

LEARN

TO

CODE



NOW

SUPERHi

Learn to Code Now

Written by

Rik Lomas

Foreword by

Frank Chimero

Art Direction by

Milan Moffatt

Thanks to

Holly Holmes

Christine Lomas

Lawrence Gosset

Adam Oskwarek

David Holmes

Simon Whybray

Deron Millay

The team at Koto

Tony the cat for his furry cuddles

NYU Langone Medical Center's café

Projective Space, Lower East Side, NYC

Tim Berners-Lee for making the web

All our students for supporting us

©2017 SuperHi Inc.

Foreword	13
-----------------	-----------

Introduction	18
---------------------	-----------

1	Different programming languages for different tasks	24
----------	--	-----------

2	How to make a website	32
	Recapping front-end web development	34
	But how do you write code?	36

3	Let's talk about HTML	44
	Oh hi, I'm a heading	48
	Blocks and inline	51
	Links	54
	Images	58
	White space + indentation	61

4	CSS — Making Code Great Again	64
	Units	72
	Connecting HTML + CSS.....	77
	Typography	80

5	HTML + CSS continued	88
	Meet the header tag.....	90
	Styling with color	94
	Background images	100
	Hover states	111
	Transitions	115
	Child selectors.....	117
	Classes	120
	Widths and heights	127

6	Let's get mathematical	130
	Ratios	132
	Margins and paddings	134
	Borders	143
	Rounded corners with border radius	147
	Filters	149

7 Get ready to be amazed by multi-column layouts with floats.....152

Wrapping elements using floats154

Overflow168

Transparency with opacity and rgba colors170

Drop shadows174

8 Mobile-friendly designs with media queries.....178

9 CSS Displays.....188

10 Positioning.....196

Cursors and mouse pointers207

11

Transforms: rotations, scaling and skews 210

Vertical alignment 219

Head tags and search engines 222

Flexbox 226

CSS animations 242

Forms 254

Audio, video and media 273

12

Web fonts 280

13

Javascript 288

Starting with Javascript 293

Manipulating data 301

For loops 307

Functions 309

Adding your scripts to your pages 312

14

jQuery 314

Events 320

What is this? 331

jQuery animations	333
CSS and jQuery together	338
Timers and intervals	346

15

Document and window	352
----------------------------------	-----

Fade in tags on scroll	365
Counting numbers in Javascript	373
Getting data out of objects	377
jQuery prepend, append, before and after	381

16

Ajax	388
-------------------	-----

JSON	398
Animation using Javascript	402
Mouse movements	407
Lightboxes	411
jQuery plug-ins	419
Fixing your own code	421
Integration with back-ends	435

Afterword	441
------------------------	-----

Foreword

by Frank Chimero

The house where I was raised was an ideal place for our family, except for one serious flaw: the house did not have my parents' fingerprints on it. So began a constant construction project that lasted almost a decade. Bathrooms were moved and walls knocked down; skylights were installed, had their leaks fixed, then uninstalled for the trouble they caused. My parents removed the attic, raised the ceilings, bought a larger Christmas tree to take advantage of the new vertical clearance, then celebrated the New Year by re-tiling the bathrooms.

One day after arriving home from school, I saw my father hovering over a giant stockpot with wooden strips fanned out over the rim like uncooked spaghetti. He was boiling the planks, he said, to soften them. He'd then slowly form each strip along the curved edge of our built-in bookshelf to use as trim for the semi-circular shelves at the end. My father was taught this method by my grandfather, another amateur furniture maker, and now it was my turn to learn the process. "Pay attention to the wood, follow the grain, and if you take care, the wood will bend and not break," he said. This was clearly intended to be a life lesson as well as a lecture in woodworking, the kind of practical

inheritance that fathers like to provide their sons. Be patient. Be gentle. Got it.

Carpentry didn't take. Ten years of growing up in a construction zone made me swear off woodworking. (Also, we got a computer the year after my lessons in woodwork. How could I resist?) Instead I design and build software, which has its own methods and tricks, but I still find myself returning to my dad's lesson. All materials, whether wood or pixels, have a grain, and that grain suggests the best way to work. Go with the grain and one will find sturdiness combined with tremendous flexibility – a natural and exciting give that grounds decisions and excites with possibilities. Work against the grain and the work becomes precarious, difficult and fragile. Instead of the elegant bending that software requires to adjust to different screens, uses, and situations, the work breaks because it cannot adapt.

This idea of a grain, however, flies in the face of our expectations for technology. Software is often presented as a wide-open, infinitely malleable material. We expect technology to help us overcome limitations, not produce more of them. Can't I do what I want? Only to an extent.

We use teak for outdoor furniture because it is weather resistant. We use white pine for wood carving because it is soft. These kinds of rationale also go for designing software. On screen, we use flat colors and simple gradients, because they're lightweight, easy to programatically draw, and can scale for areas of varying proportions. Sites have horizontal stripes of content stacked vertically, because that is how we read, and it is easier for most users to scroll vertically than horizontally. All of these design choices come from a knowledge of the materials

at hand. What is the grain of software? It has to do with fluidity. People who work on software create flexible systems that can deal with variability: content of varying lengths, connections of different speeds, users with many kinds of ability and attention span. What does the page look like if it is empty? If it is full? And every possibility in between on a mobile device? Working with software is never designing towards a fixed artifact like a chair or book. Instead, it is defining and designing conditions for a whole set of possibilities.

The easiest way to explore and test these possibilities is by working with the raw materials themselves. Learn a bit of code and fiddle with things. A sturdy knowledge of HTML, CSS and Javascript goes a long way towards understanding what is possible with the medium.



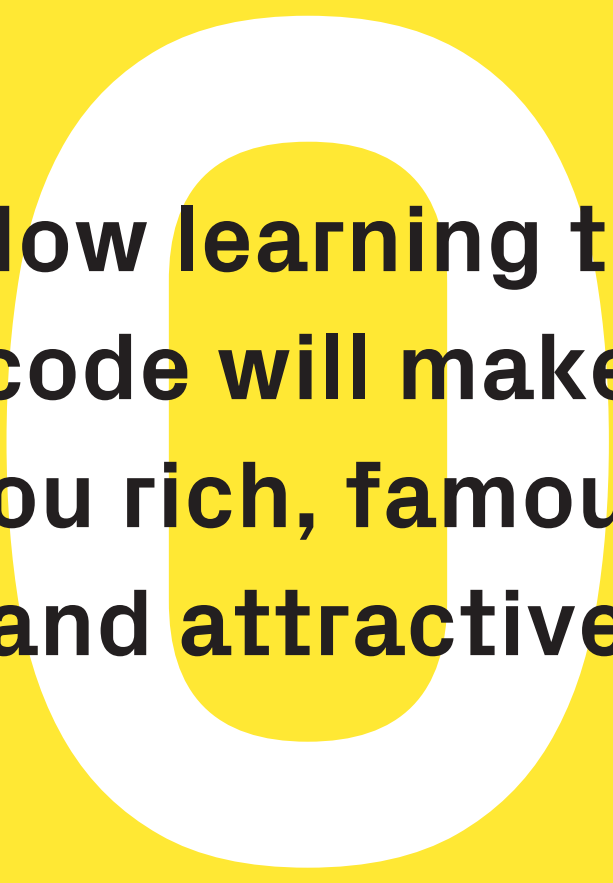
Good work is grounded in attention to detail, and knowledge of and respect for the materials.

You will make mistakes. Things won't work. But you will be in good company. We all get it wrong on the first, second, even third tries, no matter how much experience we have. Getting your hands on

the materials is a learning process for all of us. We feel the grain and discover the contours of the problem we are to solving, and revise when our efforts don't work quite as expected. Luckily, code and pixels are free, so your trials and errors should be less expensive than the considerable amount of lumber I'd waste if I ever took up woodworking.

My father never fully understood what I did for a living, but we could always find common ground in craftsmanship. Good work is grounded in attention to detail, and knowledge of and respect for the materials. The more experience I gain, the more this proves itself. Pay attention, respect the material, listen to how it guides you, and be gentle. You'll be surprised by what you can do and how flexible it all can be.

— *Frank Chimero, 2017*



**How learning to
code will make
you rich, famous
and attractive**

...and how to avoid the fake promises every single article on the internet on learning to code tells you.

Articles about learning to code often describe the possible financial rewards. How this knowledge will earn you a better salary, a promotion, even a new job.

There is some truth in these claims. We've taught people who've gone on to get pay rises, improved careers and higher freelance rates. That's because good coders are in demand.

A likely skills shortage in the next decade means a lot of jobs won't be filled. Some estimates suggest there will be a million vacancies in technology by 2020 in the United States alone.

At the same time more and more traditional jobs are set to be automated. Driverless cars for instance will likely become mainstream over the next two decades, threatening to put 3 million Americans out of work.

So there's a problem. Artificial Intelligence could lead to mass unemployment at the same time as tech jobs can't be filled due to the lack of a skilled workforce.

For children of course it's different. There are nine year olds in Estonia learning to program as part of their school curriculum. A group called Code Club holds after-school sessions around the world.

Sounds scary, right? Millions of jobs worldwide taken by robots, millions of jobs unfulfilled and nine year olds who can code better than you. The articles tell us to learn how to code so we are on the right side of this shift and won't be left behind. Often it can work. We've seen people who've paid five-figure sums to go on boot camp-style courses because they're worried about their future. But our approach is different. We're here to say don't become a coder. Don't learn to code just to be a coder.

We know what you're thinking. Why is a code school telling me not to study coding?

We're not. We're saying you should learn to code to improve and add to the skill set you have. If you're a designer, for instance, learning to code will give you a better understanding of how to design for the web. Keep being a designer, but one who knows how to use media queries in CSS to change a layout for mobile screens.

The best web designers we know can all code but they wouldn't call themselves professional coders.

Use code to be creative. Remember coding is really a design tool. Whether you're designing the next Facebook or your new portfolio, all you're doing is writing instructions to a computer. Make that box red. Make this column wider. Show the header after the user scrolls.

We're telling a fast but dumb machine how to make a site or an app do what we want.

Learning to code won't immediately turn you into the next Mark Zuckerberg or Bill Gates. What it will do is help you use your creativity and your motivation to turn your ideas into something real.

I learned to code in 2000. I was 16 and was building websites using a tool called Microsoft Publisher which was (and still is) Microsoft's version of InDesign or Quark. It's aimed primarily at designing booklets and posters but it had a website maker too.

At the time, it had a set number of templates which you could customize to a limited extent. You could change a few images and fonts and that was really it. But I wanted to do more than make a few changes, I wanted to mess up the site and make it look completely different. It's my site and I didn't want just a template.

A lot of students still have the same problems even in 2017. They don't want to stick with a Wordpress template or a SquareSpace theme. They want to make their own thing and not just tweak someone else's work.

Over a few months and years of struggling with the basics, I slowly became better at coding. I was definitely not an expert but could make it look decent.

I learned to code essentially for free. But it took months and years of struggle and continually getting stuck with problems. And with no-one to turn to for help.

The Internet is a completely different place to when I learned to code. There's a ton of resources that you can use to learn for free. But the Internet has become a lot more complex. Look at sites from the early 2000s compared with recent efforts and there is a world of difference.

Coding has become professionalised and the tools that are used to make websites and apps have increased in complexity. This can create a massive problem for beginner coders. Professionals rarely consider how beginners can use the tools they create.

Plenty of experienced coders do care about beginners, but others are snobbish about those less knowledgeable. They see beginners as a threat — more coders means more supply and less demand. They use the No True Scotsman argument to dismiss the clunky efforts of inexperienced coders — “you're no true coder if you do X, Y or Z”.

The truth is a lot cloudier. It does take a long time to get really, really good at programming. Some of the best coders I know only consider themselves as “good”. As with anything, it's the last five to 10 percent of a skillset which is the hardest to master.

Yet 90 percent of what most coders do every day can be learned quickly enough. Some junior coders I know are better than more senior colleagues. When you break down a lot of code, it's doing the same four things: listing, adding, editing and deleting items.

Take an example. My Twitter feed is just a list of posts (or tweets). When I tweet, I'm adding a new tweet to my list. I can't even edit tweets. Making a Twitter clone is in theory at least an easy task, and there are many startups trying to be the next Twitter.

Does that mean junior codes would find it easy to work at Twitter? Probably not, because of the scale and complexity of getting billions of tweets to millions of users. Should that put you off trying to build the next Twitter? Of course not.

So how do you get good at coding? Practice. Lots and lots of it. Keep making things. Keep adding features to your current projects. Keep breaking things. Keep going wrong and fixing your mistakes.

Keep learning.



**Different
programming
languages for
different tasks**

It could be the first shock when you're planning to learn to code — there are so many different languages you can use.

How come? Why can't we just have one and make it easier for everyone?

Well, it's not as simple as that. Each language is designed for different tasks. There is a saying: "If all you have is a hammer, everything looks like a nail". What we need to do is pick the best language for the job in hand, not fit the task to what the tool can do.

Let's split our languages into three sub-categories:

- ♦ **Front-end web languages** — *those that work together to make up the visual side of web sites*
- ♦ **Back-end web languages** — *those that handle the processes that a user doesn't see, such as saving data to a database, checking a user's password or paying for an order with a credit card*
- ♦ **Mobile languages** — *those used to make iPhone, iPad and Android apps*

The best way to think about front- vs back-end languages is to imagine a website being like a restaurant — you have a front-of-house staffed

by the waiters, and a back-of-house where you find the cooks. A customer interacts directly with the waiters, who then tell the kitchen staff what to prepare. The cooks make your food, hand it to a waiter, who then delivers it to your table. On a website, you only see the front-end, which is what you interact with. The back-end works away behind the scenes.

Front-end web languages

The visual elements of all websites use a combination of three languages:

- ♦ **HTML** — *which stands for HyperText Markup Language and which essentially creates and controls content such as paragraphs, headings, links and more*
- ♦ **CSS** — *or Cascading Style Sheets, which sets how the content of websites (the HTML) should look, such as this header should be red with font size 32px in the font Helvetica. It also handles how websites look on different screens such as a mobile vs a laptop*
- ♦ **Javascript** — *how a user interacts with web pages. For instance, when a user scrolls down the page, hide the header, or when a user types in an input box, check the text is an email address. Anything that happens after a page has loaded is usually controlled with Javascript*

Back-end languages

Websites can use an assortment of languages but usually you pick just one back-end language and use that. Each has slight benefits over others, but generally, they're all doing the same job: showing users web pages and talking to databases.

Some languages used on the back-end include:

- ♦ **PHP** — *used by Facebook and Wikipedia*
- ♦ **Python** — *used heavily by Google and Amazon*
- ♦ **Ruby** — *used by a lot of large startups like Basecamp*
- ♦ **Java** — *not Javascript but a completely different language only related by the fact that Javascript was invented to make two Java scripts talk to each other in a browser*
- ♦ **Elixir** — *what SuperHi uses, it's particular good for handling a lot of data at the same time*
- ♦ **Erlang** — *invented by telephone companies to handle lots of messages at the same time, now used in Whatsapp and Facebook Chat*

Some of these languages have extra libraries to help coders be even more productive. For instance, Ruby has a library for building web apps called Rails (short for Ruby on Rails). Python has one called Django and PHP has a few (Craft and Zend being two big ones). Even Javascript has a few: jQuery, React, Angular and Ember.

Mobile languages

Apps have a completely different structure to websites. For instance, if you're building an iPhone app, you wouldn't need any of the visual side of web coding as you're not building a website. Apple provides a lot of the structure you need for building apps using two languages (you can pick either), **Objective-C** and **Swift**. Swift is probably the most beginner-friendly of the two.

Android apps have a completely different structure again to the web and to iPhone apps, so developers have to build apps in **Java** using special libraries that Google provides.

If you want to build an app for both iPhone and Android, you're essentially building two completely separate apps that work differently under the hood. You may notice when new apps launch they pick one or the other, depending on where they think most of their audience might be. This is purely a time issue. Why wait until you have built two apps? Build one and see if people like it.

Around all these languages there are tools that make it easier for coders to work together. You may have heard of **Git**, which is a version control tool (think of it as being like Dropbox for coders). Git doesn't stand for anything, the inventor just thought the British word "git" meaning an ignorant, childish person was a funny word. What a silly git.

With Git, there's a site called **GitHub** where coders can store their files to share either privately among their teammates or publicly if they've selflessly made some code other people could use.

How to pick the right code language for the job

With any digital project, success can heavily depend on the right choice of code language, but that doesn't necessarily mean you should dive in with that language first.

Most languages share similar overarching processes in how they work. They take in data, do something with that data, then return something out the other side. For instance, if you're logging into a site or an app, all code will be doing the same thing: taking the user's name and password, checking then against a database, and if they are correct, logging in that user. The code might be different but the process is the same.

Before I write any code, I always use pen and paper to sketch out the overarching process. If it's taking a payment on SuperHi's site, what is the order I should ask for things? Do I ask for an email address first or do I ask for which start date the customer wants? Do I make it a stepped process or should I do it in one big form?

This overall process is the real part of coding. It's working out how to untangle a user's requirements and make things easier for them.

With this in mind, it's worth asking what code language to learn first. The real answer is it doesn't matter too much. As long as you understand the overall process, it's easy to apply it to any given language. Once you know one, it's a lot easier to pick up another. The same is true with spoken languages: bilingual speakers can pick up

another language a lot quicker than someone with only their mother tongue.

Having said that, there are definitely some languages which SuperHi's students have found a lot easier to pick up than others. Here's is a list of the languages we identified a few pages ago, ranked in order of easiness to learn (with the hardest at the bottom):

1. *HTML*
2. *CSS*
3. *Ruby*
4. *Python*
5. *Elixir*
6. *PHP*
7. *Javascript*
8. *Swift*
9. *Java*
10. *Erlang*

We won't go into detail about why we picked this order, but mostly it's to do with the structure of the languages. The hardest ones have more ways to go wrong, whereas the easiest ones are more flexible.

Does this mean you should learn Ruby before Javascript? Possibly, but the opposite can be true too: if you learn Javascript and grasp its difficulty, Ruby will make sense a lot sooner. It's whether you prefer a deep dive or a more gradual learning curve.

On our courses we learn HTML, CSS and Javascript together as it makes sense to show how all the visual parts of the web work together, but we spend the first half of courses making sure students are comfortable with HTML and CSS before we even touch Javascript. Always make sure you're comfortable with the first language before trying out a second.

2

How to make a website

...and what exactly is “front-end web development”?

Earlier we talked about the different programming languages and the two types of code, for front-end and back-end web development. The former creates the look and feel of websites, it's how our users and customers interact with what we've made. The latter is how users' information is processed. Our users don't see what's happening in this code, they put in some information such as their log-in details or their credit card number to pay for a course, the site takes that information and processes it. Our user then sees the next bit of front-end code, depending on what's happened — for instance, if your login details are wrong, you'll see a try again page, or if you've paid correctly, you'll see a confirmation page.

The web development we'll be talking about here is the front-end code. There are two reasons why we'll be concentrating on this in the course. Firstly, it's important to know how front-end code works and is put together before we tackle back-end code. Secondly, it's easier to learn front-end code than back-end.

Recapping front-end web development

There are three different code languages in front-end web development: HTML, CSS and Javascript. Each one has a different role to play.

- ♦ *HTML is the structure of websites*
- ♦ *CSS is the style of websites*
- ♦ *Javascript is the actions of websites*

Having these definitions in mind is always handy in helping you decide where you are going to put your code.

If you want to add a new paragraph to your site, or you want to remove a heading, what you're essentially doing is changing content. Therefore, the language you'd use for this is HTML.

If you want to change the typeface of the headings, or make the paragraphs a different color, this is changing how it looks (or the style of the page). Therefore, the language you'd do this in is CSS.

If you want to change how a user interacts with an input box, or you want things to move when a user scrolls down the page, you're changing the user actions or the user interactivity. Therefore, the language you'd do this in is Javascript.

Sometimes it can get a little tricky to work out. Don't worry though, as you'll pick up where the distinctions are. For instance, if you want to make the site look different on a mobile screen compared with a desktop, where would you start? A clue is in the sentence — you want to make the site look different — so you'd change the CSS of your website.

However, you may have initially thought making the site look different on mobile and desktop would have been user interactivity, as it depends on how the user is looking at it. The delineation could become clear by asking yourself, is the user interacting with the site or not? If they are interacting and changing the site after they've loaded it, it would be Javascript. If they're just seeing the site on a mobile screen and not interacting with it, then it would be CSS.

Don't worry too much about this issue just yet though. We'll talk about it a little more later in the book.

But how do you write code?

If you're reading this guide, I'm going to assume that you're a human (hi to any aliens reading this too). On our laptops, iPads and phones, we have a bunch of files. On my phone, I have some music files of songs by awful 90s indie bands, I have some videos of Tony T (my kitty), some photos sneakily taken of other people's dogs and lots, lots more.

What kind of files are they? Most likely, my music files are MP3s, my videos are .mov files and my images are JPGs. How do we know what kind of file they are? We look at the extension of the file. For example, a file called "tonythecat.jpg" has the ending ".jpg" so the computer can take a guess it is a photo. Similarly, the computer would guess that "all-star-by-smashmouth.mp3" would be a music file because of the MP3 extension at the end of the file name.

What's in a file?

All these photo, video and music files contain instructions for our laptops and phones to play or view them in a certain way. A photo file would contain a certain number of pixels and what color each pixel was, so the laptop could build up an image of Tony the cat. A music file would contain a description of the sound waves that the computer should play so I can listen to All Star by Smashmouth.

All of this is just code. Code is instructions for a particular program to run. My music app plays music files. My photo viewer shows me photos. I can't play a photo in a music player because the music player doesn't understand the instructions.

Websites are no different. The program that runs website code is called a browser. There are a few popular brand names of browser — Google Chrome, Firefox, Safari and Internet Explorer are the well known ones.

Browsers take code files that contain HTML, CSS and Javascript and turn them into visual websites for users to interact with.

So if a music file is a file that ends in “.mp3” and contains instructions to play music, website files are the same. The code for the content of a site is an HTML file, e.g. “about.html”, that contains HTML code. The code for the style of a site is a CSS file, e.g. “style.css”, that contains CSS code. The code for the user interactions of a site is a Javascript file, e.g. “scroll.js”, that contains Javascript code.

Where are my files?

A decade ago, if you were to play music on your laptop or phone, you'd have the files stored on your device. You'd have a folder called something like “music” or “iTunes” that would live on your hard drive.

But if you're looking at Google or Facebook, there isn't a folder on your hard drive called "Google" or "Facebook", so how are you getting to view these files?

If you want to play music in a more modern way, you'd use a streaming service like Spotify or Apple Music. With these services, you're not having to download all the music ever created to be able to have access to every song ever. Instead, you're streaming the songs using the Internet to the device when you need them. The songs aren't stored on your hard drive any more, they're stored on the Internet and you get them whenever you want them.

The same thing is true of web browsers. If you're looking at Google or Facebook, you're downloading HTML, CSS and Javascript files on demand. Essentially web browsers were the first streaming service.

What is streaming though?

Let's take Spotify as an example. When you're listening to a music file on Spotify, you're requesting the file from Spotify's very large hard drive (or server) instead of your own hard drive. You're using the Internet between you and Spotify to stream the file.

Because you're the one asking Spotify for the file using the Internet, this is called downloading.

On a site like Facebook, if I want to share a photo that's on my phone and put it on Facebook's very large hard drive/server, then I use the Internet between Facebook and me to upload the photo.

So all downloading really means is sending a file from someone else's hard drive to my hard drive, and all uploading means is sending a file from my hard drive to someone else's using the Internet.

When you're in a web browser like Google Chrome and you type in a web address like "www.google.com", all you're telling the web browser to do is download files from Google's server for the homepage. What files are we downloading? HTML, CSS and Javascript files to show us the content, style and actions for that page.

Wait! The Internet and the Web are different?

Most people just assume that the Internet and the Web (or World Wide Web to give it its full name) are the same thing. In reality, they're two things.

The Internet is a service that connects hard drives or servers together.

For example, you need the Internet to play songs on Spotify but you don't need to be in a web browser to use Spotify. Similarly, you don't need to be in a web browser to use Instagram — most of the time,

Instagram's users use the iPhone or Android app — not Chrome or Safari.

The Web is a subset of the Internet which deals mainly with HTML, CSS and Javascript files. We can pull other assets like images, videos and audio into our sites but they live within a web browser.

So why not just build an app? Why contain me in a browser?

The eternal question. When should I build for the Web and when should I build an app? This is a question that a lot of startup founders get really stuck with, and a lot go with what's trendy over what would be best.

For a time in around 2006, Facebook pages were all the rage. I was working at a creative agency and pretty much every one of its clients had requested a Facebook page, even if it wasn't suitable for the business. Why did they want it? Because their rivals had them so they wanted to join in the fun. Most of the time, they were a complete waste of money and their users didn't really care.

Nowadays, apps are in a similar category — you can build them but you might be wasting your time.

Why build for the Web over building an app?

Firstly, with the Web you have 100 percent of the audience. Earlier in the book, we talked about mobile programming. If you want to build an app for iOS and Android, you're essentially having to build two completely separate apps. Which, surprise surprise, takes twice as long. If you decide to do just one, you're missing a big audience using whichever mobile platform you ignore.

Secondly, there's no installation process with websites. If you look at the level of people who commit to installing an app once they've got to the installation page, it's a pretty low percentage, so not only do you need to get people to the installation page, you need to get them over the hurdle of installing the app. The median average of apps installed per person per month is zero. Most people do not install any new apps. The average American uses just three apps 80 percent of the time.

Thirdly, the publishing cycle for apps is up to the app stores and can be days and even weeks, whereas the Web is instant. Why let your customers and users wait for weeks for updates when you could do it instantly?

Lastly, discoverability is a lot tougher for apps. Most users are more likely to search via Google rather than an App Store.

Only recently have search engines like Google added apps to search results. It's also a lot harder to share an app versus a link on social media. Ninety percent of apps are deleted after being opened just once.

So for four big points, the Web wins out: more audience, no installation, instant publishing and high discoverability.

Granted, there are certain instances where apps win out, such as offering complex functionality when using cameras or geolocation, but for 90 percent of new internet businesses, publishing a website over an app makes far more sense.

What is a code editor?

When we're making our files, what program do we make them in? Is there a special program we need? If so, is it expensive?

Luckily for us, all the files that we'll be making are essentially text files with a different extension. You can open them in Notepad, TextEdit, IA Writer or any program that you would write plain old text in.

Make a folder on your computer, then store all your code and assets in there.

There are, however, easier ways to edit code. There are specialist programs that make it simpler to see what's happening when you write code.

Some of the most popular code editors include **Atom** (atom.io), **Sublime Text** (sublimetext.com), **Coda** (panic.com/coda) and **Dreamweaver** (adobe.com/dreamweaver). Some are free to download, some are paid for.

Insert product placement here

Now, usually I hate it when a company plugs their own products and tells you how great they are. I'm going to do just that, but I'll keep it very brief.

After teaching hundreds of students, I noticed a few problems when beginners were coding with the popular code editors — the same typos, the same mistakes, the same problems with hosting — so as a team we felt we could fix this for our students.

We didn't want our students feeling dumb and discouraged because they'd made a mistake. They should be able to be shown how to fix their own problems without needing to search on Google, ask a friend or give up. So we made our own code editor.

It's aimed at beginners like you. We've been working on it for more than two years and all our students have been using it.

I won't go into all the product features as we're constantly making it better and better, but it includes instant hosting, live previews, version control, artificial intelligent helpers and lots more.

Try it for yourself at **editor.superhi.com** and tell us what you think.



**Let's talk
about HTML**

Why talk about HTML first? Well, we need to have some content which we can style and interact with.

HTML stands for HyperText Markup Language. It was invented by British scientist Tim Berners-Lee in the late 1980s when he was working at a laboratory called CERN in Switzerland. Hard to imagine but in those days computer files basically sat on one machine and couldn't easily be read by another. Not only did Tim invent HTML to “mark up” (or format) text, he invented the first web browser so he could read those files from anywhere.

Then he thought: “Hey, this is pretty useful, maybe other laboratories could use this too”. Over time, other people thought something similar: “This is pretty useful, maybe normal people could use it”. Over the next few decades, the Web became mainstream.

This was in part because Tim didn't want to make money from it. He felt it was bigger than him and could be used to help increase scientific and technological progress. Little did he know it would be used years later for watching cat videos on YouTube. But it's Tim's ingenuity and selflessness that made the Web what it is.

Tim Berner-Lee love-in part two

The other part of what Tim did to make the Web successful was to make it easy to write the files. He didn't want to make them just normal text files but let the author have the ability to say what each part of the text means.

