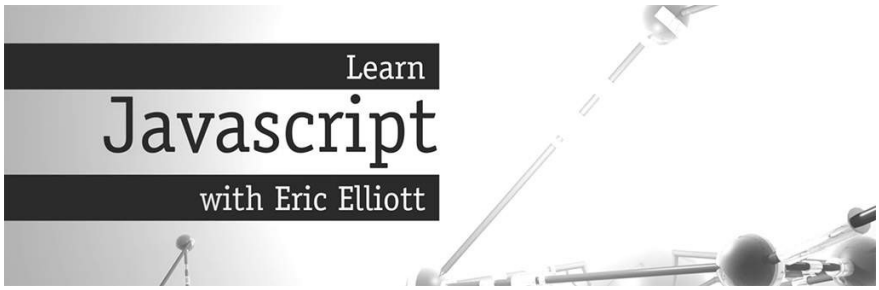
It's time to simplify, and the best way to simplify is to get to the essence. The trouble is, almost nobody in the industry has a good handle on the essentials. We as an industry have failed you, the software developer. It's our responsibility as an industry to train developers better. We must improve. We need to take responsibility. Everything runs on software today, from the economy to medical equipment. There is literally no corner of human life on this planet that is not impacted by the quality of our software. We need to know what we're doing.

It's time to learn how to compose software.

Continued in "The Rise and Fall and Rise of Functional Programming"

Learn More at EricElliottJS.com

Video lessons on function & object composition are available for members of EricElliottJS.com. If you're not a member, sign up today.



. . .

Eric Elliott *is the author of "Programming JavaScript Applications" (O'Reilly), and "Learn JavaScript with Eric Elliott". He has contributed to software experiences for* ***Adobe Systems****,* ***Zumba Fitness****,* ***The Wall Street Journal****,* ***ESPN****,* ***BBC****, and top recording artists including* ***Usher****,* ***Frank Ocean****,* ***Metallica****, and many more.*

He works remote from anywhere with the most beautiful woman in the world.

Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
Jan 7, 2016 · 5 min read

Master the JavaScript Interview: What is a Closure?

> *"Master the JavaScript Interview" is a series of posts designed to prepare candidates for common questions they are likely to encounter when applying for a mid to senior-level JavaScript position. These are questions I frequently use in real interviews.*

Go JS!
@JS_Cheerleader

#JavaScript #JSTweetInterview
What is a closure?

David K. 🎹
@DavidKPiano

Replying to @JS_Cheerleader
@JS_Cheerleader a stateful function.

I'm launching the series with a question that is often my first and last question in my JavaScript interviews. Frankly, you can't get very far with JavaScript without learning about closures.

You can muck around a bit, but will you really understand how to build a serious JavaScript application? Will you really understand what is going on, or how the application works? I have my doubts. Not knowing the answer to this question is a **serious red flag**.

Not only should you know the mechanics of what a closure is, you should know why it matters, and be able to easily answer several possible use-cases for closures.

Closures are **frequently used in JavaScript** for object data privacy, in event handlers and callback functions, and in partial applications, currying, and other functional programming patterns.

I don't care if a candidate knows the word "closure" or the technical definition. I want to find out if they understand the basic mechanics. If they don't, it's usually a clear indicator that the developer does not have a lot of experience building actual JavaScript applications.

> *If you can't answer this question, you're a junior developer. I don't care how long you've been coding.*

That may sound mean, but it's not. What I mean is that most competent interviewers will ask you what a closure is, and most of the time,

getting the answer wrong will cost you the job. Or if you're lucky enough to get an offer anyway, it will cost you potentially tens of thousands of dollars per year in pay because you'll be hired as a junior instead of a senior level developer, regardless of how long you've been coding.

Be prepared for a quick follow-up: "Can you name two common uses for closures?"

What is a Closure?

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

To use a closure, simply define a function inside another function and expose it. To expose a function, return it or pass it to another function.

The inner function will have access to the variables in the outer function scope, even after the outer function has returned.

Using Closures (Examples)

Among other things, closures are commonly used to give objects data privacy. Data privacy is an essential property that helps us program to an interface, not an implementation. This is an important concept that helps us build more robust software because implementation details are more likely to change in breaking ways than interface contracts.

> *"Program to an interface, not an implementation."*
> *Design Patterns: Elements of Reusable Object Oriented Software*

In JavaScript, closures are the primary mechanism used to enable data privacy. When you use closures for data privacy, the enclosed variables are only in scope within the containing (outer) function. You can't get at the data from an outside scope except through the object's **privileged methods**. In JavaScript, any exposed method defined within the closure scope is privileged. For example:

```
const getSecret = (secret) => {
  return {
    get: () => secret
  };
};

test('Closure for object privacy.', assert => {
  const msg = '.get() should have access to the closure.';
  const expected = 1;
  const obj = getSecret(1);

  const actual = obj.get();

  try {
    assert.ok(secret, 'This throws an error.');
```

Play with this in JSBin. *(Don't see any output? Copy and paste this HTML into the HTML pane.)*

In the example above, the `.get()` method is defined inside the scope of `getSecret()`, which gives it access to any variables from `getSecret()`, and makes it a privileged method. In this case, the parameter, `secret`.

Objects are not the only way to produce data privacy. Closures can also be used to create **stateful functions** whose return values may be influenced by their internal state, e.g.:

```
const secret = msg => () => msg;
```

```
1  // Secret - creates closures with secret messages.
2  // https://gist.github.com/ericelliott/f6a87bc41de31562d0f9
3  // https://jsbin.com/hitusu/edit?html,js,output
4
5  // secret(msg: String) => getSecret() => msg: String
6  const secret = (msg) => () => msg;
7
8  test('secret', assert => {
9    const msg = 'secret() should return a function that retur
10
11   const theSecret = 'Closures are easy.';
12   const mySecret = secret(theSecret);
13
```

Available on JSBin. *(Don't see any output? Copy and paste this HTML into the HTML pane.)*

In functional programming, closures are frequently used for partial application & currying. This requires some definitions:

Application: The process of *applying* a function to its *arguments* in order to produce a return value.

Partial Application: The process of applying a function to *some of its arguments*. The partially applied function gets returned for later use. In other words, a function that **takes a function with multiple parameters** and **returns a function with fewer parameters.** Partial application *fixes* (partially applies the function to) one or more arguments inside the returned function, and the returned function takes the remaining parameters as arguments in order to complete the function application.

Partial application takes advantage of closure scope in order to **fix** parameters. You can write a generic function that will partially apply arguments to the target function. It will have the following signature:

```
partialApply(targetFunction: Function, ...fixedArgs: Any[])
=>
  functionWithFewerParams(...remainingArgs: Any[])
```

If you need help reading the signature above, check out Rtype: Reading Function Signatures.

It will take a function that takes any number of arguments, followed by arguments we want to partially apply to the function, and returns a function that will take the remaining arguments.

An example will help. Say you have a function that adds two numbers:

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

Now you want a function that adds 10 to any number. We'll call it *`add10()`*. The result of *`add10(5)`* should be *`15`*. Our *`partialApply()`* function can make that happen:

```
const add10 = partialApply(add, 10);
add10(5);
```

In this example, the argument, *`10`* becomes a **fixed parameter** remembered inside the *`add10()`* closure scope.

Let's look at a possible *`partialApply()`* implementation:

```
// Generic Partial Application Function
// https://jsbin.com/biyupu/edit?html,js,output
// https://gist.github.com/ericelliott/f0a8fd662111ea2f569e

// partialApply(targetFunction: Function, ...fixedArgs: Any
//   functionWithFewerParams(...remainingArgs: Any[])
const partialApply = (fn, ...fixedArgs) => {
  return function (...remainingArgs) {
    return fn.apply(this, fixedArgs.concat(remainingArgs));
  };
};


test('add10', assert => {
  const msg = 'partialApply() should partially apply functi

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

Available on JSBin. *(Don't see any output? Copy and paste this HTML into the HTML pane.)*

As you can see, it simply returns a function which retains access to the `*fixedArgs*` arguments that were passed into the `*partialApply()*` function.

Your Turn

This post has a companion video post and practice assignments for members of EricElliottJS.com. If you're already a member, sign in and practice now.

If you're not a member, sign up today.

Explore the Series

- What is a Closure?
- What is the Difference Between Class and Prototypal Inheritance?
- What is a Pure Function?
- What is Function Composition?

Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
Jan 18, 2016 · 9 min read

Master the JavaScript Interview: What's the Difference Between Class & Prototypal Inheritance?

Electric Guitar — Feliciano Guimarães (CC BY 2.0)

> *"Master the JavaScript Interview" is a series of posts designed to prepare candidates for common questions they are likely to encounter when applying for a mid to senior-level JavaScript position. These are questions I*

> *frequently use in real interviews. Want to start from the beginning? See "What is a Closure?"*

> *Note: This article uses ES6 examples. If you haven't learned ES6 yet, see "How to Learn ES6".*

Objects are frequently used in JavaScript, and understanding how to work with them effectively will be a huge win for your productivity. In fact, poor OO design can potentially lead to project failure, and in the worst cases, company failures.

Unlike most other languages, JavaScript's object system is based on **prototypes, not classes**. Unfortunately, most JavaScript developers don't understand JavaScript's object system, or how to put it to best use. Others do understand it, but want it to behave more like class based systems. The result is that JavaScript's object system has a confusing split personality, which means that JavaScript developers need to know a bit about **both prototypes and classes**.

What's the Difference Between Class & Prototypal Inheritance?

This can be a tricky question, and you'll probably need to defend your answer with follow-up Q&A, so pay special attention to learning the differences, and how to apply the knowledge to write better code.

Class Inheritance: ***A class is like a blueprint—a description of the object to be created.*** Classes inherit from classes and **create subclass relationships**: hierarchical class taxonomies.

Instances are typically instantiated via constructor functions with the *`new`* keyword. Class inheritance may or may not use the *`class`* keyword from ES6. Classes as you may know them from languages like Java don't technically exist in JavaScript. Constructor functions are used, instead. The ES6 *`class`* keyword desugars to a constructor function:

```
class Foo {}
typeof Foo // 'function'
```

In JavaScript, class inheritance is implemented on top of prototypal inheritance, but *that does not mean that it does the same thing:*

JavaScript's class inheritance uses the prototype chain to wire the child `*Constructor.prototype*` to the parent `*Constructor.prototype*` for delegation. Usually, the `*super()*` constructor is also called. Those steps form **single-ancestor parent/child hierarchies** and ***create the tightest coupling available in OO design.***

> *"Classes inherit from classes and create subclass relationships: hierarchical class taxonomies."*

Prototypal Inheritance: ***A prototype is a working object instance.*** Objects inherit directly from other objects.

Instances may be composed from many different source objects, allowing for easy selective inheritance and a flat [[Prototype]] delegation hierarchy. In other words, **class taxonomies are not an automatic side-effect of prototypal OO**: *a critical distinction*.

Instances are typically instantiated via factory functions, object literals, or `*Object.create()*`.

> *"A prototype is a working object instance. Objects inherit directly from other objects."*

Why Does this Matter?

Inheritance is fundamentally a code reuse mechanism: A way for different kinds of objects to share code. The way that you share code matters because if you get it wrong, **it can create a lot of problems,** specifically:

Class inheritance creates parent/child object taxonomies as a side-effect.

Those taxonomies are virtually impossible to get right for all new use cases, and widespread use of a base class leads to **the fragile base class problem,** which makes them difficult to fix when you get them wrong.

In fact, class inheritance causes many well known problems in OO design:

- **The tight coupling problem** (class inheritance is the tightest coupling available in oo design), which leads to the next one…
- **The fragile base class problem**
- **Inflexible hierarchy problem** (eventually, all evolving hierarchies are wrong for new uses)
- **The duplication by necessity problem** (due to inflexible hierarchies, new use cases are often shoe-horned in by duplicating, rather than adapting existing code)
- **The Gorilla/banana problem** (What you wanted was a banana, but what you got was a gorilla holding the banana, and the entire jungle)

I discuss some of the issues in more depth in my talk, "Classical Inheritance is Obsolete: How to Think in Prototypal OO":

The solution to all of these problems is to favor object composition over class inheritance.

> *"Favor object composition over class inheritance."*
> *~ The Gang of Four, "Design Patterns: Elements of Reusable Object Oriented Software"*

Summed up nicely here:

Is All Inheritance Bad?

When people say “favor composition over inheritance” that is short for “favor composition over **class** inheritance” (the original quote from “Design Patterns” by the Gang of Four). This is common knowledge in OO design because **class inheritance has many flaws** and causes many problems. Often people leave off the word **class** when they talk about class inheritance, which makes it sound like *all inheritance* is bad —but it’s not.

There are actually several different kinds of inheritance, and most of them are great.

Three Different Kinds of Prototypal Inheritance

Before we dive into the other kinds of inheritance, let’s take a closer look at what I mean by **class inheritance**:

```
// Class Inheritance Example
// NOT RECOMMENDED. Use object composition, instead.

// https://gist.github.com/ericelliott/b668ce0ad1ab540df915
// http://codepen.io/ericelliott/pen/pgdPOb?editors=001

class GuitarAmp {
  constructor ({ cabinet = 'spruce', distortion = '1', volu
    Object.assign(this, {
      cabinet, distortion, volume
    });
  }
}

class BassAmp extends GuitarAmp {
  constructor (options = {}) {
    super(options);
    this.lowCut = options.lowCut;
  }
}

class ChannelStrip extends BassAmp {
  constructor (options = {}) {
    super(options);
    this.inputLevel = options.inputLevel;
  }
}

test('Class Inheritance', nest => {
  nest.test('BassAmp', assert => {
    const msg = `instance should inherit props
    from GuitarAmp and BassAmp`;

```

You can experiment with this example on Codepen.

`*BassAmp*` inherits from `*GuitarAmp*`, and `*ChannelStrip*` inherits from `*BassAmp*` & `*GuitarAmp*`. This is an example of how OO design goes wrong. A channel strip isn't actually a type of guitar amp, and doesn't actually need a cabinet at all. A better option would be to create

a new base class that both the amps and the channel strip inherits from, but even that has limitations.

Eventually, the new shared base class strategy breaks down, too.

There's a better way. You can inherit just the stuff you really need using object composition:

```
// Composition Example

// http://codepen.io/ericelliott/pen/XXzadQ?editors=001
// https://gist.github.com/ericelliott/fed0fd7a0d3388b06402

const distortion = { distortion: 1 };
const volume = { volume: 1 };
const cabinet = { cabinet: 'maple' };
const lowCut = { lowCut: 1 };
const inputLevel = { inputLevel: 1 };

const GuitarAmp = (options) => {
  return Object.assign({}, distortion, volume, cabinet, opt
};

const BassAmp = (options) => {
  return Object.assign({}, lowCut, volume, cabinet, options
};

const ChannelStrip = (options) => {
  return Object.assign({}, inputLevel, lowCut, volume, opti
};


test('GuitarAmp', assert => {
  const msg = 'should have distortion, volume, and cabinet'
  const level = 2;
  const cabinet = 'vintage';

  const actual = GuitarAmp({
    distortion: level,
    volume: level,
    cabinet
  });
  const expected = {
    distortion: level,
    volume: level,
    cabinet
  };

  assert.deepEqual(actual, expected, msg);
```

```
42    assert.end();
43  });
44
45  test('BassAmp', assert => {
46    const msg = 'should have volume, lowCut, and cabinet';
47    const level = 2;
48    const cabinet = 'vintage';
49
50    const actual = BassAmp({
51      lowCut: level,
52      volume: level,
```

Experiment with this on CodePen.

If you look carefully, you might see that we're being much more specific about which objects get which properties because with composition, ***we can***. It wasn't really an option with class inheritance. When you inherit from a class, you get everything, *even if you don't want it*.

At this point, you may be thinking to yourself, "that's nice, but where are the prototypes?"

To understand that, you have to understand that there are three different kinds of prototypal OO.

Concatenative inheritance: The process of inheriting features directly from one object to another by copying the source objects properties. In JavaScript, source prototypes are commonly referred to as **mixins.** Since ES6, this feature has a convenience utility in JavaScript called *`Object.assign()`*. Prior to ES6, this was commonly done with Underscore/Lodash's *`.extend()`* jQuery's *`$.extend()`*, and so on… The composition example above uses concatenative inheritance.

Prototype delegation: In JavaScript, an object may have a link to a prototype for **delegation**. If a property is not found on the object, the lookup is **delegated** to the **delegate prototype,** which may have a link to its own delegate prototype, and so on up the chain until you arrive at *`Object.prototype`*, which is the root delegate. This is the prototype that gets hooked up when you attach to a *`Constructor.prototype`* and instantiate with *`new`*. You can also use *`Object.create()`* for this purpose, and even mix this technique with concatenation in order to

flatten multiple prototypes to a single delegate, or extend the object instance after creation.

Functional inheritance: In JavaScript, any function can create an object. When that function is not a constructor (or `*class*`), it's called a **factory function**. Functional inheritance works by producing an object from a factory, and extending the produced object by assigning properties to it directly (using concatenative inheritance). Douglas Crockford coined the term, but functional inheritance has been in common use in JavaScript for a long time.

As you're probably starting to realize, **concatenative inheritance is the secret sauce that enables object composition in JavaScript**, which makes both prototype delegation and functional inheritance a lot more interesting.

When most people think of prototypal OO in JavaScript, *they think of prototype delegation.* By now you should see that they're missing out on a lot. Delegate prototypes aren't the great alternative to class inheritance—**object composition is**.

Why Composition is Immune to the Fragile Base Class Problem

To understand the fragile base class problem and why it doesn't apply to composition, first you have to understand how it happens:

1. `*A*` is the base class
2. `*B*` inherits from `*A*`
3. `*C*` inherits from `*B*`
4. `*D*` inherits from `*B*`

`*C*` calls `*super*`, which runs code in `*B*`. `*B*` calls `*super*` which runs code in `*A*`.

`*A*` and `*B*` contain unrelated features needed by both `*C*` & `*D*`. `*D*` is a new use case, and needs *slightly different* behavior in `*A*`'s init code than `*C*` needs. So the newbie dev goes and tweaks `*A*`'s init code. `***C***` **breaks because it depends on the existing behavior**, and `*D*` starts working.

What we have here are features spread out between *`A`* and *`B`* that *`C`* and *`D`* need to use in various ways. *`C`* and *`D`* don't use every feature of *`A`* and *`B`*… they just want to inherit some stuff that's already defined in *`A`* and *`B`*. But by inheriting and calling *`super`*, **you don't get to be selective about what you inherit**. You inherit everything:

> *"…the problem with object-oriented languages is they've got all this implicit environment that they carry around with them.* ***You wanted a banana but what you got was a gorilla holding the banana*** *and the entire jungle." ~ Joe Armstrong—"Coders at Work"*

With Composition
Imagine you have features instead of classes:

```
feat1, feat2, feat3, feat4
```

`C` needs *`feat1`* and *`feat3`*, *`D`* needs *`feat1`*, *`feat2`*, *`feat4`*:

```
const C = compose(feat1, feat3);
const D = compose(feat1, feat2, feat4);
```

Now, imagine you discover that *`D`* needs **slightly different** behavior from *`feat1`*. It doesn't actually need to change *`feat1`*, instead, **you can make a customized version of *`feat1`*** and use that, instead. You can still inherit the existing behaviors from *`feat2`* and *`feat4`* with no changes:

```
const D = compose(custom1, feat2, feat4);
```

And ***`C`*** **remains unaffected**.

The reason this is not possible with class inheritance is because **when you use class inheritance, you buy into the whole existing class**

taxonomy.

If you want to adapt a little for a new use-case, you either end up duplicating parts of the existing taxonomy (the duplication by necessity problem), or you refactor everything that depends on the existing taxonomy to adapt the taxonomy to the new use case due to **the fragile base class problem**.

Composition is immune to both.

You Think You Know Prototypes, but...

If you were taught to build classes or constructor functions and inherit from those, what you were taught was **not prototypal inheritance**. You were taught how to **mimic class inheritance using prototypes**. See "Common Misconceptions About Inheritance in JavaScript".

In JavaScript, class inheritance piggybacks on top of the very rich, flexible prototypal inheritance features built into the language a long time ago, but when you use class inheritance—even the ES6+ `*class*` inheritance built on top of prototypes, you're not using the full power & flexibility of prototypal OO. In fact, you're painting yourself into corners and **opting into all of the class inheritance problems**.

> *Using class inheritance in JavaScript is like driving your new Tesla Model S to the dealer and trading it in for a rusted out 1983 Ford Pinto.*

Stamps: Composable Factory Functions

Most of the time, composition is achieved using factory functions: functions which exist to create object instances. What if there was a standard that makes factory functions composable? There is. It's called The Stamp Specification.

Explore the Series

- What is a Closure?
- What is the Difference Between Class and Prototypal Inheritance?
- What is a Pure Function?

Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
Jan 22, 2017 · 11 min read

Master the JavaScript Interview: What is a Promise?

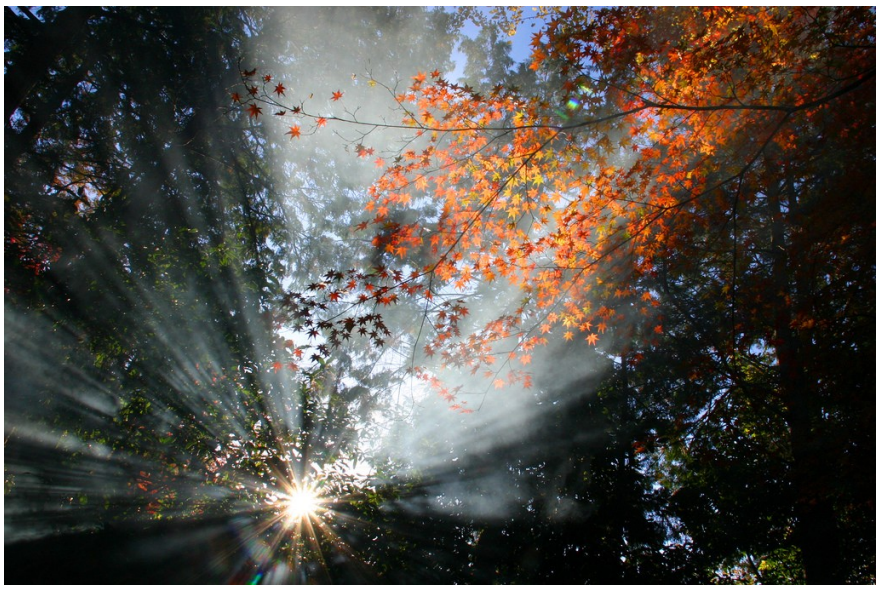


Photo by Kabun (CC BY NC SA 2.0)

> *"Master the JavaScript Interview" is a series of posts designed to prepare candidates for common questions they are likely to encounter when applying for a mid to senior-level JavaScript position. These are questions I frequently use in real interviews.*

What is a Promise?

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

Promises are eager, meaning that a promise will start doing whatever task you give it as soon as the promise constructor is invoked. If you need lazy, check out observables or tasks.

An Incomplete History of Promises

Early implementations of promises and futures (a similar / related idea) began to appear in languages such as MultiLisp and Concurrent Prolog as early as the 1980's. The use of the word "promise" was coined by Barbara Liskov and Liuba Shrira in 1988[1].

The first time I heard about promises in JavaScript, Node was brand new and the community was discussing the best way to handle asynchronous behavior. The community experimented with promises for a while, but eventually settled on the Node-standard error-first callbacks.

Around the same time, Dojo added promises via the Deferred API. Growing interest and activity eventually led to the newly formed Promises/A specification designed to make various promises more interoperable.

jQuery's async behaviors were refactored around promises. jQuery's promise support had remarkable similarities to Dojo's Deferred, and it quickly became the most commonly used promise implementation in JavaScript due to jQuery's immense popularity—for a time. However, it did not support the two channel (fulfilled/rejected) chaining behavior & exception management that people were counting on to build tools on top of promises.

In spite of those weaknesses, jQuery officially made JavaScript promises mainstream, and better stand-alone promise libraries like Q, When, and Bluebird became very popular. jQuery's implementation incompatibilities motivated some important clarifications in the promise spec, which was rewritten and rebranded as the Promises/A+ specification.

ES6 brought a Promises/A+ compliant `Promise` global, and some very important APIs were built on top of the new standard Promise support: notably the WHATWG Fetch spec and the Async Functions standard (a stage 3 draft at the time of this writing).

The promises described here are those which are compatible with the Promises/A+ specification, with a focus on the ECMAScript standard `Promise` implementation.

How Promises Work

A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states:

- **Fulfilled:** `onFulfilled()` will be called (e.g., `resolve()` was called)
- **Rejected:** `onRejected()` will be called (e.g., `reject()` was called)
- **Pending:** not yet fulfilled or rejected

A promise is **settled** if it's not pending (it has been resolved or rejected). Sometimes people use *resolved* and *settled* to mean the same thing: *not pending*.

Once settled, a promise can not be resettled. Calling `resolve()` or `reject()` again will have no effect. The immutability of a settled promise is an important feature.

Native JavaScript promises don't expose promise states. Instead, you're expected to treat the promise as a black box. Only the function responsible for creating the promise will have knowledge of the promise status, or access to resolve or reject.

Here is a function that returns a promise which will resolve after a specified time delay:

```
1  const wait = time => new Promise((resolve) => setTimeout(res
2
3  wait(3000).then(() => console.log('Hello!')); // 'Hello!'
```

wait — promise example on CodePen

Our `wait(3000)` call will wait 3000ms (3 seconds), and then log `'Hello!'`. All spec-compatible promises define a `.then()` method which you use to pass handlers which can take the resolved or rejected value.

The ES6 promise constructor takes a function. That function takes two parameters, `resolve()`, and `reject()`. In the example above, we're

only using `resolve()`, so I left `reject()` off the parameter list. Then we call `setTimeout()` to create the delay, and call `resolve()` when it's finished.

You can optionally `resolve()` or `reject()` with values, which will be passed to the callback functions attached with `.then()`.

When I `reject()` with a value, I always pass an `Error` object. Generally I want two possible resolution states: the normal happy path, or an exception—anything that stops the normal happy path from happening. Passing an `Error` object makes that explicit.

Important Promise Rules

A standard for promises was defined by the Promises/A+ specification community. There are many implementations which conform to the standard, including the JavaScript standard ECMAScript promises.

Promises following the spec must follow a specific set of rules:

- A promise or "thenable" is an object that supplies a standard-compliant `.then()` method.
- A pending promise may transition into a fulfilled or rejected state.
- A fulfilled or rejected promise is settled, and must not transition into any other state.
- Once a promise is settled, it must have a value (which may be `undefined`). That value must not change.

Change in this context refers to identity (`===`) comparison. An object may be used as the fulfilled value, and object properties may mutate.

Every promise must supply a `.then()` method with the following signature:

```
promise.then(
  onFulfilled?: Function,
  onRejected?: Function
) => Promise
```

The `.then()` method must comply with these rules:

- Both `onFulfilled()` and `onRejected()` are optional.
- If the arguments supplied are not functions, they must be ignored.
- `onFulfilled()` will be called after the promise is fulfilled, with the promise's value as the first argument.
- `onRejected()` will be called after the promise is rejected, with the reason for rejection as the first argument. The reason may be any valid JavaScript value, but because rejections are essentially synonymous with exceptions, I recommend using Error objects.
- Neither `onFulfilled()` nor `onRejected()` may be called more than once.
- `.then()` may be called many times on the same promise. In other words, a promise can be used to aggregate callbacks.
- `.then()` must return a new promise, `promise2`.
- If `onFulfilled()` or `onRejected()` return a value `x`, and `x` is a promise, `promise2` will lock in with (assume the same state and value as) `x`. Otherwise, `promise2` will be fulfilled with the value of `x`.
- If either `onFulfilled` or `onRejected` throws an exception `e`, `promise2` must be rejected with `e` as the reason.
- If `onFulfilled` is not a function and `promise1` is fulfilled, `promise2` must be fulfilled with the same value as `promise1`.
- If `onRejected` is not a function and `promise1` is rejected, `promise2` must be rejected with the same reason as `promise1`.

Promise Chaining

Because `.then()` always returns a new promise, it's possible to chain promises with precise control over how and where errors are handled. Promises allow you to mimic normal synchronous code's `try` / `catch` behavior.

Like synchronous code, chaining will result in a sequence that runs in serial. In other words, you can do:

```
fetch(url)
  .then(process)
  .then(save)
  .catch(handleErrors)
;
```

Assuming each of the functions, `fetch()`, `process()`, and `save()` return promises, `process()` will wait for `fetch()` to complete before starting, and `save()` will wait for `process()` to complete before starting. `handleErrors()` will only run if any of the previous promises reject.

Here's an example of a complex promise chain with multiple rejections:

```
const wait = time => new Promise(
  res => setTimeout(() => res(), time)
);

wait(200)
  // onFulfilled() can return a new promise, `x`
  .then(() => new Promise(res => res('foo')))
  // the next promise will assume the state of `x`
  .then(a => a)
  // Above we returned the unwrapped value of `x`
  // so `.then()` above returns a fulfilled promise
  // with that value:
  .then(b => console.log(b)) // 'foo'
  // Note that `null` is a valid promise value:
  .then(() => null)
  .then(c => console.log(c)) // null
  // The following error is not reported yet:
  .then(() => {throw new Error('foo');})
  // Instead, the returned promise is rejected
  // with the error as the reason:
  .then(
    // Nothing is logged here due to the error above:
    d => console.log(`d: ${ d }`),
    // Now we handle the error (rejection reason)
```

Promise chaining behavior example on CodePen

Error Handling

Note that promises have both a success and an error handler, and it's very common to see code that does this:

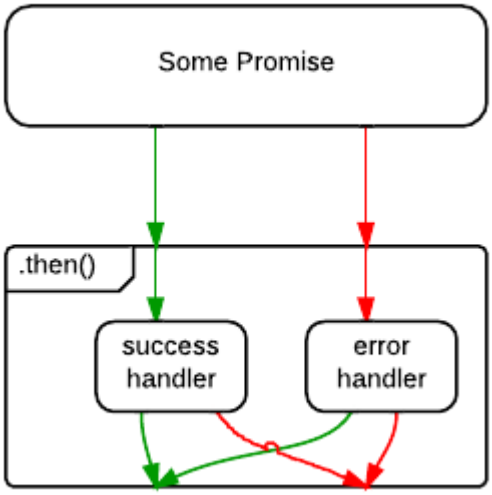
```
save().then(
  handleSuccess,
  handleError
);
```

But what happens if `handleSuccess()` throws an error? The promise returned from `.then()` will be rejected, but there's nothing there to catch the rejection—meaning that an error in your app gets swallowed. Oops!

For that reason, some people consider the code above to be an anti-pattern, and recommend the following, instead:

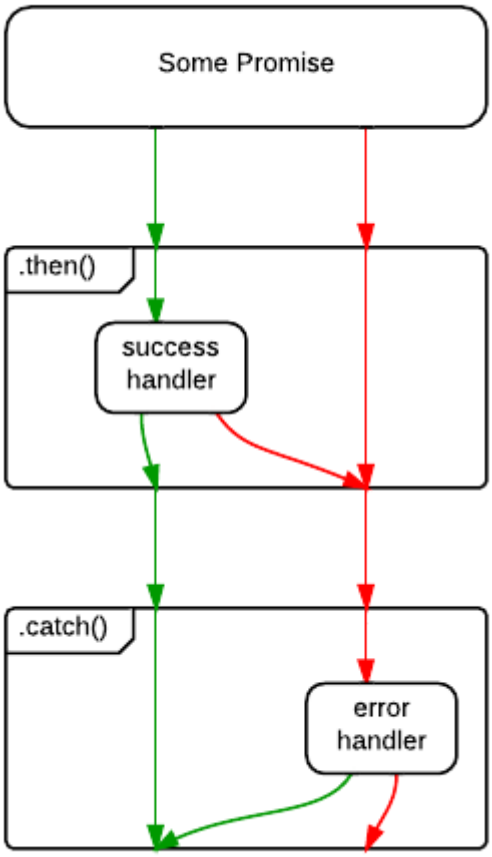
```
save()
  .then(handleSuccess)
  .catch(handleError)
;
```

The difference is subtle, but important. In the first example, an error originating in the `save()` operation will be caught, but an error originating in the `handleSuccess()` function will be swallowed.



Without .catch(), an error in the success handler is uncaught.

In the second example, `.catch()` will handle rejections from either `save()`, or `handleSuccess()`.



With .catch(), both error sources are handled. (diagram source)

Of course, the `save()` error might be a networking error, whereas the `handleSuccess()` error may be because the developer forgot to handle a specific status code. What if you want to handle them differently? You could opt to handle them both:

```
save()
  .then(
    handleSuccess,
    handleNetworkError
  )
  .catch(handleProgrammerError)
;
```

Whatever you prefer, I recommend ending all promise chains with a `.catch()`. That's worth repeating:

I recommend ending all promise chains with a `.catch()`.

How Do I Cancel a Promise?

One of the first things new promise users often wonder about is how to cancel a promise. Here's an idea: Just reject the promise with "Cancelled" as the reason. If you need to deal with it differently than a "normal" error, do your branching in your error handler.

Here are some common mistakes people make when they roll their own promise cancellation:

Adding .cancel() to the promise

Adding `.cancel()` makes the promise non-standard, but it also violates another rule of promises: Only the function that creates the promise should be able to resolve, reject, or cancel the promise. Exposing it breaks that encapsulation, and encourages people to write code that manipulates the promise in places that shouldn't know about it. Avoid spaghetti and broken promises.

Forgetting to clean up

Some clever people have figured out that there's a way to use `Promise.race()` as a cancellation mechanism. The problem with that is that cancellation control is taken from the function that creates the promise, which is the only place that you can conduct proper cleanup activities, such as clearing timeouts or freeing up memory by clearing references to data, etc...

Forgetting to handle a rejected cancel promise

Did you know that Chrome throws warning messages all over the console when you forget to handle a promise rejection? Oops!

Overly complex

The withdrawn TC39 proposal for cancellation proposed a separate messaging channel for cancellations. It also used a new concept called a cancellation token. In my opinion, the solution would have considerably bloated the promise spec, and the only feature it would have provided that speculations don't directly support is the separation

of rejections and cancellations, which, IMO, is not necessary to begin with.

Will you want to do switching depending on whether there is an exception, or a cancellation? Yes, absolutely. Is that the promise's job? In my opinion, no, it's not.

Rethinking Promise Cancellation

Generally, I pass all the information the promise needs to determine how to resolve / reject / cancel at promise creation time. That way, there's no need for a `.cancel()` method on a promise. You might be wondering how you could possibly know whether or not you're going to cancel at promise creation time.

> *"If I don't yet know whether or not to cancel, how will I know what to pass in when I create the promise?"*

If only there were some kind of object that could stand in for a potential value in the future… *oh, wait.*

The value we pass in to represent whether or not to cancel could be a promise itself. Here's how that might look:

```
const wait = (
  time,
  cancel = Promise.reject()
) => new Promise((resolve, reject) => {
  const timer = setTimeout(resolve, time);
  const noop = () => {};

  cancel.then(() => {
    clearTimeout(timer);
    reject(new Error('Cancelled'));
  }, noop);
});

const shouldCancel = Promise.resolve(); // Yes, cancel
```

Cancellable wait — try it on CodePen

We're using default parameter assignment to tell it not to cancel by default. That makes the `cancel` parameter conveniently optional. Then we set the timeout as we did before, but this time we capture the timeout's ID so that we can clear it later.

We use the `cancel.then()` method to handle the cancellation and resource cleanup. This will only run if the promise gets cancelled before it has a chance to resolve. If you cancel too late, you've missed your chance. That train has left the station.

> *Note: You may be wondering what the* `noop()` *function is for. The word noop stands for no-op, meaning a function that does nothing. Without it, V8 will throw warnings:* `UnhandledPromiseRejectionWarning: Unhandled promise rejection` *. It's a good idea to* ***always handle promise rejections,*** *even if your handler is a* `noop()` *.*

Abstracting Promise Cancellation

This is fine for a `wait()` timer, but we can abstract this idea further to encapsulate everything you have to remember:

1. Reject the cancel promise by default—we don't want to cancel or throw errors if no cancel promise gets passed in.
2. Remember to perform cleanup when you reject for cancellations.
3. Remember that the `onCancel` cleanup might itself throw an error, and that error will need handling, too. (Note that error handling is omitted in the wait example above—it's easy to forget!)

Let's create a cancellable promise utility that you can use to wrap any promise. For example, to handle network requests, etc… The signature will look like this:

```
speculation(fn: SpecFunction, shouldCancel: Promise) =>
Promise
```

The SpecFunction is just like the function you would pass into the `Promise` constructor, with one exception—it takes an `onCancel()` handler:

```
SpecFunction(resolve: Function, reject: Function, onCancel:
Function) => Void
```

```
// HOF Wraps the native Promise API
// to add take a shouldCancel promise and add
// an onCancel() callback.
const speculation = (
  fn,
  cancel = Promise.reject() // Don't cancel by default
) => new Promise((resolve, reject) => {
  const noop = () => {};

  const onCancel = (
    handleCancel
  ) => cancel.then(
      handleCancel,
      // Ignore expected cancel rejections:
      noop
```

Note that this example is just an illustration to give you the gist of how it works. There are some other edge cases you need to take into consideration. For example, in this version, `handleCancel` will be called if you cancel the promise after it is already settled.

I've implemented a maintained production version of this with edge cases covered as the open source library, Speculation.

Let's use the improved library abstraction to rewrite the cancellable `wait()` utility from before. First install speculation:

```
npm install --save speculation
```

Now you can import and use it:

```
import speculation from 'speculation';

const wait = (
  time,
  cancel = Promise.reject() // By default, don't cancel
) => speculation((resolve, reject, onCancel) => {
  const timer = setTimeout(resolve, time);

  // Use onCancel to clean up any lingering resources
  // and then call reject(). You can pass a custom reason.
  onCancel(() => {
    clearTimeout(timer);
    reject(new Error('Cancelled'));
  });
}, cancel); // remember to pass in cancel!

wait(200, wait(500)).then(
```

This simplifies things a little, because you don't have to worry about the `noop()`, catching errors in your `onCancel()`, function or other edge cases. Those details have been abstracted away by `speculation()`. Check it out and feel free to use it in real projects.

Extras of the Native JS Promise

The native `Promise` object has some extra stuff you might be interested in:

- `Promise.reject()` returns a rejected promise.
- `Promise.resolve()` returns a resolved promise.
- `Promise.race()` takes an array (or any iterable) and returns a promise that resolves with the value of the first resolved promise in the iterable, or rejects with the reason of the first promise that rejects.
- `Promise.all()` takes an array (or any iterable) and returns a promise that resolves when *all of the promises* in the iterable argument have resolved, or rejects with the reason of the first passed promise that rejects.

Conclusion

Promises have become an integral part of several idioms in JavaScript, including the WHATWG Fetch standard used for most modern ajax requests, and the Async Functions standard used to make asynchronous code look synchronous.

Async functions are stage 3 at the time of this writing, but I predict that they will soon become a very popular, very commonly used solution for asynchronous programming in JavaScript—which means that learning to appreciate promises is going to be even more important to JavaScript developers in the near future.

For instance, if you're using Redux, I suggest that you check out redux-saga: A library used to manage side-effects in Redux which depends on async functions throughout the documentation.

I hope even experienced promise users have a better understanding of what promises are and how they work, and how to use them better after reading this.

Explore the Series

- What is a Closure?
- What is the Difference Between Class and Prototypal Inheritance?
- What is a Pure Function?
- What is Function Composition?
- What is Functional Programming?
- What is a Promise?
- Soft Skills

. . .

1. *Barbara Liskov; Liuba Shrira (1988). "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems". Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation; Atlanta, Georgia, United*

Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
Mar 25, 2016 · 9 min read

Master the JavaScript Interview: What is a Pure Function?

Image: Pure — carnagenyc (CC-BY-NC 2.0)

Pure functions are essential for a variety of purposes, including functional programming, reliable concurrency, and React+Redux apps. But what does “pure function” mean?

We're going to answer this question with a free lesson from "Learn JavaScript with Eric Elliott":

Before we can tackle what a pure function is, it's probably a good idea to take a closer look at functions. There may be a different way to look at them that will make functional programming easier to understand.

What is a Function?

A **function** is a process which takes some input, called **arguments**, and produces some output called a **return value**. Functions may serve the following purposes:

- **Mapping:** Produce some output based on given inputs. A function **maps** input values to output values.
- **Procedures:** A function may be called to perform a sequence of steps. The sequence is known as a procedure, and programming in this style is known as **procedural programming**.
- **I/O:** Some functions exist to communicate with other parts of the system, such as the screen, storage, system logs, or network.

Mapping

Pure functions are all about mapping. Functions map input arguments to return values, meaning that for each set of inputs, there exists an output. A function will take the inputs and return the corresponding output.

`*Math.max()*` takes numbers as arguments and returns the largest number:

```
Math.max(2, 8, 5); // 8
```

In this example, 2, 8, & 5 are *arguments*. They're values passed into the function.

`*Math.max()*` is a function that takes any number of arguments and returns the largest argument value. In this case, the largest number we passed in was 8, and that's the number that got returned.

Functions are really important in computing and math. They help us process data in useful ways. Good programmers give functions descriptive names so that when we see the code, we can see the function names and understand what the function does.

Math has functions, too, and they work a lot like functions in JavaScript. You've probably seen functions in algebra. They look something like this:

$f(x) = 2x$

Which means that we're declaring a function called f and it takes an argument called x and multiplies x by 2.

To use this function, we simply provide a value for x:

$f(2)$

In algebra, this means exactly the same thing as writing:

4

So any place you see $f(2)$ you can substitute 4.

Now let's convert that function to JavaScript:

```
const double = x => x * 2;
```

You can examine the function's output using *`console.log()`*:

```
console.log( double(5) ); // 10
```

Remember when I said that in math functions, you could replace *`f(2)`* with *`4`*? In this case, the JavaScript engine replaces *`double(5)`* with the answer, *`10`*.

So, *`console.log( double(5) );`* is the same as *`console.log(10);`*

This is true because *`double()`* is a pure function, but if *`double()`* had side-effects, such as saving the value to disk or logging to the console, you couldn't simply replace *`double(5)`* with 10 without changing the meaning.

If you want referential transparency, you need to use pure functions.

Pure Functions

A **pure function** is a function which:

- Given the same input, will always return the same output.
- Produces no side effects.

> *A dead giveaway that a function is impure is if it makes sense to call it without using its return value. For pure functions, that's a noop.*

I recommend that you favor pure functions. Meaning, if it is practical to implement a program requirement using pure functions, you should use them over other options. Pure functions take some input and return some output based on that input. They are the simplest reusable building blocks of code in a program. Perhaps the most important design principle in computer science is KISS (Keep It Simple, Stupid). I

prefer Keep It Stupid Simple. Pure functions are stupid simple in the best possible way.

Pure functions have many beneficial properties, and form the foundation of **functional programming**. Pure functions are completely independent of outside state, and as such, they are immune to entire classes of bugs that have to do with shared mutable state. Their independent nature also makes them great candidates for parallel processing across many CPUs, and across entire distributed computing clusters, which makes them essential for many types of scientific and resource-intensive computing tasks.

Pure functions are also extremely independent—easy to move around, refactor, and reorganize in your code, making your programs more flexible and adaptable to future changes.

The Trouble with Shared State

Several years ago I was working on an app that allowed users to search a database for musical artists and load the artist's music playlist into a web player. This was around the time Google Instant landed, which displays instant search results as you type your search query. AJAX-powered autocomplete was suddenly all the rage.

The only problem was that users often type faster than an API autocomplete search response can be returned, which caused some strange bugs. It would trigger race conditions, where newer suggestions would be replaced by outdated suggestions.

Why did that happen? Because each AJAX success handler was given access to directly update the suggestion list that was displayed to users. The slowest AJAX request would always win the user's attention by blindly replacing results, even when those replaced results may have been newer.

To fix the problem, I created a suggestion manager—a single source of truth to manage the state of the query suggestions. It was aware of a currently pending AJAX request, and when the user typed something new, the pending AJAX request would be canceled before a new request was issued, so only a single response handler at a time would ever be able to trigger a UI state update.

Any sort of asynchronous operation or concurrency could cause similar race conditions. Race conditions happen if output is dependent on the sequence of uncontrollable events (such as network, device latency, user input, randomness, etc…). In fact, if you're using shared state and that state is reliant on sequences which vary depending on indeterministic factors, for all intents and purposes, the output is impossible to predict, and that means it's impossible to properly test or fully understand. As Martin Odersky (creator of Scala) puts it:

> non-determinism = parallel processing + mutable state

Program determinism is usually a desirable property in computing. Maybe you think you're OK because JS runs in a single thread, and as such, is immune to parallel processing concerns, but as the AJAX example demonstrates, a single threaded JS engine does not imply that there is no concurrency. On the contrary, there are many sources of concurrency in JavaScript. API I/O, event listeners, web workers, iframes, and timeouts can all introduce indeterminism into your program. Combine that with shared state, and you've got a recipe for bugs.

Pure functions can help you avoid those kinds of bugs.

Given the Same Input, Always Return the Same Output

With our `*double()*` function, you can replace the function call with the result, and the program will mean the same thing— `*double(5)*` will always mean the same thing as `*10*` in your program, regardless of context, no matter how many times you call it or when.

But you can't say the same thing about all functions. Some functions rely on information other than the arguments you pass in to produce results.

Consider this example:

```
Math.random(); // => 0.4011148700956255
Math.random(); // => 0.8533405303023756
Math.random(); // => 0.3550692005082965
```

Even though we didn't pass any arguments into any of the function calls, they all produced different output, meaning that *`Math.random()`* is **not pure**.

`Math.random()` produces a new random number between 0 and 1 every time you run it, so clearly you couldn't just replace it with 0.4011148700956255 without changing the meaning of the program.

That would produce the same result every time. When we ask the computer for a random number, it usually means that we want a different result than we got the last time. What's the point of a pair of dice with the same numbers printed on every side?

Sometimes we have to ask the computer for the current time. We won't go into the details of how the time functions work. For now, just copy this code:

```
const time = () => new Date().toLocaleTimeString();

time(); // => "5:15:45 PM"
```

What would happen if you replaced the *`time()`* function call with the current time?

It would always say it's the same time: the time that the function call got replaced. In other words, it could only produce the correct output once per day, and only if you ran the program at the exact moment that the function got replaced.

So clearly, *`time()`* isn't like our *`double()`* function.

A function is only pure if, given the same input, it will always produce the same output. You may remember this rule from algebra class: the same input values will always map to the same output value. However, many input values may map to the same output value. For example, the following function **is pure**:

```
const highpass = (cutoff, value) => value >= cutoff;
```

The same input values will always map to the same output value:

```
highpass(5, 5); // => true
highpass(5, 5); // => true
highpass(5, 5); // => true
```

Many input values may map to the same output value:

```
highpass(5, 123); // true
highpass(5, 6);   // true
highpass(5, 18);  // true
```

```
highpass(5, 1);   // false
highpass(5, 3);   // false
highpass(5, 4);   // false
```

A pure function must not rely on any external mutable state, because it would no longer be deterministic or referentially transparent.

Pure Functions Produce No Side Effects

A pure function produces no side effects, which means that it can't alter any external state.

Immutability

JavaScript's object arguments are references, which means that if a function were to mutate a property on an object or array parameter, that would mutate state that is accessible outside the function. Pure functions must not mutate external state.

Consider this mutating, **impure** *`addToCart()`* function:

```
// impure addToCart mutates existing cart
const addToCart = (cart, item, quantity) => {
  cart.items.push({
    item,
    quantity
  });
  return cart;
};


test('addToCart()', assert => {
  const msg = 'addToCart() should add a new item to the car
  const originalCart =     {
    items: []
  };
  const cart = addToCart(
    originalCart,
    {
      name: "Digital SLR Camera",
      price: '1495'
    },
```

It works by passing in a cart, and item to add to that cart, and an item quantity. The function then returns the same cart, with the item added to it.

The problem with this is that we've just mutated some shared state. Other functions may be relying on that cart object state to be what it was before the function was called, and now that we've mutated that shared state, we have to worry about what impact it will have on the program logic if we change the order in which functions have been called. Refactoring the code could result in bugs popping up, which could screw up orders, and result in unhappy customers.

Now consider this version:

```
// Pure addToCart() returns a new cart
// It does not mutate the original.
const addToCart = (cart, item, quantity) => {
  const newCart = lodash.cloneDeep(cart);

  newCart.items.push({
    item,
    quantity
  });
  return newCart;

};


test('addToCart()', assert => {
  const msg = 'addToCart() should add a new item to the car
  const originalCart = {
    items: []
  };

  // deep-freeze on npm
  // throws an error if original is mutated
  deepFreeze(originalCart);

  const cart = addToCart(
    originalCart,
    {
      name: "Digital SLR Camera",
```

In this example, we have an array nested in an object, which is why I reached for a deep clone. This is more complex state than you'll typically be dealing with. For most things, you can break it down into smaller chunks.

For example, Redux lets you compose reducers rather than deal with the entire app state inside each reducer. The result is that you don't have to create a deep clone of the entire app state every time you want to update just a small part of it. Instead, you can use non-destructive array methods, or `*Object.assign()*` to update a small part of the app state.

SASS Documents

airbnb / javascript

JavaScript Style Guide

#javascript #eslint #naming-conventions #arrow-functions #style-guide #style-linter #es6 #es2015 #linting #styleguide

1,597 commits | 3 branches | 81 releases | 380 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

1pete and ljharb [eslint config] [*] [fix] ensure `whitespace` entry point is compatib... ··· Latest commit 396166b 9 days ago

css-in-javascript	[guide] [css] Fixed Italic subtitle in css-in-js README.md	2 months ago
linters	Add linting for Markdown prose	9 months ago
packages	[eslint config] [*] [fix] ensure `whitespace` entry point is compatib...	9 days ago
react	[guide] [react] add comment about what JS standards are followed	17 days ago
.editorconfig	Add editorconfig	a year ago
.gitignore	Only apps should have lockfiles.	9 months ago
.npmrc	Only apps should have lockfiles.	9 months ago
.travis.yml	[Tests] on `node` `v9`; use `nvm install-latest-npm` so new npm doesn...	4 months ago
LICENSE.md	[guide] Update license year	2 months ago
README.md	[guide] Add an anchor for rule 13.7	16 days ago
package.json	Add linting for Markdown prose	9 months ago

README.md

Airbnb JavaScript Style Guide() {

A mostly reasonable approach to JavaScript

> **Note**: this guide assumes you are using Babel, and requires that you use babel-preset-airbnb or the equivalent. It also assumes you are installing shims/polyfills in your app, with airbnb-browser-shims or the equivalent.

This guide is available in other languages too. See Translation

Other Style Guides

- ES5 (Deprecated)
- React
- CSS-in-JavaScript
- CSS & Sass
- Ruby

Table of Contents

1. Types
2. References
3. Objects
4. Arrays

Types

- 1.1 **Primitives**: When you access a primitive type you work directly on its value.
    - `string`
    - `number`
    - `boolean`
    - `null`
    - `undefined`
    - `symbol`

```
const foo = 1;
let bar = foo;
```

```
bar = 9;

console.log(foo, bar); // => 1, 9
```

  - Symbols cannot be faithfully polyfilled, so they should not be used when targeting browsers/environments that don't support them natively.

- 1.2 **Complex**: When you access a complex type you work on a reference to its value.
  - `object`
  - `array`
  - `function`

```
const foo = [1, 2];
const bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

⬆️ back to top

References

- 2.1 Use `const` for all of your references; avoid using `var`. eslint: `prefer-const`, `no-const-assign`

> Why? This ensures that you can't reassign your references, which can lead to bugs and difficult to comprehend code.

```
// bad
var a = 1;
var b = 2;

// good
const a = 1;
const b = 2;
```

- 2.2 If you must reassign references, use `let` instead of `var`. eslint: `no-var` jscs: `disallowVar`

> Why? `let` is block-scoped rather than function-scoped like `var`.

```
// bad
var count = 1;
if (true) {
  count += 1;
}

// good, use the let.
let count = 1;
if (true) {
  count += 1;
}
```

- 2.3 Note that both `let` and `const` are block-scoped.

```
// const and let only exist in the blocks they are defined in.
{
  let a = 1;
  const b = 1;
}
```

```
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

⬆️ back to top

Objects

- 3.1 Use the literal syntax for object creation. eslint: `no-new-object`

```
// bad
const item = new Object();

// good
const item = {};
```

- 3.2 Use computed property names when creating objects with dynamic property names.

> Why? They allow you to define all the properties of an object in one place.

```
function getKey(k) {
  return `a key named ${k}`;
}

// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;

// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```

- 3.3 Use object method shorthand. eslint: `object-shorthand` jscs: `requireEnhancedObjectLiterals`

```
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

- 3.4 Use property value shorthand. eslint: `object-shorthand` jscs: `requireEnhancedObjectLiterals`

> Why? It is shorter to write and descriptive.

```
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

- 3.5 Group your shorthand properties at the beginning of your object declaration.

> Why? It's easier to tell which properties are using the shorthand.

```
const anakinSkywalker = 'Anakin Skywalker';
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker,
};

// good
const obj = {
  lukeSkywalker,
  anakinSkywalker,
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4,
};
```

- 3.6 Only quote properties that are invalid identifiers. eslint: `quote-props` jscs: `disallowQuotedKeysInObjects`

> Why? In general we consider it subjectively easier to read. It improves syntax highlighting, and is also more easily optimized by many JS engines.

```
// bad
const bad = {
  'foo': 3,
  'bar': 4,
  'data-blah': 5,
};

// good
const good = {
  foo: 3,
  bar: 4,
  'data-blah': 5,
};
```

- 3.7 Do not call `Object.prototype` methods directly, such as `hasOwnProperty`, `propertyIsEnumerable`, and `isPrototypeOf`.

> Why? These methods may be shadowed by properties on the object in question - consider `{ hasOwnProperty: false }` - or, the object may be a null object (`Object.create(null)`).

```
// bad
console.log(object.hasOwnProperty(key));

// good
console.log(Object.prototype.hasOwnProperty.call(object, key));

// best
const has = Object.prototype.hasOwnProperty; // cache the lookup once, in module scope.
/* or */
import has from 'has'; // https://www.npmjs.com/package/has
// ...
console.log(has.call(object, key));
```

- 3.8 Prefer the object spread operator over `Object.assign` to shallow-copy objects. Use the object rest operator to get a new object with certain properties omitted.

```
// very bad
const original = { a: 1, b: 2 };
const copy = Object.assign(original, { c: 3 }); // this mutates `original` ಠ_ಠ
delete copy.a; // so does this

// bad
const original = { a: 1, b: 2 };
const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }

// good
const original = { a: 1, b: 2 };
const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }

const { a, ...noA } = copy; // noA => { b: 2, c: 3 }
```

⬆️ back to top

Arrays

- 4.1 Use the literal syntax for array creation. eslint: `no-array-constructor`

```
// bad
const items = new Array();

// good
const items = [];
```

- 4.2 Use Array#push instead of direct assignment to add items to an array.

```
const someStack = [];

// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

- 4.3 Use array spreads `...` to copy arrays.

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i += 1) {
```

```
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

- 4.4 To convert an array-like object to an array, use spreads `...` instead of Array.from.

```
const foo = document.querySelectorAll('.foo');

// good
const nodes = Array.from(foo);

// best
const nodes = [...foo];
```

- 4.5 Use Array.from instead of spread `...` for mapping over iterables, because it avoids creating an intermediate array.