Remember, the user doesn't see the actual code that the author writes. The user sees the final version of the website — the browser turns the HTML code into a web page, just like Spotify turns a music data file into an audio of Smashmouth's All Star.

The first thing we'd do is create a new file and name it something related to the page, then with the HTML extension. For example, about.html is a good file name for the about page on a website. We then open it and start writing our content.

Tim wanted the author to let whoever was viewing the site that a particular part of the text was a heading and that another part was a paragraph.

The way he thought about doing this was to surround the text with things called “tags”. He wanted to start the header at one place and finish it at another. For instance, if we have some text that reads:

Isn't this scientific document cool?

We would start the sentence with a paragraph open tag, then finish it with a paragraph closing tag:

```
<p>Isn't this scientific document cool?</p>
```

Notice the slight difference between the two tags. The open tag doesn't have a slash in it and the closing tag does. When we have a slash in a tag, it means stop doing this tag now.

The name of the tag is a “p” tag — the p standing for paragraph. Tim liked to use shorter names for two reasons — it's easier to write and back in the 80s the Internet was a lot slower, so the less text the less time the page took to load.

There are around 100 types of HTML tags but on a day to day basis, most coders only use around 15 to 20 of them. Paragraph tags (<p>) are among the most common.

Tags can also be reused too. There's a high probability that there's more than one tag on your page, so to “mark up” or describe the text, we can just use another <p> tag.

```
<p>Isn't this scientific document cool?</p>
```

```
<p>I just discovered how to make lead into gold.
```

```
I thought that wasn't possible?</p>
```

Oh hi, I'm a heading

Of course, not everything is a paragraph. On any document, you're likely to have titles and sub-titles. Just like at the top of this section, where there's a heading saying hi to you.

As with paragraphs, we have to mark up our text with which bits are headings and sub-headings, and as with any document there's a hierarchy to our headings and titles.

We start with the main headings and work our way down to sub-headings and sub-sub-headings.

Let's take a résumé. The main heading on the page would be your name e.g. Rik Lomas. You'd then have sub-headings such as Education, Experience and References. Then within each of those you'd have sub-sub-headings, so in Experience, you might have company names (e.g. SuperHi, Steer etc).

In HTML, there are tags to reflect these headings, sub-headings and sub-sub-headings (down to sub-sub-sub-sub-sub-headings!).

Your main heading would be a `<h1>` tag.

```
<h1>Rik Lomas</h1>
```

Your sub-heading would be a `<h2>` tag.

```
<h2>Education</h2>
```


Your sub-sub-heading would be a `<h3>` tag.

```
<h3>SuperHi Inc.</h3>
```

There are even `<h4>`, `<h5>` and `<h6>` tags too. Notice that it's not just in descending number, it's text of an equal importance. For instance, I might have a tag `<h3>` above another `<h2>`. I can continue to use paragraphs above and below my headings too.

```
<h1>Rik Lomas</h1>
```

```
<h2>Experience</h2>
```

```
<h3>SuperHi Inc</h3>
```

```
<p>I was the founder and coder for SuperHi...</p>
```

So very quickly, we've covered six more tags, taking our total to seven. Remember, we said that most professional coders only use 15 to 20 tags, so we're 30 to 40 percent through all the tags you might ever need to make websites.

index.html

We talked earlier about naming our HTML files after the page they represent. An “about” page would be named “about.html”, a “contact” page would be named “contact.html” or “contact-us.html” (remember you can't use spaces in file names).

There is a special name however for the first page you expect your user to visit. We'd call it the home page, but its file name is "index.html".

Whenever I go to www.facebook.com, I'm essentially asking for the index.html on Facebook. All sites will look for the index.html if you don't specify a particular page.

Why index? Remember Tim Berners-Lee made this for scientific reasons so he thought that the first page would a list of other pages to read — an index or record of pages on that site.

Blocks and inline

All the tags we've talked about so far are kind of bigger boxes of content. Visually they'd go across the whole page and the next tag would go below them. Your header is at the top, then a paragraph below, then another paragraph below that.

These tags are called block tags because they fill the width of the page and make the next tag go beneath them.

Most tags are block tags, but there are other tags where you don't want to fill up the whole width of the page. For example, you might just want to highlight one or two words in a paragraph.

There are a few tags that can work within a line of text, these are called inline tags.

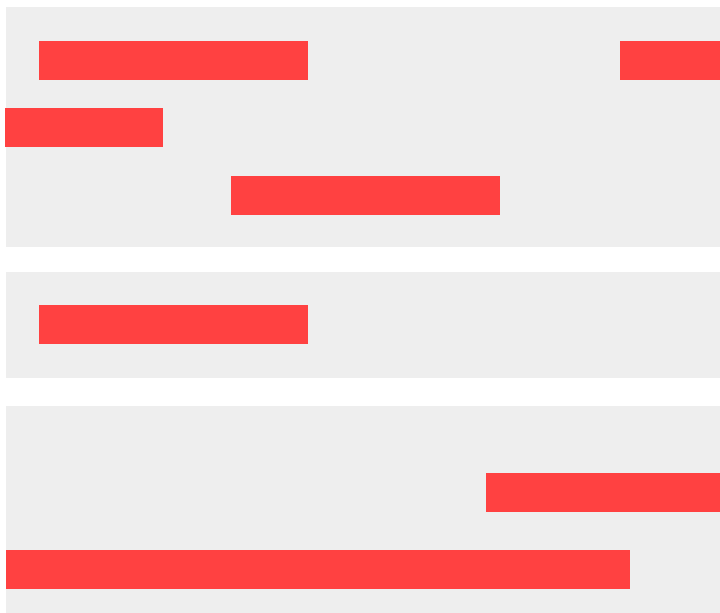
An example is a bold tag, the `` tag. You might only want to highlight one or two words in the paragraph.

```
<p>At my last job, conversions improved by <b>over 50%  
</b> in just 4 months.</p>
```

To paraphrase Xzibit in Pimp My Ride: I heard you liked tags so I put tags inside your tags. We'll be putting more tags inside other tags later.

Other inline tags include for italics (`<i>`) and to underline something (`<u>`). In this paragraph, we're having "over 50%" as italic and "just 4" as underline. That's a `<p>` tag with two different tags inside.

Block tags and inline tags



BLOCK



INLINE

```
<p>At my last job, conversions improved by <i>over 50%  
</i> in <u>just 4</u> months.</p>
```

One thing to note is that an inline tag could go to multiple lines of text — for instance in the previous example, the word “over” could be the last word of the first line and the “50%” could be the first word of the second line.

A good rule of thumb is that block tags go down the page and inline tags go across the page.

Links

One important tag that made the web successful was the ability to link one page to another, so users could click between the pages of not only one author's website but to other author's sites. The original idea came from scholarly referencing and footnotes, but it massively helped the discoverability of other people's websites.

A link tag in HTML is called the `<a>` tag — the “a” stands for anchor because it might link to another site, another page or another section within the page so it doesn't always go away from the same place.

Just like the bold, italic and underline tags, a link tag is an inline tag, as you could just link one word in a paragraph to somewhere else.

The link is a little more complex than the other tags we've covered. We need to introduce a new part of HTML called attributes.

Attributes

Let's give an example of a link tag.

```
<a>Go to Facebook</a>
```

This is some text that says “Go to Facebook” and is surrounded by the `<a>` link tags at the start and end. But if a user clicks the link, where

should they go to? Should they go to the Facebook.com homepage? Should they go to the Wikipedia entry on Facebook? Should they go to the Google Finance page for Facebook's stock price? It depends.

We need to give this some more information.

We don't want to write out the URL for where the link is going to in the text, otherwise our user will see an ugly URL when they just want the words "Go to Facebook".

To add this extra information, we need to add in an HTML attribute — an extra bit of information for this tag. To do this, we are going to add a special HTML attribute to the link called the "href" attribute (short for hypertext reference).

```
<a href="http://www.facebook.com">Go to Facebook</a>
```

The way we add attributes is to add them to opening tags in HTML. We add a space after the name of the tag (e.g. "a", "h1"), then the name of attribute (e.g. "href") with an equals symbol, then something in quotation marks.

There are more attributes that we'll talk about later, but for now we'll only be using the "href" attribute on link tags.

Different types of links

The link above goes to a different website completely. These links are called “absolute” links — basically something that isn’t on your own site.

You can write the href attribute as a “relative” link if you’re linking to another page on your own website.

```
<a href="about.html">About page</a>
```

Notice you don’t need the full URL, just the name of the file on your site.

Other links — email me...

You don’t have to link to other HTML pages on a link. You can do other actions such as pop open a new email compose box.

```
<a href="mailto:rik@superhi.com">Email me!</a>
```

Notice this href attribute starts with “mailto:” then the email address of where you want to send it.

Joining everything together

So we've covered paragraphs, titles, sub-titles, inline tags and links. We can combine them all together to get something like this.

```
<h1>Rik Lomas</h1>
<h2>Experience</h2>
<h3>SuperHi Inc</h3>
<p>I worked at <a href="https://www.superhi.com">
SuperHi</a> from 2014-2017.</p>
<p>When I was there, I helped increase profits by
<b>30%</b> over the course of those years</p>
<p>You can <a href="about.html">contact me here</a>.</p>
```

Images

Text only is pretty boring, as Tim Berners-Lee found out a few years after creating HTML and the Web. Within CERN the Web was starting to be used for non-scientific purposes. Some of the science staff formed an all-female pop band called Les Horribles Cernettes and did parody pop songs replacing lyrics with science jokes. They had the words on their website but they wanted their band photo too, so they asked Tim how.

At the time the whole of the web was text-only, so Tim created an image tag called `` especially for the band, whose place in the history of the Web was thus assured.

How to use the `` tag

Tim noticed two issues with the new `` tag. The first is similar to the `<a>` tag, you need to know what image to pull into the page. Tim didn't want to use the "href" attribute because that meant go somewhere else, whereas this was pulling the image into the page. To solve this, he made a new attribute called "src" — short for "source", where the image lives.

The second problem Tim noticed was that no other content can go inside an image — we can't put a paragraph or a header inside one for instance. So to solve this, he made the image a "single tag". Most tags

have an open and a close tag. Images just have one open tag and no close tag.

```

```

No close tag there and a brand new attribute called “src”.

You can use multiple images together and put them inside other tags too.

```
<p>  
  
  
</p>
```

To put in two images inside a paragraph, one with a cat image and another with a dog image.

Note these “src” attributes are using “relative” sources, just like in the link tags being able to use “about.html”. You can also use “absolute” sources too.

```

```

Image file types

There are only a few types of image file that we can use on the web. Certain image types are good for certain jobs. The file types are:

- `.jpg` JPG files are great for photographs and detailed color imagery.
- `.gif` Are they called “giffs” or “jiffs”? Who knows... but GIF files are good for animated images.
- `.png` PNG files are good for flatter color images and if you have any transparency in your image.
- `.svg` SVG stands for scalable vector graphics which means they don't pixelate when really large. Great for icons and logos.

White space + indentation

We mentioned earlier that HTML is all about content and CSS is all about style. Because of this, HTML doesn't really care about how your code looks, it will ignore any white space you put in your code and make it look “normal”.

For instance, this code...

```
<p>Hi there, Rik!</p>
```

...would look exactly the same as...

```
<p>  
    Hi  
there,    Rik!</p>
```

HTML ignores all the spacing and treats it as if it's one sentence. If we want to control how the website looks, we need to use CSS — more on that later.

Indentation

As HTML doesn't care about gaps, it means we can make our code cleaner to read so that when we come back to it in a few months, or if we're working with another coder, it can make more sense.

For instance, this is hard to read:

```
<p><a href="about.html">About</a></p>
```

So to make it easier, I would write it like:

```
<p>  
    <a href="about.html">About</a>  
</p>
```

Notice that not only have I put the `<a>` tag with some space above and below, but I've also indented the code so it goes into the page a little. I do this by pressing the tab key to push it further in.

This is just my personal preference. There's no right and wrong way to do HTML — both ways would look the same to a user.

Indentation is a powerful tool when you start getting more complex code.

One more example to make it more apparent. We could have code like this:

```
<p><a href="about.html">About</a> <a href="blog.html">
Blog</a> <a href="contact.html">Contact</a></p>
```

Simply one paragraph with three one-word links. We can clean it up like this:

```
<p>
  <a href="about.html">About</a>
  <a href="blog.html">Blog</a>
  <a href="contact.html">Contact</a>
</p>
```

All I've done is separate the links onto different lines, then indent (or tab) the contents of the paragraph. Nice, clean and looks the same.



CSS — Making Code Great Again

So far, we've been focusing on the content of our websites.

By default, the browser gives them a style that looks like the web would have looked in the late 1980s — a simple white background with black text in a default font. At the time this simple styling was fine for scientific research and basic websites, but as the Web became more popular, the Web's authors wanted to tweak the look and feel of their sites.

Over a few years between 1989 and 1996, to change the style, authors used plenty of “hacks” — non-standard ways to get around the limitations of the code they could play with. HTML started getting more complicated as more hacks were added, to the point where HTML started to look unusable to normal people.

Now, remember that HTML was made so regular folks could write it. The people that were making and updating the web browsers at the time decided enough was enough. They needed a new way to Make Coding Great Again, something for the people.

Back at CERN, one of Tim Berners-Lee's colleagues, a Norwegian called Håkon Wiem Lie, had an idea based on other word processing programs. The idea that if you were using a header or a paragraph in several places across a website, you probably want them to look

consistent. The idea that you could make the colors and typography consistent until you wanted to overwrite something later on.

Håkon had come up with the idea of Cascading Style Sheets (or CSS for short).

The cascading part of CSS is the idea that you begin to style your site on a general basis, then get more particular as you get into the nitty-gritty. For instance, you might want to make the typeface Arial across the whole site, except for headers where you prefer Georgia.

So it's always good to keep this in mind. Start off thinking about styling in the most generic way possible. What is the most used color? What is the most used typeface?

How does CSS work?

The first thing we do is similar to how we make HTML files. Instead of having a file that ends in “.html”, we just make one with “.css”. The name doesn't matter too much, whatever makes sense to you. I like to call my style sheets something like “style.css” so it's very obvious. You could break your files up into smaller style sheets later on, e.g. “about.css” and “homepage.css”, but it's personal preference.

The next thing to know is that CSS is not HTML. It looks different because it does a completely different job. It's not marking up content, it's describing what content should look like.

There are two main parts to CSS: selectors and rules. The selector picks which part of HTML you want to style, then the rules tell the HTML how to look.

Let's see an example:

```
h1 {  
    font-size: 48px;  
    font-family: Arial;  
}
```

The selector in this case is the word "h1" — we're picking all the <h1> tags across our website. We're then using an open curly bracket "{" to start selecting. There are two rules in the style, the first telling the font size to be 48 pixels large and the second telling the typeface to be Arial. We're then finished with styling so we use a close curly bracket "}" to tell the browser to stop styling.

How rules work is you have a defined list of what you can change about the styling (more on them later), then you add a colon ":" to say do this rule, then give it a particular value depending on the rule. We then finish the rule with a semi-colon ";" to let the browser know we're done and we can move on to the next rule — think of it like a period or full stop when we finish a sentence. Like this. Or. This.

Basic CSS selectors

In the previous example, we picked all the `<h1>` tags on the page by using the CSS selector “`h1`”. Take a guess at how we’d pick all the `<p>` paragraph tags on the page? It’s simply “`p`”.

```
p {  
    font-size: 14px;  
}
```

Here is our CSS picking all the paragraphs on our website and just applying one simple rule to them all, that they should all be 14 pixels font size.

Take a guess at how we’d pick all the links on the page? Well, we add links to the page with `<a>` tags, so to style them in CSS, we’d use:

```
a {  
    color: red;  
}
```

Here I’ve selected every link on the page, then added one rule to make the text color red. Notice to any one in the UK, Canada, Australia, etc., the word is “color” not “colour”. You’d think Englishman Tim Berners-Lee would have vetoed it, but no.

Body selector

Let's talk about the cascading part of CSS a little more. We could go through our website and apply styles to every single tag, but that gets a little laborious over time.

Håkon Wien Lie thought it would be better to do things once and then overwrite individually. To make a style apply across the whole website we can write things in the “body” selector.

```
body {  
    font-family: Arial;  
    font-size: 16px;  
    color: black;  
}
```

This CSS style means that by default, everything in the “body” of our web pages (e.g. the visual content of the page) would get these styles applied to them. Every tag would have the typeface Arial, the font size 16 pixels and the text color black until you overwrite them with another style.

Multiple and overwriting styles

In our style sheets, we're very likely to have multiple styles in the same file. The way we do this is just to put them on top of each other:

```
body {  
    font-family: Arial;  
    font-size: 16px;  
    color: black;  
}  
  
h1 {  
    font-size: 24px;  
}
```

In this example, we have two styles, one for the whole of the page, then one style just for the `<h1>` tags. To add a third, we can just do it at the bottom of the file.

What typeface will the `<h1>` tag be? We haven't explicitly said in the style sheet. The answer is it would be Arial because we said in the stylesheet that all the content should be Arial in the "body" selector. The styles get passed down into all tags until we overwrite them.

What font size would a `<p>` tag in our HTML be? Well, by default again we said the "body" selector has a size of 16 pixels, so everything in the HTML will be that size until we overwrite it. We have overwritten the default 16px font size in the "h1" selector by making our heading 24 pixels high.

Remember we can put tags inside other tags? What font size would an italic `<i>` tag inside a `<p>` paragraph tag be? Same as the paragraph — all the styles get passed down to the tags inside bigger tags.

One more question... if we have the style sheet:

```
body {  
    font-size: 16px;  
}
```

```
p {  
    font-size: 24px;  
}
```

What size would an `<i>` italic tag be inside a `<p>` tag be? In this case, the default font size for the page is 16 pixels, but then we overwrite the `<p>` paragraph tag and everything inside it to be 24 pixels, so the `<i>` tag would be 24 pixel font size too.

Units

Mostly in this guide, we'll be using pixel units. A pixel is a single square of color on a screen. Until recently, a pixel was made up of three sub-pixels — one red, one green and one blue. Each of the sub-pixels adds together to make different colors. However, more recently, retina screens have been able to double the amount of sub-pixels both down and across the screen, so on retina screens like the iPhone, there are actually 12 sub-pixels (four red, four green and four blue). There are other types of unit that we'll be using in the guide.

In CSS, a pixel unit is defined as a number (positive or negative) that ends with "px":

```
header {  
    font-size: 16px;  
}
```

Percentages

We can also use percentages in our CSS instead of pixel units. These are mainly useful in layouts:

```
section {  
    width: 75%;  
}
```


Em units

An em unit is a size relative to the current font size of that tag. By default, the font size would 16px, so we could use em units instead of px if we're unsure what pixel size to use but want to make it relative sized:

```
header {  
    font-size: 2em;  
}
```

By default, that would be twice 16px (32px). However, if we have something inside the header and use em units, it would be now based on 32px sizes:

```
header h1 {  
    font-size: 1.5em;  
}
```

The pixel size for this `<h1>` tag would be 1.5 times 32px (not 16px), so 48px large in pixel units.

Rem units

Similar to em units, rem units are based on the current font size of the page (not the current tag). In the last example, the header size would be

32px if we had “2em” or “2rem”, but the `<h1>` tag would be 24px if we were to use “1.5rem” instead of “1.5em”

vw, vh, vmin and vmax

We can base our sizes on the “viewport” — essentially the part of our page that the browser can currently see. If our unit is “1vw”, this is 1 percent of the viewport width, so if our browser is 1200px across, 1vw is 12px. “vh” is 1 percentage of the viewport height — so if we can only see 800px of our page in the browser, 1vh is 8px.

Sometimes we might not know how wide or tall our user’s browsers are, so using “vw” and “vh” units can work well. For instance, having a section at exactly the height of the browser could be useful for an intro to the page:

```
section {  
  height: 100vh;  
}
```

The “vmin” units look at the sizes of “vh” and “vw” and see which one of the two is the smallest. The “vmax” unit does the same, except looking for the larger of “vh” and “vw”.

Degrees

For some CSS rules, we might use angles, such as in background gradients or rotations. To use degrees we use “deg”. For instance, “5deg” is rotate from the top by 5 degrees clockwise. We can use the opposite direction by making it negative, i.e. “-5deg”.

Multipliers

Some CSS rules can also take multiplier numbers, such as line heights and scale. This would be just a single number, which might include decimal places. For instance:

```
body {  
    line-height: 1.5;  
}
```

This means make the line height (or leading), 1.5 times the current font size.

Zero is zero

Some students ask, what's the difference between using “0” and “0px” or “0em”. The answer is nothing — no pun intended — zero is zero no matter what unit you use!

Connecting HTML + CSS

Earlier, we were a little bit naughty and we skipped a bit. We didn't talk about the general structure of HTML files and we need to add a few more tags to our page to split our sets of tags into two. One set of tags is the content of the page that our users see, such as image tags, paragraph tags and header tags. The other set of tags is to do with any other information. The content tags are put in a `<body>` tag and the other tags are put in a `<head>` tag (note: not `<header>`). Then both of these tags are put inside an `<html>` tag:

```
<html>
  <head>
    <title>My website</title>
  </head>

  <body>
    <h1>Hey there!</h1>
  </body>
</html>
```

We've also added one more tag, the `<title>` tag, that lives in the `<head>` tag. The title isn't included within the page's content, but you'll see the phrase "My website" in your browser tab and for your page's headline in Google results.

To link up our HTML file with our CSS file, we need to add one more tag to our `<head>` tag because it's not part of our content. If our CSS file is called "style.css", we need to add a `<link>` tag:

```
<html>
  <head>
    <title>My website</title>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Hey there!</h1>
  </body>
</html>
```

We can add multiple stylesheets on the same page by adding another `<link>` tag:

```
<html>
  <head>
    <title>My website</title>
    <link rel="stylesheet" href="style.css">
    <link rel="stylesheet" href="slideshow.css">
  </head>

  <body>
    <h1>Hey there!</h1>
  </body>
</html>
```

The order of the stylesheet is only important if you're overwriting styles in more than one file — the bottom file will be the one that overwrites styles in the top one.

For now, most of the tags we'll use will be within the body tag of our page.

Typography

Now we know how to use some basic styles, let's show some of the CSS rules for typography and how you use them.

Font sizing

We talked about this earlier but we use:

```
p {  
    font-size: 16px;  
}
```

to give the font a particular size. Notice there's no space between the number and the “px” — it's all one word.

Font weight and how to make fonts bold

To make a font bold, we give it a font weight:

```
p {  
    font-weight: 700;  
}
```


This number correlates to a typographic weight. There are essentially nine weights to pick from:

100	thin
200	extra light
300	light
400	regular
500	medium
600	semi bold
700	bold
800	extra bold
900	black

The default font weight is 400. Watch out, as not all typefaces have all the weights. If that is the case, your text will default to the nearest weight it can find (e.g. if there are only 400 and 700 weights and you select 900, it'll display as 700).

Typography and text

To change the typeface of the site, you can change the font's family:

```
p {  
    font-family: Arial;  
}
```

By default, there are not many fonts to pick from, just the standard ones such as Arial, Georgia, Helvetica and Times New Roman. We'll talk later about how to add web fonts into your site.

Make the font italic

To make the font italic:

```
p {  
    font-style: italic;  
}
```

To overwrite an italic font and make it normal again:

```
p {  
    font-style: normal;  
}
```

Leading or line heights

To change the ratio between the height of the lines between text, we change the line height:

```
p {  
    line-height: 1.5;  
}
```

The number is a ratio based on the font size, so for instance if the font size is 16 pixels, the space between each line would be 24 pixels (16 times 1.5). A tight leading would be around 1.2 and a looser leading would be around 1.6.

You can also use pixel sizes for this:

```
p {  
    line-height: 24px;  
}
```

But it's better to use ratios as you may change the font size later on.

Letter spacing or tracking

To change the distance between characters in text, we can change the letter spacing. To increase the spacing by one pixel per character, we can add:

```
p {  
    letter-spacing: 1px;  
}
```

To reduce the space, we can use negative pixels:

```
p {  
    letter-spacing: -1px;  
}
```

Aligning text

When it comes to aligning text, you're not really handling the typography but how the text works, so the rules look slightly different:

```
p {  
    text-align: left;  
}
```

There are four different values you can put in: left, center, right and justify (watch out for “center” not “centre”, British people!).

Underlining text and removing underlines

You might want to add underlines to your code:

```
p {  
    text-decoration: underline;  
}
```

You may notice that by default links have underlines, if you want to get rid of them, add:

```
a {  
    text-decoration: none;  
}
```

MAKING TEXT SHOUT

Sometimes you need to make your text shout out! We could just go back to our HTML and delete the content and replace with an upper case version, but there's no need to rewrite your HTML. Having upper case letters could be a branding style and therefore we could do it in CSS.

You can make all the letters upper case by adding:

```
p {  
    text-transform: uppercase;  
}
```

You could also make each first letter capitalized too:

```
p {  
  text-transform: capitalize;  
}
```

Putting them all together

Remember you can add several of these rules together in one style or across multiple styles:

```
p {  
  font-family: Arial;  
  font-weight: 900;  
  font-size: 16px;  
  line-height: 1.5;  
  text-align: center;  
}
```

The order of the rules doesn't matter as long as you don't include two of the same rule. If you do, the latter one will take effect. For example:

```
p {  
  font-size: 14px;  
  font-size: 16px;  
}
```

The font size will be 16 pixels as the first one gets overwritten by the second.

5

HTML + CSS
continued

Previously, we talked about basic HTML tags that are mainly used for text.

For instance, the headings and the paragraphs are generally used to contain text and media (like `` image tags).

Sometimes you want to collect a bunch of HTML tags together and style them separately. For instance, you might have a header on your page that looks different to the main content, or you might have content in your footer that you want to have a smaller font size.

To do this we can group our tags together using more HTML tags. We have a few specialized tags and then some more generic tags to do this. All of the tags we use are “block” tags — ones that fill the width of the page.

Meet the header tag

The first tag we have is the `<header>` tag and its job unsurprisingly is to hold content for a header. This could contain just text or other tags too. A good example of a `<header>` tag in action would be:

```
<header>
  <h1>Rik Lomas</h1>
  <p>Coder for SuperHi</p>
</header>
```

Inside our `<header>` tag, we've got a heading and a paragraph tag. The `<header>` tag itself has no styling whatsoever until we give it something, so at the moment it looks the same with or without it. The reason why we would add the tag is so we can style it with CSS.

How do we pick the `<header>` in CSS? Same as all the other tags:

```
header {
  font-family: Georgia;
}
```

This would mean that the `<h1>` and the `<p>` tags inside the header would be in different typefaces. Pretty useful and it stops us repeating ourselves.

The navigation tag

Whenever you're looking to add navigation to your site, the `<nav>` tag is the perfect one to add:

```
<nav>
  <a href="about.html">About</a>
  <a href="blog.html">Blog</a>
  <a href="contact.html">Contact</a>
</nav>
```

You can add a `<nav>` tag within a `<header>` tag or separately in the page — whatever makes sense to you. It also doesn't have to be at the top of the page, you could have multiple navigations on the page, so add as many as you like.

The footer tag

Unsurprisingly, the opposite end of a page to the header would be the footer. That has a tag too. And unsurprisingly this is the `<footer>` tag:

```
<footer>
  <a href="https://www.instagram.com">Instagram</a>
  <a href="https://www.facebook.com">Facebook</a>
  <a href="https://www.twitter.com">Twitter</a>
</footer>
```

And to style it up, we'd select it with the footer selector:

```
footer {  
    font-size: 14px;  
}
```

The section tag

Within the page itself, between the header and footer, you might want divide the page up into sections. To do this, there's the section tag:

```
<section>  
    <h2>Education</h2>  
    <h3>Durham University</h3>  
    <p>I went to...</p>  
</section>
```

To style this up in CSS, you'd use the "section" selector.

General block tag, the <div> tag

If you have part of a page that doesn't really fit as a header, section or footer, but you still want to divide it up somehow, there's a more

general “block” tag called the `<div>` tag, “div” being short for “division” or “divider”, and which doesn’t have any special meaning:

```
<div>
  <h3>References</h3>
  <p>Holly Holmes</p>
</div>
```

To style it up in CSS, use the “div” selector.

Search engine friendliness

If `<header>`, `<section>`, `<footer>` and `<div>` don’t actually style up the page, then why use them? Well, not only does it make it easier in the long term (more on this later), but search engines like Google want to know how you’re labeling your content. They’ll rank you higher in search results if they can better understand your HTML content, so it’s good to use `<header>` for all your header content and `<footer>` for your footer content.