```
// bad
const baz = [...foo].map(bar);

// good
const baz = Array.from(foo, bar);
```

- 4.6 Use return statements in array method callbacks. It's ok to omit the return if the function body consists of a single statement returning an expression without side effects, following 8.2. eslint: `array-callback-return`

```
// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map(x => x + 1);

// bad - no returned value means `acc` becomes undefined after the first iteration
[[0, 1], [2, 3], [4, 5]].reduce((acc, item, index) => {
  const flatten = acc.concat(item);
  acc[index] = flatten;
});

// good
[[0, 1], [2, 3], [4, 5]].reduce((acc, item, index) => {
  const flatten = acc.concat(item);
  acc[index] = flatten;
  return flatten;
});

// bad
inbox.filter((msg) => {
  const { subject, author } = msg;
  if (subject === 'Mockingbird') {
    return author === 'Harper Lee';
  } else {
    return false;
  }
});

// good
inbox.filter((msg) => {
  const { subject, author } = msg;
  if (subject === 'Mockingbird') {
    return author === 'Harper Lee';
  }
```

```
  return false;
});
```

- 4.7 Use line breaks after open and before close array brackets if an array has multiple lines

```
// bad
const arr = [
  [0, 1], [2, 3], [4, 5],
];

const objectInArray = [{
  id: 1,
}, {
  id: 2,
}];

const numberInArray = [
  1, 2,
];

// good
const arr = [[0, 1], [2, 3], [4, 5]];

const objectInArray = [
  {
    id: 1,
  },
  {
    id: 2,
  },
];

const numberInArray = [
  1,
  2,
];
```

⬆️ back to top

Destructuring

- 5.1 Use object destructuring when accessing and using multiple properties of an object. eslint: `prefer-destructuring` jscs: `requireObjectDestructuring`

> Why? Destructuring saves you from creating temporary references for those properties.

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// good
function getFullName(user) {
  const { firstName, lastName } = user;
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

- 5.2 Use array destructuring. eslint: `prefer-destructuring` jscs: `requireArrayDestructuring`

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

- 5.3 Use object destructuring for multiple return values, not array destructuring. jscs: `disallowArrayDestructuringReturn`

> Why? You can add new properties over time or change the order of things without breaking call sites.

```
// bad
function processInput(input) {
  // then a miracle occurs
  return [left, right, top, bottom];
}

// the caller needs to think about the order of return data
const [left, __, top] = processInput(input);

// good
function processInput(input) {
  // then a miracle occurs
  return { left, right, top, bottom };
}

// the caller selects only the data they need
const { left, top } = processInput(input);
```

⬆️ back to top

Strings

- 6.1 Use single quotes `''` for strings. eslint: `quotes` jscs: `validateQuoteMarks`

```
// bad
const name = "Capt. Janeway";

// bad - template literals should contain interpolation or newlines
const name = `Capt. Janeway`;

// good
const name = 'Capt. Janeway';
```

- 6.2 Strings that cause the line to go over 100 characters should not be written across multiple lines using string concatenation.

> Why? Broken strings are painful to work with and make code less searchable.

```
// bad
const errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// bad
const errorMessage = 'This is a super long error that was thrown because ' +
```

```
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';

// good
const errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about
```

- 6.3 When programmatically building up strings, use template strings instead of concatenation. eslint: `prefer-template` `template-curly-spacing` jscs: `requireTemplateStrings`

> Why? Template strings give you a readable, concise syntax with proper newlines and string interpolation features.

```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// bad
function sayHi(name) {
  return `How are you, ${ name }?`;
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

- 6.4 Never use `eval()` on a string, it opens too many vulnerabilities. eslint: `no-eval`
- 6.5 Do not unnecessarily escape characters in strings. eslint: `no-useless-escape`

> Why? Backslashes harm readability, thus they should only be present when necessary.

```
// bad
const foo = '\'this\' \i\s \"quoted\"';

// good
const foo = '\'this\' is "quoted"';
const foo = `my name is '${name}'`;
```

⬆️ back to top

Functions

- 7.1 Use named function expressions instead of function declarations. eslint: `func-style` jscs: `disallowFunctionDeclarations`

> Why? Function declarations are hoisted, which means that it's easy - too easy - to reference the function before it is defined in the file. This harms readability and maintainability. If you find that a function's definition is large or complex enough that it is interfering with understanding the rest of the file, then perhaps it's time to extract it to its own module! Don't forget to explicitly name the expression, regardless of whether or not the name is inferred from the containing variable (which is often the case in modern browsers or when using compilers such as Babel). This eliminates any assumptions made about the Error's call stack. (Discussion)

```
// bad
function foo() {
```

```
  // ...
}

// bad
const foo = function () {
  // ...
};

// good
// lexical name distinguished from the variable-referenced invocation(s)
const short = function longUniqueMoreDescriptiveLexicalFoo() {
  // ...
};
```

- 7.2 Wrap immediately invoked function expressions in parentheses. eslint: `wrap-iife` jscs: `requireParenthesesAroundIIFE`

> Why? An immediately invoked function expression is a single unit - wrapping both it, and its invocation parens, in parens, cleanly expresses this. Note that in a world with modules everywhere, you almost never need an IIFE.

```
// immediately-invoked function expression (IIFE)
(function () {
  console.log('Welcome to the Internet. Please follow me.');
}());
```

- 7.3 Never declare a function in a non-function block (`if` , `while` , etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears. eslint: `no-loop-func`
- 7.4 **Note:** ECMA-262 defines a `block` as a list of statements. A function declaration is not a statement.

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
  };
}
```

- 7.5 Never name a parameter `arguments` . This will take precedence over the `arguments` object that is given to every function scope.

```
// bad
function foo(name, options, arguments) {
  // ...
}

// good
function foo(name, options, args) {
  // ...
}
```

- 7.6 Never use `arguments` , opt to use rest syntax `...` instead. eslint: `prefer-rest-params`

> Why? `...` is explicit about which arguments you want pulled. Plus, rest arguments are a real Array, and not merely Array-like like `arguments` .

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

- 7.7 Use default parameter syntax rather than mutating function arguments.

```
// really bad
function handleThings(opts) {
  // No! We shouldn’t mutate function arguments.
  // Double bad: if opts is falsy it'll be set to an object which may
  // be what you want but it can introduce subtle bugs.
  opts = opts || {};
  // ...
}

// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}

// good
function handleThings(opts = {}) {
  // ...
}
```

- 7.8 Avoid side effects with default parameters.

> Why? They are confusing to reason about.

```
var b = 1;
// bad
function count(a = b++) {
  console.log(a);
}
count();  // 1
count();  // 2
count(3); // 3
count();  // 3
```

- 7.9 Always put default parameters last.

```
// bad
function handleThings(opts = {}, name) {
  // ...
}

// good
function handleThings(name, opts = {}) {
  // ...
}
```

- 7.10 Never use the Function constructor to create a new function. eslint: `no-new-func`

> Why? Creating a function in this way evaluates a string similarly to eval(), which opens vulnerabilities.

```
// bad
var add = new Function('a', 'b', 'return a + b');

// still bad
var subtract = Function('a', 'b', 'return a - b');
```

- 7.11 Spacing in a function signature. eslint: `space-before-function-paren` `space-before-blocks`

> Why? Consistency is good, and you shouldn't have to add or remove a space when adding or removing a name.

```
// bad
const f = function(){};
const g = function (){};
const h = function() {};

// good
const x = function () {};
const y = function a() {};
```

- 7.12 Never mutate parameters. eslint: `no-param-reassign`

> Why? Manipulating objects passed in as parameters can cause unwanted variable side effects in the original caller.

```
// bad
function f1(obj) {
  obj.key = 1;
}

// good
function f2(obj) {
  const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? obj.key : 1;
}
```

- 7.13 Never reassign parameters. eslint: `no-param-reassign`

> Why? Reassigning parameters can lead to unexpected behavior, especially when accessing the `arguments` object. It can also cause optimization issues, especially in V8.

```
// bad
function f1(a) {
  a = 1;
  // ...
}

function f2(a) {
  if (!a) { a = 1; }
  // ...
}

// good
function f3(a) {
  const b = a || 1;
  // ...
}

function f4(a = 1) {
  // ...
}
```

- 7.14 Prefer the use of the spread operator `...` to call variadic functions. eslint: `prefer-spread`

> Why? It's cleaner, you don't need to supply a context, and you can not easily compose `new` with `apply`.

```
// bad
const x = [1, 2, 3, 4, 5];
console.log.apply(console, x);

// good
const x = [1, 2, 3, 4, 5];
console.log(...x);

// bad
new (Function.prototype.bind.apply(Date, [null, 2016, 8, 5]));

// good
new Date(...[2016, 8, 5]);
```

- 7.15 Functions with multiline signatures, or invocations, should be indented just like every other multiline list in this guide: with each item on a line by itself, with a trailing comma on the last item.

```
// bad
function foo(bar,
             baz,
             quux) {
  // ...
}

// good
function foo(
  bar,
  baz,
  quux,
) {
  // ...
}

// bad
console.log(foo,
  bar,
  baz);

// good
console.log(
  foo,
  bar,
  baz,
);
```

⬆️ back to top

Arrow Functions

- 8.1 When you must use an anonymous function (as when passing an inline callback), use arrow function notation. eslint: `prefer-arrow-callback` , `arrow-spacing` jscs: `requireArrowFunctions`

  > Why? It creates a version of the function that executes in the context of `this` , which is usually what you want, and is a more concise syntax.

  > Why not? If you have a fairly complicated function, you might move that logic out into its own named function expression.

```
// bad
[1, 2, 3].map(function (x) {
  const y = x + 1;
  return x * y;
});
```

```
// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

- 8.2 If the function body consists of a single statement returning an expression without side effects, omit the braces and use the implicit return. Otherwise, keep the braces and use a `return` statement. eslint: `arrow-parens`, `arrow-body-style` jscs: `disallowParenthesesAroundArrowParam`, `requireShorthandArrowFunctions`

  > Why? Syntactic sugar. It reads well when multiple functions are chained together.

```
// bad
[1, 2, 3].map(number => {
  const nextNumber = number + 1;
  `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map(number => `A string containing the ${number}.`);

// good
[1, 2, 3].map((number) => {
  const nextNumber = number + 1;
  return `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map((number, index) => ({
  [index]: number,
}));

// No implicit return with side effects
function foo(callback) {
  const val = callback();
  if (val === true) {
    // Do something if callback returns true
  }
}

let bool = false;

// bad
foo(() => bool = true);

// good
foo(() => {
  bool = true;
});
```

- 8.3 In case the expression spans over multiple lines, wrap it in parentheses for better readability.

  > Why? It shows clearly where the function starts and ends.

```
// bad
['get', 'post', 'put'].map(httpMethod => Object.prototype.hasOwnProperty.call(
    httpMagicObjectWithAVeryLongName,
    httpMethod,
  )
);

// good
['get', 'post', 'put'].map(httpMethod => (
  Object.prototype.hasOwnProperty.call(
    httpMagicObjectWithAVeryLongName,
    httpMethod,
```

```
  )
));
```

- 8.4 If your function takes a single argument and doesn't use braces, omit the parentheses. Otherwise, always include parentheses around arguments for clarity and consistency. Note: it is also acceptable to always use parentheses, in which case use the "always" option for eslint or do not include `disallowParenthesesAroundArrowParam` for jscs. eslint: `arrow-parens` jscs: `disallowParenthesesAroundArrowParam`

  > Why? Less visual clutter.

```
// bad
[1, 2, 3].map((x) => x * x);

// good
[1, 2, 3].map(x => x * x);

// good
[1, 2, 3].map(number => (
  `A long string with the ${number}. It's so long that we don't want it to take up space on the .map line!`
));

// bad
[1, 2, 3].map(x => {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

- 8.5 Avoid confusing arrow function syntax (`=>`) with comparison operators (`<=` , `>=`). eslint: `no-confusing-arrow`

```
// bad
const itemHeight = item => item.height > 256 ? item.largeSize : item.smallSize;

// bad
const itemHeight = (item) => item.height > 256 ? item.largeSize : item.smallSize;

// good
const itemHeight = item => (item.height > 256 ? item.largeSize : item.smallSize);

// good
const itemHeight = (item) => {
  const { height, largeSize, smallSize } = item;
  return height > 256 ? largeSize : smallSize;
};
```

⬆️ back to top

Classes & Constructors

- 9.1 Always use `class` . Avoid manipulating `prototype` directly.

  > Why? `class` syntax is more concise and easier to reason about.

```
// bad
function Queue(contents = []) {
  this.queue = [...contents];
}
Queue.prototype.pop = function () {
```

```
  const value = this.queue[0];
  this.queue.splice(0, 1);
  return value;
};

// good
class Queue {
  constructor(contents = []) {
    this.queue = [...contents];
  }
  pop() {
    const value = this.queue[0];
    this.queue.splice(0, 1);
    return value;
  }
}
```

- 9.2 Use `extends` for inheritance.

> Why? It is a built-in way to inherit prototype functionality without breaking `instanceof`.

```
// bad
const inherits = require('inherits');
function PeekableQueue(contents) {
  Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function () {
  return this.queue[0];
};

// good
class PeekableQueue extends Queue {
  peek() {
    return this.queue[0];
  }
}
```

- 9.3 Methods can return `this` to help with method chaining.

```
// bad
Jedi.prototype.jump = function () {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function (height) {
  this.height = height;
};

const luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

// good
class Jedi {
  jump() {
    this.jumping = true;
    return this;
  }

  setHeight(height) {
    this.height = height;
    return this;
  }
}
```

```
const luke = new Jedi();

luke.jump()
  .setHeight(20);
```

- 9.4 It's okay to write a custom toString() method, just make sure it works successfully and causes no side effects.

```
class Jedi {
  constructor(options = {}) {
    this.name = options.name || 'no name';
  }

  getName() {
    return this.name;
  }

  toString() {
    return `Jedi - ${this.getName()}`;
  }
}
```

- 9.5 Classes have a default constructor if one is not specified. An empty constructor function or one that just delegates to a parent class is unnecessary. eslint: `no-useless-constructor`

```
// bad
class Jedi {
  constructor() {}

  getName() {
    return this.name;
  }
}

// bad
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
  }
}

// good
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
    this.name = 'Rey';
  }
}
```

- 9.6 Avoid duplicate class members. eslint: `no-dupe-class-members`

> Why? Duplicate class member declarations will silently prefer the last one - having duplicates is almost certainly a bug.

```
// bad
class Foo {
  bar() { return 1; }
  bar() { return 2; }
}

// good
class Foo {
  bar() { return 1; }
}

// good
```

```
class Foo {
  bar() { return 2; }
}
```

⬆️ back to top

Modules

- 10.1 Always use modules (`import` / `export`) over a non-standard module system. You can always transpile to your preferred module system.

  > Why? Modules are the future, let's start using the future now.

```
// bad
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;

// ok
import AirbnbStyleGuide from './AirbnbStyleGuide';
export default AirbnbStyleGuide.es6;

// best
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- 10.2 Do not use wildcard imports.

  > Why? This makes sure you have a single default export.

```
// bad
import * as AirbnbStyleGuide from './AirbnbStyleGuide';

// good
import AirbnbStyleGuide from './AirbnbStyleGuide';
```

- 10.3 And do not export directly from an import.

  > Why? Although the one-liner is concise, having one clear way to import and one clear way to export makes things consistent.

```
// bad
// filename es6.js
export { es6 as default } from './AirbnbStyleGuide';

// good
// filename es6.js
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- 10.4 Only import from a path in one place. eslint: `no-duplicate-imports`

  > Why? Having multiple lines that import from the same path can make code harder to maintain.

```
// bad
import foo from 'foo';
// … some other imports … //
import { named1, named2 } from 'foo';

// good
import foo, { named1, named2 } from 'foo';

// good
```

```
import foo, {
  named1,
  named2,
} from 'foo';
```

- 10.5 Do not export mutable bindings. eslint: `import/no-mutable-exports`

> Why? Mutation should be avoided in general, but in particular when exporting mutable bindings. While this technique may be needed for some special cases, in general, only constant references should be exported.

```
// bad
let foo = 3;
export { foo };

// good
const foo = 3;
export { foo };
```

- 10.6 In modules with a single export, prefer default export over named export. eslint: `import/prefer-default-export`

> Why? To encourage more files that only ever export one thing, which is better for readability and maintainability.

```
// bad
export function foo() {}

// good
export default function foo() {}
```

- 10.7 Put all `import` s above non-import statements. eslint: `import/first`

> Why? Since `import` s are hoisted, keeping them all at the top prevents surprising behavior.

```
// bad
import foo from 'foo';
foo.init();

import bar from 'bar';

// good
import foo from 'foo';
import bar from 'bar';

foo.init();
```

- 10.8 Multiline imports should be indented just like multiline array and object literals.

> Why? The curly braces follow the same indentation rules as every other curly brace block in the style guide, as do the trailing commas.

```
// bad
import {longNameA, longNameB, longNameC, longNameD, longNameE} from 'path';

// good
import {
  longNameA,
  longNameB,
  longNameC,
  longNameD,
  longNameE,
} from 'path';
```

- 10.9 Disallow Webpack loader syntax in module import statements. eslint: `import/no-webpack-loader-syntax`

> Why? Since using Webpack syntax in the imports couples the code to a module bundler. Prefer using the loader syntax in `webpack.config.js`.

```
// bad
import fooSass from 'css!sass!foo.scss';
import barCss from 'style!css!bar.css';

// good
import fooSass from 'foo.scss';
import barCss from 'bar.css';
```

⬆️ back to top

Iterators and Generators

- 11.1 Don't use iterators. Prefer JavaScript's higher-order functions instead of loops like `for-in` or `for-of`. eslint: `no-iterator` `no-restricted-syntax`

> Why? This enforces our immutable rule. Dealing with pure functions that return values is easier to reason about than side effects.

> Use `map()` / `every()` / `filter()` / `find()` / `findIndex()` / `reduce()` / `some()` / ... to iterate over arrays, and `Object.keys()` / `Object.values()` / `Object.entries()` to produce arrays so you can iterate over objects.

```
const numbers = [1, 2, 3, 4, 5];

// bad
let sum = 0;
for (let num of numbers) {
  sum += num;
}
sum === 15;

// good
let sum = 0;
numbers.forEach((num) => {
  sum += num;
});
sum === 15;

// best (use the functional force)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;

// bad
const increasedByOne = [];
for (let i = 0; i < numbers.length; i++) {
  increasedByOne.push(numbers[i] + 1);
}

// good
const increasedByOne = [];
numbers.forEach((num) => {
  increasedByOne.push(num + 1);
});

// best (keeping it functional)
const increasedByOne = numbers.map(num => num + 1);
```

- 11.2 Don't use generators for now.

> Why? They don't transpile well to ES5.

- 11.3 If you must use generators, or if you disregard our advice, make sure their function signature is spaced properly. eslint: `generator-star-spacing`

  > Why? `function` and `*` are part of the same conceptual keyword - `*` is not a modifier for `function`, `function*` is a unique construct, different from `function`.

```
// bad
function * foo() {
  // ...
}

// bad
const bar = function * () {
  // ...
};

// bad
const baz = function *() {
  // ...
};

// bad
const quux = function*() {
  // ...
};

// bad
function*foo() {
  // ...
}

// bad
function *foo() {
  // ...
}

// very bad
function
*
foo() {
  // ...
}

// very bad
const wat = function
*
() {
  // ...
};

// good
function* foo() {
  // ...
}

// good
const foo = function* () {
  // ...
};
```

⬆️ back to top

Properties

- 12.1 Use dot notation when accessing properties. eslint: `dot-notation` jscs: `requireDotNotation`

```
const luke = {
  jedi: true,
  age: 28,
};

// bad
const isJedi = luke['jedi'];

// good
const isJedi = luke.jedi;
```

- 12.2 Use bracket notation `[]` when accessing properties with a variable.

```
const luke = {
  jedi: true,
  age: 28,
};

function getProp(prop) {
  return luke[prop];
}

const isJedi = getProp('jedi');
```

- 12.3 Use exponentiation operator `**` when calculating exponentiations. eslint: `no-restricted-properties`.

```
// bad
const binary = Math.pow(2, 10);

// good
const binary = 2 ** 10;
```

⬆️ back to top

Variables

- 13.1 Always use `const` or `let` to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace. Captain Planet warned us of that. eslint: `no-undef` `prefer-const`

```
// bad
superPower = new SuperPower();

// good
const superPower = new SuperPower();
```

- 13.2 Use one `const` or `let` declaration per variable. eslint: `one-var` jscs: `disallowMultipleVarDecl`

> Why? It's easier to add new variable declarations this way, and you never have to worry about swapping out a `;` for a `,` or introducing punctuation-only diffs. You can also step through each declaration with the debugger, instead of jumping through all of them at once.

```
// bad
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
    goSportsTeam = true;
```

```
    dragonball = 'z';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

- 13.3 Group all your `const` s and then group all your `let` s.

> Why? This is helpful when later on you might need to assign a variable depending on one of the previous assigned variables.

```
// bad
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// good
const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

- 13.4 Assign variables where you need them, but place them in a reasonable place.

> Why? `let` and `const` are block scoped and not function scoped.

```
// bad - unnecessary function call
function checkName(hasName) {
  const name = getName();

  if (hasName === 'test') {
    return false;
  }

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}

// good
function checkName(hasName) {
  if (hasName === 'test') {
    return false;
  }

  const name = getName();

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}
```

- 13.5 Don't chain variable assignments. eslint: `no-multi-assign`

> Why? Chaining variable assignments creates implicit global variables.

```
// bad
(function example() {
  // JavaScript interprets this as
  // let a = ( b = ( c = 1 ) );
  // The let keyword only applies to variable a; variables b and c become
  // global variables.
  let a = b = c = 1;
}());

console.log(a); // throws ReferenceError
console.log(b); // 1
console.log(c); // 1

// good
(function example() {
  let a = 1;
  let b = a;
  let c = a;
}());

console.log(a); // throws ReferenceError
console.log(b); // throws ReferenceError
console.log(c); // throws ReferenceError

// the same applies for `const`
```

- 13.6 Avoid using unary increments and decrements (++, --). eslint `no-plusplus`

> Why? Per the eslint documentation, unary increment and decrement statements are subject to automatic semicolon insertion and can cause silent errors with incrementing or decrementing values within an application. It is also more expressive to mutate your values with statements like `num += 1` instead of `num++` or `num ++`. Disallowing unary increment and decrement statements also prevents you from pre-incrementing/pre-decrementing values unintentionally which can also cause unexpected behavior in your programs.

```
// bad

const array = [1, 2, 3];
let num = 1;
num++;
--num;

let sum = 0;
let truthyCount = 0;
for (let i = 0; i < array.length; i++) {
  let value = array[i];
  sum += value;
  if (value) {
    truthyCount++;
  }
}

// good

const array = [1, 2, 3];
let num = 1;
num += 1;
num -= 1;

const sum = array.reduce((a, b) => a + b, 0);
const truthyCount = array.filter(Boolean).length;
```

- 13.7 Avoid linebreaks before or after `=` in an assignment. If your assignment violates `max-len`, surround the value in parens. eslint `operator-linebreak`.

  > Why? Linebreaks surrounding `=` can obfuscate the value of an assignment.

```
// bad
const foo =
  superLongLongLongLongLongLongLongFunctionName();

// bad
const foo
  = 'superLongLongLongLongLongLongLongString';

// good
const foo = (
  superLongLongLongLongLongLongLongFunctionName()
);

// good
const foo = 'superLongLongLongLongLongLongLongString';
```

⬆️ back to top

Hoisting

- 14.1 `var` declarations get hoisted to the top of their closest enclosing function scope, their assignment does not. `const` and `let` declarations are blessed with a new concept called Temporal Dead Zones (TDZ). It's important to know why typeof is no longer safe.

```
// we know this wouldn't work (assuming there
// is no notDefined global variable)
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// creating a variable declaration after you
// reference the variable will work due to
// variable hoisting. Note: the assignment
// value of `true` is not hoisted.
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// the interpreter is hoisting the variable
// declaration to the top of the scope,
// which means our example could be rewritten as:
function example() {
  let declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}

// using const and let
function example() {
  console.log(declaredButNotAssigned); // => throws a ReferenceError
  console.log(typeof declaredButNotAssigned); // => throws a ReferenceError
  const declaredButNotAssigned = true;
}
```

- 14.2 Anonymous function expressions hoist their variable name, but not the function assignment.

```
function example() {
  console.log(anonymous); // => undefined
```

```
  anonymous(); // => TypeError anonymous is not a function

  var anonymous = function () {
    console.log('anonymous function expression');
  };
}
```

- 14.3 Named function expressions hoist the variable name, not the function name or the function body.

```
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
    console.log('Flying');
  };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  var named = function named() {
    console.log('named');
  };
}
```

- 14.4 Function declarations hoist their name and the function body.

```
function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
  }
}
```

- For more information refer to JavaScript Scoping & Hoisting by Ben Cherry.

⬆️ back to top

Comparison Operators & Equality

- 15.1 Use `===` and `!==` over `==` and `!=`. eslint: `eqeqeq`
- 15.2 Conditional statements such as the `if` statement evaluate their expression using coercion with the `ToBoolean` abstract method and always follow these simple rules:
  - **Objects** evaluate to **true**
  - **Undefined** evaluates to **false**
  - **Null** evaluates to **false**
  - **Booleans** evaluate to **the value of the boolean**
  - **Numbers** evaluate to **false** if **+0, -0, or NaN**, otherwise **true**
  - **Strings** evaluate to **false** if an empty string `''`, otherwise **true**

```
if ([0] && []) {
  // true
  // an array (even an empty one) is an object, objects will evaluate to true
}
```

- 15.3 Use shortcuts for booleans, but explicit comparisons for strings and numbers.

```
// bad
if (isValid === true) {
  // ...
}

// good
if (isValid) {
  // ...
}

// bad
if (name) {
  // ...
}

// good
if (name !== '') {
  // ...
}

// bad
if (collection.length) {
  // ...
}

// good
if (collection.length > 0) {
  // ...
}
```

- 15.4 For more information see Truth Equality and JavaScript by Angus Croll.
- 15.5 Use braces to create blocks in `case` and `default` clauses that contain lexical declarations (e.g. `let`, `const`, `function`, and `class`). eslint: `no-case-declarations`

> Why? Lexical declarations are visible in the entire `switch` block but only get initialized when assigned, which only happens when its `case` is reached. This causes problems when multiple `case` clauses attempt to define the same thing.

```
// bad
switch (foo) {
  case 1:
    let x = 1;
    break;
  case 2:
    const y = 2;
    break;
  case 3:
    function f() {
      // ...
    }
    break;
  default:
    class C {}
}

// good
switch (foo) {
```

```
  case 1: {
    let x = 1;
    break;
  }
  case 2: {
    const y = 2;
    break;
  }
  case 3: {
    function f() {
      // ...
    }
    break;
  }
  case 4:
    bar();
    break;
  default: {
    class C {}
  }
}
```

- 15.6 Ternaries should not be nested and generally be single line expressions. eslint: `no-nested-ternary`

```
// bad
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null;

// split into 2 separated ternary expressions
const maybeNull = value1 > value2 ? 'baz' : null;

// better
const foo = maybe1 > maybe2
  ? 'bar'
  : maybeNull;

// best
const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

- 15.7 Avoid unneeded ternary statements. eslint: `no-unneeded-ternary`

```
// bad
const foo = a ? a : b;
const bar = c ? true : false;
const baz = c ? false : true;

// good
const foo = a || b;
const bar = !!c;
const baz = !c;
```

- 15.8 When mixing operators, enclose them in parentheses. The only exception is the standard arithmetic operators (`+` , `-` , `*` , & `/`) since their precedence is broadly understood. eslint: `no-mixed-operators`

> Why? This improves readability and clarifies the developer's intention.

```
// bad
const foo = a && b < 0 || c > 0 || d + 1 === 0;

// bad
const bar = a ** b - 5 % d;

// bad
// one may be confused into thinking (a || b) && c
```

```
if (a || b && c) {
  return d;
}

// good
const foo = (a && b < 0) || c > 0 || (d + 1 === 0);

// good
const bar = (a ** b) - (5 % d);

// good
if (a || (b && c)) {
  return d;
}

// good
const bar = a + b / c * d;
```

⬆️ back to top

Blocks

- 16.1 Use braces with all multi-line blocks. eslint: `nonblock-statement-body-position`

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
function foo() { return false; }

// good
function bar() {
  return false;
}
```

- 16.2 If you're using multi-line blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace. eslint: `brace-style` jscs: `disallowNewlineBeforeBlockStatements`

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}

// good
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

- 16.3 If an `if` block always executes a `return` statement, the subsequent `else` block is unnecessary. A `return` in an `else if` block following an `if` block that contains a `return` can be separated into multiple `if` blocks. eslint: `no-else-return`

```
// bad
function foo() {
  if (x) {
    return x;
  } else {
    return y;
  }
}

// bad
function cats() {
  if (x) {
    return x;
  } else if (y) {
    return y;
  }
}

// bad
function dogs() {
  if (x) {
    return x;
  } else {
    if (y) {
      return y;
    }
  }
}

// good
function foo() {
  if (x) {
    return x;
  }

  return y;
}

// good
function cats() {
  if (x) {
    return x;
  }

  if (y) {
    return y;
  }
}

//good
function dogs(x) {
  if (x) {
    if (z) {
      return y;
    }
  } else {
    return z;
  }
}
```

⬆️ back to top

Control Statements

- 17.1 In case your control statement (`if` , `while` etc.) gets too long or exceeds the maximum line length, each (grouped) condition could be put into a new line. The logical operator should begin the line.

  > Why? Requiring operators at the beginning of the line keeps the operators aligned and follows a pattern similar to method chaining. This also improves readability by making it easier to visually follow complex logic.

```
// bad
if ((foo === 123 || bar === 'abc') && doesItLookGoodWhenItBecomesThatLong() && isThisReallyHappening()) {
  thing1();
}

// bad
if (foo === 123 &&
  bar === 'abc') {
  thing1();
}

// bad
if (foo === 123
  && bar === 'abc') {
  thing1();
}

// bad
if (
  foo === 123 &&
  bar === 'abc'
) {
  thing1();
}

// good
if (
  foo === 123
  && bar === 'abc'
) {
  thing1();
}

// good
if (
  (foo === 123 || bar === 'abc')
  && doesItLookGoodWhenItBecomesThatLong()
  && isThisReallyHappening()
) {
  thing1();
}

// good
if (foo === 123 && bar === 'abc') {
  thing1();
}
```

- 17.2 Don't use selection operators in place of control statements.

```
// bad
!isRunning && startRunning();

// good
if (!isRunning) {
  startRunning();
}
```

⬆️ back to top

Comments

- 18.1 Use `/** ... */` for multi-line comments.

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}
```

- 18.2 Use `//` for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment unless it's on the first line of a block.

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}

// also good
function getType() {
  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}
```

- 18.3 Start all comments with a space to make it easier to read. eslint: `spaced-comment`

```
// bad
//is current tab
const active = true;

// good
// is current tab
const active = true;

// bad
/**
 *make() returns a new element
 *based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}
```

- 18.4 Prefixing your comments with `FIXME` or `TODO` helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to the problem that needs to be implemented. These are different than regular comments because they are actionable. The actions are `FIXME: -- need to figure this out` or `TODO: -- need to implement`.

- 18.5 Use `// FIXME:` to annotate problems.

```
class Calculator extends Abacus {
  constructor() {
    super();

    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

- 18.6 Use `// TODO:` to annotate solutions to problems.

```
class Calculator extends Abacus {
  constructor() {
    super();

    // TODO: total should be configurable by an options param
    this.total = 0;
  }
}
```

⬆️ back to top

Whitespace

- 19.1 Use soft tabs (space character) set to 2 spaces. eslint: `indent` jscs: `validateIndentation`

```
// bad
function foo() {
····let name;
}

// bad
function bar() {
·let name;
}

// good
function baz() {
··let name;
}
```

- 19.2 Place 1 space before the leading brace. eslint: `space-before-blocks` jscs: `requireSpaceBeforeBlockStatements`

```
// bad
function test(){
  console.log('test');
}

// good
function test() {
  console.log('test');
}

// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});

// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});
```

- 19.3 Place 1 space before the opening parenthesis in control statements (`if` , `while` etc.). Place no space between the argument list and the function name in function calls and declarations. eslint: `keyword-spacing` jscs: `requireSpaceAfterKeywords`

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

// good
function fight() {
```

```
  console.log('Swooosh!');
}
```

- 19.4 Set off operators with spaces. eslint: `space-infix-ops` jscs: `requireSpaceBeforeBinaryOperators`, `requireSpaceAfterBinaryOperators`

```
// bad
const x=y+5;

// good
const x = y + 5;
```

- 19.5 End files with a single newline character. eslint: `eol-last`

```
// bad
import { es6 } from './AirbnbStyleGuide';
  // ...
export default es6;
```

```
// bad
import { es6 } from './AirbnbStyleGuide';
  // ...
export default es6;↵
↵
```

```
// good
import { es6 } from './AirbnbStyleGuide';
  // ...
export default es6;↵
```

- 19.6 Use indentation when making long method chains (more than 2 method chains). Use a leading dot, which emphasizes that the line is a method call, not a new statement. eslint: `newline-per-chained-call` `no-whitespace-before-property`

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// bad
$('#items').
  find('.selected').
    highlight().
    end().
  find('.open').
    updateCount();

// good
$('#items')
  .find('.selected')
    .highlight()
    .end()
  .find('.open')
    .updateCount();

// bad
const leds = stage.selectAll('.led').data(data).enter().append('svg:svg').classed('led', true)
    .attr('width', (radius + margin) * 2).append('svg:g')
    .attr('transform', `translate(${radius + margin},${radius + margin})`)
    .call(tron.led);

// good
const leds = stage.selectAll('.led')
    .data(data)
```

```
    .enter().append('svg:svg')
      .classed('led', true)
      .attr('width', (radius + margin) * 2)
    .append('svg:g')
      .attr('transform', `translate(${radius + margin},${radius + margin})`)
      .call(tron.led);

// good
const leds = stage.selectAll('.led').data(data);
```

- 19.7 Leave a blank line after blocks and before the next statement. jscs: `requirePaddingNewLinesAfterBlocks`

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;

// bad
const obj = {
  foo() {
  },
  bar() {
  },
};
return obj;

// good
const obj = {
  foo() {
  },

  bar() {
  },
};

return obj;

// bad
const arr = [
  function foo() {
  },
  function bar() {
  },
];
return arr;

// good
const arr = [
  function foo() {
  },

  function bar() {
  },
];

return arr;
```

- 19.8 Do not pad your blocks with blank lines. eslint: `padded-blocks` jscs: `disallowPaddingNewlinesInBlocks`

```
// bad
function bar() {

  console.log(foo);

}

// bad
if (baz) {

  console.log(qux);
} else {
  console.log(foo);

}

// bad
class Foo {

  constructor(bar) {
    this.bar = bar;
  }
}

// good
function bar() {
  console.log(foo);
}

// good
if (baz) {
  console.log(qux);
} else {
  console.log(foo);
}
```

- 19.9 Do not add spaces inside parentheses. eslint: `space-in-parens` jscs: `disallowSpacesInsideParentheses`

```
// bad
function bar( foo ) {
  return foo;
}

// good
function bar(foo) {
  return foo;
}

// bad
if ( foo ) {
  console.log(foo);
}

// good
if (foo) {
  console.log(foo);
}
```

- 19.10 Do not add spaces inside brackets. eslint: `array-bracket-spacing` jscs: `disallowSpacesInsideArrayBrackets`

```
// bad
const foo = [ 1, 2, 3 ];
console.log(foo[ 0 ]);

// good
```

```
const foo = [1, 2, 3];
console.log(foo[0]);
```

- 19.11 Add spaces inside curly braces. eslint: `object-curly-spacing` jscs: `requireSpacesInsideObjectBrackets`

```
// bad
const foo = {clark: 'kent'};

// good
const foo = { clark: 'kent' };
```

- 19.12 Avoid having lines of code that are longer than 100 characters (including whitespace). Note: per above, long strings are exempt from this rule, and should not be broken up. eslint: `max-len` jscs: `maximumLineLength`

> Why? This ensures readability and maintainability.

```
// bad
const foo = jsonData && jsonData.foo && jsonData.foo.bar && jsonData.foo.bar.baz && jsonData.foo.bar.baz.quux &&

// bad
$.ajax({ method: 'POST', url: 'https://airbnb.com/', data: { name: 'John' } }).done(() => console.log('Congratula

// good
const foo = jsonData
  && jsonData.foo
  && jsonData.foo.bar
  && jsonData.foo.bar.baz
  && jsonData.foo.bar.baz.quux
  && jsonData.foo.bar.baz.quux.xyzzy;

// good
$.ajax({
  method: 'POST',
  url: 'https://airbnb.com/',
  data: { name: 'John' },
})
  .done(() => console.log('Congratulations!'))
  .fail(() => console.log('You have failed this city.'));
```

⬆️ back to top

Commas

- 20.1 Leading commas: **Nope.** eslint: `comma-style` jscs: `requireCommaBeforeLineBreak`

```
// bad
const story = [
    once
  , upon
  , aTime
];

// good
const story = [
  once,
  upon,
  aTime,
];

// bad
const hero = {
    firstName: 'Ada'
  , lastName: 'Lovelace'
```

```
  , birthYear: 1815
  , superPower: 'computers'
};

// good
const hero = {
  firstName: 'Ada',
  lastName: 'Lovelace',
  birthYear: 1815,
  superPower: 'computers',
};
```

- 20.2 Additional trailing comma: **Yup.** eslint: `comma-dangle` jscs: `requireTrailingComma`

> Why? This leads to cleaner git diffs. Also, transpilers like Babel will remove the additional trailing comma in the transpiled code which means you don't have to worry about the trailing comma problem in legacy browsers.