Styling with color

With our newer layout tags in place, we can start moving on from just plain old typography. We can start adding more color and backgrounds to our web pages. We're going to start with two more CSS rules that we can use to style our pages. Observant readers may have noticed that we used a rule called `color` earlier, where we styled up a `<a>` link tag and make the text color red:

```
a {  
  color: red;  
}
```

There are several text color values that we can use. Most of them are just plain old color names like “yellow”, “white”, “black”, etc. But what if the color we want isn't just plain red — what if it's a pale reddy-orange color? We can't just put that into our style sheet. We need to add in a very particular color. To do this, we use a hex value.

Hex values

A hex value is short for a hexadecimal value. Sounds nerdy, right?

In everyday terms, we can count from 1 to 9 and after that we don't have any more single numbers left to use. The single numbers we can

use are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. There are 10 of them — this is called a decimal system (decem is Latin for 10).

By contrast, computers count from 1 to 9, then have more single numbers because they can use a, b, c, d, e and f as well. The number after 9 in computer counting is actually a. Weird, right? And what's after that? No surprise it's "b".

So the computer number system is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e and f.

What's after "f", the last "number" computers can count with? "10". What's after "19"? Say hello to "1a"! What's after "1f"? "20".

So the human version of 10 and the computer version of 10 are two different numbers — humans would have counted from 1 to 17. So the number system a computer uses isn't "decimal" but "hexadecimal" ("hexa" + "deca" are Ancient Greek for 6 and 10).

As computers count in these hexadecimal numbers, our colors are represented by six of these hexadecimal numbers, which looks like:

```
#ff0099
```

The first character, the "hash", is just to denote that this is a color. The next six are what the color is.

The first two ("ff") represent the red channel, the second two ("00") represent the green channel, and the last two ("99") represent the

Hex Values

f f 0 0 9 9

RED GREEN BLUE

The diagram illustrates the hex color #ff0099. The hash symbol # is black. The first two digits 'ff' are red, the next two '00' are teal, and the last two '99' are blue. Brackets below the digits group them into three pairs: 'ff' for RED, '00' for GREEN, and '99' for BLUE.

blue channel. Most computer screens use red, green and blue light to mix colors together.

The minimum value for each one is “00” (fully off) and the maximum value is “ff” (fully on as “f” is the top number). The mid point is around “88” as 8 is halfway between “00” and “ff” — not 0 and 9.

It might be hard to know what exact color this represents, but we can take a rough guess. If for the #ff0099 value, we can tell that the red is fully on, the green is fully off and the blue is about half way on, we’d know it’d be roughly two parts red and one part blue, so a pinky purple color.

There are a lot of online resources for picking colors (see the resources section at the back), but one I particularly like can be found here:

<https://color.hailpixel.com>

The SuperHi Editor also has color pickers available in its style sheets so you can pick colors within the editor itself. No need to go searching online.

Hex and text

So now with our brand new knowledge of hex values, we can add it into our CSS rules in the following way:

```
a {  
  color: #ff0099;  
}
```

Remember you can change the colors across whole tags or even whole pages:

```
body {  
  color: #333333;  
}  
  
header {  
  color: #ff4141;  
}
```

This will make the whole page have a text color of #333333, which is a dark grey, except for the header and all the tags inside the header, which are a warm red.

Background colors

For changing the background color of the tags, you need to use a new rule called background-color:

```
footer {  
  background-color: #998899;  
}
```

This will give the footer a background color of mid grey.

Remember you can mix and match your text and background colors:

```
section {  
    background-color: #111118;  
    color: #ffffff;  
}
```

This gives our section tags a background color of near black and a text color of white.

Background images

Some of the time when designing your website, you might not want to use a solid background color. The first thing you need to do is make sure you have some images in your site that you can pull into the site. Remember you can only use particular image types that browsers can use, such as “.jpg”, “.gif”, “.png” and “.svg” files.

To add an image saved as “sally.jpg” into the background of a tag:

```
header {  
    background-image: url(sally.jpg);  
}
```

Notice the “url(...)” around the file name. This is so the browser knows to pull it from a URL into the page. There are other types of background images we can add, such as linear gradients, that we’ll come to later.

Background-repeat

By default, if your image is smaller than the tag of which it’s the background, the image will tile. You can control the way it repeats, or turn off the tiling altogether.

To turn off tiling:

```
header {  
    background-repeat: no-repeat;  
}
```

To only repeat horizontally across the page:

```
header {  
    background-repeat: repeat-x;  
}
```

It's repeat "x" because the "x" direction in mathematics is horizontal (or from left to right).

To only repeat vertically down the page:

```
header {  
    background-repeat: repeat-y;  
}
```

If "x" is across the page, the "y" direction in mathematics is vertical (or from top to bottom).

If you ever want to put the tiling effect back on:

```
header {  
    background-repeat: repeat;  
}
```

Background-position

Now you've added your image and controlled how it tiles, you want to control where it's placed within the tag. To do this you use "background-position".

This takes two values, the first being the "x" direction (across the page) and the second being the "y" direction (down the page).

If you want to move it 20px across the page, then 40px down the page you can do:

```
header {  
    background-position: 20px 40px;  
}
```

There are some cheats if you want to do things like place it in the top, left, bottom, right or center of the tag. For instance, if you want to place it on the right hand side in the center you can add in:

```
header {  
    background-position: right center;  
}
```

If you want to place it right in the middle of the tag, you can put it in the center horizontally and in the center vertically:

```
header {  
    background-position: center center;  
}
```

You can also use percentages, if you want to be specific. For instance you can move it 80 percent across and 20 percent down by adding in:

```
header {  
    background-position: 80% 20%;  
}
```

Background-size

You can also control how the background image is sized by using the `background-size` rule. This also takes two values — the horizontal size (“x”) then the vertical size (“y”) — just like with `background-position`:

```
header {  
    background-size: 200px 100px;  
}
```

This will make your background image 200px across by 100px down.

Blurry backgrounds and retina screens

If you're on a retina screen, for instance on a new iPhone or a new MacBook, you may notice your background images looking blurry by default.

Non-retina screens have three sub-pixels per pixel (one red, one green and one blue sub-pixel, remember the hex colors!). On retina screens, to make the screen look crisper, they have 12 sub-pixels per pixel (four red, four green and four blue).

With images, each pixel always has three sub-pixels, so when you view a “normal” image on a retina screen, it’ll look a bit blurry. To fix this we need to double the size of the image to give it more definition on those screens.

One good technique to do this is, if we’re making an image that would fill 300px by 200px on a retina screen, we export the image from Photoshop or Sketch as twice the size (600px by 400px), then in our CSS, we make it half the size again to give it a higher density of pixels:

```
header {  
  background-size: 300px 200px;  
}
```

Exporting the image at double size and halving the size in CSS will stop any blurring.

Stretch the background

There are some special values in the background-size rule we can use if we don’t know the specific size the image is meant to be.

Sometimes we might want to stretch our background image to completely fill the tag's background at any size, as we might not know how wide the header is or how high the section tag.

If we're not too precious about the edges of the image, we might want to crop some of the background image. To do this, we add in:

```
header {  
    background-size: cover;  
}
```

If we do want to keep the whole image with no cropping, but resize to fit, we can add in:

```
header {  
    background-size: contain;  
}
```

Both “cover” and “contain” are special values for background-size that we can use to great effect.

Parallax effect by fixing the background to the browser

Usually when we place a background image, it stays with the tag when we scroll up and down the page.

In some circumstances, we might want to attach the background image to the browser itself, rather than the tag. Why? Sometimes we might want to add in a “wipe” effect on the background. Some designers call this the parallax effect. It’s a pretty basic version of parallax but it can be highly effective.

To add it in we use a new rule called “background-attachment” as we’re changing where the background is attached (the browser rather than the tag):

```
header {  
    background-attachment: fixed;  
}
```

Combining them all together

We can combine all the background rules together to get some decent effects:

```
header {  
    background-color: #9900ff;  
    background-image: url(field.jpg);  
    background-repeat: no-repeat;  
    background-position: center top;  
    background-size: cover;  
    background-attachment: fixed;  
}
```

This will give us a background which is the image “field.jpg”, not repeated, positioned in the top center, stretched and fixed to the browser scroll.

Background gradients

One last thing we can add into our backgrounds is a gradient. Weirdly this doesn't use “background-color”, as that can only take solid colors. We have to use “background-image” to do this. Bit of a strange one, but hey, that's what we were given, so let's do it.

To keep this simple, we're going to use the default colors like “red” and “yellow” rather than hex values. We can use hex values in our code though.

Let's say we want to add a gradient of yellow to red that goes across the page, we can add it in as:

```
header {  
    background-image: linear-gradient(90deg, yellow, red);  
}
```

Bit more complicated than usual, but it should read fairly well. We're adding a background image that isn't a URL as usual, but a linear gradient that goes at a 90 degree (or “90deg”) angle from left to right, starting at yellow and ending at red.

The commonly used degrees for web design are:

<i>0deg</i>	from top to bottom
<i>90deg</i>	from left to right
<i>180deg</i>	from bottom to top
<i>270deg</i>	from right to left

We can add multiple colors into our gradient. Let's say we want to have a gradient that starts at the top as red, goes to pink, then goes to white at the bottom:

```
header {  
    background-image: linear-gradient(0deg, red, pink,  
    white);  
}
```

Remember we can be more particular with our colors by using hex values instead of default colors:

```
header {  
    background-image: linear-gradient(30deg, #000000,  
    #666666);  
}
```

Where to use HTML `` tags or CSS background images?

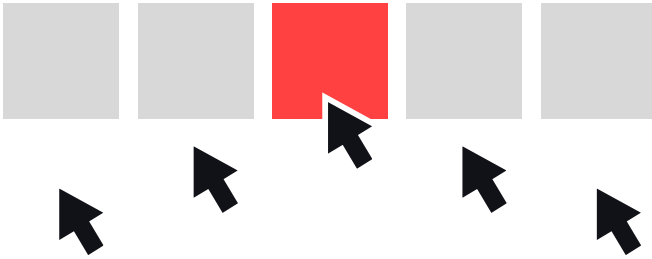
One question that's trickier to answer is, what kind of image should we use when we're designing our sites? We talked about the `` tag that we can use in HTML earlier in the guide, but now we've introduced the `background-image` in CSS. Which should we use? It depends.

The way that I like to answer the question is to ask: "Does the image form part of the content of the website, or does it form part of the design?". For example, let's say we have an online store, I would say that the image of the products are part of the content of the website, as our customers would expect to see what they're buying, whereas a pattern on the header of the site isn't part of the content, but would be part of the design. If we redesigned the site in a year, I doubt customers would require it before they made a purchase.

Another way to think about this issue would be to consider if you'd want the images to appear in a Google image search. Google will look at the `` tags on your website because they're content of your site, but not at the background images in the CSS because they're not content.

Again, there's no right or wrong answer to this. Remember that HTML and CSS are design tools, it's up to the author how they use them. Whatever works best for you is the real answer.

Hovering a tag with a mouse cursor



Hover states

So far, the way we've selected our HTML tags in CSS has been pretty straightforward — whatever the tag name is, that's the CSS selector. If we're picking all our `<a>` tags, we select them with "a", if we're picking all our `<section>` tags, we select them with "header", but what if we're doing something more complex? How about if we want to style them only when a user hovers the tag? We're going to introduce the hover state — a new CSS selector — that styles tags only when a user hovers it.

Let's say we want to style up our links on the page. We want to make the default look a blue color with no underline:

```
a {  
    color: #2727e6;  
    text-decoration: none;  
}
```

Whenever a link has no underline, it might make it more difficult for users to notice that this link is a link — without the underline they may just think it's another bit of text. Let's make any of the links change color when we hover them.

To select an `<a>` tag on hover, we need to select the "a" first, then only on hover. To do this we use the selector "a:hover". If we want to select a `<section>` tag on hover, we'd use "section:hover". No space between the colon and "hover".

In our CSS, we'd need two styles, the first being the default from above — this is what it looks like normally. Then we need a second style to overwrite what we want to change — in this case, from blue text to black text:

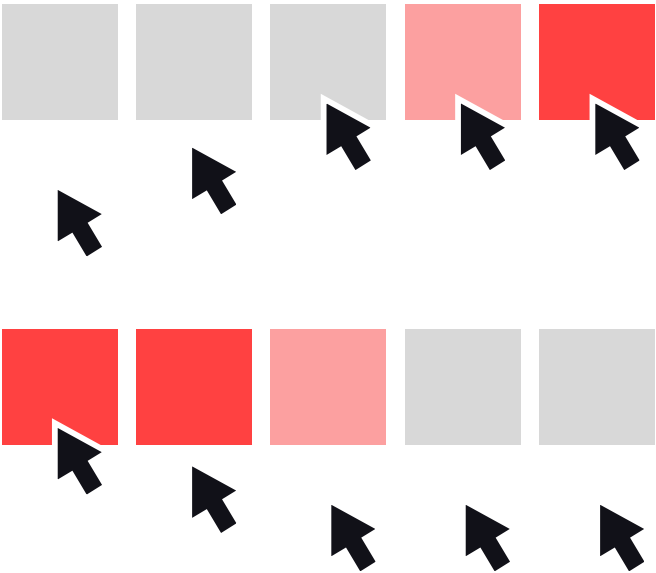
```
a {  
  color: #2727e6;  
  text-decoration: none;  
}  
  
a:hover {  
  color: #111118;  
}
```

Notice in the second rule that we don't need to repeat the text decoration rule. We only need to overwrite the things we want to change.

In some circumstances, we might want to overwrite a default that isn't there. For instance, we might want to add a background only on hover — the default styling would just be transparent and then the hover state would be a background color:

```
header {  
  color: red;  
}  
  
header:hover {  
  background-color: yellow;  
}
```


Timeline of a hover effect with a transition



In this case, on our header the default styling would be a red text color and no background color, then on hover, we are overwriting the background color to be yellow. The text color stays red on default and on hover.

Transitions

At the moment, whenever we change between states of default and hover, we get an instant flick of change. In a lot of instances, this can look jarring to users, especially if the user is hovering over a large area. Wouldn't it be great if we could add a fade effect? To do this, we can add a CSS transition.

Let's say we want to fade our link's text color from blue to black again, but this time we want to fade it between the two colors over half a second:

```
a {  
    color: #2727e6;  
    text-decoration: none;  
    transition: color 0.5s;  
}  
  
a:hover {  
    color: #111111;  
}
```

Here we've transitioned the color rule over 0.5 seconds ("0.5s").

One question you may have is, "why did we put the transition rule into the default style rather than the hover style?" Surely we want it to fade when we hover, right? Yes we do! But we also want the fade to happen when we move away from the tag, so we put the transition in the

default style to have it in all instances — in this case, both hovering over the tag and moving away from the tag with our mouse.

What if we want to transition more than one thing? Let's say a background color and a text color? Instead of the name of the attribute (e.g. "color"), we can just say "all":

```
header {  
    color: black;  
    background-color: white;  
    transition: all 2s;  
}  
  
header:hover {  
    color: red;  
    background-color: yellow;  
}
```

In this case, we're changing both the header's background color and the text color over two seconds ("2s").

Child selectors

Let's look at some typical HTML:

```
<header>
  <a href="index.html">Rik Lomas</a>
</header>

<nav>
  <a href="index.html">Home</a>
  <a href="contact.html">Contact</a>
  <a href="blog.html">Blog</a>
</nav>
```

It's pretty normal to have links in a few different places on a page. Here we have one link in a `<header>` tag and three in a `<nav>` tag. But in our design, our header could look vastly different to our navigation, so that could mean our header's links could look vastly different to our navigation's links too. How do we pick and choose the correct styles?

Remember, CSS stands for cascading style sheets — the cascading bit means we can go into tags and overwrite styles when we need to. We can also use the cascading part of CSS to select particular tags too.

Let's say we just want to select the header's links. We would first of all select our `<header>` tag. Within the `<header>` tag we would then want to just select our `<a>` tags. The way we do this in CSS is by using a space between each selector. We select "header", then we select "a":

```
header a {  
  color: red;  
}
```

If we want to select all the links within the navigation, we'd select the “<nav>” then the “<a>” tags inside the navigation:

```
nav a {  
  color: blue;  
}
```

It's quite common to do this in CSS, you'll see selectors that look like “section p”, “header img”, “footer a” and more.

Adding hover states to child selectors

Sometimes you'll want to do tasks like when a user hovers just the links inside the header. We have all the tools we need to do this, we just need to combine them.

When we want to do a hover state on any link we would do:

```
a:hover {  
  color: blue;  
}
```

And if we want to select just the links inside the header, hovered or not:

```
header a {  
    color: red;  
}
```

To combine them together, we would do:

```
header a:hover {  
    color: white;  
}
```

In this example, we're first of all picking the `<header>` tag. Next we want to go inside the header tag so we add a space. Then we're picking the link in a hovered state — no spaces in the selector here!

Classes

We're going to add one last level of complexity to our selectors. I promise it'll be the last bit of complexity you'll need in a selector for 99.9% of whatever you do.

Let's look at some more HTML:

```
<section>
  <h2>Experience</h2>
</section>
```

```
<section>
  <h2>Education</h2>
</section>
```

```
<section>
  <h2>References</h2>
</section>
```

Most of the time I want to make both sections look the same — the same typography, the same color scheme, the same spacings. But let's say we want to change one of the sections very, very slightly. I want to change the Education section to have a different background color to bring attention to it.

None of the selectors I've used so far make sense to use, as using "section" in CSS would change all of the <section> tags. I don't want to do anything with hover either.

One way around it could be to change the `<section>` tag to another tag, but it would need a lot of duplicate code in my style sheet to do two tags looking mostly the same.

The more sensible and more common way is to use a “class” attribute.

We want to add a hook into our HTML to say we want to style this up in a particular way. The hook we'll add doesn't add any styles until we style it up in CSS.

How to add class attributes in HTML

We talked about attributes earlier in the guide when we talked about the “href” attribute for link tags and the “src” attributes for image tags. Attributes are extra information for the web browser to use — for “href” it's for the browser to know where to send the user next, for “src” it's to know which file to pull into the page.

With the “class” attribute, we're adding a hook into our HTML for our CSS to use later on.

To add a class attribute, we add the extra information on to the opening HTML tag:

```
<section>
  <h2>Experience</h2>
</section>
```

```
<section class="highlighted">
  <h2>Education</h2>
</section>

<section>
  <h2>References</h2>
</section>
```

In this example, we've added a class attribute called "highlighted" to the middle tag. We can call the attribute whatever makes sense to us to use. Remember, the user doesn't see the name of this. The only rules you have to remember is no spaces and it has to be letters, numbers and dashes.

If a user looked at the web page at the moment, it would look exactly the same as it did without a class attribute. It doesn't do anything until we style it.

You can reuse the class attribute on multiple tags too, for instance:

```
<section class="highlighted">
  <h2>Experience</h2>
</section>

<section class="highlighted">
  <h2>Education</h2>
</section>
```

```
<section>
  <h2>References</h2>
</section>
```

Here, I've added the class attribute of "highlighted" on to the first two tags but not the last tag.

Adding CSS class selectors

So far we've added some class attributes into our HTML but they're unused at the moment, so let's hook some CSS up to them.

In our CSS, if we want to style up all three section tags, we would do so like this:

```
section {
  background: white;
  color: black;
}
```

However, if we then wanted to overwrite the background color of the sections with the class attribute "highlighted", we would need to select them separately to all the sections. To do this, we would pick all the `<section>` tags, then filter them by their class using a "." (period or full stop), then the name of the class attribute:

```
section {  
    background: white;  
    color: black;  
}  
  
section.highlighted {  
    background: yellow;  
}
```

Our highlighted sections have a yellow background color, and because we're not overwriting the text color, this is black from the default section style.

We can use the class attributes in lots of different ways. For instance in our HTML we could have:

```
<p class="intro">  
    In this example...  
</p>  
  
<p>  
    However, you may notice...  
</p>
```

Then to style up the `<p>` tag with the class attribute, we could do in our CSS style sheet:

```
p.intro {  
    font-size: 18px;  
    font-weight: 700;  
}
```

Combining class attributes with hover states

Okay, just one more complicated selector. I promise.

In the last example, where we had a `<p>` tag with the class attribute of “intro”, what if we wanted to do something when a user hovered the `<p>` tag? We’d combine the selectors:

```
p {  
    font-size: 16px;  
}  
  
p.intro {  
    font-size: 18px;  
    font-weight: 700;  
}  
  
p.intro:hover {  
    color: red;  
}
```

Here we have three styles. The first is picking any `<p>` tag. The second is picking any `<p>` tag, then filtering for any with the class attribute of “intro”. The last one is picking any `<p>` tag, then filtering for the class attribute of “intro” and then only applying the style on hover. Phew!

So for this style sheet, the tag with `<p class="intro">` would be font size 18px and bold font weight (or 700 weight) by default. Then we apply the red color only on hover.

Multiple classes

We can also add more than one class on to a tag. The way we do this is by using a space between the two class names:

```
<p class="highlighted large">  
  I'm a large and highlighted paragraph  
</p>
```

We can now select this style by using either class name. Both “p.highlighted” and “p.large” will select this tag. We could also be incredibly particular and select any paragraphs with both classes by adding to our CSS:

```
p.highlighted.large {  
  font-size: 24px;  
  background-color: yellow;  
}
```

Widths and heights

So far, we've been able to change the colors, backgrounds and typography of our web pages. In the next few chapters, we'll move on to the layout of web pages. As we're talking about how our site looks, rather than the content or the actions, we'll be doing all the layout in our style sheets.

One of the easier things to pick up in terms of layouts is how to size things. We can add widths and heights to any block tag (<header>, <div> and <p> are examples of block tags) but not inline tags (<a>, and <u> examples of inline tags). We'll talk later in the book about how to give inline tags width and height, but for now, we'll concentrate on block tags.

To add width to any block tag, we can add in:

```
p {  
    width: 500px;  
}
```

This style will make any paragraph tag 500 pixels wide. We can add different units into this, not just pixel units. For example, if we wanted our paragraph to be half its current width:

```
p {  
    width: 50%;  
}
```

Unsurprisingly to add heights, we would add:

```
header {  
    height: 100px;  
}
```

For mobile-friendly designs, we can also add in a minimum and maximum width to each:

```
header {  
    min-width: 500px;  
    max-width: 1000px;  
}
```

And we can do the same for heights too:

```
header {  
    min-height: 300px;  
}  
  
footer {  
    max-height: 500px;  
}
```

We'll talk more about mobile-friendly web design later in the guide, where we'll use some of these rules.



**Let's get
mathematical**

Almost all of HTML, CSS and Javascript has nothing to do with mathematics.

Beginners might assume there are a lot of numbers being crunched, but in fact the figures we really use are based in the design of web sites. Here we're going to look briefly at two important mathematical principles that we use in web design and therefore coding.

Ratios

Humans are born with a natural instinct to prefer patterns and sizes that are based on mathematical ratios. Musical notes sound pleasant together because they have a nice mathematical ratio — the notes C and G for example have a 3:2 ratio in sound frequency. Photographs usually have aspect ratios such as 16:9, 3:2, 4:3, or in the case of Instagram, 1:1. These are not picked at random — the human eye just prefers to look at things bearing these ratios.

We can use the idea of ratios in our web site designs too — mainly in our CSS, as this is where we are working with the look and feel of the sites. Let's say we have a site that has a default font size of 14px:

```
body {  
    font-size: 14px;  
}
```

It might be tempting to have titles in font sizes like 20px or 25px, which are nice round numbers, but the ratio of 25:14 or 20:14 (which factors down to 10:7) isn't as nice of a ratio as 21px, which factors down from 21:14 to 3:2, just like in the musical notes and photograph aspect ratios we mentioned earlier:

```
h1 {  
    font-size: 21px;  
}
```

Other font sizes that might work well would include 28px (2:1), 35px (5:2) and 42px (3:1). With a default font size of 16px, nice ratios we could use include 24px (3:2), 32px (2:1) and 40px (5:2). Notice how the sizes are close to the recommendations above, but visually they will make a lot of difference.

Dividing numbers

Some numbers are used pretty often in web design. You may see the numbers 360, 640, 960 and 1128 appear quite regularly, but why?

Mathematically, these numbers divide into lots of other rounded numbers. For instance, 960 divides nicely into 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, etc, whereas 980 or 940 don't divide as neatly into lots of other numbers. It's one of the reasons you see a lot of web pages which are 960px wide because it's easier to make visually nicer layouts when you can divide in plenty of ways.

Margins and paddings

A lot of web design in our pages will use spacing — the idea that there will be some distance between different tags, or there's some buffer around the tag.

If we want to move tags away from each other, we'll use a new CSS rule called “margin”.

If we want to add a buffer around the tag, we'll use a new CSS rule called “padding”.

Margins

Let's say we want to add a 20 pixel space at the top and bottom of all our paragraphs so our content is easier to read. We want to push other tags away from each individual paragraph tag:

```
<p>  
    Going freelance means...  
</p>
```

```
<p>  
    The first step is...  
</p>
```

```
<p>  
    I'm not a fan...  
</p>
```

For our users, we want there to be a gap between the first `<p>` tag and the second `<p>` tag, then a gap between the second `<p>` tag and the third `<p>` tag. If we add a fourth `<p>` tag later, we want there to be a gap too.

What we need to do is add a margin. In CSS, our margin takes four values: top, right, bottom, left. We start at the top and go clockwise around the tag. Some people remember the order as “TRouBLLe” too (the capitals TRBL for the directions).

For our paragraphs, we want 20 pixels at the top and 20 pixels at the bottom. We're not too fussed about the left or right having a margin so we can make them have zero margin:

```
p {  
    margin: 20px 0 20px 0;  
}
```

We have four values in our margin rule, all separated by spaces.

If we decided that in fact we did want the left and the right sides to have the same margin as the top and bottom, we'd then have:

```
p {  
    margin: 20px 20px 20px 20px;  
}
```

This can be shortened to just:

```
p {  
    margin: 20px;  
}
```

Having one value means all sides will have the same margin.

One thing to remember with margins is that the gap between two tags is an overlap not an addition. For example, if we have two paragraphs, each with 20 pixels margin, the gap between them is 20 pixels, not 40 pixels.

However if we have a header with a 30 pixel margin next to a paragraph with just a 20 pixel margin, the gap between would be 30 pixels, as whichever is the larger will push further.

Paddings

Margins push other tags further away from each other, but some of the time you might not want this to happen, but instead increase the size of the tag:

```
header {  
    background-color: red;  
}
```

In our style sheet, we have a `<header>` tag with the background color red. The background will fill the space that the content fits into — it will stretch across the width of the tag and fill vertically, depending on the amount of content inside the tag. If we want to add a buffer around the content of the tag to make the red background color extend, we can use the padding rule.

This rule uses the same format as the margin rule — four values to go clockwise from the top to denote the top, right, bottom then left (remember “TRouBLLe”):

```
header {  
    background-color: red;  
    padding: 50px 100px 50px 100px;  
}
```

In this rule, we are now making the background color have a buffer around the edge of any header tags.

Margins vs paddings

Hi there,

we only have

margins between

our tags



We only

have padding

between tags



We have

both padding

and margin



Again we can use just one value if we're using the same padding around every side:

```
header {  
    background-color: red;  
    padding: 50px;  
}
```

I personally prefer to use all four values for both margins and paddings when I'm making web sites. A lot of web design is trial and error, so I'm constantly tweaking the layout, and it makes sense for me to tweak with four values rather than have to switch between four and one.

Combining margins and paddings

Margins and paddings aren't a battle of "this or that", you can use them both together in a lot of circumstances:

```
<p>  
    Going freelance means...  
</p>
```

```
<p class="highlight">  
    The first step is...  
</p>
```

```
<p>
    I'm not a fan...
</p>
```

I've taken the same HTML as before and added a class of "highlight" to the middle paragraph.

I want all my paragraphs to have a gap between them — 20 pixels on the top and bottom, 0 on the left and right.