```
// bad - git diff without trailing comma
const hero = {
     firstName: 'Florence',
-    lastName: 'Nightingale'
+    lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing']
};

// good - git diff with trailing comma
const hero = {
     firstName: 'Florence',
     lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing'],
};



// bad
const hero = {
  firstName: 'Dana',
  lastName: 'Scully'
};

const heroes = [
  'Batman',
  'Superman'
];

// good
const hero = {
  firstName: 'Dana',
  lastName: 'Scully',
};

const heroes = [
  'Batman',
  'Superman',
];

// bad
function createHero(
  firstName,
  lastName,
  inventorOf
) {
  // does nothing
}

// good
function createHero(
  firstName,
  lastName,
```

```
  inventorOf,
) {
  // does nothing
}

// good (note that a comma must not appear after a "rest" element)
function createHero(
  firstName,
  lastName,
  inventorOf,
  ...heroArgs
) {
  // does nothing
}

// bad
createHero(
  firstName,
  lastName,
  inventorOf
);

// good
createHero(
  firstName,
  lastName,
  inventorOf,
);

// good (note that a comma must not appear after a "rest" element)
createHero(
  firstName,
  lastName,
  inventorOf,
  ...heroArgs
);
```

⬆️ back to top

Semicolons

- 21.1 **Yup.** eslint: `semi` jscs: `requireSemicolons`

> Why? When JavaScript encounters a line break without a semicolon, it uses a set of rules called Automatic Semicolon Insertion to determine whether or not it should regard that line break as the end of a statement, and (as the name implies) place a semicolon into your code before the line break if it thinks so. ASI contains a few eccentric behaviors, though, and your code will break if JavaScript misinterprets your line break. These rules will become more complicated as new features become a part of JavaScript. Explicitly terminating your statements and configuring your linter to catch missing semicolons will help prevent you from encountering issues.

```
// bad - raises exception
const luke = {}
const leia = {}
[luke, leia].forEach(jedi => jedi.father = 'vader')

// bad - raises exception
const reaction = "No! That's impossible!"
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}())

// bad - returns `undefined` instead of the value on the next line - always happens when `return` is on a line by
function foo() {
  return
    'search your feelings, you know it to be foo'
```

```
}

// good
const luke = {};
const leia = {};
[luke, leia].forEach((jedi) => {
  jedi.father = 'vader';
});

// good
const reaction = "No! That's impossible!";
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}());

// good
function foo() {
  return 'search your feelings, you know it to be foo';
}
```

Read more.

⬆️ back to top

Type Casting & Coercion

- 22.1 Perform type coercion at the beginning of the statement.
- 22.2 Strings: eslint: `no-new-wrappers`

```
// => this.reviewScore = 9;

// bad
const totalScore = new String(this.reviewScore); // typeof totalScore is "object" not "string"

// bad
const totalScore = this.reviewScore + ''; // invokes this.reviewScore.valueOf()

// bad
const totalScore = this.reviewScore.toString(); // isn’t guaranteed to return a string

// good
const totalScore = String(this.reviewScore);
```

- 22.3 Numbers: Use `Number` for type casting and `parseInt` always with a radix for parsing strings. eslint: `radix` `no-new-wrappers`

```
const inputValue = '4';

// bad
const val = new Number(inputValue);

// bad
const val = +inputValue;

// bad
const val = inputValue >> 0;

// bad
const val = parseInt(inputValue);

// good
const val = Number(inputValue);
```

```
// good
const val = parseInt(inputValue, 10);
```

- 22.4 If for whatever reason you are doing something wild and `parseInt` is your bottleneck and need to use Bitshift for performance reasons, leave a comment explaining why and what you're doing.

```
// good
/**
 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 */
const val = inputValue >> 0;
```

- 22.5 **Note:** Be careful when using bitshift operations. Numbers are represented as 64-bit values, but bitshift operations always return a 32-bit integer (source). Bitshift can lead to unexpected behavior for integer values larger than 32 bits. Discussion. Largest signed 32-bit Int is 2,147,483,647:

```
2147483647 >> 0; // => 2147483647
2147483648 >> 0; // => -2147483648
2147483649 >> 0; // => -2147483647
```

- 22.6 Booleans: eslint: `no-new-wrappers`

```
const age = 0;

// bad
const hasAge = new Boolean(age);

// good
const hasAge = Boolean(age);

// best
const hasAge = !!age;
```

⬆️ back to top

Naming Conventions

- 23.1 Avoid single letter names. Be descriptive with your naming. eslint: `id-length`

```
// bad
function q() {
  // ...
}

// good
function query() {
  // ...
}
```

- 23.2 Use camelCase when naming objects, functions, and instances. eslint: `camelcase` jscs: `requireCamelCaseOrUpperCaseIdentifiers`

```
// bad
const OBJEcttssss = {};
const this_is_my_object = {};
function c() {}
```

```
// good
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 23.3 Use PascalCase only when naming constructors or classes. eslint: `new-cap` jscs: `requireCapitalizedConstructors`

```
// bad
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}

const good = new User({
  name: 'yup',
});
```

- 23.4 Do not use trailing or leading underscores. eslint: `no-underscore-dangle` jscs: `disallowDanglingUnderscores`

> Why? JavaScript does not have the concept of privacy in terms of properties or methods. Although a leading underscore is a common convention to mean “private”, in fact, these properties are fully public, and as such, are part of your public API contract. This convention might lead developers to wrongly think that a change won’t count as breaking, or that tests aren’t needed. tl;dr: if you want something to be “private”, it must not be observably present.

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';
this._firstName = 'Panda';

// good
this.firstName = 'Panda';

// good, in environments where WeakMaps are available
// see https://kangax.github.io/compat-table/es6/#test-WeakMap
const firstNames = new WeakMap();
firstNames.set(this, 'Panda');
```

- 23.5 Don’t save references to `this` . Use arrow functions or Function#bind. jscs: `disallowNodeTypes`

```
// bad
function foo() {
  const self = this;
  return function () {
    console.log(self);
  };
}

// bad
function foo() {
  const that = this;
  return function () {
    console.log(that);
  };
}
```

```
// good
function foo() {
  return () => {
    console.log(this);
  };
}
```

- 23.6 A base filename should exactly match the name of its default export.

```
// file 1 contents
class CheckBox {
  // ...
}
export default CheckBox;

// file 2 contents
export default function fortyTwo() { return 42; }

// file 3 contents
export default function insideDirectory() {}

// in some other file
// bad
import CheckBox from './checkBox'; // PascalCase import/export, camelCase filename
import FortyTwo from './FortyTwo'; // PascalCase import/filename, camelCase export
import InsideDirectory from './InsideDirectory'; // PascalCase import/filename, camelCase export

// bad
import CheckBox from './check_box'; // PascalCase import/export, snake_case filename
import forty_two from './forty_two'; // snake_case import/filename, camelCase export
import inside_directory from './inside_directory'; // snake_case import, camelCase export
import index from './inside_directory/index'; // requiring the index file explicitly
import insideDirectory from './insideDirectory/index'; // requiring the index file explicitly

// good
import CheckBox from './CheckBox'; // PascalCase export/import/filename
import fortyTwo from './fortyTwo'; // camelCase export/import/filename
import insideDirectory from './insideDirectory'; // camelCase export/import/directory name/implicit "index"
// ^ supports both insideDirectory.js and insideDirectory/index.js
```

- 23.7 Use camelCase when you export-default a function. Your filename should be identical to your function's name.

```
function makeStyleGuide() {
  // ...
}

export default makeStyleGuide;
```

- 23.8 Use PascalCase when you export a constructor / class / singleton / function library / bare object.

```
const AirbnbStyleGuide = {
  es6: {
  },
};

export default AirbnbStyleGuide;
```

- 23.9 Acronyms and initialisms should always be all capitalized, or all lowercased.

> Why? Names are for readability, not to appease a computer algorithm.

```
// bad
import SmsContainer from './containers/SmsContainer';
```

```
// bad
const HttpRequests = [
  // ...
];

// good
import SMSContainer from './containers/SMSContainer';

// good
const HTTPRequests = [
  // ...
];

// also good
const httpRequests = [
  // ...
];

// best
import TextMessageContainer from './containers/TextMessageContainer';

// best
const requests = [
  // ...
];
```

⬆️ back to top

Accessors

- 24.1 Accessor functions for properties are not required.
- 24.2 Do not use JavaScript getters/setters as they cause unexpected side effects and are harder to test, maintain, and reason about. Instead, if you do make accessor functions, use getVal() and setVal('hello').

```
// bad
class Dragon {
  get age() {
    // ...
  }

  set age(value) {
    // ...
  }
}

// good
class Dragon {
  getAge() {
    // ...
  }

  setAge(value) {
    // ...
  }
}
```

- 24.3 If the property/method is a `boolean`, use `isVal()` or `hasVal()`.

```
// bad
if (!dragon.age()) {
  return false;
}
```

```
// good
if (!dragon.hasAge()) {
  return false;
}
```

- 24.4 It's okay to create get() and set() functions, but be consistent.

```
class Jedi {
  constructor(options = {}) {
    const lightsaber = options.lightsaber || 'blue';
    this.set('lightsaber', lightsaber);
  }

  set(key, val) {
    this[key] = val;
  }

  get(key) {
    return this[key];
  }
}
```

⬆️ back to top

Events

- 25.1 When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass an object literal (also known as a "hash") instead of a raw value. This allows a subsequent contributor to add more data to the event payload without finding and updating every handler for the event. For example, instead of:

```
// bad
$(this).trigger('listingUpdated', listing.id);

// ...

$(this).on('listingUpdated', (e, listingID) => {
  // do something with listingID
});
```

prefer:

```
// good
$(this).trigger('listingUpdated', { listingID: listing.id });

// ...

$(this).on('listingUpdated', (e, data) => {
  // do something with data.listingID
});
```

⬆️ back to top

jQuery

- 26.1 Prefix jQuery object variables with a `$`. jscs: `requireDollarBeforejQueryAssignment`

```
// bad
const sidebar = $('.sidebar');

// good
const $sidebar = $('.sidebar');
```

```
// good
const $sidebarBtn = $('.sidebar-btn');
```

- 26.2 Cache jQuery lookups.

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...

  $('.sidebar').css({
    'background-color': 'pink',
  });
}

// good
function setSidebar() {
  const $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...

  $sidebar.css({
    'background-color': 'pink',
  });
}
```

- 26.3 For DOM queries use Cascading `$('.sidebar ul')` or parent > child `$('.sidebar > ul')`. jsPerf
- 26.4 Use `find` with scoped jQuery object queries.

```
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```

⬆️ back to top

ECMAScript 5 Compatibility

- 27.1 Refer to Kangax's ES5 compatibility table.

⬆️ back to top

ECMAScript 6+ (ES 2015+) Styles

- 28.1 This is a collection of links to the various ES6+ features.

1. Arrow Functions
2. Classes

3. Object Shorthand
4. Object Concise
5. Object Computed Properties
6. Template Strings
7. Destructuring
8. Default Parameters
9. Rest
10. Array Spreads
11. Let and Const
12. Exponentiation Operator
13. Iterators and Generators
14. Modules

- 28.2 Do not use TC39 proposals that have not reached stage 3.

> Why? They are not finalized, and they are subject to change or to be withdrawn entirely. We want to use JavaScript, and proposals are not JavaScript yet.

⬆️ back to top

Standard Library

The Standard Library contains utilities that are functionally broken but remain for legacy reasons.

- 29.1 Use `Number.isNaN` instead of global `isNaN`. eslint: `no-restricted-globals`

> Why? The global `isNaN` coerces non-numbers to numbers, returning true for anything that coerces to NaN. If this behavior is desired, make it explicit.

```
// bad
isNaN('1.2'); // false
isNaN('1.2.3'); // true

// good
Number.isNaN('1.2.3'); // false
Number.isNaN(Number('1.2.3')); // true
```

- 29.2 Use `Number.isFinite` instead of global `isFinite`. eslint: `no-restricted-globals`

> Why? The global `isFinite` coerces non-numbers to numbers, returning true for anything that coerces to a finite number. If this behavior is desired, make it explicit.

```
// bad
isFinite('2e3'); // true

// good
Number.isFinite('2e3'); // false
Number.isFinite(parseInt('2e3', 10)); // true
```

⬆️ back to top

Testing

- 30.1 **Yup.**

```
function foo() {
  return true;
```

```
}
```

- 30.2 **No, but seriously**:
  - Whichever testing framework you use, you should be writing tests!
  - Strive to write many small pure functions, and minimize where mutations occur.
  - Be cautious about stubs and mocks - they can make your tests more brittle.
  - We primarily use `mocha` at Airbnb. `tape` is also used occasionally for small, separate modules.
  - 100% test coverage is a good goal to strive for, even if it's not always practical to reach it.
  - Whenever you fix a bug, *write a regression test*. A bug fixed without a regression test is almost certainly going to break again in the future.

⬆️ back to top

Performance

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Are Javascript functions like `map()`, `reduce()`, and `filter()` optimized for traversing arrays?
- Loading...

⬆️ back to top

Resources

Learning ES6+

- Latest ECMA spec
- ExploringJS
- ES6 Compatibility Table
- Comprehensive Overview of ES6 Features

Read This

- Standard ECMA-262

Tools

- Code Style Linters
  - ESlint - Airbnb Style .eslintrc
  - JSHint - Airbnb Style .jshintrc
  - JSCS - Airbnb Style Preset (Deprecated, please use ESlint)
- Neutrino preset - neutrino-preset-airbnb-base

Other Style Guides

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

Other Styles

Getting Started with CSS – Part 1

Guil Hernandez

This is the first of a two-part article on CSS, the most powerful tool we have available to us as web designers and developers. In this article, we'll cover basic CSS concepts to get us started.

What is CSS?

CSS (Cascading Style Sheets) is responsible for describing how HTML elements are displayed. Colors, font sizes and page layout are just a few of the presentational responsibilities of CSS. With CSS we create rules to specify how content should appear on a page. We then associate the rules with HTML elements.

CSS was introduced in HTML 4.0 by the W3C to help solve development and maintenance issues developers were having at the time with HTML 3.2, which included several presentational tags and attributes. Because HTML was never intended to contain presentational tags for formatting pages, such as the `<font>` and `<center>` tags, it made it difficult and impractical to develop and maintain websites.

By separating content from presentation, CSS made HTML more semantic and maintainable. CSS also improved the accessibility of web content by allowing access and adaptability to various devices like desktops, mobile devices, large screens and printers.

Current Status

Currently, CSS 2.1 is the official web standard with parts of the CSS3 spec in the recommendation stage—the stage where a

spec is considered a final standard. The good news is the W3C has split the CSS3 spec as a series of modules being implemented separately and independently from each other. This has enabled portions of the spec to move faster and it encourages browser vendors to implement modules without having to wait for the entire spec to be considered finished.

To follow the development of CSS3 and view all completed specifications and drafts, visit the W3C.

Fundamental Concepts

The Cascade

One of the most fundamental concepts of CSS is the cascade. The cascade determines which style properties are assigned to an element based on declarations that have cascaded down from other sources; it's what allows us to keep styles consistent throughout a website without having to repeat them.

There are 3 main concepts that determine which properties are assigned to an element: **importance**, **specificity** and **source order**.

Importance

Importance depends on the origin of the style sheet. Style sheets can have different sources, such as user agent styles, user styles and author styles.

Style sheet sources.

User Agent (UA) styles are the styles the browser applies by default. Each browser has its own UA style sheet. This is why there are some visual differences when viewing a web page on different browsers.

User styles are styles specified by the user. Users can have special styles predefined that are different from the browser's default styles. These are oftentimes used for accessibility—for users with special needs who, for example, need larger font sizes, specific colors or contrast.

Author styles are styles specified by an author—more likely the site's designer. By default, rules in author style sheets have more weight than user agent and user style sheets.

Specificity

Specificity determines which CSS rule is applied by the browser. Once the cascade decides the importance of a rule, it assigns a specificity to it. If two rules select the same element and the properties contradict one another, the rule with the more specific selector takes precedence. Once you're familiar with how

specificity works it can save you time customizing and troubleshooting your CSS.

Source Order

Source Order is the order in which styles are linked, included or imported. The order is important because the cascade assigns a priority to a rule based on what order they appear. If there are two linked style sheets in the `<head>` section, the last one will take precedence over the first. Or, if two declarations affect the same element, have the same importance and the same specificity, the declaration that appears later in the style sheet will be applied.

```
<head>
    <title>My Website</title>
    <link rel="stylesheet" href="style.css">
  <link rel="stylesheet" href="more-styles.css">
</head>
```

The last linked style sheet takes precedence.

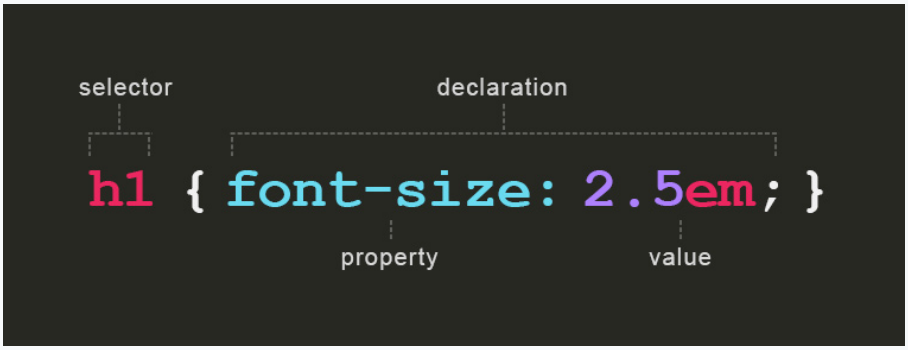
After importance, specificity and source order have been determined, each rule is assigned a weight that determines which property will be applied to an element.

Inheritance

Inheritance means that the value of a property is copied from the value of the parent element. So when a property is said to be inherited, it simply means that the value will be determined by that inheritance unless another property value is specified.

For example, if the `body` element of a page is set to a specific font, that font will be inherited by other elements, such as headings and paragraphs. Not all properties are inherited, but we can force them to be by using the `inherit` value.

The CSS Rule



A simple CSS Rule

CSS rules tell browsers how to render elements. A rule (or rule set) is comprised of two main parts: the selector followed by one or more declarations. The rules then apply the declarations listed to all elements matched by the selector.

Selector

A selector is the part of the CSS rule that matches HTML elements. Selectors can contain one or more simple selectors, such as type, class or ID selectors separated by combinators, which explain the relationship between the selectors. Examples of combinators are whitespace, greater-than sign (>), plus sign (+) and tilde (~).

Declaration

The declaration is the part that sits inside the curly braces. It consists of a property name followed by a colon, then a property value. Multiple declarations can be made by organizing them into semicolon separated groups.

Properties

Properties are a set of parameters that define the rendering of a document; they indicate the part of the element we want to

change. Each property is followed by a value, which specifies the settings we want to use for a property.

In part 2, we'll cover ways to add CSS to a page, the CSS box model, selectors, and more!

code | css | css3 | make a website

2 Responses to "Getting Started with CSS – Part 1"

iCann on June 28, 2017 at 4:29 am said:
Hi there! Do you know if they make any plugins to help with SEO?
I'm trying to get my blog to rank for some targeted
keywords but I'm not seeing very good gains. If you know of any please share.
Appreciate it!

Reply

Startledpumkin on June 27, 2013 at 8:02 pm said:
Such a good lesson Guil. Thanks so much!

Reply

Leave a Reply

Learn

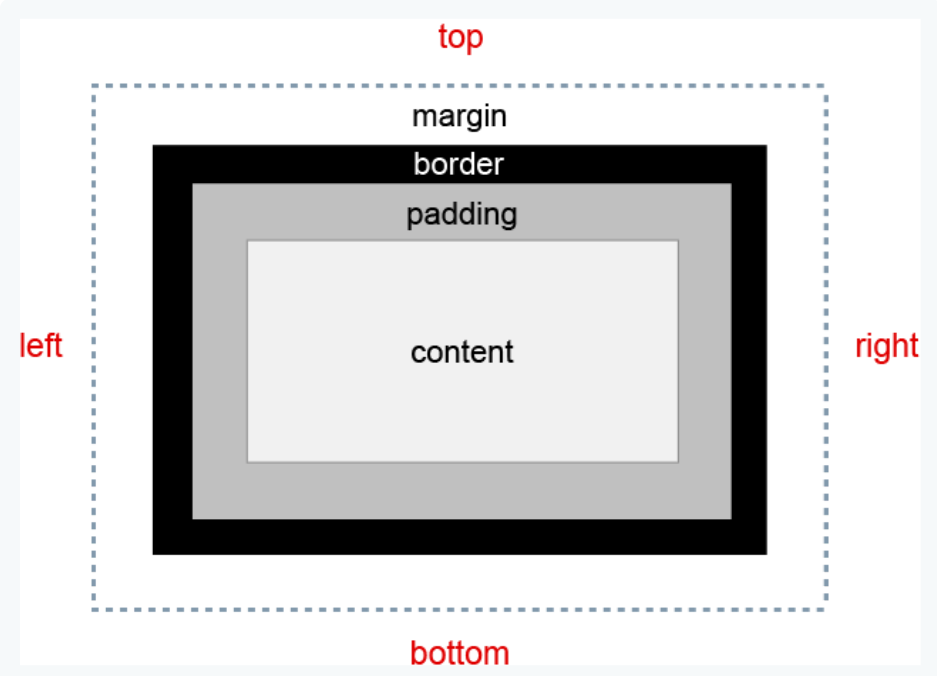
Getting Started with CSS – Part 2

Guil Hernandez

In part 1 we covered basic CSS concepts like the cascade and CSS rules. In this article, we'll take things a step further with the CSS box model and basic selector types.

The CSS Box Model

Understanding how the box model works is the very basis of designing with CSS. Every HTML element, no matter how big or small, can be thought of as a rectangular box made up of content, padding, borders and margins. The box model applies to all HTML elements – it's essentially a box that wraps the elements. So first, we'll need to think of every HTML element as a rectangle or box.



The CSS Box Model example.

The innermost area — and core component — is the **content area**, which is the area containing the element's actual content, such as text or images.

The **padding area** surrounds the content area; it separates the content from the border area above.

The **border area** of the box is the outermost part of the box — we can think of it as an outline to the box. Although borders are optional, different styles such as color and thickness can be applied to them.

Finally, the **margin area** exists outside of the box; it's the space around an element that separates it from other elements. So just remember that with margins we create space around the elements.

Adding CSS to a Page

We can add CSS to a page several different ways, and as we'll learn, each has its advantages or disadvantages. Let's review the different methods we can use.

Inline Styles

One of the ways we can add styles to a page is with inline styles. Inline styles mix content with presentation. To add them, we use the style attribute inside the HTML element we want to style.

```
<body style="background: lightblue;">
```

Inline styles are at the lowest level possible of the cascade, so that means they will override any styles declared in external or internal style sheets. They should be used sparingly because if we need to make any changes it means we'll need to go directly to our markup, so we're not really separating content from presentation — making maintenance difficult and impractical. Inline styles might be good for creating quick HTML/CSS

mockups and prototypes, or for temporary styles not meant to be shared with other elements on the page.

Internal Styles

Internal styles are included in the `<head>` section of the document and are defined using the `<style>` tag. With this method we can write all of our CSS rules within the HTML document.

```
<style>
  h1 {
    color: white;
    background: black;
  }
</style>
```

Internal styles can be useful for temporary use, testing a new feature or small iteration.

The downside to using internal styles on a larger website is that since the styles are on the same page they are required to be downloaded each time the page is loaded. It also means that we may need to duplicate a lot of the same styles across multiple pages.

External Style Sheet

With an external style sheet we can change the look of an entire website with just one file, then reference it in the `<head>` section of the document using the `<link>` element.

```
<link rel="stylesheet" type="text/css" href="style.css">
```

In our example above, the `rel` attribute specifies the relationship between the current document and the linked document, which in this case is a style sheet. The `type` attribute specifies the

media type of the linked document. However, in HTML5, `type` is no longer required, so we can omit it if we prefer. Finally, the `href` attribute points to the location of the CSS file.

@import Method

The `@import` rule can be used in external or internal style sheets. This method allows us to import CSS from other files, so it shares some of the same advantages as linking a style sheet, such as browser caching and maintenance efficiency. When using this method, the `@import` statement must precede all other declarations in order for it work properly.

This method can make our CSS more modular by splitting it into different files. The page can link to one style sheet which can import other style sheets for reset, layout, typography, etc…

```
@import 'reset.css';
@import 'layout.css';
@import 'typography.css';
```

The drawback to using this method is performance. Each `@import` statement is also a new HTTP request.

More on Selectors

Selectors are one of the most important and powerful parts of CSS. They are essentially patterns used to allow us to select the elements we want to style based on their type, attributes, or even a position in the document. We'll cover four basic selector types to get us started.

Type Selectors

A type selector selects an element type in the document with the corresponding name. For example, we can add a background color to our page by selecting the `body` and adding a background property.

```
body {
  background: lightblue;
}
```

Descendant Selectors

We can combine type selectors to create what are called descendant selectors. A descendant selector selects an element that is a descendant of another; it's made up of two or more selectors separated by whitespace.

```
h2 em {
  color: green;
}
```

Class Selectors

A class selector selects an element based on its class attribute. Classes are specified with the dot (.) character followed by the class name. Classes are not unique, so multiple elements on a page can have the same class applied to them, and an element can also have multiple classes in its class attribute.

```
.intro {
  font-weight: bold;
}
```

ID Selectors

An ID selector selects an element based on its ID attribute using the pound (#) symbol. IDs are unique, so an element can only have one ID and a page can only have one element with that ID. We can apply both a class and an ID to an element, just remember that IDs carry more specificity than classes do.

```
#wrapper {
  width: 90%;
  margin: 0 auto;
}
```

IDs also have browser functionality. They can be used as fragment identifiers for creating landmarks in pages. Let's say we give the `h1` at the top of our page an ID of "top", then add a link to the bottom of the page with an href attribute of "top" preceded by a "#":

```
<h1 id="top">My Website Heading</h1>
...
<a href="#top">Back to top</a>
```

When the link is clicked, the browser will locate the element with the ID of "top".

These basic selectors are just some of the many ways we can target elements. In future articles, we'll learn how we can get more complex and specific with our CSS selectors.

code | css | css3 | make a website

Leave a Reply

Name *

Email Address (will not be published) *

Website


Jason Arnold Follow
May 6, 2017 · 5 min read

Getting started with SASS

The first questions I asked myself when starting to use SASS were what is it and why should I use it? I try to not change my workflow whenever I hear about a new language or technology because if I'm constantly changing things around then I'm fighting that workflow more than I am building new things.

Having said that, however, I do also think it is a good idea to check out something new from time to time to keep your skills fresh and to make sure you are using the best tools for you.

What is SASS?

First off, it might be a good idea to explain what SASS is and what it does. The browser can only read CSS and those CSS files are the ones linked in the HTML files. So that part doesn't change. What does change is that you no longer write CSS directly. You write SASS and it is then preprocessed into a CSS file. That file looks like just any CSS file, but it is just created on-the-fly from the SASS file.

SASS looks very similar to CSS but does have some differences. There are no curly braces `{}` or semi-colons used so there are less syntactic errors to worry about. Also, SASS uses indentation and nesting so you no longer have descendent selectors that dig several layers deep into a site.

Why SASS?

I've heard about SASS for a while now but hadn't yet taken the time to get to know exactly what it is or how to use it. When learning a new language, I usually start at the language's website to get a feel for it. Sass's website calls it 'CSS with superpowers' and gives this definition:

> *Sass is the most mature, stable, and powerful professional grade CSS extension language in the world.*

Sounds cool. But why should I use it over vanilla CSS?

The reasons given on the site are that SASS adds features to CSS that help to keep it organized and more flexible. One of the great things about CSS is that there is really no method that one must follow when using it. If you know the basics of the language you can write the CSS you need. It gives you a lot of freedom. This can also quickly become a bad thing since your CSS files can get out of control. Even in modestly sized projects, CSS files can get very big, disorganized, and hard to maintain. SASS's features are here to help with these problems.

I just started digging into SASS but two of the features that I'm impressed with so far are variables and nesting.

Variables in SASS are similar to variables in any other programming language. You define one to hold a piece of information you want to use throughout your SASS file. The first thing I think of here is keeping track of colors on a site. If I'm reusing text, background, or border colors, I don't want to have to keep referencing a list to keep track of which one is which. Or If I need to change one of those colors, I then have to find everywhere in the CSS file I've used it. With a SASS variable I set it one place and that is it.

Nesting in SASS is another feature that helps with keeping a file more organized and easier to read. If you have a complex site and need to set a CSS property on a deeply nested element, then you'll probably have

some pretty nasty descendant selectors to get to those elements. Nesting does away with this. Instead of descendent selectors, SASS uses indentation to determine selector hierarchy. This makes for much cleaner, more maintainable code.

Other SASS features include things like partials and imports which allow you to make your CSS more modular and break larger files down to smaller sizes. Also SASS mixins allow you to create CSS declarations that you can use throughout your code.

So, SASS offers quite a few features that vanilla CSS doesn't. So, if this has convinced you to give SASS a try, then let's get started adding it to a project.

Installing and Configuring SASS

You can install SASS through the command line just like most other tools. SASS is installed through a Ruby gem.

```
gem install sass
```

After SASS is installed, you have to tell it where the SASS files are located and then where the CSS files should go when compiled. The syntax for this is pretty simple, you call the `sass` command and then provide an input file and an output file.

```
sass style.sass:style.css
```

Another option for compiling is to add a 'watch' flag so that SASS watches a directory or an individual file and whenever that file changes, the CSS file is automatically compiled.

```
sass --watch style.sass:style.css
```

When you watch files, the terminal will pause until you change the SASS file. When you do you'll see either of these messages depending on if you are using `sass` or `node-sass`

```
// ♥ sass --watch style.sass:style.css
>>> Sass is watching for changes. Press Ctrl-C to stop.
>>> Change detected to: style.sass
      write style.css
      write style.css.map
```

Watch message in sass

SASS in action

Now that you know how to get SASS set up and running the rest is pretty easy. Just create a `.sass` file in your project directory and write your SASS there. When you run the processor (or the file is being watched for changes) the code in that file will be changed to vanilla CSS.

Here is an example I wrote that creates a really basic navbar with a logo on the left and three links along the length of the rest of the page.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="style.css">
    <title>SASS</title>
</head>
<body>
    <div id="navbar">
        <div id="logo"></div>
        <div id="links">
            <ul>
                <li>About</li>
                <li>Blog</li>
                <li>Contact</li>
            </ul>
        </div>
    </div>
</body>
</html>
```

The HTML for the example. Note that the <link> tag is pointing to the usual .css file.

```sass
$logoColor: tomato
$divBorder: black 2px solid

#navbar
    display: flex
    flex-direction: row

    #logo
        background: $logoColor
        height: 100px
        width: 100px
        border-radius: 50%
        border: $divBorder

    #links
        width: 100%

        ul
            display: flex
            flex-direction: row
            justify-content: space-around

            li
                list-style: none
```

SASS code

I created several nested sections for this page so you could see that part of SASS being used. I also included a couple of variables at the top of the file. Here is what the CSS looks like after all of this.

```
#navbar {
  display: flex;
  flex-direction: row; }
  #navbar #logo {
    background: tomato;
    height: 100px;
    width: 100px;
    border-radius: 50%;
    border: black 2px solid; }
  #navbar #links {
    width: 100%; }
    #navbar #links ul {
      display: flex;
      flex-direction: row;
      justify-content: space-around; }
      #navbar #links ul li {
        list-style: none; }
```

I've just started using SASS, but I already see some of the great things that it can provide and how it can help my workflow in future projects. I hope that this post has done the same for you. Please leave any questions or comments below. Thanks!

If you want to see the code for this post, you can find it here.

Note for “Medium’s CSS is actually pretty f***ing good”

This document refers to LESS, an alternate CSS preprocessor that is similar to SASS. This document is also one of the best I have found on managing a big CSS project using a preprocessor.

There are two relevant differences between SASS and LESS:

1. Variables in LESS are declared starting with an “@” while variables in SASS start with a “$”, so the variable “@zIndex-1” in this article would be “$zIndex-1” in SASS

2. Mixins are declared fairly differently. In LESS, the code for a vendor prefix mixin looks like this:

```
.user-page-avatar {
  .transition(width .2s ease-in);
}

.transition(@transition) {
  -webkit-transition: @transition;
     -moz-transition: @transition;
      -ms-transition: @transition;
       -o-transition: @transition;
          transition: @transition;
}
```

In SASS, it looks like this:

```
.user-page-avatar {
  transition(width .2s ease-in);
}

@mixin transition($transition) {
  -webkit-transition: $transition;
     -moz-transition: $transition;
      -ms-transition: $transition;
       -o-transition: $transition;
          transition: $transition;
}
```

fat Follow
computer loser / captioned co-founder (captioned.com)
Aug 28, 2014 · 9 min read

Medium's CSS is actually pretty f***ing good.

Listen to the audio version above? 🔈 😱

> *I always believe that to be the best, you have to smell like the best, dress like the best, act like the best. When you throw your trash in the garbage can, it has to be better than anybody else who ever threw trash in the garbage can.*
>
> *— Lil Wayne*

I've been meaning to write something about the CSS at Medium for a while because I'm not completely ashamed of it…

So how did that happen? What did we do differently? OMG, how can you do the same thing? Or learn from us, or something?

What follows are some notes on our CSS, the steps we've taken to get it to where it is, and the world we live in today.

The best at drawing pictures of myself

. . .

In the beginning (some history)

Roughly 2 years ago I joined Obvious Corp. to work on the software application, *medium.com* (which, you're ~hopefully~ reading on).

At the time, Medium had already gone through a series of "re-styles" (*A re-style is where designers ruin your life by coming up with something prettier than they had previously come up with, resulting in you having to rewrite a bunch of CSS/LESS/SASS/etc*). Because of these re-styles, there was a lot of built up cruft – the company was leaning pretty heavily on LESS's language features, with page driven semantics, non-sprited/non-retina image assets, and other such things.

I dug through git and managed to pull out our old profile page implementation from back in 2012 ~shudder~. Check it out below and notice some of these unfortunate patterns:

- everything nested under a .**profile-page** class, with zero reusable components (also ~almost~ everything is prefixed with **.profile**)
- super generic variable names like **@link-color** which weren't scoped to anything, but presumably only to be used in **profile-page.less**

- deep nesting (**.profile-page .profile-posts .home-post-summary .home-post-collection-image img** is an actual selector being generated below—super rough on perf)
- no image spriting
- no z-index scale, no font scale, no color scale, no scalesss…

```
// Copyright 2012 The Obvious Corporation.

@column-margin: 9px;
@link-color: rgb(78, 78, 78);
@link-hover-color: rgb(160, 160, 160);

.profile-page {
  margin: 0 0 0 55px;
  width: auto;
  font-family: helvetica, arial, sans-serif;
  line-height: 1.2em;

  .profile-left {
    width: 925px;
    float: left;

    h1 {
      height: 55px;
      line-height: 55px;
      text-transform: uppercase;
      font-size: 15px;
      font-weight: 700;
      padding-left: 30px;
    }

    .profile-details {
      background-color: rgb(47, 47, 47);
      padding: 47px 37px;
      position: relative;

      .profile-settings-link {
        position: absolute;
        width: 48px;
        height: 48px;
        right: 37px;
        top: 22px;
        text-indent: 1000px;
        overflow: hidden;
        background: url(/img/placeholder-gray.png) no-repe
      }

```

```
42          .profile-image {
43            float: left;
44            margin-right: 47px;
45
46            img {
47              width: 128px;
48              height: 128px;
49              .border-radius(64px);
50            }
51          }
52
53          .profile-name {
54            font-family: georgia, "Times New Roman", serif;
55            font-size: 32px;
56            font-style: italic;
57            font-weight: 400;
58            color: white;
59            text-shadow: 0 1px 0 rgba(0, 0, 0, 0.6);
60            text-transform: capitalize;
61          }
62
63          .profile-bio {
64            margin-left: 175px; // = 128 + 47
65            font-size: 16px;
66            line-height: 1.2em;
67            font-weight: 400;
68            color: rgb(224, 224, 224);
69          }
70        }
71
72        .profile-stats {
73          .clearfix;
74          height: 66px;
75          line-height: 66px;
76          padding-left: 28px;
77          border: 1px solid rgb(211, 211, 211);
78          border-top: none;
79          background: rgb(245, 245, 245);
80
81          .profile-stats-label {
82            float: left;
83            height: 66px;
```

```
84          line-height: 66px;
85          text-transform: uppercase;
86          font-size: 15px;
87          font-weight: 700;
88          color: rgb(77, 77, 77);
89        }
90
91        .profile-stats-item {
92          width: 139px;
93          height: 66px;
94          line-height: 66px;
95          border-left: 1px solid rgb(211, 211, 211);
96          float: right;
97          text-align: center;
98          font-family: georgia, "Times New Roman", serif;
99          font-size: 14px;
100         font-weight: 700;
101         color: rgb(153, 153, 153);
102         background: transparent url(/img/placeholder-gray.
103         padding-top: 16px;
104         .box-sizing(border-box);
105       }
106     }
107
108     .profile-posts {
109       border: 1px solid rgb(211, 211, 211);
110       border-top: none;
111       background: white;
```

https://gist.github.com/fat/a4af78882d0003d2345e

. . .

The 1st Project: OMG Images

At the time, I had been working a lot on library code (Bootstrap, Ratchet, etc.) and sweating all the details, trying to write the best CSS humanly possible.

Medium's CSS was clearly different. Not a neutral difference, but a soul crushing, shitty difference. And I wanted to fix that.