With the middle paragraph, I want this to increase the gap between the other two, then have a yellow background with a buffer around it:

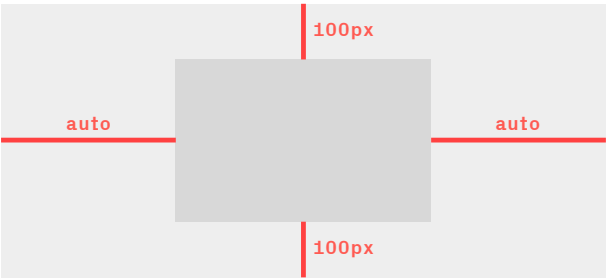
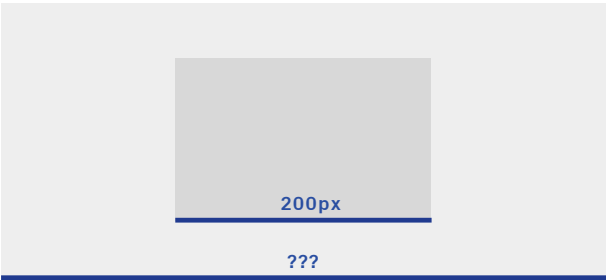
```
p {
    margin: 20px 0 20px 0;
}

p.highlight {
    margin: 30px 0 30px 0;
    padding: 40px 40px 40px 40px;
    background-color: yellow;
}
```

By using a class attribute, I've increased the margin to 30 pixels at the top and the bottom on the highlight `<p>` tags, then added a 40 pixel buffer around the whole tag with a yellow background on it.

For tags without a background, the visual difference between margin and paddings will be pretty much the same, either you're extending a see-through background or you're pushing away by that distance. In the

*Automatic margins on the
left and right of a tag*



next section we will talk about borders which will make a difference to which you pick.

Automatic Margins

Margins combined with widths are good for pushing block tags like `<header>` and `<div>` into the middle of other tags or the whole page, but sometimes we may not know how much margin we need.

For example, if we want the whole page to have a set width and to be in the center of the browser, we won't know how much margin the whole page needs on the left and on the right. A user's browser could be 900 pixels or 1900 pixels wide, so how do we tell the browser to add a margin that we don't know the size of? We use the word "auto".

Let's say we want the page to be 800 pixels wide, we can move it into the middle by combining the width and the left and right margins:

```
body {  
    width: 800px;  
    margin: 0 auto 0 auto;  
}
```

Unfortunately, "auto" only works for the left and right margin and it's a bit trickier to move content vertically into the middle of page — we'll talk more about that later.

Borders

To add borders to our tags, we can add a border rule in CSS. A border takes three values: a width, a border style and a color.

The width of the border is similar to the pixel values that we've been using so far. A thin border would be 1px, whereas a thicker border could be 10px.

The border styles we can use are dotted, dashed and solid. Almost all the time we'll be using a solid one.

To add a thin solid white border to our `<header>` tag, we add:

```
header {  
    border: 1px solid white;  
}
```

If we want to add a thicker solid bright yellow border to our `<p>` tags, we can add:

```
p {  
    border: 10px solid #fff182;  
}
```

Specifying borders

You may not want borders around every part of the tag. You may just want a border on the left or on the bottom. You can be more specific by using the `border-top`, `border-right`, `border-bottom` and `border-left` rules:

```
p {  
    border-bottom: 1px solid #eeeeee;  
}  
  
header {  
    border-left: 5px solid #ff4141;  
}  
  
footer {  
    border-right: 10px solid white;  
    border-top: 1px solid #9900ff;  
}
```

In our example, we're giving our `<p>` tags a thin light grey bottom border, our `<header>` tags a thicker solid red left border, then our `<footer>` tags a thick white right border and a thin purple top border.

Using borders for underlined links

One of the more common ways to use borders is for underlines on links. Usually a link's underline is a thin 1px border which is the same color as the text. By turning off the default underline (using the `text-decoration` rule) and replacing it with a border on the bottom, we can have a lot more control over how the underline looks.

We can even use transitions to make the border fade between two colors on hover:

```
a {  
    text-decoration: none;  
    border-bottom: 2px solid white;  
    transition: border-bottom 0.5s;  
}  
  
a:hover {  
    border-bottom: 2px solid red;  
}
```

Sometimes we want to fade in a border from nothing on hover. We would want to transition between a border with no color but a width, to a border with color and the same width:

```
a {  
  text-decoration: none;  
  border-bottom: 2px solid transparent;  
  transition: border-bottom 0.5s;  
}  
  
a:hover {  
  border-bottom: 2px solid red;  
}
```

The box model — borders, paddings and margins

If we combine borders, paddings and margins, the border will go around the buffer zone of the padding but not change with the size of the margin.

If we start from the very outer edge of the tag, we start by applying the margin, then add the border, apply a background if there is one, then add in the padding's buffer zone, and then we have our actual content inside that.

The technical name for this is the box model — it's how the boxes (or tags) of HTML are styled up and in what order.

Rounded corners with border radius

Most of the web design we've done so far has been in rectangles and squares, but sometimes we want to add rounded rectangles and circles.

Let's say we want to make our `<header>` tag have slightly rounded corners, we can use the CSS rule `border-radius` to round the corners of our tag. Most of the time we would want our tag to have all rounded corners the same, so all we need to do is add one size value:

```
header {  
    background-color: red;  
    border-radius: 10px;  
}
```

In rarer cases, we might want to have all four corners rounded differently. Similarly to paddings and margins, we can add in four values. These values start in the top left corner and go around clockwise to the top right, bottom right then bottom left:

```
header {  
    background-color: red;  
    border-radius: 10px 0 0 20px;  
}
```

This example will give a rounded corner to just the top left (10px) and the bottom left (20px).

Making a circle

Let's say we have a `<div>` tag and we give it a set height of 100px and a width of 100px. Without any border radius, we just have a square. When we start increasing our radius, the corners get further and further rounded. Once we hit 50px, half the height and half the width, our corners are so rounded that the whole tag will look circular:

```
div {  
  width: 100px;  
  height: 100px;  
  background-color: red;  
  border-radius: 50px;  
}
```

If we try an even higher value, there's no more corner to round, so it will still look like a 50px border radius, even if it says 200px.

Filters

Recently Adobe started adding some of their own code to how CSS works to let us use some of the powerful Photoshop filters in our browsers. Because they are so new they may not work in some older browsers. For 90 percent of users they will work fine, but for those using older computers it may look as if these filters were never there. So for the moment, use sparingly until everyone else catches up (or buys new laptops).

We're going to add a new rule called `filter` to our CSS. The first one lets us blur tags. Blur tags have a pixel unit, the larger the number, the more blur we have:

```
img {  
    filter: blur(2px);  
}
```

We can also fade our image to gray by increasing another filter called `grayscale`:

```
img {  
    filter: grayscale(100%);  
}
```

The default for this is 0% `grayscale` (so normal), but we can fade to gray by increasing the percentage to 100%.

Instead of gray, we could fade our image out to an antique sepia effect by using a similar filter:

```
img {  
  filter: sepia(50%);  
}
```

Similar to grayscale, we can increase the saturation of the image too:

```
img {  
  filter: saturation(200%);  
}
```

100% is the default saturation and anything less than 100% will reduce the saturation in the same way that grayscale works.

We can also change the contrast of the image in a similar way to saturation:

```
img {  
  filter: contrast(500%);  
}
```

This will increase the contrast by five times the original. We can also change the brightness of the image:

```
img {  
  filter: brightness(200%);  
}
```

We can invert the color of the tag too by adding an invert filter:

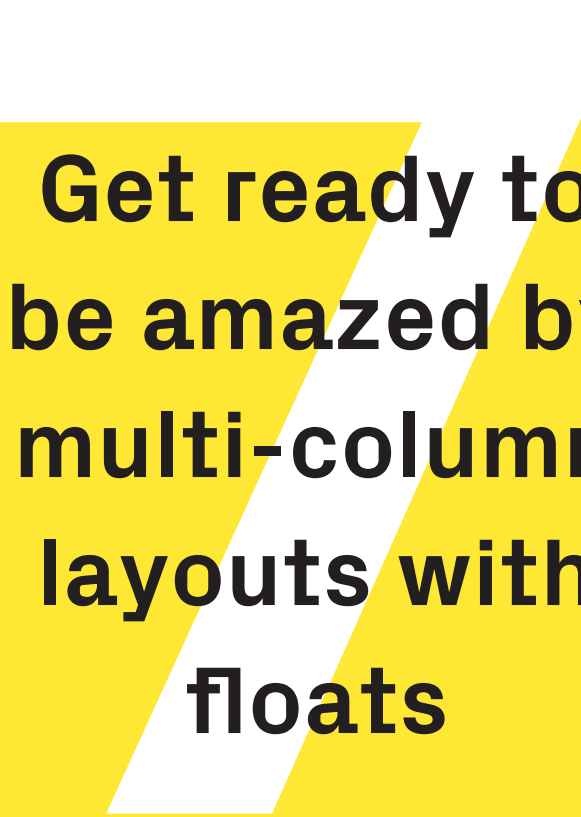
```
img {  
    filter: invert(100%);  
}
```

We can also combine filters:

```
img {  
    filter: contrast(200%) grayscale(100%) blur(3px);  
}
```

Where filters can come in really useful is in hover effects, for instance if we wanted to fade an image from gray into full color:

```
img {  
    filter: grayscale(100%);  
    transition: filter 1s;  
}  
  
img:hover {  
    filter: grayscale(0);  
}
```



**Get ready to
be amazed by
multi-column
layouts with
floats**

We'll be talking about three different ways of making layouts in the next few chapters: floats, position and display.

Floats are among the most commonly used layout tools in CSS. They're in pretty much every web page that you'll see, so they are an important part of going from one-column pages.

Floats weren't originally intended for layouts, they were actually intended for wrapping tags to the left or right of other content or tags, to create something more like a newspaper layout.

Let's explain this original intent first before we talk about using floats for layouts.

Wrapping elements using floats

Let's imagine our HTML is the following...

```
<p>  
    
  Today, the president was very happy...  
</p>
```

Here we have a `<p>` tag which is a “block” tag so it fills across the page and any new tags after the `<p>` tag will go underneath. Inside our `<p>` tag, we have an image and some content. The content is just inline content as it's regular old text. The `` tag is also an inline tag, so by default the image sits as if it was part of text, its bottom sitting at the text's baseline.

How do we make the text content wrap around the image's right hand side? We make the `` tag float to the left:

```
img {  
  float: left;  
}
```

What if we wanted the text to wrap on the left and the image to go to the right?

```
img {  
    float: right;  
}
```

Unfortunately, there's no center floating at the moment, so you can't wrap text on both sides of an image — we only get the choice of left or right.

Making layouts with floats

We can use the idea of moving tags to the left or the right to build up our content. Let's take a header where we want to move our `<h1>` tag to the left corner and our `<nav>` tag to the right corner:

```
<header>  
    <h1>Furneaux's Flower Shop</h1>  
    <nav>  
        <a href="about.html">About</a>  
        <a href="blog.html">Blog</a>  
        <a href="contact.html">Contact</a>  
    </nav>  
</header>
```

First of all, in our style sheet, we can use floats to move the `<h1>` tag to the left:

```
h1 {  
    float: left;  
}
```

This will make all the other content in the header wrap around the `<h1>` tag on the right hand side. We can then use floats to move the `<nav>` tag to the right hand side:

```
h1 {  
    float: left;  
}  
  
nav {  
    float: right;  
}
```

Great! This will move the `<h1>` to the left and the `<nav>` to the right. However, you might notice — especially if you had a background on your `<header>` tag — that some of the layout could be messed up.

Fixing layouts with clearfix

Whenever you're floating any tags, it's a good idea to remember what you're actually doing — you're taking the tag out of its usual position to make other content and tags wrap around it. As you're doing this, the tag that surrounds the floated tag — we'll call this the “parent” tag — doesn't realize that you've floated the tag. Cascading style sheets can

only go deeper into tags so any higher-level tags are unaware of what you're doing inside them.

When we float all of the content inside a tag (like in the `<header>` in the previous example), it doesn't realize that there's any content inside it as you're floating every tag inside it. The tag gets confused and collapses down as if there's no content inside it.

You might see this happen when your background disappears or your floated content starts to escape the edges of your tag.

Why does this happen? Well, the truth is that floats were never designed to make layouts, but us coders found a “hack” — a way that just happened to work even if it wasn't meant to fix the problem. Remember CSS was made by scientists in the mid 1990s when web sites looked a lot simpler, and they didn't think web pages would need any complex layouts. So when we use floats to make layouts, we're “hacking” the system and have to deal with its limitations.

To get around this, we use another fix called a “clearfix”. There are a lot of different fixes and no single agreed version. The simplest way is to add one rule to your outer tag:

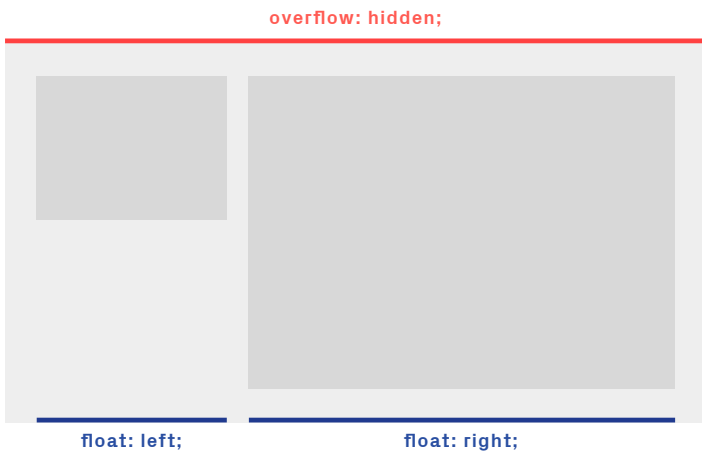
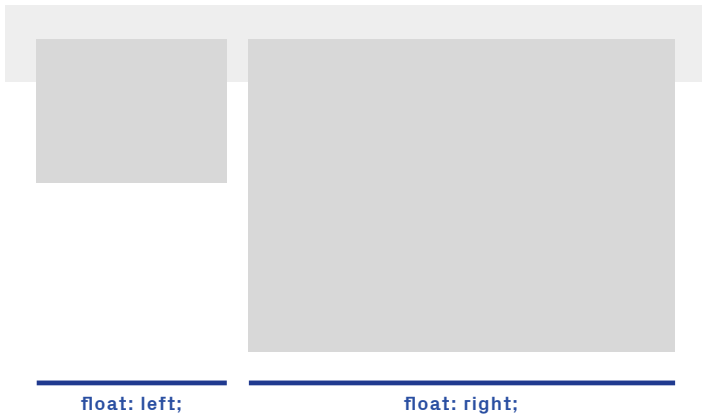
```
header {  
    overflow: hidden;  
}  
  
h1 {  
    float: left;  
}  
  
nav {  
    float: right;  
}
```

We'll talk about what about overflow is in a later section, but for now, we'll add it in and leave it unexplained!

Three column layouts

To add more columns to your layout, for instance if you have a blog with two sidebars and one main area for blog posts, we can structure our HTML in this way:

Floating content without and with clearfix



```
<section>
  <div class="sidebar">
    Post by Rik Lomas
  </div>

  <div class="content">
    Here's a really long blog post about...
  </div>

  <div class="comments">
    I love this post! It's...
  </div>
</section>
```

Here we have one outer `<section>` tag that acts as the “parent” to the three `<div>` divider tags. We’ve given each `<div>` tag a different class so we can style them up differently.

To style them up, I’m going to give the sidebar around 20 percent of the width, the main content 50 percent, then give the remainder (30 percent) to the comments.

Remember as we’re floating all the tags inside the `<section>` tag, we need to add our “clearfix”:


```
section {  
    overflow: hidden;  
}
```

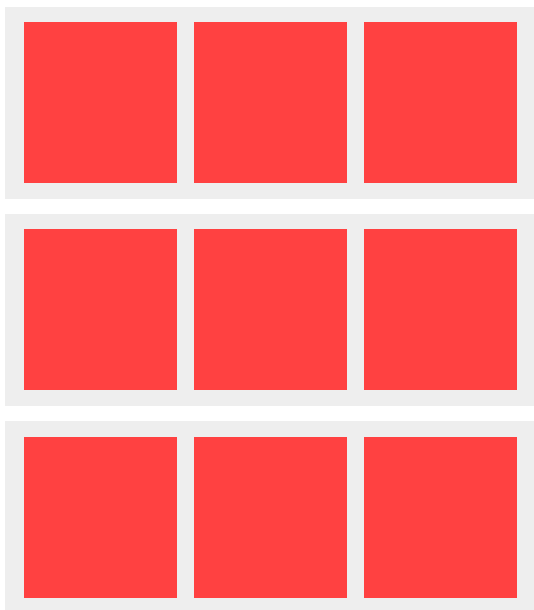
```
div.sidebar {  
    float: left;  
    width: 20%;  
}
```

```
div.content {  
    float: left;  
    width: 50%;  
}
```

```
div.comments {  
    float: right;  
    width: 30%;  
}
```

Notice that we're floating two tags to the left and one to the right and all the numbers add up to 100 percent. The height of the section tag will be based on whatever the highest content is in the sidebar, content or comments `<div>` tags.

A grid layout with floats



overflow: hidden;



float: left;

Making a grid with floats

Let's say we want to make a grid of 3 by 2 using floats. There are a few ways to do this. The first is if we have content that is equal heights — let's say we have images that are all 300 by 300 pixels. We would set up the HTML like so:

```
<section>
  
  
  
  
  
  
</section>
```

We can make the first image float left. That will then wrap the second image on the right hand side of the first image. If we float the second image to the left, the third image will wrap on the right hand side. If we float the third image, there's no more room for the fourth image to fit into so it'll go on to the next line down. In our CSS we can add:

```
section {
  overflow: hidden;
  width: 900px;
}
```

```
img {  
    float: left;  
    width: 300px;  
    height: 300px;  
}
```

Sometimes we don't know how tall the items in our grid might be. If we use the previous HTML and CSS, we might get our grid looking a little messed up. To fix our HTML and CSS to make a grid with unknown heights, we can add:

```
<section>  
      
      
      
</section>  
  
<section>  
      
      
      
</section>
```

Then our CSS will be:

```
section {  
    overflow: hidden;  
    width: 900px;  
}
```

```
img {  
    float: left;  
    width: 300px;  
}
```

You may have noticed that the style sheet is pretty much the same as before. Nothing's changed. All we've done is change the HTML which is the structure of the web page. Sometimes the fixes we need aren't in CSS, they can be in just the HTML, or even both HTML and CSS.

Clearing floats

Sometimes we might not want a particular tag to wrap around another floated tag, but go underneath. To add this is we can use the CSS rule “clear”. We can use the options to avoid (or “clear”) floated left tags, floated right tags or both.

To clear floated left tags only:

```
div {  
    clear: left;  
}
```

To clear floated right tags only:

```
div {  
    clear: right;  
}
```

To clear both left and right floated tags:

```
div {  
    clear: both;  
}
```

Clearing tags in this way can be a really useful way of building up your web designs.

The different kinds of overflow

default

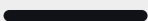
CSS
IS
AWESOME

hidden

CSS
IS
AWESOM

auto +
scroll

CSS
IS
AWESOM



Overflow

We talked earlier about a new rule called overflow that we didn't explain. In this chapter we'll talk a little more about what overflow actually is and what it does.

Let's say we have a `<div>` tag that's 300 by 300 pixels. Most of the time we'd expect our content to fit inside the tag, but what if we have more content than can do that?

Think of a drinking glass. Most of the time, liquid would fit inside it comfortably, but if we have too much, what happens? It overflows.

Luckily in CSS, we can control what happens. By default, just like with a liquid, the content will overspill and escape the container, but we can change how that works.

If we want to cut off any extra content so it's clipped, we can give the `<div>` tag an overflow of "hidden":

```
div {  
    overflow: hidden;  
}
```

If we decide that within our tag, we want a little scroll bar so a user can see the rest of the clipped content, we can add:


```
div {  
    overflow: scroll;  
}
```

On some web browsers, this will give the `<div>` tag a permanent scroll bar, even if the content isn't bigger than the tag. To make it automatically have a scroll bar if the content is bigger and no scroll bar if it's smaller, we can add the overflow automatically:

```
div {  
    overflow: auto;  
}
```

The overflow hack we used in the last chapter for fixing floats works because we're forcing the parent to stretch around all of the content, floated or not, by using a hidden overflow.

Transparency with opacity and rgba colors

In some web designs we might want to make parts of our page slightly transparent. There are two ways to do this. The first is using a CSS rule called `opacity`. By default, every tag is full opacity (“1”). We can hide tags by making them have no opacity (“0”), or we can pick somewhere in between.

Let’s say we want to make an `<h1>` tag 30 percent opaque (or mostly see-through), we can add:

```
h1 {  
  opacity: 0.3;  
}
```

We can even make it so we can have it fade back into full opacity on hover:

```
h1 {  
  opacity: 0.3;  
  transition: opacity 0.5s;  
}
```

```
h1:hover {  
    opacity: 1;  
}
```

In this example we've even added a transition so we can fade it over half a second.

The opacity CSS rule will make the whole tag transparent, including the background and the content inside it. However, sometimes we might just want the background to be see-through, or the text to be slightly transparent.

To do this we need to talk about rgba colors.

rgba colors

Usually in our colors we use hex values, as discussed previously. We start with a hash, then six numbers or letters.

```
header {  
    background-color: #ff0099;  
}
```

Forgetting the hash tag, the first two letters denote red, the next two denote green and the last two denote blue, but there's no way to add a transparency effect in here.

However we can use a different style of color called `rgba`, which stands for red, green, blue and alpha.

This works slightly differently to hex values. Instead of writing in computer counting, we can add human/Arabic numbers from 0 to 255 (or “ff” in computer speak) to determine each separate color. We can then add a value from 0 to 1 for how much opacity the color should have.

We can convert the color above and make it half see-through:

```
header {  
    background-color: rgba(255, 0, 153, 0.5);  
}
```

It's not something I can easily convert just by looking at the hex value, I generally use an online tool, Photoshop or Sketch to work out what the hex value's colors are.

Remember this will just change the background's opacity, not the content. We can use a hover state to transition between two `rgba` colors in the same way:

```
header {  
    background-color: rgba(255, 0, 153, 0.5);  
    transition: background-color 0.5s;  
}
```

```
header:hover {  
    background-color: rgba(255, 0, 153, 1);  
}
```

Drop shadows

For both text and tags, we can add drop shadows to apply some subtle (or not so subtle) effects to our designs. There are two types of drop shadows we can use, one for the text and one for the tag itself.

Text drop shadows

The CSS rule for text shadows takes four values, the “x” direction, the “y” direction, the “blur”, then the color of the shadow.

The “x” direction is across the page. If you want to move the shadow to the right of the tag, you can use a positive number of pixels (e.g. 5px). To keep the same, just use “0”. If you want to move the shadow to the left, you can use negative pixels (e.g. -5px). Same thing for the “y” direction, 5px would be further down the page, -5px would be back up the page, 0 would be no difference up or down.

The blur is also a pixel value referring to how far the drop shadow should be blurred over. If it's 0, there's no blur to the shadow and it looks sharp. The bigger the blur value, the more area the shadow spreads over and the lighter it will be.

The last is a color value. This can be a hex or rgba value. In fact, rgba colors are great for drop shadows as they can give them a transparency that hex values can't.

Let's say we want to give our `<h1>` tag a 50 percent transparent black text shadow that is slightly blurred, straight down and to the right. We would add:

```
h1 {  
    text-shadow: 0 5px 2px rgba(0, 0, 0, 0.5);  
}
```

We're not putting any left value to make it go straight down. Then we're putting the value 5px downwards with a 2px blur, then giving it a rgba color.

Outer drop shadow

The other drop shadow effect is the “box-shadow” — instead of just being on the text, it's on the overall tag itself. It has pretty much the same values, “x”, “y”, blur and color, just this time it's a different rule, the “box-shadow” rule:

```
h1 {  
    box-shadow: 0 5px 2px rgba(0, 0, 0, 0.5);  
}
```

As before, we're having a shadow that goes straight down by 5px with a 2px blur and a 50 percent transparent black color.

Glow effect

We can use box-shadow to give a tag a glow effect too. All we'd need to do is not move the shadow down or to the left, then give it a large blur. In this example, we'll blur by 15px with a 40 percent transparent white color:

```
header {  
    box-shadow: 0 0 15px rgba(255, 255, 255, 0.4);  
}
```

Inner shadows

Most shadows go around the edge of the tag itself, but we can add one more value to make the box shadow go from the outer edge to the inner edge. All we do is add the word “inner” to the start of our box-shadow rule.

```
header {  
    box-shadow: inner 0 0 15px rgba(255, 255, 255, 0.4);  
}
```




**Mobile-friendly
designs with
media queries**

In one of my classes a student was talking to me about how the company she worked for had recently spent thousands of dollars converting its website into a mobile-friendly version.

I was pretty surprised when I saw a simple desktop site turned into a basic mobile equivalent.

“Thousands of dollars?”, I thought, “that can’t be right!”. After delving into the code, I found just 14 lines had been added. It worked out at roughly \$500 per line.

A few hours later, after being taught how to make sites mobile-friendly, the student came back to me with an angry thank-you. Happily she now understood how to make sites work on all different screens. Less pleasingly she knew her company been ripped off by so-called professionals.

Mobile-friendly — or “responsive” — designs in the web are surprisingly straightforward to add to your sites. You don’t need to make another version of the site, all you’re focusing on is the look and feel of the site and how it changes from one browser size to another.

As you’re changing the look and feel, all you need to do is add more code into your style sheets.

What we'll add to our CSS is a new concept called "media queries". Essentially, we'll be turning on or off particular CSS styles depending on the size of the browser itself.

Very early in this guide, we talked about the idea that we can put HTML tags inside other tags — Pimp my Ride style. We can do something similar in CSS. We can say "if the browser is smaller than 600px then do these styles ... if not, ignore".

Let's show an example where we have the whole site having a font size of 18px. On a smaller browser, the font size might look too big, so we want to reduce it to 14px if the browser is smaller than 600px wide:

```
body {  
    font-size: 18px;  
}  
  
@media (max-width: 600px) {  
    body {  
        font-size: 14px;  
    }  
}
```

Notice how we have an @ (or ampersand) with the word "media" straight after it. This is telling CSS to do a browser check. In our rounded brackets, our check is to ask: "Is the browser 600px or less?". If it is, then add the styles. We can put multiple styles within a media query too:

```
body {  
    font-size: 18px;  
}  
  
h1 {  
    font-size: 32px;  
}  
  
@media (max-width: 600px) {  
    body {  
        font-size: 14px;  
    }  
  
    h1 {  
        font-size: 21px;  
    }  
}
```

You can also have multiple media queries in the same style sheet. Let's say we want to have a mid point below 900px and above 600px where the font size should be 16px:

```
body {  
    font-size: 18px;  
}  
  
@media (max-width: 900px) {  
    body {  
        font-size: 16px;  
    }  
}  
  
@media (max-width: 600px) {  
    body {  
        font-size: 14px;  
    }  
}
```

The order of the media queries matters, as the second one overwrites the first. If they were the other way around, the font size would 16px if the browser was less than 900px and less than 600px too.

Media queries for layouts

Media queries are heavily used to turn multi-column layouts into single-column layouts in responsive design. Let's take a two-column desktop layout in HTML:

```
<section>
  <div class="main">
    Here is the main content
  </div>

  <div class="side">
    Here is a sidebar
  </div>
</section>
```

In your desktop version of your CSS, you'd have something like:

```
section {
  overflow: hidden;
  width: 920px;
  margin: 0 auto 0 auto;
}

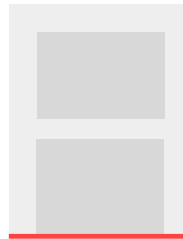
div.main {
  width: 600px;
  float: left;
}

div.side {
  width: 300px;
  float: right;
}
```

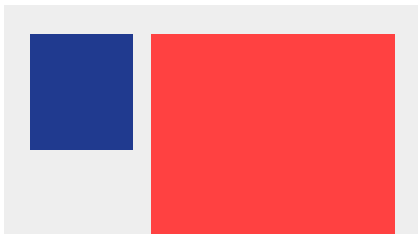
Media queries with floated layout changes



960px
or larger

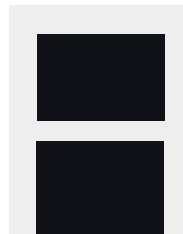


959px
or smaller



float: left;

float: right;



float: none;

As we want to start breaking the layout at around 960px (920px with a bit of room either side), we can add media queries for browser widths less than 960px.

How do we break that layout? Firstly, we want to get rid of the width on the `<section>` tag, otherwise it's bigger than the width of the browser.

Next, we want to stop the two `<div>` tags from wrapping anything and get rid of the widths so they span the whole width. To get rid of the floating, we can turn it off with "none". To let the width revert back to normal, we can just say "auto":

```
@media (max-width: 960px) {  
  section {  
    width: auto;  
  }  
  
  div.main {  
    float: none;  
    width: auto;  
  }  
  
  div.side {  
    float: none;  
    width: auto;  
  }  
}
```

Turning off floats and reverting the widths to automatic size is a very common technique in responsive design. In fact, these lines of code were similar to the thousands of dollars worth of code I mentioned at the start of the chapter.



CSS Displays

Earlier in this guide, we talked about different tags having different types — some tags were “blocks” that push other tags down the page, others were “inline” that follow along the usual style of text.

Some examples of “block” tags are `<header>`, `<h1>`, `<p>` and `<div>`. When we place them next to each other, they go down the page (ignoring any floats we might do).

Some examples of “inline” tags are `<a>`, ``, `` and ``. When we place these inline tags next to each other, they go across the page and on to the next line if they can't fit, just like words in a sentence would.

Sometimes we may want to change the type of tag in CSS depending on its context. A good example of this is when we have links in a navigation. On a desktop layout, we may want the links next to each other, but on a mobile, it may make more sense to have them go down the page instead.

Let's take an example piece of HTML:

```
<nav>
  <a href="about.html">About</a>
  <a href="blog.html">Blog</a>
  <a href="contact.html">Contact</a>
</nav>
```

On a desktop layout, it would make sense for this to go across the page and we can let our users read from left to right.

However, on a mobile device, we might not have enough room to display them from left to right. We can switch how these links look from their default “inline” display to looking like a “block” tag. To do this we will use a CSS rule called `display` to change how the tag acts.

First of all we'll set a media query for mobile only at around 600px browser width, then select the links in the nav, and change how they act:

```
@media (max-width: 600px) {
  nav a {
    display: block;
  }
}
```

Making buttons with inline-block

CSS was invented in the mid 1990s by scientists and little did they think that it would become quite as used as it has. The downside of having scientists make tools used for design is they didn't really think about how designers would use the tools in 20 years' time.

Perhaps the most annoying thing about inline tags like `<a>` and `` is that they don't listen to a few of the more important CSS rules, mainly width and height.

If you try and give them a width and height, they'll just ignore you. Ugh. What's with that? The reason is inline tags were always meant to act like text, and technically, they should fit to the content inside them and not be sized.

Several years later, the new committee that updates CSS called the W3C (or the World Wide Web Consortium) decided that if they suddenly gave inline tags the power to control width and height, they may break a lot of older websites. So instead they came up with a half-way house: the "inline-block". A rule that lets tags go across the page, just like inline tags, but can be given widths and heights.

So where would be good places to use this? For me, this would be for buttons.

Let's take an example of some HTML:

```
<nav>
  <a href="about.html">About</a>
  <a href="login.html">Log in</a>
  <a href="signup.html">Sign up</a>
</nav>
```

Let's say I want to highlight the sign up link. At the moment, all the links would look the same, so the first thing I'd want to do is add a class to the sign up link. We'll call the class "button", but you could call it anything you like (e.g. "signup", "attention", etc):

```
<nav>
  <a href="about.html">About</a>
  <a href="login.html">Log in</a>
  <a href="signup.html" class="button">Sign up</a>
</nav>
```

Now in our CSS we can make this look more like a button. We can give it a width, some padding, make the text centered, rounded corners and more!

```
nav a {
  text-decoration: none;
  color: black;
}
```



```
nav a.button {  
    display: inline-block;  
    width: 100px;  
    background-color: red;  
    color: white;  
    text-align: center;  
    padding: 10px 20px 10px 20px;  
    border-radius: 5px;  
}
```

Just by adding in the “inline-block” we have a lot more control over how it looks.

Hiding tags completely

Alongside the option to change the display type of tags between “inline”, “inline-block” and “block”, we also have the option of completely hiding the tag.

A first question may be, why not just delete the content from the HTML? Yes, you could if you’re not going to need it any more, but what if you want to be selective? What if you want to see something on a desktop screen but not a mobile?

Let’s say we want to remove any <section> tag with the class of “timeline” on a mobile screen. Roughly we want to say, at around 600px or less on the browser width, hide this completely.

We don't want to make its opacity zero as the area will still be taken up by a see-through box. We want to pretend it wasn't there at all, as if it wasn't in the HTML:

```
@media (max-width: 600px) {  
  section.timeline {  
    display: none;  
  }  
}
```

We could do the opposite too, which would be to hide on a desktop but show on the mobile browser:

```
section.timeline {  
  display: none;  
}  
  
@media (max-width: 600px) {  
  section.timeline {  
    display: block;  
  }  
}
```

This way we start with the default (totally hidden), then show it when we're on a mobile version of the site.

10

Positioning

When making more complex layouts, sometimes we might have to use styles that don't fit into the usual floated layout pattern.

As designers, we want to be able to do more with our designs!

We're going to talk about three different ways to change our design. Some of this gets a bit complex and weird so don't worry if you don't understand it straight away.

Relative positioning

Sometimes we may want to move a tag based on where it currently is. Let's say we want to adjust a `<header>` tag by moving it relative to its normal position, to be down 10 pixels and further right by 20 pixels.

We can use some new CSS rules to achieve this. They are the position rule, the top rule and the left rule. We're positioning the tag relative to where it would usually be, then moving it from the top by 10 pixels, then moving it from the left by 20 pixels:

```
header {  
  position: relative;  
  top: 10px;  
  left: 20px;  
}
```

We can also move it in the opposite direction by using two other rules called “bottom” and “right”:

```
header {  
  position: relative;  
  bottom: 10px;  
  right: 20px;  
}
```

Moving the tag will leave a “shadow” — basically a gap where the tag would have usually sat — so even if we move it a large distance, there will still be a gap where it was originally.

If we don't want a shadow, we can use the next type of positioning, called absolute positioning.

Absolute positioning

Absolute positioning means that we are positioning the tag not based on its normal place but relative to two things: the first is based on whether it's inside a relatively positioned tag. For example, if we have some HTML that looks like this:

```
<header>
  <h1>Hi there</h1>
</header>
```

If we have any relative positioning on the `<header>` tag, any absolute positioning on the `<h1>` will be based on the internal coordinates of the `<header>` tag:

```
header {
  position: relative;
  top: 0;
  left: 0;
}
```

```
h1 {  
  position: absolute;  
  top: 20px;  
  right: 20px  
}
```

Even if we add no movement to the header here, our `<h1>` will be moved without a shadow of where it once was to the top right corner of the `<header>` tag.

The second rule of absolute positioning is the absolutely position tag isn't inside any relative tag, it will be positioned to the whole page itself.

If our CSS was simply:

```
h1 {  
  position: absolute;  
  top: 20px;  
  right: 20px;  
}
```

The `<h1>` would say bye-bye to the `<header>` tag and attach itself to the whole page to the top right corner, even if the header was nowhere near the top right corner of the page. Crazy, right?

So two rules for absolute positioning, if you want the “co-ordinates” of the tag (the left, right, top or bottom) based on another tag, put that tag as a relatively positioned tag. If you want the “co-ordinates” to be based

on the whole page, make sure that tag doesn't have any parent tags which are relatively positioned. Phew. It's a tough one.

One thing you might notice with absolute position is that the tag sticks to the page itself when you scroll. But what if you want the tag to be fixed to the browser rather than the page?

Fixed positioning (or how to make things sticky)

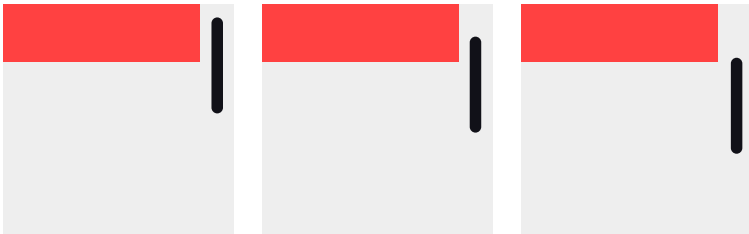
You may have seen a lot of the next CSS rule on the Internet. The idea of having tags that follow you down the page. Facebook and Twitter both have “sticky” headers.

We talked a little about how to fix things to the browser back in CSS backgrounds, when we used “background-attachment: fixed” to fix the background to the browser itself rather than the tag. We can use the similar idea of “fixed” in the tag's position too:

```
header {  
    position: fixed;  
    top: 0;  
    left: 0;  
}
```

This will attach the tag to the page in the top left corner and keep it there, even if scrolling up and down the page.

Fixed vs absolute positions on browser scroll



position: fixed;



position: absolute;

One thing you might noticed is that when you take the tag out of the usual context of the page, it might lose its width. That's because it has no clue about what size it should be. If you want to fill up the whole width of the browser, you can add in:

```
header {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
}
```

This will stretch the header across the whole of the page. We could also start to add heights, background colors and more to this header to fill it out.

Combining positions and floats

One thing you might be tempted to do is start building your layouts using positions rather than floats, because it looks easier — it's similar to design programs like Photoshop and Sketch where you set an “x” and “y” position and away you go. But resist the temptation. Floats are a more powerful tool, especially when it comes to responsive design. The more positioning you use, the more tangles you'll end up in, when it comes to fixing your layout for mobile.

We can however combine positions and floats in great ways. Let's say we want a sticky header where our title is on the left and navigation is on the right. Our HTML would look something like:

```
<header>
  <h1>Boyce</h1>

  <nav>
    <a href="portfolio.html">Portfolio</a>
    <a href="about.html">About</a>
    <a href="contact.html">Contact</a>
  </nav>
</header>
```

So first we want to fix our header to the top of the page. Next we want to move our title to the left, then we want to move our navigation to the right. Remember, because we're floating everything inside the header, we need to fix the layout with a hidden overflow:

```
header {
  position: fixed;
  top: 0;
  left: 0;
  overflow: hidden;
}
```

```
h1 {  
    float: left;  
}  
  
nav {  
    float: right;  
}
```

It might be very tempting to make the floats into absolute positions, but don't do this as it makes it harder when we move to responsive later!

Overlapping positions

Sometimes you might have tags that overlap due to using position rules in lots of ways. The default order is based on whatever tag is further down in the HTML but sometimes you want to switch them out.

As we said earlier in the guide, going across the page is called the “x” direction and going down the page is called the “y” direction. There's one more direction, the “z” direction, which is to do with how close a tag is to the user.

By default, every tag is at the base level, zero, and then sorted by the order of the HTML. We can change that by increasing the “z-index” of the tag. The higher the “z-index”, the “closer” to the user the tag will seem.

To change the z-index, just increase or decrease the number:

```
header {  
    z-index: 1;  
}  
  
nav {  
    z-index: 2;  
}
```

This would make the nav be always above the header. Both would be always above any tag without a z-index as the default is 0.

Cursors and mouse pointers

Whenever our users are using a device with a mouse attached, they will get a mouse pointer on the screen. The default cursor is dependent on what the cursor is hovering over — if it's over a link, it's a little hand-style pointer, if it's over an input box, it's a text cursor, otherwise it's just the default angled arrow.

We can change it to be whatever cursor type we like, even a custom cursor too!

Let's say we want to make the header look like it's clickable with a hand pointer (on a Mac this looks like a Mickey Mouse glove):

```
header {  
    cursor: pointer;  
}
```

If we want the cursor to look like a text cursor, we can add:

```
header {  
    cursor: text;  
}
```

If we want to make the cursor like it's not allowed to click or do something, we can add in:

```
header {  
  cursor: not-allowed;  
}
```

If we want to add our own cursor from an image, we can add in:

```
header {  
  cursor: url(cursor.svg), auto;  
}
```

What we're saying here is that we can make the cursor an image that we've named "cursor.svg", similar to how we have a background image. We then put in a fallback (the comma, then "auto") to be the automatic cursor if a browser can't find our image or can't display a custom cursor.

Pointer events

We can also override tags to ignore any mouse events such as clicking and dragging, by adding in a different CSS rule called pointer-events:

```
a {  
  pointer-events: none;  
}
```


This is particularly useful if we're using positioning to place tags over each other — adding pointer-events. None will let the cursor click through to the next layer down.

We might want to turn it back on for other tags, for instance we might want to ignore the fixed header but not the links inside it:

```
header {  
    pointer-events: none;  
}
```

```
header a {  
    pointer-events: all;  
}
```

**Transforms:
rotations,
scaling and
skews**

Back in 2010, Steve Jobs wrote an open letter about Adobe Flash, the multimedia tool that most of the Web was using for animation and video content.

Jobs said he didn't want to include Flash as part of the iPhone's web browser, because of what he said was its "rapid energy consumption, poor performance on mobile devices, abysmal security, lack of touch support, and desire to avoid a third party layer of software".

Some in the industry were appalled — how dare Apple not include Flash? It was everywhere on the Web!

But Jobs was right. It made no sense to have a website that contained Flash — why have two separate areas, one using web technologies and one using Flash, when we could forget the latter and just build sites using the former?

Over the last few years, Flash has lost most of its power over the Web. Friends of mine who were Flash coders have had to change careers as they found less and less demand for their specialism.

So the Web has had to catch up to fill the gaps that Flash left behind. One of the things that CSS didn't have was a way to transform tags — meaning the ability to rotate, skew and scale tags. So the clever people at the World Wide Web Consortium (or W3C, the body which

coordinates the development of web standards) made the transform rule in CSS.

Rotations

Previously in this guide we talked about linear gradients and how we can angle them using the units “deg” rather than “px”. We can reuse them in this instance. To rotate a tag clockwise by 10 degrees we can use:

```
div {  
    transform: rotate(10deg);  
}
```

If we want to rotate in a counter-clockwise (or anti-clockwise) way, we can use negative degrees:

```
div {  
    transform: rotate(-10deg);  
}
```

Skewing

Skewing is the mathematical way to make a tag turn from a rectangle into a parallelogram. We can skew in two direction, the first being the “x” direction (across the page), the second being the “y” direction. Both take degree units too. We’re telling the corners to increase or decrease their angles by a certain amount. For instance, if we want to add a little bit of skew across the page, we would add:

```
div {  
    transform: skew(10deg, 0);  
}
```

This would mean the bottom of the tag would be further across the page than the top.

To skew down the page, we can do:

```
div {  
    transform: skew(0, 10deg);  
}
```

This would mean the right of the tag would be further down the page than the left.

Adding scale

Sometimes you might want to add a scale to your tag. For instance, you may want the tag to get bigger as you hover over it. Scale takes a multiple number, similar to line heights, so by default all tags start at 1 times the size. If we want to make the tag 20 percent smaller, we would make it 0.8 times the size:

```
div {  
    transform: scale(0.8);  
}
```

We can also use this to make tags pop out on hover, let's say we want to make all the images in a section pop out by 120 percent on hover:

```
section img {  
    transition: transform 1s;  
}  
  
section img:hover {  
    transform: scale(1.2);  
}
```

We don't need to add the transform in the non-hovered style because the default is just 1. We're also using transition to fade between 1x and 1.2x the size over 1 second. It's easy to get the words transition and transform mixed up so watch out!

Translate

We spoke earlier in the guide about moving tags around using relative positions. There is another way to do this, and that's to use `translate`.

Again, we're moving things in an "x" direction, then a "y" direction using pixels. We can also use negative pixels if we want to move up or to the right:

```
div {  
    transform: translate(0, 20px);  
}
```

This would move the tag 20px further down the page.

You might be thinking "why do this ... what's the point"? Good question! Let's give a good example. Let's say we have some images that we want to be slightly transparent by default and then pop up into full view on hover. We can add:

```
img {  
    opacity: 0.5;  
    transform: translate(0, 10px);  
    transition: all 1s;  
}  
  
img:hover {  
    opacity: 1;  
    transform: translate(0, 0);  
}
```

Not only does it pop to full opacity on hover, but moves up by 10 pixels too. Nice little effect!

Combining transforms

We don't only have the option of picking one transform (e.g. rotate, scale, etc), we can put more than one together. We might want to scale and rotate in some cases:

```
img {  
  transform: rotate(5deg) scale(0.9);  
}  
  
img:hover {  
  transform: rotate(-5deg) scale(1);  
}
```

Transform origin

Usually the transforms that you put on a tag will be right in the center of the tag, both horizontally and vertically, but you may want change that. For instance, you may want to rotate around the top left corner rather than the middle.

Alongside the transform rule, we can add the transform-origin rule to our styles:

```
img {  
    transform: rotate(5deg);  
    transform-origin: left top;  
}
```

This will rotate the image around the top left corner. We can also add pixel units to the transform origin (across then down). Another useful unit to use would be a percentage from the top left corner. Let's say we want to rotate around 25 percent across and down — we may not know how big the tag is. We can do:

```
img {  
    transform: rotate(5deg);  
    transform-origin: 25% 25%;  
}
```

A nice thing about translates

Going back to translate for a moment, we can also use percentages for moving a tag around. The percentage is based on the how big the tag is, so for instance if I say move 100 percent to the left, it's based on the width of the tag and will go one whole tag length across the page.

Why is this useful? One of the things we can do is move a tag so the middle of it is in the usual place of the top left. Most tags expand across and down the page, but we might want to expand from the center of the tag instead. To do this, we can move the tag 50 percent of its width to the left and 50 percent of its height up the page:

```
img {  
    transform: translate(-50%, -50%);  
}
```

It's minus 50 percent because we're moving it back to the top and back to the left of the page, instead of across and down the page. In the next chapter, we'll talk about why this can be useful for vertically aligning large sections of HTML.

Vertical alignment

There are two types of way to vertically align tags, and the best to use really depends on the type of display that your tags has.

Aligning within text

For “inline” and “inline-block” tags, there’s a pretty straightforward way to do this. There’s a rule in CSS called `vertical-align` that lets you change how the tag sits within text. For instance, if you want to sit a link to the top of a line of text, if we have bigger tags in that line of text, we can add:

```
a {  
    vertical-align: top;  
}
```

Other alignments we can use include “middle”, “bottom” and “baseline” (based on where the letters in text sit).

We can also use pixel or percentage units to push the tag up from the baseline:

```
a {  
    vertical-align: 10px;  
}
```

We can use negative pixels or percentage units to push the tag down from the baseline:

```
a {  
    vertical-align: -30%;  
}
```

Aligning with block elements

Unlike with “inline” and “inline-block” tags, “block” tags are a little trickier to align. The problem is that CSS sucks at aligning things down the page, because technically the page shouldn’t know anything about browser size. Of course, we sometimes do want to make things look like they know about the size of the page!

There’s no one right way to do this. You can use margins and paddings to move tags around which works up to a point. You can use positions to move tags too.

One way to move the tag into the middle of the page uses positions and transforms together. We move the top left corner to the center of the page, then move it back across and up the page with a transform:

```
section.intro {  
    position: absolute;  
    top: 50%;  
    left: 50%;  
    transform: translate(-50%, -50%);  
}
```

Just using the position absolute (or fixed) to move the tag means the top left corner is right in the middle of the page, not the center of the tag. By using transform we can shift the whole tag upwards by half its height, then across by half its width.

Head tags and search engines

Most of the content in our HTML that we've added so far has been between our `<body>` tag in our HTML, then we've styled it up using connected CSS in our `<head>` tag. Let's explore a few more of the tags that live inside the head of our page (not our `<header>`!).

Previously all of our content has gone inside a structure that looks like this:

```
<html>
  <head>
    <title>My page</title>

    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <header>
      <h1>Hi there</h1>
    </header>
  </body>
</html>
```

All the new information in this chapter will be between the opening and close tags.

Search Engine Optimization

We talked earlier about the `<title>` tag and how it is used in search engines to say what your page is in search results. It's also really important to make sure the content in your title tag is relevant to what people are searching for. Google uses the title tag in ranking search results. If for example you're selling children's clothes, make sure you put that in your `<title>` tag too:

```
<head>  
  <title>SuperHi – children's hand-made clothes</title>  
</head>
```

The second most important thing is to add a new tag called the `<meta>` tag. Meta is the Greek word for “post” or “after”, but it's come to mean self-referential. In this case we're adding some more information about the page itself — content describing the content!

The first meta tag is the description. Google and other search engines will use this to add a little description under a search result:

```
<head>  
  <title>SuperHi – children's hand-made clothes</title>  
  
  <meta name="description" content="We made organic...">  
</head>
```

Google doesn't consider this in its search ranking, but it's useful for potential visitors to our site to know what the site is about before they click.

Facebook social tags

We can also add some tags called Open Graph tags that Facebook and some other social networks use to display a “card” when users share your page. We can add them into our `<head>` and change according:

```
<meta property="og:title" content="SuperHi online
course">
<meta property="og:type" content="article">
<meta property="og:url" content="http://superhi.com/">
<meta property="og:image" content="http://superhi.com/
image.jpg">
<meta property="og:description" content="Learn to
code...">
<meta property="og:site_name" content="SuperHi" />
```

Notice the use of the attribute “property” inside of “name” for this one. I’m not sure what the reasoning is as most other `<meta>` tags just use the name “attribute” for everything!

Twitter social tags

For Twitter cards, we can use something very similar to Facebook’s meta tags:


```
<meta name="twitter:card" content="summary_large">
<meta name="twitter:site" content="@superhi_">
<meta name="twitter:title" content="SuperHi online
course">
<meta name="twitter:description" content="Learn to
code...">
<meta name="twitter:image" content="http://superhi.com/
image.jpg">
```

There are some tools online for testing your “cards” on both Twitter and Facebook — just search for “Facebook Sharing Debugger” and “Twitter Sharing Debugger”.

Flexbox

So we have four different options: “inline” to act as if text and can possibly wrap on two lines or more, “inline-block” to act as a box that goes horizontally and not wrap on two lines, “block” to act as a box that goes vertically down the page, and “none” to hide tags.

More recently, the nerds at W3C decided to add another display type to the list. They found that a lot of web sites were being used more for “web apps” — more complicated web pages that act like desktop apps — so they added some simple and powerful tools for distributing space and aligning content in ways that these “web apps” and complex web pages often need. These are based on a new display type called “flex”. In the coder community we talk about using “flexbox”, because we’re using boxes (or tags) that have flex.

Let’s say we have some HTML that looks like:

```
<section>
  
  
  
  
</section>
```

Most of the time, we’d be using flexbox for more complicated layouts, but let’s keep it simple for now. To activate flexbox, we use:

```
section {  
    display: flex;  
}
```

Flex direction

Next we have a choice about the tags inside our flexboxed section. Do we want the images to be arranged across or down the section?

If we want to add the direction to be across, we'd add:

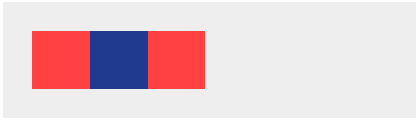
```
section {  
    display: flex;  
    flex-direction: row;  
}
```

It's "row" as the image tags would go across the page. If we want to make the direction be down the tag, we'd add:

```
section {  
    display: flex;  
    flex-direction: column;  
}
```

I get "row" and "column" mixed up all the time. The best way to remember which is which is that it's to do with the tags inside the flexbox and how you want them to sit.

*Justifying contents in a flexbox
with a row direction*



flex-start



center



flex-end



space-between



space-around

Justify contents

So far, most of what we've done can be made with “blocks” and “inline-blocks”. Where flexbox excels is with the next few CSS rules.

We're going to add justify-content to say where the image tags should start within the section tag. This depends on whether we've picked a row or column in our flex direction.

We're going to say we have a row, which means the start of the flex would be on the left, and the end of the flex would be on the right (because rows go from left to right).

To make our images start on the left, we can do the default:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: flex-start;  
}
```

We could make our images start on the right:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: flex-end;  
}
```

Or even the middle:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: center;  
}
```

One thing we can't do with any other display type is make the tags be justified in two different ways. However, we can add space between all the tags to make them evenly spaced between but no spacing on the outside:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-between;  
}
```

We can also make them have an even space around all sides of the image tags:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-around;  
}
```

Notice we're not doing this CSS style on the images themselves but on the parent container tag. The section tag is in control over the layout here.

To wrap or not to wrap

Let's say our section tag runs out of room on responsive design, what should we do with the images inside it? Do we go on to multiple lines with the images? Should we make them wrap or not?

If we do want to make the images wrap on to multiple lines, we can add:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-around;  
    flex-wrap: wrap;  
}
```

But if we want to force the images to be in one single row, even if it crops the images, we can force the layout not to wrap:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-around  
    flex-wrap: nowrap;  
}
```

Aligning in the other direction

So far, we've said our images should go across the section tag in a row, with space around on the "x" direction (the direction across the page). What happens in the "y" direction, if the section tag is bigger than the size of the images?

We can control how they work by adding a new CSS rule called `align-items`. If we want to align the items at the top of the flex (in this case, as it's a row, the top), we can do:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-around;  
    flex-wrap: wrap;  
    align-items: flex-start;  
}
```

If we want to align at the bottom (or right if it's a column), we can add:


```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-around;  
    flex-wrap: wrap;  
    align-items: flex-end;  
}
```

If we want to align in the middle:

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-around;  
    flex-wrap: wrap;  
    align-items: center;  
}
```

We can even make the items stretch from the top to the bottom (or left to right on a column direction):

```
section {  
    display: flex;  
    flex-direction: row;  
    justify-content: space-around;  
    flex-wrap: wrap;  
    align-items: stretch;  
}
```

Multiple lines of content

If we have a few lines of content after we turn on the wrap, how should those individual items look? We can add `align-content`.

The difference between `align-content` and `align-items` is that the latter is based on how each individual wrapped row should act, whereas `align-content` is how all the rows should act.

These use the same five options as `justify-content` and `align-items`. The options are `flex-start`, `flex-end`, `center`, `space-between` and `space-around`:

```
section {  
  display: flex;  
  flex-direction: row;  
  justify-content: space-around;  
  flex-wrap: wrap;  
  align-items: stretch;  
  align-content: space-around;  
}
```

Phew, that's a lot! For images in a section, we're done, but if we want more complex layout, we can also use one more rule to make layouts.

Flex on items itself

Let's say we have a more complex layout in our HTML, for instance something like:

```
<header>
  <h1>Logo</h1>
  <p>Tag line</p>
  <nav>
    <a href="login.html">Log in</a>
    <a href="signup.html">Sign up</a>
  </nav>
</header>
```

The idea of flexbox itself is to let tags directly inside the flexbox have some flexibility in their width or height (depending on the direction of your flex-flow rule). For instance we might be trying to make a sidebar and a main content which has the ratio of 1:4. We can also make them stretch to fit the whole of the container — something that's particularly tricky to do with just floats alone.

Let's say we have a section tag with two div tags inside:

```
<section>
  <div class="sidebar">
    Here goes our sidebar...
  </div>

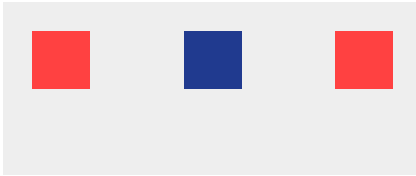
  <div class="content">
    Here's our content...
  </div>
</section>
```

Usually, we could add in floating and widths to give it some layout, but if one of our divs has more content than the other, it doesn't stretch both of them to fit. To show this, let's give the main container a minimum height:

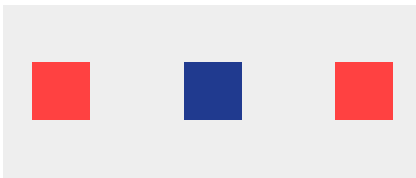
```
section {
  display: flex;
  flex-flow: row;
  align-items: stretch;
  min-height: 400px;
}

div.sidebar {
  flex: 1;
  background-color: red;
}
```

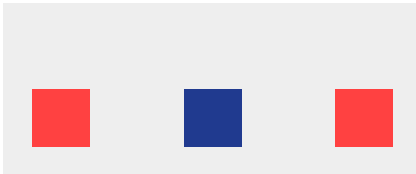
*Aligning items in a flexbox with row direction,
justified with space between*



flex-start



center



flex-end



stretch

```
div.content {  
    flex: 4;  
    background-color: blue;  
}
```

Not only will our two divs stretch to fit the height of the section tag, they will also have flexible widths at a ratio of 1:4.

Flex and static

Where flexible tags come in useful is when we want to mix set pixel widths with flexible ones. Let's say we want our sidebar to always be 200 pixels wide but our content div tag to change — there isn't a way to know how wide this is going to be, so let's make the browser do the hard work!

We can keep the section CSS the same as before. We just need to change the sidebar to a set width and our content to fill the space:

```
section {  
    display: flex;  
    flex-flow: row;  
    align-items: stretch;  
    min-height: 400px;  
}
```

```
div.sidebar {
    width: 200px;
    background-color: red;
}

div.content {
    flex: 1;
    background-color: blue;
}
```

Flex grow, shrink and basis

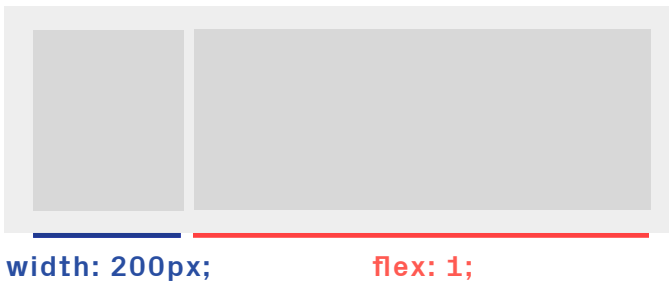
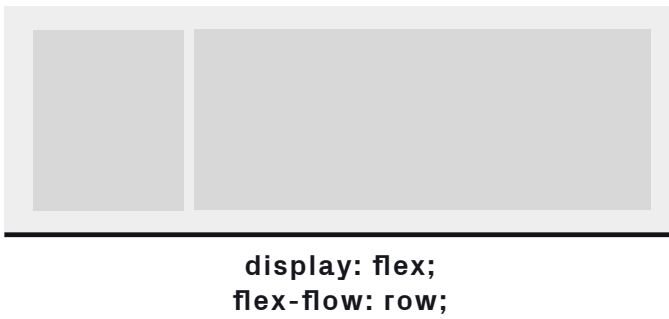
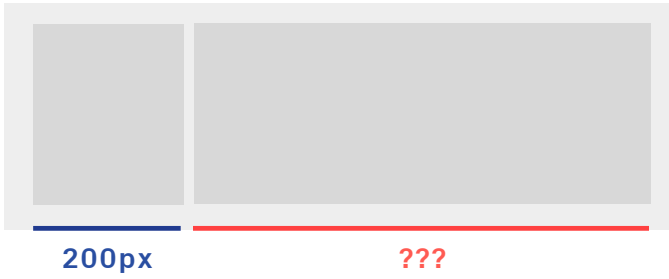
We can add some more information into our flex rule. If we just give it a single number, this will be the amount it can grow (e.g. 1:4 ratio from before), but let's say we don't want our sidebar to be less than 200 pixels wide. We can add some more information into our flex rule.

We can add up to three values into the rule: the ratio of flexible growth, the ratio of flexible shrinking and then the basis size for the tag.

For our sidebar, we're happy for it to stretch at a ratio of 1:4 bigger than 200px, but we don't want it shrinking at any ratio at all under 200px:

```
div.sidebar {
    flex: 1 0 200px;
    background-color: red;
}
```

Flexbox with static and flexible sized tags




```
div.content {  
    flex: 4;  
    background-color: blue;  
}
```

What this is now doing is saying to the browser, at sizes bigger than 1000px (1 + 4 ratio times 200px minimum), be in a ratio of 1:4, but at under 1000px, keep the sidebar at 200px and make the content size flexible. At very small browser sizes, our content tag could be very small too, while our sidebar stays the same!

CSS animations

So far the only animations that we've had are between states, for instance when we hover over a tag, we can make a change. So how do we make animations that automatically play without any user interaction? We can use CSS animations!

There are two parts to adding a CSS animation — the first is adding a keyframe which details what the animation is, the second is adding the keyframe to the tags you want animating in that way.

Keyframes

Think of a keyframe as a timeline of what we want to animate. We might for instance want to animate the background color between two colors, or we might just want to fade in a tag on load. Let's say we want to fade in a tag. The first thing we need to do is think of a name for our keyframe. This can be anything that describes it as one word — think like an HTML class name — so let's call ours “fadein”.

To start the keyframe, we add it away from any other styles, a little like adding a media query:

```
@keyframes fadein {  
  
}
```

We use “@keyframes” to say we want to define an animation, then name it whatever we like. We then need to say how we are going to animate it between the start and the end of the animation.

To do this we can use percentages of the keyframe. To fade something in, we'd start at 0 percent with not seeing anything, then at 100 percent of the animation, we'd be able to see the tag:

```
@keyframes fadein {  
  0% {  
    opacity: 0;  
  }  
  100% {  
    opacity: 1;  
  }  
}
```

We can add other percentages in here, for instance if want to start with a tag faded out, then fade in, and then fade back out, we could do:

```
@keyframes fadeinandout {  
  0% {  
    opacity: 0;  
  }  
  50% {  
    opacity: 1;  
  }  
  100% {  
    opacity: 0;  
  }  
}
```

At the moment, the keyframe lives on its own — we've only defined what we want to do but we haven't said which tags to apply our animation to.

The animation rule

Alongside the @keyframes, we need to say which tags have that animation, how long the animation should take, and how many times it should happen.

To add the keyframes to a tag, we can use a new CSS rule called the animation rule. We need to first of all use the name of the keyframe (e.g. if you named it “fadein” we can use that), then say how long you want the animation to take:

```
@keyframes fadein {
  0% {
    opacity: 0;
  }
  100% {
    opacity: 1;
  }
}

header {
  animation: fadein 2s;
}
```

Here we've said: add the fadein keyframes to our header and animate over two seconds. Currently this will animate just once, but if we want to loop the animation, let's say if we have a glow on the page, we can add "infinite":

```
@keyframes glow {
  0% {
    background-color: #ff4141;
  }
  50% {
    background-color: #ff8888;
  }
  100% {
    background-color: #ff4141;
  }
}
```

```
header {  
  animation: glow 10s infinite;  
}
```

This will repeat the glow keyframe every 10 seconds on a loop.

To delay the animation starting, we add two other things. The first is delay time. The other (“both”) is to tell the animation to apply the start of the animation before it runs and keep the styles after it has finished:

```
@keyframes jumpin {  
  0% {  
    opacity: 0;  
    transform: translate(20px, 0);  
  }  
  100% {  
    opacity: 1;  
    transform: translate(0, 0);  
  }  
}  
  
header {  
  animation: jumpin 1s 2s both;  
}
```

This will apply a fade and a movement over one second, with a two second delay on starting. It will also keep the tag hidden before the animation, and keep it faded in after the animation.

We can apply the same keyframe on multiple tags too!

```
@keyframes fadein {
  0% {
    opacity: 0;
  }
  100% {
    opacity: 1;
  }
}

header {
  animation: fadein 1s 2s both;
}

section.intro {
  animation: fadein 1s 4s both;
}
```

This will fade in the header after two seconds and then a section with the class of “intro” after four seconds.

Using steps to animate

At the moment, our animation takes place smoothly between various different points. For example, we start at no opacity, and it animates smoothly to full opacity.

Let's say we want to go in steps rather than go smoothly between two values, we can add in one more extra to make it stepped:

```
@keyframes fadein {  
  0% {  
    opacity: 0;  
  }  
  100% {  
    opacity: 1;  
  }  
}  
  
header {  
  animation: fadein 1s steps(4) both;  
}
```

This will tell the header to run the animation over four steps: it will stay at 0, then after $\frac{1}{4}$ of a second, it will jump to 0.25 opacity. Then after $\frac{1}{2}$ a second, it will jump to 0.5 opacity. After $\frac{3}{4}$ of a second, it will jump to 0.75 opacity, and finally after 1 second it will finish at full opacity.

The header will never be 0.6 opacity as it will instantly jump between 0.5 and 0.75.

Also notice that there are only four steps, not five, as we're not including the starting step. Think of it like taking a physical step, it's the amount of times you move your feet!

We can also use steps in the CSS transition rule as well as animations.

Using steps to animate images

Most of the time we don't want to use steps in our animations (or transitions), but where they can come in very useful is making complex image animations. For instance we might want to make a logo look like it's bouncing up and down like on the SuperHi website, or a "like" button that animates when we click it, as on the Twitter website.

The big secret is these animations are using a single image that is one long strip and has been made in an image editor like Photoshop or Sketch, rather than anything complex in CSS or Javascript.

Let's say we have an image that we want to animate over 10 frames that usually is 100 pixels by 100 pixels. We could make one single image that would be 10 times the length at 1000 pixels by 100 pixels. We can then use that image to "pretend" we're animating in the browser.

Let's say we have a link tag, with the class of "heart", that we want to animate:

```
<a href="like.html" class="heart"></a>
```

Let's leave it empty of content but style it using CSS instead — we'll give it an inline-block display to let it have a width and height, then add a background:

Animating background positions with steps



animation without steps



animation with steps



using one long image



**a background position
animation without steps**



**a background position
animation with steps**

```
a.heart {  
  display: inline-block;  
  width: 100px;  
  height: 100px;  
  
  background-image: url(heart.svg);  
  background-position: top left;  
}
```

The rest of the image will be cut off on the right hand side. If we wanted to see frame number 2, we would need to move it back across the page by 100px, so we'd change the background position to “-100px 0” instead. To see the last frame, number 10, we can change the background position to “-900px 0”.

So we want to animate the background position between the starting position “0 0” (or top left) to “-900px 0”:

```
@keyframes heart {  
  0% {  
    background-position: 0 0;  
  }  
  
  100% {  
    background-position: -900px 0;  
  }  
}
```

```
a.heart {
  display: inline-block;
  width: 100px;
  height: 100px;
  background-image: url(heart.svg);
  background-position: top left;
  animation: heart 0.25s;
}
```

At the moment, the animation will see the background position speed across the page from 0 to -900 pixels across, but we see the the image glide from one end to the other. We want to step this instead!

As there's 10 frames we need to do nine steps as we're not including the starting position:

```
@keyframes heart {
  0% {
    background-position: 0 0;
  }

  100% {
    background-position: -900px 0;
  }
}
```

```
a.heart {  
  display: inline-block;  
  width: 100px;  
  height: 100px;  
  background-image: url(heart.svg);  
  background-position: top left;  
  animation: heart 0.25s steps(9);  
}
```

Forms

All the code we've been writing so far has been using just HTML and CSS, two parts of the "front-end" of the web. Earlier in the guide, we talked about web sites being like restaurants, where the "front-end" waiters take a customer's order and send it to the "back-end" cooks to prepare.

So far we've only talked about the front-end of the site. Now let's talk about the back-end.

On the web, there are just two ways of getting around web pages. The first is by clicking on a link to go to another page. The second is by letting your users enter data into a form. The front-end of our web sites take the data and passes it to the back-end, just like a waiter would pass a customer's order to the kitchen and what is cooked is dependent on that order.

The waiters don't so much care about the preparation of customers' meals, they're just messengers who ensure the right food is delivered to the table.

Similarly the front-end of your website doesn't care about the data as it's the back-end's job to process the data, store it, delete it if needed, send emails, charge credit cards and so on. Then, once it's finished, to tell the user what next page or message they should see.

For instance, let's say we want to log into a web site. How does this work technically? Our front-end lets a user type a username and

password into two different input fields on a log in page. There will be a submit button underneath that a user will click. The front-end will then pass the user's data to the back-end.

The back-end will take the username and look in its database for a match. If there is one, and the password matches, it tells the front-end that the user is correct and any new information will be added to that user's account on the web site. If there isn't a match for the username or the password, it tells the user there was an error logging them in — essentially the front-end will tell the user that the back-end couldn't log them in, just like a cook telling a waiter to inform a hungry customer they're out of the chicken soup.

The form tag

On some web pages, there may be multiple ways to process data. There could be a sign up form, a log in form and a forgotten password form all on one page. We want each one of our options to be processed in completely separate ways — we need some place for our data in each circumstance to go to be processed.

Let's focus on our log in form. We're going to pretend there is a URL that will process any data for us on a page called "login.html". Usually if we go to the web page in our browser, either by typing it in or clicking a link to the page, we are "getting" the page. However, if we are submitting a form, we are "posting" data to the page.

Our form takes two attributes: what page we're going to next and how we're submitting the data. Most of the time we'll be "posting" data:

```
<form action="login.html" method="post">  
  
</form>
```

If we don't want to "post" data and instead use the method "get", all of our user's data will be in the next page's address bar, which is bad for security.

Input tags

The next tag we need to add is our user's username input field. We need to introduce a new tag called the input tag. There are a few types of input tag that we'll cover, but the first one is plain old text input.

Inputs take the name of the data they contain. Just like HTML classes, the name of the field is dependent on us and the back-end — if the back-end expects it to be "username" data and not "email" data, then we need to match accordingly:

```
<form action="login.html" method="post">  
  <input type="text" name="username">  
</form>
```


Notice that the input tag is a single tag, as no other tag will ever go inside it. You can't put a header or a link inside an input, right?

Password fields

So we have our username, next we need our password. This is still an input tag, but with a different type, so the browser knows to conceal this data from prying eyes:

```
<form action="login.html" method="post">
  <input type="text" name="username">
  <input type="password" name="password">
</form>
```

Here the type of the input field is “password”, but coincidentally the data's name is going to be “password” too. On a sign up field we may have “password” data and “password-confirmation” data — both would be the type “password” though:

```
<form action="signup.html" method="post">
  <input type="text" name="username">
  <input type="password" name="password">
  <input type="password" name="password-confirmation">
</form>
```

How do we submit?

At the moment, we have our input fields where users can fill out data, but no way to submit the form. Technically, our users could press “return” on the keyboard, but it would be nice to have something to click on too.

Weirdly, another input field is the submit type, which turns into a button. A user can’t input any text but, technically, they’re inputting that they’ve finished added data.

We’re going to add one more attribute, which is the “value” attribute. This give any input (not just the submit type) default text:

```
<form action="login.html" method="post">
  <input type="text" name="username">
  <input type="password" name="password">
  <input type="submit" value="Log in">
</form>
```

Other “text” types

There are some specialized versions of the text input field that on a laptop or desktop computer look the same, however for a touch-screen it will give our users a different specialized keyboard. For instance, if we have a telephone number to type in, it’s fine if we have a physical

keyboard, but on a touch-screen keyboard, we might as well just use the number keys, so we can use:

```
<input type="tel" name="home-phone">
```

If we have an email address, we can use:

```
<input type="email" name="email-address">
```

If we just have a number, we can use:

```
<input type="number" name="quantity">
```

If we want to add the “.com” key to be able to quickly add a URL in our input field, we can use:

```
<input type="url" name="twitter-profile">
```

Labelling our inputs

At the moment, it's hard for our users to know which each input field is which — is that an email field, is that a password, is that a username field? We need to let our users know what's going on!

To do this, we can use the label tag to wrap around our inputs:

```
<form action="login.html" method="post">
  <label>
    Username
    <input type="text" name="username">
  </label>

  <label>
    Password
    <input type="password" name="password">
  </label>

  <input type="submit" value="Log in">
</form>
```

This also gives users the benefit of when they click on the text for the input field, our text cursor will automatically move to that field.

Sometimes we might have our label and our input apart, but we can still connect them by using “for” on our label and “id” on our input:

```
<label for="username">Password</label>
<input type="text" name="username" id="username">
```

Larger text inputs

Most input tags can only handle up to 255 characters of text — slightly less than two tweets — but what happens if we want to add 256

characters, or 20,000 characters? Or 20,000 words? We have to use a different tag called the `textarea` tag.

Let's say we have a create new blog post form. We might have the title of the blog post. Usually that would be less than 255 characters, but the body of the blog post would be more than that. We'd keep the title as a text input field, but for the body of the post, we'd use a `textarea` tag:

```
<form action="newpost.html" method="post">
  <label>
    Post title
    <input type="text" name="title">
  </label>

  <label>
    Post entry
    <textarea name="body"></textarea>
  </label>

  <input type="submit" value="Create new post">
</form>
```

You may have noticed that unlike input tags, which don't have closing tags, the `textarea` tag does. Why? `textarea` tags can contain multiple lines of text, whereas input tags cannot. Technically no other tag can go inside the `textarea`, but we can put many return/enter characters in there to separate lines of text.

A good rule of thumb to help choose between text input tags and textarea tags is that if you are going to have multiple lines of text, or if your users could enter more than 255 characters, use a textarea tag.

Dropdown select boxes

For some inputs, you might want them to be from a pre-defined list, rather than have a user type in a free-form answer.

To have a defined list, we use two new tags. The first is the select tag, which works similarly to the input tag. We give it a “name” attribute to tell the back-end what the user has picked. Then inside the select tag, we have a set of options tags.

Let's have two selections for our user. The first is the color of the clothing item they've picked, the second is the size of the item:

```
<form action="buy.html" method="post">
  <label>
    Size
    <select name="size">
      <option value="s">Small</option>
      <option value="m">Medium</option>
      <option value="l">Large</option>
      <option value="xl">Extra Large</option>
    </select>
  </label>
```

```
<label>
  Color
  <select name="color">
    <option value="red">Red</option>
    <option value="black">Black</option>
    <option value="white">White</option>
  </select>
</label>

  <input type="submit" value="Order">
</form>
```

Notice how in the option tags, the text of the option that the user sees doesn't have to match the value attribute. The value attribute will be the item sent to the back-end and the text in between is what the user will see.

Check boxes

Phew, we've pretty much covered every tag that we'll be using, but there are still some other input types that we haven't talked about.

Sometimes we might want to add a simple checkbox to say whether we agree or disagree with something. There's no need to make a user type in "yes" to agree — why not have a check box!

To add a checkbox, we use the input tag but with the type attribute of “checkbox”:

```
<form action="iagree.html" method="post">
  <label>
    <input type="checkbox" name="agree">
    I agree to the changes
  </label>

  <input type="submit" value="Save">
</form>
```

We can also make the default of the check box to be checked by adding in the word “checked” to the end of the tag:

```
<input type="checkbox" name="agree" checked>
```

The default value that gets sent back to the back-end when a user submits the form will be a “1” if the user has checked the box and a “0” if the user hasn’t — basically an “on” or “off”.

Radio buttons

Something that sits between the check box and the select tag is the radio button input, where a user can pick one of a select group of options. Unlike the select field, a user can see the options on the page without having to click to open a drop down.

The reason they're called radio buttons is due to their similarity to a old fashioned car radio, where if you pressed one of the buttons to choose a station, a previously selected button would pop back out.

For our example, let's choose a review for a movie:

```
<h2>Review the movie</h2>

<form action="rate.html" method="post">
  <label>
    <input type="radio" name="rating" value="3">
    Excellent, I loved it
  </label>

  <label>
    <input type="radio" name="rating" value="2">
    It was okay, nothing special
  </label>

  <label>
    <input type="radio" name="rating" value="1">
    I didn't enjoy it
  </label>

  <input type="submit" value="Submit review">
</form>
```

Notice how each of the radio buttons has the same "name" attribute? It's the way that the page knows to unselect all the other linked buttons if you click on one of them.

We can also preselect one of the radio buttons using the word `checked`, just like with check boxes:

```
<input type="radio" name="breakfast" value="Continental"
checked>
```

File uploads

One last input type we can have is for letting our users upload files from their phone or computer to the back-end of our website, for instance if we want them to add an image to their profile, or if you want them to upload a music file of their hottest new jam.

For uploads, we need to introduce two new things. The first is an input field with the type `“file”`:

```
<input type="file" name="photo">
```

The second thing we need to talk about is the form tag itself. By default, form tags are good for submitting text information to the back-end of our website, but they’re not so great at handling both text and file data unless we say we want to handle both types of data.

To tell the whole form to control both text and files, we can add a new attribute to our form tag called the `“enctype”` (short for `“encryption type”`, just like `“src”` is short for `“source”`). The particular encryption type

we need is a “multipart form data” type — essentially multiple parts to the form, text and files.

You don't have to remember this — this is something I regularly Google:

```
<form action="upload.html" method="post"
  enctype="multipart/form-data">
  <label>
    <input type="file" name="profile-pic">
  </label>

  <input type="submit" value="Upload new profile
  photo">
</form>
```

Phew, it makes the form tag quite long, but without the encryption type, any files uploaded won't get added.

Styling inputs

Once we're happy with our form's content in HTML, we want to start styling it up to make it easier for our users to interact with.

Just like with any tag, we can select them directly, such as:

```
input {  
    color: #000000;  
}  
  
select {  
    width: 200px;  
}
```

However, with so many different input types, how can we control all them when our check boxes look vastly different to our email input boxes?

The first option would be to add classes on our input tags. This works very well, but we would have to go back through our HTML and add the attributes in there.

Another way is to use the name of the attribute itself to select all the tags with the same type. To do this, let's say we want to just select all the inputs, then filter them for all the type attributes that are "email":

```
input[type=email] {  
    width: 250px;  
}
```

This selector looks for all the input tags first, then using the square brackets, finds any type attribute exactly equal to "email". We can use this selector in lots of different types too:

```
input[type=email] {  
    width: 250px;  
}
```

```
input[type=file] {  
    width: 100px;  
}
```

We could also combine these with classes:

```
input[type=email].newsletter {  
    width: 400px;  
}
```

This would pick any input tag, then filter by the type of email, then filter by the class of “newsletter”. It would need to match both to style!

Placeholders

In a lot of instances we might want to add a suggestion about what to put into the input tag. For instance, on an email address field we might want to suggest a fake email address so it's more obvious what to put in there. We use a placeholder attribute to make that suggestion:

```
<input type="email" name="email" placeholder="rik@  
superhi.com">
```

Placeholders work on pretty much all the input tag types. To style up the placeholder text itself, we have to use a weird bit of CSS styling. Let's say we want the user's text to be in black text color, but the placeholder in grey:

```
input {  
    color: #000000;  
}  
  
input::placeholder {  
    color: #999999;  
}
```

The “::placeholder” is a bit like a state (e.g. a hover state), but more to do with the actual internals of the tag rather than the user interaction.

This could also combine with the type selector:

```
input[type=email]::placeholder {  
    color: #cccccc;  
}
```

Errors and validation

There are a few more extra attributes we can add into our input tag's HTML to add some checks to make sure we can require or validate.

We can add in the “required” attribute to check whether the input field is filled in or not:

```
<input type="text" name="username" required>
```

The browser will check when a form is submitted that this field is required, and if it isn't, it will let the user know.

There's one big downside of using the required attribute — it's not yet fully supported in some browsers, such as Safari and older versions of Internet Explorer. How do we make sure the data is filled in?

Technically, the front-end shouldn't care about data that is or isn't filled in. The front-end is just the delivery system between the user and the back-end of the web site.

Let's go back to our restaurant analogy. Say we have two waiters, one who knows that the soup has run out and the other who doesn't. What would happen if a user orders the soup from the waiter who doesn't know it has run out? The waiter would go to the kitchen and the chef would say, “oh the soup has ran out, they'll have to have something else”. Then the waiter would go back to the customer and suggest ordering an alternative.

The waiter who knew the soup had run out wouldn't need to do the trip to the kitchen and would ask for a different order there and then. Even if she or he forgot, the last line of defense is still the chef, in the kitchen, with no soup.

In our web site, the back-end is the place that tells our users the field should be required. It's a nice thing to tell the front-end that the field is required, but when some browsers don't support it, we can't rely on that. Expect that field to be optional in some circumstances, even if it says required, and have a back-up!

Audio, video and media

So far in this guide we've talked about adding text, images and colors to our site. What about other media like audio and video?

Audio is pretty straightforward. The first thing we need is to add our audio in MP3 format. Other file types like WAV and OGG work in particular browsers, but MP3 works in all modern browsers.

To add an audio track we can add:

```
<audio src="smashmouth.mp3"></audio>
```

By default, we won't see anything. The audio track is there but what do we want to do with it? Do we want to let our users play the track and give them some playback controls, or do we want the audio to auto play?

To make the user have some controls we can add:

```
<audio src="smashmouth.mp3" controls></audio>
```

To make the audio auto play, we can add:

```
<audio src="smashmouth.mp3" autoplay></audio>
```

We could even add both:

```
<audio src="smashmouth.mp3" autoplay controls></audio>
```

We can also make the audio repeat by adding “loop”, which can be useful for background sounds:

```
<audio src="smashmouth.mp3" autoplay loop></audio>
```

Video

To place a video in our HTML, we can add:

```
<video src="kitty.mp4"></video>
```

If we want to add controls, we can add:

```
<video src="kitty.mp4" controls></video>
```

If we want to make the video auto play, we can add:

```
<video src="kitty.mp4" autoplay></video>
```

If we make the video loop by adding “loop”:

```
<video src="kitty.mp4" autoplay loop></video>
```

We could add all three attributes to have an auto-playing, looped video our users can control:

```
<video src="kitty.mp4" autoplay loop controls></video>
```

Slight hitch to auto playing

There's one big downside to adding the "autoplay" attribute. Mobile browsers will ignore you. There's a legitimate reason for this — if you're on a phone, you're likely to not be connected to wifi, but 3G or 4G, which is slower to load. Browsers would prefer to ask users if they really want to play the video or audio, so they aren't automatically using up all their mobile data allowance.

There's no quick way around this auto playing restriction, it's something we have to design around. Some users detest auto playing videos and audio, so it's always a friendly gesture for us as authors of web pages to ask the user what they'd like to do.

Other websites' media

So we have our own audio and videos playing on our own web site, but what if we want to grab a YouTube video or a Spotify playlist? Unfortunately we can't use the audio or video tags for this. But there is another way.

Back when YouTube was in its infancy, it didn't have a lot of traffic coming to its web site. But it did notice a lot of other web sites were having trouble adding videos to their web pages. By using a pretty old technique called "embedding content", YouTube offered a service where other people could upload content and put it on their own sites. This made YouTube's traffic skyrocket.

Other sites followed with the embedding content idea and most larger media sites now offer this. How you embed content depends on the site.

For instance, if you want to embed a YouTube video, go to the video you want to place on your site. Underneath, there will be a "share" link. This will open a new section that will have another link called "embed". In here, there will be some HTML code that we can copy and paste into our own HTML.

Other sites will have their own way, but a lot hide the "embed code" under share links. Look out for HTML code that starts with an `<iframe>` tag. This is an "interactive frame" that essentially puts a small web page inside your web page. A web page inside a web page sounds crazy, but it's exactly what it's doing. YouTube, Soundcloud, Spotify and more all have special web pages solely devoted to putting their content into a box on your web page and millions of others.

Sometimes it can be quicker and easier to add your own content to YouTube and Soundcloud to convert it into a web friendly format. They'll do all the work for you and there's just a small piece of HTML to add once it's done.

Video background

Sometimes you might want to add a video background to your website. We'll do this with a video tag, but you could do this with an embedded video too.

In our HTML, we'd add at the very top of our content something like:

```
<video src="background.mp4" autoplay loop></video>
```

We can then place this in the background of the page:

```
video {  
    position: fixed;  
    z-index: -1;  
  
    min-width: 100vw;  
    min-height: 100vh;  
  
    top: 50%;  
    left: 50%;  
    transform: translate(-50%, -50%);  
}
```

Here we're keeping the video in the background as we scroll by using fixed position and making our z-index -1 (remember default is 0, so we move it back). We're then making sure the video fills the background by doing a minimum width and height. We're then moving the video's top left corner into the middle of the page, then moving it back up and left with transform.

Remember this will not auto play in mobile browsers. Maybe add a media query at smaller sizes to hide the video and show a default background instead:

```
@media (max-width: 900px) {  
  video {  
    display: none;  
  }  
}
```




Web fonts

One last thing before we move on to Javascript.

We have already talked about typography on the Web and about how to change typefaces, and how they look using CSS rules. For a long time after the Web's inception, us coders were only able to use a small number of typefaces — the ones that most users had installed as a default on their computers.

In early studies it was shown that there were only 14 typefaces that were installed on most computers — known as “web safe fonts”, they include Arial, Georgia, Times New Roman and Comic Sans.

When it comes to web design, having just 14 “safe” typefaces to work with gives us limitations over how we design. Designers previously had to cheat this limitation by either using images of the text they wanted, or by using Flash with the correct fonts packaged inside — both are really awkward, convoluted ways of getting a typeface on to a web page.

Again, the nerds at W3C decided to solve this problem by coming up with a new system called “web fonts”, which does what it says — it gets a huge variety of typefaces into the browser, even if users don't have them all installed on their computers.

Think of it like an image. The user doesn't have the image on their computer, instead they download it when they need it. In this case,

because it's to do with the look and feel of the site, rather than the content, we will be adding our changes into the CSS files.

There are two ways to add in web fonts. The first is self-hosted files, the second is adding fonts hosted elsewhere, such as Google Fonts.

Self hosted fonts

The first way is to have fonts that you have within your files, similar to having your own hosted images.

You need to make sure that you're legally allowed to self-host the font files. You can't just upload any old file without the permission of the font foundry that made them. There are some amazing font foundries such as Grilli Type, Lineto and Commercial Type that put in a lot of effort and time into making fonts and they deserve to be paid for doing so.

Just like with images, audio and video, there are certain file types that we work with for font files. The most commonly used file types are OTF, TTF and WOFF — make sure your typeface is in one of these formats.

There are two parts to adding fonts in your CSS: the first is to say what the font is called and what file it is, the second is which tags you want that typeface for.

Let's say we want to use the typeface GT America and we have a file called "gt-america.otf". At the top of our CSS file, we would add:

```
@font-face {
    font-family: GT America;
    src: url(gt-america.otf);
}
```

The "@font-face" is a CSS command, similar to "@media" or "@keyframes". We are then giving a name to the font to use later, we can name this whatever we like but it's generally a good idea to name it after the font itself. The last part is to tell the font where the file is — in this case, the source is a URL called "gt-america.otf".

Next, we need to say what tags can use GT America. We might just want our h1 tags to have this font, so because we called it "GT America" earlier, we can now use it:

```
h1 {
    font-family: GT America;
}
```

We could even set the whole page to be in GT America if we wanted to:

```
body {
    font-family: GT America;
}
```

But what if we have a few different fonts in our typeface? We might have a regular font, a bold font and an italic font. How do we make sure we match everything up?

Let's pretend that our regular font is the one we've hooked up already: "gt-america.otf". We might also have a bold font called "gt-america-bold.otf", a regular italic font called "gt-america-italic.otf", and even a bold italic font called "gt-america-bold-italic.otf". We need to add three more @font-face statements to match them up:

```
@font-face {  
    font-family: GT America;  
    font-weight: 700;  
    src: url(gt-america-bold.otf);  
}
```

```
@font-face {  
    font-family: GT America;  
    font-style: italic;  
    src: url(gt-america-italic.otf);  
}
```

```
@font-face {  
    font-family: GT America;  
    font-style: italic;  
    font-weight: 700;  
    src: url(gt-america-bold-italic.otf);  
}
```

Essentially we're telling our CSS files to look for the right font source when we're using CSS rules that match. If we use font weight or font style in our rules, find the right font file to use.

For instance, now we can use the bold weight in our tags:

```
h1 {  
    font-family: GT America;  
    font-weight: 700;  
}
```

Or maybe the bold italic weight in our h2 tags:

```
h2 {  
    font-family: GT America;  
    font-weight: 700;  
    font-style: italic;  
}
```

The amount of @font-face statements you have depends on the amount you need. Try not to add too many to avoid slowing down your web page loading time. A good rule of thumb is to just use one or two fonts in your designs — any more is excessive.

Hosted fonts

Other font foundries might want to host the fonts for you for licensing reasons. They might want to let you have a yearly subscription to the typeface rather than a continual license, or they may just want to add new characters over time. Either way, it's the foundries' decision and you have to stick with it.

Each foundry will have different instructions on their site about how to add their font to your site. Just follow those and it should be an easy process.

One of the free hosted options is Google Fonts. Google has a range of hundreds of fonts that you can use legally for free — 818 at the time of publication. To use a Google hosted font, just go to fonts.google.com and pick the font you like. Just as with YouTube and Soundcloud, there will be an “embed” option that will give you a snippet of code that you can code and paste into your code.

The image features a solid yellow background. On the left side, there is a large, white, stylized number '1'. On the right side, there is a large, white, stylized number '3'. Both numbers are rendered in a clean, sans-serif font. In the center, between the two numbers, the word 'Javascript' is written in a bold, black, sans-serif font.

Javascript

Our efforts so far have been concerned with two of the three parts of building the front-end of web pages — HTML and CSS.

These are what make up the content and style of pages. Up to now we've only been able to add one level of interaction — CSS states such as hover. How do we add more interactivity? Step forward Javascript!

This is what gives us the ability to alter a page after it has loaded for our users. The best way to think about where to use Javascript is to think “when the user clicks ...”, “when the user submits a form ...”, “when the user scrolls ...”, “when the user resizes the browser ...”, etc.

All of these are actions, the third and last part of front-end web development.

In other words, HTML is the content, CSS is the style, Javascript is the user action.

A quick history of Javascript — and what is Java?

Back in the mid 1990s, before the days of Adobe Flash (or Macromedia Flash as it was called then), there was another way to integrate miniature media programs into web pages. This was Java Applets.

Java is a programming language created in the early 90s, originally with the intent to make interactive television. After a few years, it was realized that this technology could be used in the fast growing World Wide Web. Over a few years, all the major browsers added the ability to use these miniature programs in the HTML of web pages. Java was the programming language and applet just meant a small application.

As more sites used Java applets in their pages, browser makers decided there needed to be a way to control these Java applets, in case they took a while to load, or if two applets needed to communicate. They wanted to make a “glue” between HTML, CSS and Java applets.

One of the large browser companies at the time, Netscape, hired a programmer called Brendan Eich to make this “glue” programming language. Initially, he called it Mocha, since Mocha and Java were both coffee beans. When it was released, Netscape first changed its name to LiveScript, then a few months later to Javascript, to reflect the fact it was intended to work with Java applets.

The code behind Java applets and Javascript look completely different. There isn't too much in common apart from the name. It was a branding decision by Netscape to increase the new language's popularity with Java developers.

Think of it like a grape and a grapefruit — both have similar names but are completely different. Or a horse and a horsefly.

Over time, Java applets became less and less popular, partly due to Adobe's Flash, which is a quicker and powerful version of Java applets. Then over even more time, Flash became less popular, partly due to its removal from the iPhone system and partly due to Javascript itself getting quicker and more powerful over the course of 20 years.

So whenever you hear anyone talk about HTML, CSS and Java, you can put on your best “well, actually” voice and tell them: “I think you mean Javascript, not Java”. Cue smug feeling.

One of the harder languages to learn

As Javascript was invented for a slightly different purpose than what it is used for today, this means the thought process and planning for it has been a little less structured than other languages.

Think of Javascript as a small house that, over years, has more and more extensions and wings built on to it, to the point where it looks like a bit of a mess. And because of this, it means that Javascript itself can be a difficult language to learn. You'll notice a lot of (round brackets) and {curly brackets} that can look quite confusing at first, but once we understand what they actually do, it hopefully gets a little clearer.

On the back-end of web sites, us coders have the choice of a variety of languages to work in, it depends partly on whether the tool is right for the job and partly if we like the language.

Unfortunately for us coders, we are tied to Javascript for aspects of the front-end. We don't have a choice in what to use for the web's "action" language, so we have to know it to be a front-end web developer.

There are, however, tools we can use to make it easier. We will talk soon about jQuery, which means that a lot of common tasks we would need to do get reduced from tens of lines of normal Javascript into one line of jQuery code. We'll cover this later in the guide.

Starting with Javascript

The one thing to remember with Javascript is you are essentially holding and changing data. That data could be HTML, that data could be CSS, that data could be a username, but all Javascript (and all coding languages) does is change things.

So first we need to know how to store data in Javascript, then later, how to change data.

Numbers

The first thing we can store data as is a number. Numbers are written just as if you would on a piece of paper. They can be decimal numbers, they can be negative numbers, and they can be negative decimal numbers:

12

-12

1.23

-1.234

Strings

The next thing we can store data in is a string, which is essentially just a set of letters, numbers and characters. The way we store data in a string is by using double quote marks around the data:

```
"riklomas"  
"Rik Lomas"  
"Welcome to SuperHi"  
"p@ssw0rd1"
```

Examples of data you might want to store in a string are usernames, welcome messages, error messages and blog post titles.

Booleans

Booleans are named after George Boole, the 19th Century English mathematician who invented a lot of the formal ideas behind the logic that computers still use. As Mr. Boole was all about logic, the booleans are also about logic. There are two special words, without any quotation marks:

```
true  
false
```

Notice they're both all lower case letters. Javascript knows to look out for these two words as they're special.

Examples of where we might use booleans are to check if something is sold out, to check if a user is logged in, or to check if a username is too long.

Arrays (or lists)

With our numbers, strings and booleans, we might want to store them in a list. This could be a list of numbers, a list of strings or a mixture of everything.

To make a list we use square brackets to start and end the list, then a comma to separate each item in a list:

```
["bacon", "sausages", "eggs"]
```

```
[48, 32, 15, 3]
```

```
[true, true, false]
```

```
["red", false, 12.23]
```

To make an empty list, for example if we were having an empty shopping basket, we can use:

```
[]
```

Objects

Arrays are used to describe multiple items that are grouped together, but sometimes we might want to describe one single item. For instance, if we are describing a user, it would be just me, one single user, versus an array of multiple users.

To describe a user, we can use curly brackets to start and end the object we're describing. Then inside we use a "key" to give the attributes we want to describe, then a colon, then the "value" of the attribute. Think of it a little like a CSS rule. We then separate them with commas:

```
{  
  name: "Rik Lomas",  
  username: "@riklomas",  
  age: 32,  
  admin: false  
}
```

Notice how the "keys" don't have quotation marks around them, only the values that are strings do. One of the values is a boolean (I'm not an admin) and one of the values is a number (I'm getting old).

One more example would be of a course that we run:


```
{  
  title: "Introduction to HTML",  
  teacher: "Lawrence Gosset",  
  isSoldOut: false,  
  students: ["Holly", "Sarah", "Dani"],  
}
```

Notice here that we have a key — “is sold out” — that we have condensed down to one weird word. Keys have to be just one word, no spaces allowed.

Also notice that we have two values that are strings, one that is a boolean and the last value is an array. We can mix and match our objects and arrays so we could have an array of objects or a objects with arrays in it. Or even an array of objects that contain arrays. Or an array of arrays. Phew!

Variables

A lot of the time we might want to hold on to data so that we can change it later on. To do this, we tell the data to attach itself to a variable.

The name “variable” sums it up pretty well — it’s a bit of data that could change later.

The way we say something is a variable is we tell a word that we make up to be the variable, then tell it to equal something:

```
var username = "@riklomas"
```

We always use “var” then a space to start a variable. Then we give the variable a name (no quotes!) that we assign to it. In this case we called it username as we felt that was most appropriate, but we could have called it “ourbestusernameever” if we wanted to. We then say it’s equal to something — a string, a number, an array, an object, anything we want to store:

```
var age = 32
var canEdit = false
var colors = ["red", "blue", "white"]
```

Here we’ve added three variables, one called “age”, one called “canEdit” (has to be one word), then one called “colors”. All are equal to completely different things.

Later, we might want to update our variable to be something different. We might want to change the age, name, colors, etc. To do this, we don’t need to use “var” again, we can just use the name of the variable:

```
var age = 32
age = 33
age = 52
```

At the end of this code, “age” will now be 52, not 33 or 32. We only need to say “var” once.

We can also alter the variable based on the variable itself:

```
var slide = 1
```

```
slide = slide + 1
```

At the end of this code, “slide” will be 2. Originally it was one, but then we said add one to “slide” and overwrite the variable “slide”. We can also subtract, multiple, divide and more to our variables:

```
var account = 900
```

```
account = account - 1000
```

My variable “account” in my bank website is now -100. This is also a reflection of my real life bank account. If I wanted to double my real life bank account, I could use the multiply sign “*” or divide with “/”. So:

```
var account = 900
```

```
account = account * 3 / 2
```

Here, I’m tripling the original account number, then dividing the total by two (account is 1350 at the end).

Numbers aren’t the only thing we can add together. We can also add strings to together:

```
var welcome = "Hi there, "  
var name = "Rik"  
var message = welcome + name
```

Here we're making a string called `welcome` and a string called `name`, then we're making a brand new variable called `message` which adds the two previous strings together. The output for the variable `message` would be "Hi there, Rik".

We could also add numbers to strings too and they automatically convert into strings:

```
var slide = 2  
var max = 4  
var text = slide + " out of " + max
```

The variable called "text" would have the output of "2 out of 4" — we're adding something that changes (2) to something that stays the same (" out of ") to something that changes (4). Unfortunately, you can't subtract, multiply or divide strings, Javascript doesn't know what you mean when you say divide "rik" by 2.

Manipulating data

So far we've spoken about how to store data, one big part of Javascript. The other even larger part is manipulating and changing that data.

We'll talk about a few ways to do so, but first let's quickly talk about how data can be formatted.

We'll be making "statements" about how Javascript should be changed. In some cases we'll be using an "if" statement, in some cases we'll be using a "for" statement, in some cases we'll be using a "function" statement. We'll talk about what they mean later, but the way they are all set up is using the format of name, round brackets, curly brackets:

```
if () {}  
for () {}  
function () {}
```

To make them a bit easier to read, generally we open up the curly brackets so ...

```
function () {}
```

Turns into ...

```
function () {  
  
}
```

That last trailing “}” can confuse a lot of new coders, but what we’re doing is just making it easier to read when we start adding more code between the “{” and the “}”.

What do the round brackets and the curly brackets do? The round brackets take some “arguments”, or the information to let the statement know what to do with it. The “if” statement checks whether to do the “statement”. And the “for” statement checks how many times to do the “statement”.

We saw this earlier in CSS’s media queries. We had:

```
@media (max-width: 640px) {  
  
}
```

The @media is a CSS statement to check the browser. The round brackets then do a check to see if the browser is less than 640 pixels wide, and if it is, do the things in the curly brackets.

In Javascript, something similar might have looked like:

```
var width = 500  
  
if (width < 640) {  
  
}
```

In this case, the variable “width” is smaller than (“<”) 640, because 500 is less than 640.

What are we doing inside the curly brackets? Well, in Javascript, this could be anything. We'll talk more about this later.

if statements

To add an “if” statement, we add the word “if”, then round brackets, then curly brackets:

```
if () {}
```

We can then put our check in the round brackets, and what we want to do if it passes the check in the curly brackets. We can space out our curly brackets to give ourselves a bit of breathing room:

```
if (account < 0) {  
    var error = "Not enough money"  
}
```

Here our check is: “is our variable account less than 0”? If it is, make a new variable called “error” that has the string “Not enough money”.

There are a few types of check we can put in our “if” statements. The most common is the equals check. This isn't a single equals but a triple equals:

```
if (username === "rik") {  
    var isAdmin = true  
}
```

What is a triple equals doing there? And why do we also have a single equals below it?

The single equals means “make this the new thing” (e.g. `isAdmin` is now true). The triple equals means “check is this the same” (e.g. if the `username` variable is exactly “rik”).

There is also a double equals, which does a similar thing to a triple equals but is less strict, which can lead to weird things happening, so we’ll stick with a stricter check. A good example of this is something like “0” is double equals to the boolean false, e.g. “0” == false would pass but “0” === false wouldn’t. Weird, right?

We can also have a few more checks:

- ♦ **`age !== 32`** — means age is not 32
- ♦ **`age > 32`** — means age is greater than 32 (not 32 though)
- ♦ **`age >= 32`** — means age is greater than or equal to 32
- ♦ **`age < 32`** — means age is less than 32 (not 32 though)
- ♦ **`age <= 32`** — means age is less than or equal to 32

We could also have a simple boolean to check:


```
var isAdmin = true

if (isAdmin) {
    var message = "Welcome"
}
```

To add a double check (e.g. if this AND that), we can add in the “and” symbol which is “&&”:

```
if (isAdmin && age >= 21) {
    var message = "Welcome to the admin, adult!"
}
```

To add a double check where it's one OR the other, we can add in the “or” symbol which is “||”:

```
if (isAdmin || age >= 21) {
    var message = "Welcome, admin or adults!"
}
```

Else and else if

There is more we can add to the “if” statement. We can add in something to happen when the check doesn't pass, using an “else” statement:

```
if (age >= 21) {  
    var message = "Welcome"  
} else {  
    var message = "You must be 21 or older..."  
}
```

We can also add in another check if the first one doesn't pass using an "else if" statement:

```
if (age >= 21) {  
    var message = "Welcome"  
} else if (age >= 18) {  
    var message = "Come back in a few years"  
} else {  
    var message = "You must be 21 or older..."  
}
```

You can put as many "else if" statements as you like and you don't necessarily need an "else" if nothing else matches too:

```
if (key === "w") {  
    var direction = "up"  
} else if (key === "a") {  
    var direction = "left"  
} else if (key === "s") {  
    var direction = "down"  
} else if (key === "d") {  
    var direction = "right"  
}
```

For loops

Sometimes you might want to loop over a bit of code a few times, for instance when counting or checking items in an array. The way we do this is with a “for” statement, or as most coders called it, a “for loop” (as it loops!).

The syntax for “for” loops is a little weird. It starts off very similar to before:

```
for () {}
```

What goes inside our round brackets? Three parts: where we start from, where we end, and the steps to get there.

We usually start with a number variable:

```
for (var i = 1) {}
```

I’m going to make a variable called “i” because it’s short. The variables “i”, “j” and “k” are often used in loops by coders, just for their brevity.

Next we need to say where we finish. In this case, I want “i” to count up to five. We separate the first part from the second part using a semi-colon:

```
for (var i = 1; i <= 5) {}
```

Next, we need to say how we increase that variable “i” on each loop. For most examples I want to count up by one each time (e.g. 1, 2, 3, 4, 5):

```
for (var i = 1; i <= 5; i = i + 1) {}
```

There is a shortcut for “i = i + 1” that a lot of coders use called the “plus plus”. It just adds 1 to itself:

```
for (var i = 1; i <= 5; i++) {}
```

We can then separate out the curly brackets to add our code in our loop:

```
for (var i = 1; i <= 5; i++) {  
  
}
```

We'll talk more about what we can put in our curly brackets later!

Functions

Functions are probably one of the most common parts of what we'll be using for our interactions. They are the underlying part of what we want to happen when our users interact.

I like to explain functions as if they were a paper shredder. Usually the machine just sits in the corner not doing anything, waiting to be used. When we want to use it, we flip it on, put paper into the top, and at the bottom, out comes our shredded paper. Inside the shredder, we don't really care too much about the mechanics, we can probably guess there are some blades that cut up the paper, but it doesn't matter too much, as long as it does the job.

But let's say we are a shredder manufacturer. In which case we do care what's getting put into our products and how they work. We don't want there to be any paper jams and we don't want our users to cut their fingers. We have to make sure that everything is in good working order.

Back to functions. The way we write one is simple. We assign it to a variable (because we might want to use it later), then use our "function" statement, then round brackets, then curly brackets:

```
var shredder = function () {}
```

At the moment, we've made a shredder, but it doesn't process anything:

```
var num = 1

var adder = function () {
    num = num + 1
}
```

At the end of this code, `num` is still 1 because we haven't used the `adder` function. We've written the mechanics of the shredder but haven't turned it on. To turn it on, we know that we've saved the function to a variable called "adder". To use it, we use:

```
adder()
```

So if our code was:

```
var num = 1

var adder = function () {
    num = num + 1
}

adder()
adder()
```

We have run the `adder` function twice, so we add two 1s to our `num`, so the `num` variable is now 3.

We can also pick and choose what goes into the `adder` function. For example, in our real life shredder, if we put pink paper in there, we get

pink shreds. If we put in yellow paper, we get yellow shreds. The output is dependent on the input.

To make our output dependent on the input we can add an argument. Something that our function takes in its round brackets to use within the curly brackets:

```
var num = 1

var adder = function (plus) {
    num = num + plus
}

adder(3)
adder(5)
```

Here on the first run of the adder function, we are saying that “plus” is equal to 3, so num is now 4 (3 plus 1). On the second run of the adder function, we are saying that “plus” is now equal to 5 so now num is 9 (4 plus 5).

These examples of functions are not really very useful because they’re all pretty theoretical. What we really want to do is start to manipulate data within the confines of HTML and CSS too. Let’s start playing with our web pages instead!

Adding your scripts to your pages

At the moment, our fake code has just lived in the ether. We haven't set a place for us to write our code within our web pages. How do we do this?

The first thing we need to do is add our code into a file. Just like HTML and CSS, we need to use an extension that the browser knows is a Javascript file. We are going to use the “.js” file extension. For instance if we are making a slideshow Javascript file, we might want to call this “slideshow.js” or maybe some kind of parallax effect in the file called “parallax.js”.

Next we need to add it to the web pages we want. Not all pages will need all Javascript files. For instance, it might just be your homepage that needs the slideshow file.

To do this we need to pull our script into our HTML using a script tag. We add this at the very bottom of our HTML content:


```
<footer>
    Footer goes here...
</footer>

    <script src="slideshow.js"></script>
    <script src="parallax.js"></script>
</body>
</html>
```

Here we are adding two scripts to the bottom of our HTML, one with the filename “slideshow.js” and the other with the filename “parallax.js” — inside our files is where we will write our Javascript code.

Why do we add the code to the bottom of our HTML? We need the rest of our HTML and CSS to load before we can mess around with it!

jQuery

To make our lives a little bit easier, we're going to be working with a Javascript library called jQuery.

This was created in 2006 by a coder called John Resig, who at the time was 21 years old. He wanted to make a way to simplify writing Javascript code, especially when working with HTML and CSS, so he made a collection of functions that make the most common tasks quicker and easier to write.

He then packaged all these functions up into one big file and called it jQuery for anyone else to use.

Why did it he do that? Why not just sell it and make a ton of money? Well, John is a very nice, altruistic person and wanted to help other people write code quicker. And the more people saw and used his code, the better his reputation became. (Think of how mixtapes in rap music are given away to promote the artist.)

This type of free code is called open source programming and a lot of such code is used heavily in everything we coders do — HTML, CSS and Javascript are all open source and anyone can build a browser and contribute to its development.

John bought the domain for jquery.com, made a logo based on his favorite band Devo's "energy dome" hats, and added a download link.

Over time, more and more sites used the jQuery code library to help speed up development.

We'll be using jQuery in the rest of the guide, so to use it, we download it from **jquery.com** (the latest “production” version is good), then add it our websites.

We also need to add it our HTML as the first script so we can use it in the second, third, and so on, scripts:

```
<footer>
  Footer goes here...
</footer>

<script src="jquery.js"></script>
<script src="slideshow.js"></script>
```

How to work with jQuery

When creating jQuery, John Resig wanted to pick a short, simple symbol to use whenever he used the jQuery code, so that other coder's code looked shorter and cleaner. He picked the dollar sign “\$”. Whenever we use jQuery in our code, it will start with the \$.

The `$` is essentially a big function, just like our “adder” function or a “shredder” function. To use it we need to put round brackets after it:

```
$()
```

We then need to pick what HTML we want to alter. To do this, we can add a CSS-style selector in a string. For instance, where we want to pick a section with the class of “intro”, in CSS we’d pick it as “section.intro” — we can use this in jQuery too:

```
$("section.intro")
```

We need to put the selector in a string, as “section.intro” doesn’t mean anything in Javascript — it’s not a special word like true or false.

If we wanted to just pick all the links on the page, in CSS we’d do “a”, so in jQuery, we’d do:

```
$("a")
```

There are many jQuery functions we can use once we’ve picked the tags we want to play with. You can see them all on the jQuery website under “API” (short for “Application Programming Interface”). We’ll talk through some of the most popular ones.

Changing HTML in jQuery

Once we've picked our tag, we can change all the HTML inside it. To do this we pick the tags we want to change, then (“.” in jQuery) use the `HTML` function in jQuery:

```
$("#section.intro").html("New text!")
```

Because HTML is all letters and numbers, we use a string in our round brackets for HTML. However if we have something that variable we can add it in there too:

```
var num = 3
```

```
$("#section.intro").html(num + " items in basket")
```

Here we're adding the variable “num” to a string that stays the same.

We can even add a variable to a string to a variable:

```
var step = 2
```

```
var max = 5
```

```
$("#div.steps").html(step + " out of " + max)
```

Changing the CSS in jQuery

Just like with changing the HTML of the tag, we first need to select the tags we want to change. Rather than the HTML, we need to change the CSS, and in jQuery, there's a quick way to do this — the “css” function.

The previous “html” function took just one argument — the new HTML to change. The new “css” function takes two arguments (or parts between the round brackets). The first is the CSS rule you want to change, the second is what you want to change it to. These are usually in strings because they're not special words in the Javascript:

```
$("#header").css("background-color", "red")
```

```
$("#section.intro").css("font-family", "Arial")
```

```
$("#input[type=email]").css("padding", "0 15px 0 15px")
```

All are examples of the jQuery “css” function. As you can see most of the arguments are strings, but on some, you could use a number which will default to a pixel value:

```
$("#header").css("height", 60)
```

This is the same as:

```
$("#header").css("height", "60px")
```

I like to add the second version for clarity.

Events

So far, our jQuery code only changes the HTML and CSS when we load the page. This is similar to if we altered our HTML and CSS files manually — our users don't see any difference and aren't interacting with the page, they're just seeing an instant change. How do we make it so that when our user does something such as a click, only then do we change the page?

Let's take some HTML:

```
<header>  
    Furneux's Florist  
</header>
```

Let's introduce the concept in Javascript of when a user clicks on the header on our HTML, the HTML inside it changes. To do this we need to introduce events.

Events are a large part of how Javascript is useful in conjunction with HTML and CSS. Whenever we think of the phrases “when a user clicks...”, “when a user scrolls...”, “when a user resizes the browser...”, etc. we are adding in a Javascript event.

An event is essentially some code that is waiting to happen. One part of code may be run once, twice, or 200 times by a user, or never at all.

For instance, in our site, when a user clicks the header, we want the HTML inside to change. A user could click the header once, 10 times or never at all.

In jQuery, we can add some code into a linked Javascript file by selecting the tag we want the event to happen on:

```
$("header")
```

We then can add an event by adding the jQuery code “on”:

```
$("header").on()
```

What event do we want the event to happen on? This could be multiple events (such as when the mouse moves over and when the mouse clicks), but in our case we just want to do add an event on click:

```
$("header").on("click")
```

At the moment, when we click the header, nothing happens. Just like the `css()` jQuery function, we can add a second argument into the `on()` function. This will not be a string, but we want to run some code when a user clicks on the header:

```
$("header").on("click", function () {})
```

The end of this line of code looks a bit odd — we have a two round brackets, two curly brackets then a close round bracket. This closing round bracket belongs to the open round bracket at the start of “click”.

Just like with our other code, we can open up our curly brackets to give ourselves a bit of room to write our code in:

```
$("#header").on("click", function () {  
  
})
```

Now this looks even weirder. We have this close curly bracket and a close round bracket together on a separate line, like a strange “unibrow smiling” emoji, but just like before they belong to the first line of code.

In plain English, we are picking the header, then waiting for something to happen on a “click” event. When it does happen (and it may never happen), run this function.

What do we want to do in this function? We want to change the HTML, just like we did in a previous chapter:

```
$("#header").on("click", function () {  
    $("#header").html("Oh hi there")  
})
```

I like to indent (press “tab”) the code inside the function so it looks a little bit easier to read. You don’t need to, but it helps when you come back to it in a few months’ time or if you’re working with other people.

We don’t have to pick to change the header in this case. We can pick any other part of the page. If I click on the header, we could change the footer:

```
$("#header").on("click", function () {  
    $("#footer").html("A new secret footer!")  
})
```

Why you would want to do this may remain a mystery in this particular case, but what it does mean is we can now have control over other tags that aren't nearby to the tag our user interacts with.

We can also change anything we want to in our function. We could change the footer's HTML and the background color on the whole page too:

```
$("#header").on("click", function () {  
    $("#footer").html("A secret new footer!")  
  
    $("body").css("background-color", "pink")  
})
```

As long as all the changes are inside our curly brackets, we're good.

We can also have multiple events running on one page. For instance, we might have when we click a header, do something, or when we click a footer, do something else:

```
$("#header").on("click", function () {  
    $("#header").html("Clicked the header")  
})
```

```
$("#footer").on("click", function () {  
    $("#footer").html("Clicked the footer")  
})
```

Notice that because they're two completely separate user interactions, we keep them apart. Why? Remember an event could happen just once, multiple times or never at all. In this case, we could click the footer before the header. This means the bottom part of the code happens before the top part. It can get a bit confusing at first glance, but we're at the mercy of our users and we're preparing for anything that may happen and in any order!

Links and forms with events

Let's take an HTML navigation that looks like this:

```
<nav>  
    <a href="about.html">About</a>  
    <a href="blog.html">Blog</a>  
    <a href="contact.html">Contact</a>  
</nav>
```

When we click a link, we might want it to change background color, so based on the previous code we had, we might add something like this:

```
$("#a").on("click", function () {  
    $("#a").css("background-color", "red")  
})
```

When a user clicks one of these links, what happens? The first thing is the background color will change to red, but the second is the user will go to the next page. If they clicked on “About”, they will go to “about.html”. Both of these things will happen almost instantly, so the user will probably never see the background change because it’s too quick for them to notice.

How do we stop the event from going to the next page? We need to stop the default thing from happening. The default in this case is, when we click a link, we go to the next page. We need to stop this, and to do so we need to handle the event, then stop it. To do this we need to add an argument into our function:

```
$("#a").on("click", function (event) {  
    $("#a").css("background-color", "red")  
})
```

As this is an argument, we can call it whatever we want. It doesn’t have to be called “event” — it could be called “e”, “ev”, “potato”, “riksbigevent”, “abc” — but as we’re dealing with the default event, it makes sense to label it accordingly.

The next thing we need to do is stop that event. We use the Javascript function “preventDefault()” to do this on the original event:

```
$("#a").on("click", function (event) {  
    $("#a").css("background-color", "red")  
  
    event.preventDefault()  
})
```

I like to add it as the last thing in my event to remind myself that the final part is to stop the usual link click from happening.

We don't need to add this to tags like "header", "footer", "p", "span", etc., because when we usually click them, they don't do anything. We only need to do this when we click links or submit forms:

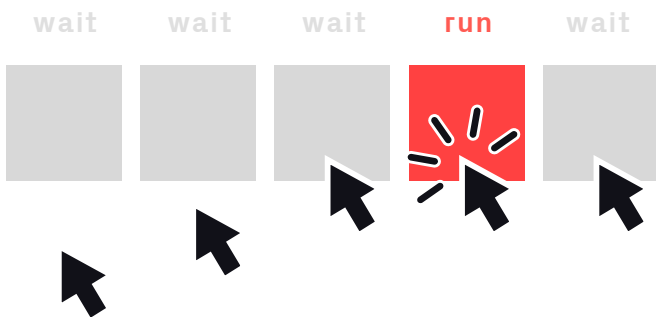
```
$("#form").on("submit", function (event) {  
    $("#body").css("background-color", "pink");  
  
    event.preventDefault()  
})
```

In this case, when we submit the form, we will change the whole page's background to pink and stop the form going to the next page.

Events in jQuery

We don't just have to use "click" and "submit" — there are plenty more! Here is just a small selection that we can use in our "on()" function:

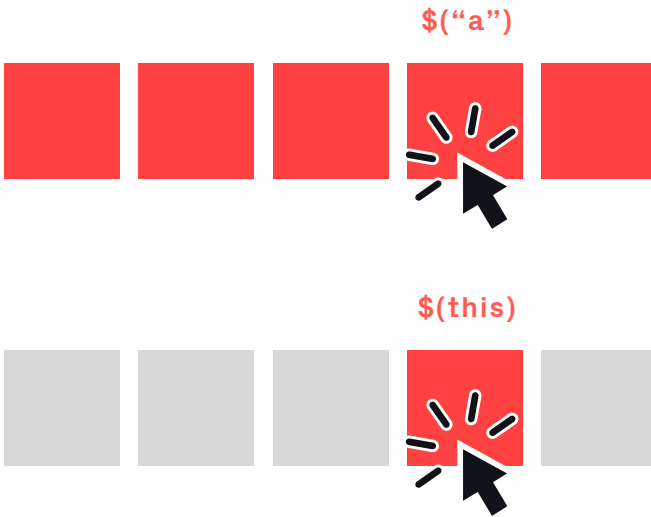
Running a function on click



- ♦ **dblclick** — double click rather than single click
- ♦ **mousedown** — when a user has their mouse pressed down
- ♦ **mouseup** — when a user has finished pressing their mouse down, a click is a combination of mousedown then mouseup
- ♦ **mousemove** — when a user moves the cursor with the mouse by any amount of pixels
- ♦ **mouseenter** — when a user moves the cursor over a tag, similar to CSS hover states
- ♦ **mouseleave** — when a user moved the cursor outside a tag after being inside it, similar to CSS when you unhover a tag
- ♦ **mousewheel** — when the user moves their mouse wheel in any direction
- ♦ **scroll** — when a user scrolls a tag if it has an overflow on it
- ♦ **resize** — when a tag is resized, usually used for textareas in forms
- ♦ **keypress** — when a user presses a key on their keyboard, usually used in form inputs
- ♦ **focus** — when a user gets a typeable cursor in a form input, little bit like click for form tags
- ♦ **blur** — when a user loses the typeable cursor from a form input, little bit like an “unclick” for input tags

- ♦ ***change*** — *when a user changes a form input's value, could be with typing into a text field, checking an checkbox or changing a select tag*
- ♦ ***contextmenu*** — *when a user right-clicks on a tag*

The special word — this — just selecting itself



What is this?

Let's go back to our links example:

```
$("#a").on("click", function (event) {  
    $("#a").css("background-color", "red")  
  
    event.preventDefault()  
})
```

What happens? When we click any link, we stop going to the next page and also change every single link's background to red. How do just make the link we clicked go red instead?

We need to introduce a new special word. We've had two special words so far — `true` and `false` — which are Javascript words that don't need quotation marks around them because Javascript knows what to do with them. We're going to add one more: `this`.

The special word, `this`, relates to the particular thing the function is related to. It doesn't have to be a tag, it could be other related items — we'll talk a little more about that later.

Another way to think about this is that it's the context for the function — in jQuery, it's the tag we're talking about.

When we use the word `this` inside the function's curly brackets in the code above, it refers to the tag the user has just clicked on. So instead

of making all the links change by using `$(“a”)`, we can use this to just refer to the single tag:

```
$(“a”).on("click", function (event) {  
    $(this).css("background-color", "red")  
  
    event.preventDefault()  
})
```

Notice that there's no quote marks around this. If we did add them, jQuery would look for a `<this>` tag which doesn't exist!

We can use this in any jQuery function, not just links. For instance, if we want to highlight paragraphs with a yellow color on click, we want to highlight just the tag our user clicked on:

```
$(“p”).on("click", function () {  
    $(this).css("background-color", "#fff182")  
})
```

Notice we don't need `“preventDefault()”` here as it's not a link or form. We could add it in, but the default event does nothing anyway so we'd be stopping nothing!

jQuery animations

So far, we've only talked about three jQuery functions: "html()", "css()" and "on()". jQuery has around 100 different functions we can use — most coders only use a selection of them and it's dependent on what we want our sites to do. In the next examples, we're mainly going to focus on animation functions in jQuery.

Let's make a navigation that shows itself when we click "Show menu". Our HTML may look something like:

```
<a href="menu.html" class="show-menu">Show menu</a>

<nav>
  <a href="about.html">About</a>
  <a href="blog.html">Blog</a>
  <a href="contact.html">Contact</a>
</nav>
```

Now the first thing we might want to do is hide that navigation by default. As we're doing this instantly on the page, we can start our script with:

```
$("nav").hide()
```

This script essentially gives the tag CSS that makes the display type "none".

Next we want to wait for our user to click the link with the class "show-menu":

```
$("#nav").hide()

$("#a.show-menu").on("click", function () {})
```

As it's a function, let's open our function out to make it a bit easier to read:

```
$("#nav").hide()

$("#a.show-menu").on("click", function () {

})
```

Next, as it's a link our user is clicking on, stop it going to the next page:

```
$("#nav").hide()

$("#a.show-menu").on("click", function (event) {
    event.preventDefault()
})
```

Now we can make the navigation show itself again:

```
$("#nav").hide()

$("#a.show-menu").on("click", function (event) {
    $("#nav").show()

    event.preventDefault()
})
```

So by default, the navigation is hidden, but only when a user clicks on the link with the class “show-menu” does the navigation appear again. First line instant, the rest later.

The “show()” and “hide()” jQuery functions happen instantly, but we may want to fade the navigation in from hidden, so we can use a new function called “fadeIn()”:

```
$("#nav").hide()

$("#a.show-menu").on("click", function (event) {
    $("#nav").fadeIn()

    event.preventDefault()
})
```

Just one simple tweak can change it from instant to a fade effect. Nice! Another alternative could be the “slideDown()” method:

```
$("#nav").slideDown()
```

At the moment, once we show the navigation, if we click the same link again, the navigation will continue to show. How do we make it disappear if it's shown? We need to use a different jQuery function!

If we want to make it instant show and hide, we can use the “toggle()” jQuery function. We want to fade between in and out. To do this, we can use the “fadeToggle()” method:

```
$("#nav").hide()

$("#a.show-menu").on("click", function (event) {
    $("#nav").fadeToggle()

    event.preventDefault()
})
```

By default, the fade happens over one second. Javascript uses milliseconds for its timings (unlike CSS that uses “0.5s” for example). One second is 1,000 milliseconds. If we want to run a quicker fade at say half a second, we need to say 500 milliseconds.

To change the function, we need to add one argument to it, which is the number of milliseconds to fade over:


```
$("#nav").hide()

$("#a.show-menu").on("click", function (event) {
    $("#nav").fadeToggle(500)

    event.preventDefault()
})
```

CSS and jQuery together

The jQuery animations are pretty good, but they don't give us great control over how they work. For instance, we might want to fade in and have a small movement upwards at the same time. jQuery doesn't give us as much control as we might need as web designers.

Let's go back to the definitions of HTML, CSS and Javascript. HTML is for content, CSS is for our look and feel, and Javascript is for our user's interactions. If we were to say how quick a fade should be, or if there should be a slight movement on a transition, shouldn't it really go into CSS, as that's where the look and feel is? The answer is yes, probably.

There's no 100 percent right or wrong answer to where you add your transition — if a jQuery fade in is good for you, do it, if you want more control, CSS and jQuery together might be better.

As we're creative people, we probably want more control over our design work. To do this, we'll add our design control in CSS instead, but how?

To this, we still need to work with our user's interactions in Javascript, but instead of changing the CSS directly, we're going to add an HTML class instead, then let CSS handle the work.

Let's take some HTML that looks like this:

```
<nav>
  <a href="about.html">About</a>
  <a href="blog.html">Blog</a>
  <a href="contact.html">Contact</a>
</nav>
```

Previously, we had some Javascript code to make the background red on any clicked link:

```
$("#a").on("click", function (event) {
  $(this).css("background-color", "red")

  event.preventDefault()
})
```

We're picking any link, then if any is clicked, we change the background to red, then stop the link going to the next page.

The first bit we want to fix is the fact that we have some styling (background going to red) in Javascript rather than CSS.

If we were to change one of the links in HTML to look different than the others, we'd add an HTML class, for example:

```
<a href="blog.html" class="highlighted">Blog</a>
```

We're not going to do this manually as it depends on which link the user clicks.

In our Javascript, we're going to replace our "css()" with a new jQuery function called "addClass()" — watch out for the capital C:

```
$( "a" ).on( "click", function (event) {  
    $(this).addClass("highlighted")  
  
    event.preventDefault()  
})
```

We're going to add a class of "highlighted" to our link when we click it. The class name of "highlighted" is up to us, whatever makes sense when we hook it up.

At the moment, there's no style in our stylesheet to make this go red, but we can add one. Our default link background may be white but we need one with red too:

```
a {  
    background-color: white;  
}  
  
a.highlighted {  
    background-color: red;  
}
```

Notice that our original HTML stays the same. It's the Javascript adding the class to our HTML instead. Now we have our styling defined in our style sheets, which makes way more sense!

Because we have our styling in CSS rather than Javascript, we can use CSS transitions to fade between the backgrounds:

```
a {  
    background-color: white;  
    transition: background-color 0.5s;  
}  
  
a.highlighted {  
    background-color: red;  
}
```

There isn't a way in jQuery alone to fade in a background, however with CSS we can make it fade and it makes more logical sense to keep it in there too.

We may want to turn the class off on click if it's already on. A little bit like the “fadeToggle()” function we added. The function we would need to toggle a class would be “toggleClass()”.

```
$("#a").on("click", function (event) {  
    $(this).toggleClass("highlighted")  
  
    event.preventDefault()  
})
```

By default, the class isn't on any of the links, so when we click any of them, it will add the class of “highlighted”. Next time we click, if it's already on, take it away. If click another link, turn that on.

Fade and transitions with CSS and jQuery

In the last example we used a simple background transition, but what if we want to copy and change the “fade()” jQuery function? How do we do that?

Let's use the same HTML as an earlier example:

```
<a href="menu.html" class="show-menu">Show menu</a>

<nav>
  <a href="about.html">About</a>
  <a href="blog.html">Blog</a>
  <a href="contact.html">Contact</a>
</nav>
```

And start with the accompanying Javascript:

```
$("#nav").hide()

$("#a.show-menu").on("click", function (event) {
  $("#nav").fadeToggle(500)

  event.preventDefault()
})
```

The first thing we're going to do is move some of the code into our CSS. Let's move the hidden nav code from jQuery into CSS:

```
nav {  
    opacity: 0;  
}
```

We can now change our Javascript to toggle a class using “toggleClass()” instead of the “fadeToggle()” animation:

```
$("#a.show-menu").on("click", function (event) {  
    $("#nav").toggleClass("show")  
  
    event.preventDefault()  
})
```

Again, the class of “show” is a name we picked to describe what we’re doing. We can now use this class of “show” in our CSS:

```
nav {  
    opacity: 0;  
    transition: all 0.5s;  
}  
  
nav.show {  
    opacity: 1;  
}
```

This will fade the menu over half a second in both fading in and fading out.

We might also want to add a movement to our fade in and fade out. We can do this in our CSS files, rather than our Javascript files:

```
nav {  
  opacity: 0;  
  transform: translate(0, 10px);  
  transition: all 0.5s;  
}  
  
nav.show {  
  transform: translate(0, 0);  
  opacity: 1;  
}
```

We're starting the navigation as hidden and 10 pixels further down the page, based on where it would usually be. When we add the class of "show", we are giving it full opacity and moving the nav where it should be (no translate across or down).

If your navigation is using positioning to place it over other tags, by default, a user won't see it, but a user's mouse will be able to interact with it, as it's there but see-through. We can add one more CSS rule called "pointer-events" to let the mouse ignore the tag when hidden, but don't ignore when shown:

```
nav {  
  opacity: 0;  
  transform: translate(0, 10px);  
  pointer-events: none;  
  transition: all 0.5s;  
}
```



```
nav.show {  
    transform: translate(0, 0);  
    pointer-events: auto;  
    opacity: 1;  
}
```

Again, the Javascript stays the same, but we're controlling look and feel in our CSS files instead.

Timers and intervals

Javascript not only deals with user interactions but can also control other events. We might want to run a function every second if we have a clock on the page, or we might want to run some code after five seconds to move to the next slide. These aren't necessarily user-driven, but they're to do with the interactivity of web pages.

We'll start by talking about intervals — events that run at a specified number of milliseconds. Intervals are part of Javascript, not jQuery, so they don't start with the dollar sign. It's also good practice to save an interval to a variable so we could stop it later.

What we'll make is a small counter. We'll start with “1” and every three seconds, we'll add one to it. First let's make a variable with the count:

```
var count = 1
```

Next we want to add our interval that adds one every three seconds. To do this, we'll be using the Javascript “setInterval()” — watch out for the capital I in Interval — that take two arguments. The first is a function to run, the second is how often we want to run it:

```
var count = 1
```

```
var timer = setInterval(function () {}, 3000)
```

This should look a little similar to “on()” in jQuery. We’re adding 3,000 as Javascript counts in milliseconds, not seconds. We can open up the curly brackets to make our code a bit easier to read:

```
var count = 1

var timer = setInterval(function () {

}, 3000)
```

Next we want to add one to the count every three seconds:

```
var count = 1

var timer = setInterval(function () {
    count = count + 1
}, 3000)
```

At the moment the variable count goes up but our user can’t see it. We might want to update our HTML every three seconds to reflect this new variable:

```
var count = 1

var timer = setInterval(function () {
    count = count + 1

    $("div.counter").html("The count is now: " + count)
}, 3000)
```

In our HTML, let's imagine we have a `div` tag with the class of "counter" — we're setting its content to a string that stays the same ("The count is now:") added to a variable (`count`).

What if we want to stop the interval from happening when we count up to 10? We need to stop it!

To stop intervals, we need to use "clearInterval" (capital I again!) which takes one argument — the variable name for the interval. In our case, it's called "timer" as we said "var timer = ...". We need to do a check — if the count is equal to 10, stop the timer:

```
var count = 1

var timer = setInterval(function () {
    count = count + 1

    $("div.counter").html("The count is now: " + count)

    if (count === 10) {
        clearInterval(timer)
    }

}, 3000)
```

One-off delays — timeouts

In the same manner, we might not want to do to a constant interval but a delay. In Javascript, this is called “`setTimeout()`”.

Just like with “`setInterval()`”, we have two arguments, the function to run and then the time we want to delay the function from running.

Let's say we want to give our user a warning if they haven't done a task after 10 seconds:

```
var delay = setTimeout(function () {}, 10000)
```

We can separate our curly brackets to make it easier to read:

```
var delay = setTimeout(function () {  
  
}, 10000)
```

Then we can do any code inside our function:

```
var delay = setTimeout(function () {  
    $("div.warning").fadeIn()  
}, 10000)
```

We can cancel the delay anywhere else in our code. For example, let's say we click our “done” div tag, we might want to stop our warning fading in, using the “`clearTimeout()`” function. This works in a very similar way to “`clearInterval()`”:

```
var delay = setTimeout(function () {  
    $("div.warning").fadeIn()  
}, 10000)  
  
$("div.done").on("click", function () {  
    clearTimeout(delay)  
})
```

We can also have delays within events. For instance, if we wanted to delay the HTML from changing for a second after a click event, we could add:

```
$("#header").on("click", function () {  
    var delay = setTimeout(function () {  
        $("h1").html("I was delayed by 1 second")  
    }, 1000)  
})
```

This is looking a little more complex, but we have a function for the click event. Inside that is a function for a timeout that runs after 1,000 milliseconds.

The background is a solid bright yellow. Overlaid on this are two large, white, stylized letters. On the left is a large 'L' that is partially cut off by the left edge of the frame. On the right is a large 'S' that is also partially cut off by the right edge. The text 'Document and window' is centered in the space between these two letters.

**Document
and window**

Some of the more fun Javascript effects don't use the "click" event but the "scroll" event.

But first we need to talk about two new special words in Javascript: "document" and "window", which are to describe two concepts of the web page and the browser. What's the difference between the two?

Let's take a long blog post web page. The length of this page depends on several things — how many words there are, how large the font size is, how much padding or margin each paragraph has. The page might be longer on a mobile device as there's less horizontal space to fit all the words in.

The page in Javascript is described by the special word document.

The page is the thing we scroll within our browser. If we go from page to page, the length of the page will most likely change, but the browser itself doesn't. On laptops and desktop computers, you can resize your browser by dragging the corners but generally it stays the same size.

In Javascript, the browser has the special word window.

Basically we're looking through our window (browser) to view a portion of our document (page).

When it comes to events, which one you pick depends on what your user is doing. If your user is scrolling the page, we will add an event to our document special word. If your user is resizing the browser, we will add an event to our window.

We'll concentrate on scrolling the page for now. These effects are collectively known as the "parallax" effect. We added in a basic version with the CSS rule "background-attachment: fixed" earlier in the guide, but with Javascript, we can have much finer control of the effect.

Let's say we have a long page where after a certain amount of pixels down we want to change the background color of the whole page. We might have in our CSS something that looks like this:

```
body {  
    background-color: white;  
    transition: background-color 1s;  
}  
  
body.scrolled {  
    background-color: yellow;  
}
```

By default, we have a background color of white on the whole page, but when we add a class of "scrolled" to the whole page, we'll transition the background to yellow over one second.

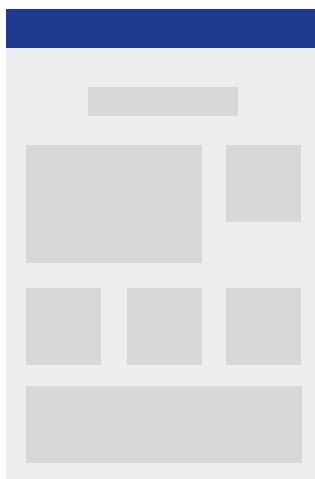
At the moment, the "scrolled" version will never happen until we add it into our Javascript.

*The whole page is called the document,
the part we can see is in the window*

window



document



First of all we need to select our new special word in jQuery:

```
$(document)
```

We're going make it do an event on scroll:

```
$(document).on("scroll", function () {})
```

Let's open out our curly brackets:

```
$(document).on("scroll", function () {  
  
})
```

This event will happen every single pixel a user moves the page. Generally it will happen a lot more than click events. Next we want to find how far down the page we've scrolled. We can do this with a jQuery function called "scrollTop()" which gives us the number of pixels we've scrolled. Let's save it as a variable to use later in the event:

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
})
```

We want to see if this new variable called pixels is bigger than 200 or not. If it is, add this new class to the "body" of the web page (not the document itself as there is no concept of "document" in CSS):

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
  
    if (pixels > 200) {  
        $("body").addClass("scrolled")  
    }  
})
```

At the moment, this works just one-way. As we scroll down the page, we'll get the yellow background transition, but if we try to scroll back up, the opposite doesn't happen. There's no fade to white. To add it in, we need to remove the class if the pixels are less than 200:

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
  
    if (pixels > 200) {  
        $("body").addClass("scrolled")  
    } else {  
        $("body").removeClass("scrolled")  
    }  
})
```

We can use this same technique for showing and hiding fixed headers. For example, in our CSS we might have a header that is styled like this:

```
header {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
    height: 80px;  
    transition: top 1s;  
}  
  
header.hide {  
    top: -80px;  
}
```

By default, we are showing the header, but we might want to hide it after scrolling with the class of “hide” (again, hide is a class we’ve just made up, it could be “potato” if that made more sense).

The Javascript for hiding and showing the header is very similar to changing the background color on the page:

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
  
    if (pixels > 200) {  
        $("header").addClass("hide")  
    } else {  
        $("header").removeClass("hide")  
    }  
})
```

The only thing we've changed is which tag and what class to add or remove.

Javascript parallax — movement based on scroll

The previous parallax examples have been binary — fade in or out, change background to white or yellow. What if we want to do more complex parallax effects? We can add some mathematics to our Javascript.

Before you run away, let me reassure you. All the math we'll add is pretty simple algebra that the average 11 year old could understand. Don't panic!

Let's say we have a header tag that we want to scroll slightly faster than the rest of the page, to give the page some depth. This effect would be solely based on the amount of pixels down the page we are. We can't add anything other than the default in CSS, but we want to change the CSS transform of the header.

For example, we want to change:

```
header {  
    top: 10px;  
}
```

We want to tweak the “10px” to be “11px”, then “12px”, then “13px” as we scroll further and further.

To do this we have something similar to what we did before:

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
})
```

When the pixels variable is 0 (right at the top of the page), we want the header to not move. When the pixels variable is 100px down, we want the header to be 5px down. When the pixels variable is 200px ...

What's the mathematics for that? The header movement should be 1/20th of the page scroll (0/20 is 0, 100/20 is 5, 200/20 is 10, etc). Let's save this to a new variable called movement:

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
  
    var movement = pixels / 20  
})
```

We then want to move our header at this slower movement down the page using the CSS top position:


```
$(document).on("scroll", function () {
    var pixels = $(document).scrollTop()

    var movement = pixels / 20

    $("header").css("top", movement + "px")
})
```

Remember we're adding a variable to a string ("px") to give the number a unit (we could do it as a "vh", "em" or "%" as an alternative).

Progress bar scroll

On some longer websites, you'll see a progress bar that gets wider the further down the page you've scrolled. This depends on a few things — the height of the page, how far down the page you've scrolled, but also the height of the browser.

Why does this depend on the height of the browser? Let's say our page is 3,000 pixels tall and our browser is 600 pixels tall. At the top of the page, we are at 0 pixels down, but where are we when we hit the bottom of the page? We are 2,400 pixels down — we can see the bottom 600 pixels out of 3,000, so the top of the scroll is 2,400 pixels (3,000 minus 600).

If our browser was shorter at 300 pixels, the bottom of the page's scroll position would be 2,700 pixels down (3,000 minus 300). If our browser

was taller at 1,200 pixels, there's less page to scroll, so the bottom of the page scroll would be 1,800 pixels down (3,000 minus 1,200).

For our progress bar, when it's at 0 percent, the page is at the top (0 pixels down). When it's at 100 percent though, the page is at the bottom of scroll, which is the page height minus the browser height.

To make our progress bar, we would add an empty tag at the top of our HTML, something like:

```
<div class="progress"></div>
```

We'd then style it up in CSS:

```
div.progress {  
  position: fixed;  
  top: 0;  
  left: 0;  
  height: 4px;  
  
  background-color: red;  
}
```

Notice there is no width rule here. We'll be adding this in our Javascript:

```
$(document).on("scroll", function () {  
  var pixels = $(document).scrollTop()  
})
```

We start with something quite familiar — whenever a user scrolls, save the scroll top position to a variable called `pixels`.

Let's work out the total number of pixels we could scroll over — the document height minus the window height — and save it to a variable called `total`:

```
$(document).on("scroll", function () {
    var pixels = $(document).scrollTop()

    var total = $(document).height() - $(window).height()
})
```

We have two variables that are numbers. `Pixels` is how far we are down the page, and `total` is the maximum we can scroll. When `pixels` is 0, we want the progress bar to be 0 percent. When `pixels` is equal to `total`, we want the progress bar to be 100 percent. When `pixels` is half the `total`, the progress bar should be 50 percent. We need to make the width based on the ratio of `pixels` to the `total`:

```
$(document).on("scroll", function () {
    var pixels = $(document).scrollTop()

    var total = $(document).height() - $(window).height()

    var percentage = 100 * pixels / total
})
```

If the `total` is 1,000 and `pixels` scrolled is 500, the new variable `percentage` would be 100 times 500 divided by 1,000, which is 50. If

pixels is 0, percentage is 0. If pixels is the same as total, percentage is 100.

We can then set the progress bar to be the percentage across:

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
  
    var total = $(document).height() - $(window).height()  
  
    var percentage = 100 * pixels / total  
  
    $("div.progress").css("width", percentage + "%")  
})
```

Remember that percentage is a variable of a number. We need to add it to a unit, otherwise it would default to "px". We want the progress bar to go from 0 to 10 percent, then 20 percent, then finally up to 100 percent, not between 0 and 100px.

Fade in tags on scroll

A common web design technique that uses the parallax effect is the idea of scrolling down a web page and then having certain elements pop into life as your user goes further and further down.

So how do we know this is using a parallax-style effect? As our users scroll pixel by pixel down the page, we can do a check to see whether or not we should animate the tag in or not. The telltale signs of a parallax effect is whether the effect depends on pixel by pixel scrolling.

Let's build up this effect and think about what we want to happen. We're going to say as soon as certain tags are "in view" (e.g. the user can see some portion of them), then we want to fade that tag in. Every time the user scrolls, we want to do a check, and that check will be look at each tag and if it's within the browser window, fade it in.

We're going to put this effect on each `<section>` tag on the page but you can change this to fit which tags or CSS selection you want.

We want to do this fade effect in CSS itself as this is the style of the page. We want to toggle a class on each section using Javascript. We'll start with the sections all having no opacity but when they get a newly added class, they'll fade in. Let's call this newly added class "visible".

```

section {
    opacity: 0;
    transition: opacity 1s;
}

section.visible {
    opacity: 1
}

```

Here we're saying by default, make all section tags have no opacity and if the opacity does change, transition its change over a second. Later, we'll add in a class using Javascript called "visible" to then fade it to full opacity.

Next, in our Javascript file, we want to do an event on scroll. Where do we add that event? On the whole page as we scroll it, so adding it to the document.

```

$(document).on("scroll", function () {

})

```

Next we want to find out how far down the page we've scrolled. Let's save this to a variable called pageTop (remember, you can call variables whatever you like). We can use jQuery's scrollTop() function to calculate how far down the page we are.

```

$(document).on("scroll", function () {
    var pageTop = $(document).scrollTop()
})

```

As the names `pageTop` and `scrollTop()` suggest, we're finding the amount of pixels down the document the top of the browser currently is. If we haven't scrolled, the top of the browser window is currently 0 pixels down the document. But the threshold for tags coming into view isn't the top of the browser window but the bottom of the browser window. How do we get the bottom instead of the top?

It's easier than it sounds. We just need to add the height of the browser window to the `pageTop` variable. We're going to save this as another variable called `pageBottom`. How do we get the height of the browser window? We don't want to select the document this time (that's the whole page itself), but the window and get its height.

```
$(document).on("scroll", function () {
    var pageTop = $(document).scrollTop()
    var pageBottom = pageTop + $(window).height()
})
```

A quick reminder: we don't need to put quotation marks around `window` and `document` as they're both special words in Javascript to denote the browser and the page respectively.

So far, when we scroll down the page, we're saving numbers to two variables – not very exciting. What we want to do next is independently check each section tag to see if that tag's top position is above the bottom of the page, and if it is, fade it in using the CSS class we added.

Firstly, let's save the tags we want to animate to another variable:

```
$(document).on("scroll", function () {  
    var pageTop = $(document).scrollTop()  
    var pageBottom = pageTop + $(window).height()  
    var tags = $("section")  
})
```

Next, we want to loop over each one of those tags to independently check it. Half the tags might want to fade in but the other half might not. Let's use a loop to go over each one.

```
$(document).on("scroll", function () {  
    var pageTop = $(document).scrollTop()  
    var pageBottom = pageTop + $(window).height()  
    var tags = $("section")  
  
    for (var i = 0; i < tags.length; i