Looking at all there was to do, the first thing I tackled was images. I remember being pretty appalled that we weren't doing any sort of spriting in 2012… accumulating hundreds of image assets like placeholders, arrows, icons, etc. in this crazy /img/ directory which was something like a graveyard for icons.

To get away from that world I did two things—first, I wrote a little CSS utility script called SUS (which we're still using today and I've just open sourced here: https://github.com/medium/sus). This was largely written in IRC with the help of Guillermo Rauch, and is used to extract images from stylesheets and lazy load them in a separate, data-uri file. Not an original concept, just something simple that could be done to help us get to where we needed to go.

The second thing was Geoff Teehan and I sat down and created Medium's first icon font. At the time, we largely had no idea what we were doing… but with the help of icomoon, a github blog post, and a few long nights, we were able to ship something ~pretty~ good to Medium. This was huge because it meant we could delete like 97% of the img folder, everything looked great on my retina MacBook Pro, and we were requesting significantly fewer resources.

. . .

The 2nd Project: Scales

The next major project for me was the z-index scale. Z-index is one of those things which can get out of hand super easily and has always driven me crazy. I didn't want this to be the same at Medium.

Before this project began, it was very common to see one element with a **z-index: 5**, next to a sibling with a **z-index: 1000000;** and another sibling **z-index: 1000001;** and another **z-index: 99999;**.

The styles for these were spread all across the code base and there was no clear way of figuring out how to sort these.

So I took on the laborious task of manually auditing the entire codebase for z-index values. I then introduced a private scale (private to z-index.less) which could be used for z-indexing components (limiting it to 1-10). And finally, I moved all the actual assignments of this scale internal to z-index.less, so you could see how different components were situated relative to each other (which is actually pretty handy).

Below is the z-index file we are using on medium.com today.

```
// Copyright 2014 A Medium Corporation
//
// z-index.less
// Medium.com's z-index scale. Z-index values should always
// allows us to at a glance determine relative layers of ou
// arrising from arbitrary z-index values. Do not edit the
// scoped z-index values.


// Z-Index Scale (private vars)
// ---------------------------------------------------
@zIndex-1:   100;
@zIndex-2:   200;
@zIndex-3:   300;
@zIndex-4:   400;
@zIndex-5:   500;
@zIndex-6:   600;
@zIndex-7:   700;
@zIndex-8:   800;
@zIndex-9:   900;
@zIndex-10: 1000;


// Z-Index Applications
// ---------------------------------------------------
@zIndex-1--screenForeground:        @zIndex-1;
@zIndex-1--followUpVisibility:      @zIndex-1;
@zIndex-1--prlWelcome:              @zIndex-1;
@zIndex-1--appImageDropdown:        @zIndex-1;
@zIndex-1--surfaceUnder:            @zIndex-1;
@zIndex-1--blockGroup:              @zIndex-1;

@zIndex-2--surfaceOver:             @zIndex-2;
@zIndex-2--imagePickerControl:      @zIndex-2;
@zIndex-2--collectionCardButton:    @zIndex-2;
@zIndex-2--blockGroupButtonGroup:   @zIndex-2;
@zIndex-2--blockGroupFocused:       @zIndex-2;
@zIndex-2--blockGroupOverlay:       @zIndex-2;

@zIndex-3--caption:                 @zIndex-3;
@zIndex-3--blockInsertControl:      @zIndex-3;
```

```
42
43  @zIndex-5--figureOverlay:           @zIndex-5;
44  @zIndex-5--highlightMenu:           @zIndex-5;
45  @zIndex-5--metabar:                 @zIndex-5;
46  @zIndex-5--profileAvatar:           @zIndex-5;
47  @zIndex-5--noteRecommendations:     @zIndex-5;
48  @zIndex-5--collectionLogo:          @zIndex-5;
49
```

https://gist.github.com/fat/1f6da6b3bd0311a1f8a0

Of course, this z-index scale was quickly followed by a color scale (variables for blacks, grays, whites, brand-colors, etc) and a type scale (variables for font-sizes, weights, letter-spacing, line-heights, etc.).

Also, you might notice the variable names have taken on some stricter, semantic value – more on that later…

. . .

The 3rd Project: New Style Guide

Not long after introducing scales to Medium, we started in on a big re-style code named “Cocoon”.

Cocoon deprecated several post templates (Medium originally had image templates and quote templates, not just a single article template) and also introduced post lists instead of post “cards.”

We took this as a chance to step back and write a new style guide for Medium.

This initial effort was shared between myself, Dave Gamache, Dustin Senos, and the talented Chris Erwin.

We took some time to really tighten up our thinking around writing production

CSS/LESS at Medium—the major updates being:

- Limit LESS to variables and mixins (no nesting, guards, extend, etc.)—while these language features ~can~ be powerful, we found that inexperienced LESS developers often got in trouble with them fairly easily. We also just preferred the visual aesthetic of pure CSS (and the consistency it afforded) and wanted to move our codebase as far in that direction as reasonable.
- Classes and IDs are lowercase with words separated by a dash—this is how most of us had been writing CSS at the time with Bootstrap, Skeleton, Ratchet, etc. And we thought it made sense to do the same here. Another reason is it followed the naming conventions set forth by the CSS language itself, i.e. border-radius-top-left, etc.
- Prefer components over page level styles—we wanted our front end to feel like library code, and began to break files like profile.less into smaller more focused files like button.less, dialog.less, tooltip.less, etc.

Here it is in its entirety:

LESS Coding Guidelines

The Obvious Corporation uses LESS as its CSS preprocessor. It's helpful to become familiar with its dynamic behavior before reading these guidelines.

Naming Conventions

Classes and IDs are lowercase with words separated by a dash:

Right:

```
.user-profile {}
.post-header {}
#top-navigation {}
```

Wrong:

```
.userProfile {}
.postheader {}
#top_navigation {}
```

Image file names are lowercase with words separated by a dash:

Right:

```
icon-home.png
```

Wrong:

```
iconHome.png
icon_home.png
iconhome.png
```

Image file names are prefixed with their usage.

Right:

```
icon-home.png
bg-container.jpg
bg-home.jpg
sprite-top-navigation.png
```

Wrong:

```
home-icon.png
container-background.jpg
bg.jpg
top-navigation.png
```

IDs vs. Classes

You should almost never need to use IDs. Broken behavior due to ID collisions are hard to track down and annoying.

Color Units

When implementing feature styles, you should only be using color variables provided by colors.less.

When adding a color variable to colors.less, using RGB and RGBA color units are preferred over hex, named, HSL, or HSLA values.

Right:

```
rgb(50, 50, 50);
rgba(50, 50, 50, 0.2);
```

Wrong:

```
#FFF;
#FFFFFF;
```

```
white;
hsl(120, 100%, 50%);
hsla(120, 100%, 50%, 1);
```

z-index scale

Please use the z-index scale defined in z-index.less. `@z-index-1 - @z-index-9` are provided. @z-index-9 is the `kill -9` of our z-index scale and really should never be used.

Font Weight

With the additional support of web fonts `font-weight` plays a more important role than it once did. Different font weights will render typefaces specifically created for that weight, unlike the old days where `bold` could be just an algorithm to fatten a typeface. Obvious uses the numerical value of `font-weight` to enable the best representation of a typeface. The following table is a guide:

Raw font weights should not be specified. Instead, use the appropriate font mixin:

`.wf-sans-i7, .wf-sans-n7, etc.`

The suffix defines the weight and style:

```
n = normal
i = italic
4 = normal font-weight
7 = bold font-weight
```

Refer to type.less for type size, letter-spacing, and line height. Raw sizes, spaces, and line heights should be avoided outside of type.less.

```
ex:

@type-micro
@type-smallest
```

```
@type-smaller
@type-small
@type-base
@type-large
@type-larger
@type-largest
@type-jumbo
```

See Mozilla Developer Network — font-weight for further reading.

Componentizing

Always look to abstract components. Medium has a very strong, very consistent style and the reuse of components across designs helps to improve this consistency at an implementation level.

A name like `.homepage-nav` limits its use. Instead think about writing styles in such a way that they can be reused in other parts of the app. Instead of `.homepage-nav`, try instead `.nav` or `.nav-bar`. Ask yourself if this component could be reused in another context (chances are it could!).

Components should belong to their own less file. For example, all general button definitions should belong in buttons.less.

Name-spacing

Name-spacing is great! But it should be done at a component level – never at a page level.

Also, namespacing should be made at a descriptive, functional level. Not at a page location level. For example, `.profile-header` could become `.header-hero-unit`.

Wrong:

```
.nav,
```

```
.home-nav,
.profile-nav,
```

Right:

```
.nav,
.nav-bar,
.nav-list
```

Style Scoping

Medium pages should largely be reusing the general component level styles defined above. Page level name-spaces however can be helpful for overriding generic components in very specific contexts.

Page level overrides should be minimal and under a single page level class nest.

```
.home-page {
  .nav {
    margin-top: 10px;
  }
}
```

Nesting

Don't nest. Ever.

Nesting makes it harder to tell at a glance where css selector optimizations can be made.

Wrong:

```
.list-btn {
  .list-btn-inner {
    .btn {
      background: red;
    }
    .btn:hover {
      .opacity(.4);
    }
  }
}
```

Right:

```
.list-btn .btn-inner {
  background: red;
}
.list-btn .btn-inner:hover {
  .opacity(.4);
}
```

Spacing

CSS rules should be comma seperated but live on new lines:

Right:

```
.content,
.content-edit {
  …
}
```

Wrong:

```
.content, .content-edit {
  …
}
```

CSS blocks should be seperated by a single new line. not two. not 0.

Right:

```
.content {
  …
}
.content-edit {
```

https://gist.github.com/fat/b27700946c777adacdf4

It wasn't perfect by any means, but it cleaned up a few basic ideas and got us headed towards what felt like the right direction.

Unfortunately, even after this style update, people were still struggling with when to make a component, when to make a subcomponent, etc… And we were getting the occasional unfocused, run-on classname like: **.nav-on-light-background-button** or **.button-primary-sidebar-over-blur.**

People weren't prefixing classes at a page level anymore (which is great), but instead they were stringing together really long lists of arbitrary words separated by dashes. The evolution being: **.button → .button-primary → .button-primary-dark → .button-primary-dark-container → .button-primary-dark-container-label**, ad nauseam.

. . .

The 4th Project: Identifying the Future

About this time I started writing ~a lot~ about CSS internally at Medium with the lofty goal of it becoming the best in the world. I didn't really know what that meant, but I knew our current direction just wasn't working.

> *People were confused. And what's worse, they were thinking they were writing CSS really well, when in reality they weren't.*

So, I started looking around—poking at frameworks, trying different tools, philosophies, talking to friends, talking to friends of friends, etc. I soon identified three major projects that I wanted to tackle in order to get our CSS to where I thought it needed to be.

1. Introduce new CSS variable, mixin, and classname semantics to avoid run-on classnames and improve readability
2. Move us off of LESS and onto Rework for more powerful mixin support and a syntax even closer to vanilla CSS
3. Add tooling around CSS performance (load time, fps, layout time, etc)—to make it easier to track style changes and regressions over time

. . .

The 5th Project: Semantics

I wanted stricter semantics for our code base because for a team of our size I felt like it would be easier to have rules for us to lean on. And I'd rather have something a bit more complicated if that also meant people had to be more mindful when creating new CSS classes. At all costs, I wanted to avoid the run-on classname, or at least make it harder to create.

I began talking at length about it with Daryl Koopersmith and my good friend Nicolas Gallagher.

Nicolas and I have an interesting relationship in that Nicolas always tells me something, I say he's wrong, call him names in french (Va te faire foutre, enculé), and dance around for a few weeks, before inevitably realizing he's right, and taking his idea as my own.

This time was no different, and after a few late nights, I was eventually convinced of a style similar to the one Nicolas outlined in SUITCSS. Similar, but ~slightly~ better.

So I began by plagiarizing the hell out of Nicolas. I threw away our old style guide and copy pasted large portions of his, editing a few things here and there.

Eventually I came up with our current guide, which you can read in its entirety below, the main additions being:

- **.js-** prefixed class names for elements being relied upon for javascript selectors
- **.u-** prefixed class name for single purpose utility classes like .**u-underline**, .**u-capitalize**, etc.
- Introduction of meaningful hypens and camelCase—to underscore the separation between component, descendant

components, and modifiers

- **.is-** prefixed classes for stateful classes (often toggled by js) like **.is-disabled**
- New CSS variable semantics: [property]-[value]--[component]
- Mixins reduced to polyfills only and prefixed with **.m-**

LESS Coding Guidelines

Medium uses a strict subset of LESS for style generation. This subset includes variables and mixins, but nothing else (no nesting, etc.).

Medium's naming conventions are adapted from the work being done in the SUIT CSS framework. Which is to say, it relies on *structured class names* and *meaningful hyphens* (i.e., not using hyphens merely to separate words). This is to help work around the current limits of applying CSS to the DOM (i.e., the lack of style encapsulation) and to better communicate the relationships between classes.

Table of contents

- Performance
  - Specificity

JavaScript

syntax: `js-<targetName>`

JavaScript-specific classes reduce the risk that changing the structure or theme of components will inadvertently affect any required JavaScript behaviour and complex functionality. It is not neccesarry to use them in every case, just think of them as a tool in your utility belt. If you are creating a class, which you dont intend to use for styling, but instead only as a selector in JavaScript, you should probably be adding the `js-` prefix. In practice this looks like this:

```
<a href="/login" class="btn btn-primary js-login"></a>
```

Again, JavaScript-specific classes should not, under any circumstances, be styled.

Utilities

Medium's utility classes are low-level structural and positional traits. Utilities can be applied directly to any element; multiple utilities can be used together; and utilities can be used alongside component classes.

Utilities exist because certain CSS properties and patterns are used frequently. For example: floats, containing floats, vertical alignment, text truncation. Relying on utilities can help to reduce repetition and provide consistent implementations. They also act as a philosophical alternative to functional (i.e. non-polyfill) mixins.

```
<div class="u-clearfix">
  <p class="u-textTruncate">{$text}</p>
```

```
    <img class="u-pullLeft" src="{$src}" alt="">
    <img class="u-pullLeft" src="{$src}" alt="">
    <img class="u-pullLeft" src="{$src}" alt="">
  </div>
```

u-utilityName

Syntax: `u-<utilityName>`

Utilities must use a camel case name, prefixed with a `u` namespace. What follows is an example of how various utilities can be used to create a simple structure within a component.

```
<div class="u-clearfix">
  <a class="u-pullLeft" href="{$url}">
    <img class="u-block" src="{$src}" alt="">
  </a>
  <p class="u-sizeFill u-textBreak">
    …
  </p>
</div>
```

Components

Syntax: `<componentName>[--modifierName|-descendantName]`

Component driven development offers several benefits when reading and writing HTML and CSS:

- It helps to distinguish between the classes for the root of the component, descendant elements, and modifications.
- It keeps the specificity of selectors low.
- It helps to decouple presentation semantics from document semantics.

You can think of components as custom elements that enclose specific semantics, styling, and behaviour.

ComponentName

The component's name must be written in camel case.

```
.myComponent { /* … */ }
```

```
<article class="myComponent">
  …
</article>
```

componentName--modifierName

A component modifier is a class that modifies the presentation of the base component in some form. Modifier names must be written in camel case and be separated from the component name by two hyphens. The class should be included in the HTML *in addition* to the base component class.

```
/* Core button */
.btn { /* … */ }
/* Default button style */
.btn--default { /* … */ }
```

```
<button class="btn btn--primary">…</button>
```

componentName-descendantName

A component descendant is a class that is attached to a descendant node of a component. It's responsible for applying presentation directly to the descendant on behalf of a particular component. Descendant names must be written in camel case.

```
<article class="tweet">
  <header class="tweet-header">
    <img class="tweet-avatar" src="{$src}" alt="{$alt}">
    …
  </header>
  <div class="tweet-body">
    …
  </div>
</article>
```

componentName.is-stateOfComponent

Use `is-stateName` for state-based modifications of components. The state name must be Camel case. **Never style these classes directly; they should always be used as an adjoining class.**

JS can add/remove these classes. This means that the same state names can be used in multiple contexts, but every component must define its own styles for the state (as they are scoped to the component).

```
.tweet { /* … */ }
.tweet.is-expanded { /* … */ }
```

```
<article class="tweet is-expanded">
  …
</article>
```

Variables

Syntax: `<property>-<value>[--componentName]`

Variable names in our CSS are also strictly structured. This syntax provides strong associations between property, use, and component.

The following variable defintion is a color property, with the value grayLight, for use with the highlightMenu component.

```
@color-grayLight--highlightMenu: rgb(51, 51, 50);
```

Colors

When implementing feature styles, you should only be using color variables provided by colors.less.

When adding a color variable to colors.less, using RGB

When adding a color variable to colors.less, using RGB and RGBA color units are preferred over hex, named, HSL, or HSLA values.

Right:

```
rgb(50, 50, 50);
rgba(50, 50, 50, 0.2);
```

Wrong:

```
#FFF;
#FFFFFF;
white;
hsl(120, 100%, 50%);
hsla(120, 100%, 50%, 1);
```

z-index scale

Please use the z-index scale defined in z-index.less.

`@zIndex-1 - @zIndex-9` are provided. Nothing should be

https://gist.github.com/fat/a47b882eb5f84293c4ed

It's one thing to copy paste a style guide, it's another to rewrite all of your CSS application.

Thankfully I was able to convince the company of the importance of this project, if only for my sanity, and was given 2½ weeks to rewrite all the CSS on Medium.com—which if you're following along, is a fraction of the amount of time it took for us to fix CSS underlines.

That said, this rewrite was not only cathartic, it cleaned up loads of dead styles, tightened up and generalized implementations across the site, broke files out into smaller subcomponents, and to my surprise, only caused a handful of rollbacks.

Of course, refactoring CSS meant also refactoring our templates—adding more encapsulation and stricter semantics across the board. Now today, rather than having hundreds of one off <IMG> tags with random avatar classes for example, we have a single centralized avatar template, which accepts boolean options to trigger different modifiers

like size, style, etc. This makes updating styles easier and implementation detail bugs much less frequent.

. . .

X Project: The future?

Undoubtedly we are in a better place then we were 2 years ago. Writing CSS at Medium is actually pretty pleasant and devs with different experience levels are contributing some pretty great stuff.

That said, the next CSS project will have to be around performance. As we continue to grow our story pages and push them to the next level, you can imagine how getting accurate, reliable measuring tools around layout and rendering performance is incredibly important… and kinda just sad that we don't already have in 2014.

So that's it. There's more to do, but I'm feeling pretty good about it all at the moment. Which again, is better than usual I think.

Cheers if you made it to the bottom of this long boring post.

Thanks so much to Katie, Dave, Mark, Koop, Kristofer, Nicolas, and others for helping me write this thing (and fix Medium) without even probably realizing it ❤

PS

(If you enjoy this?! Consider reading my take on React and Redux: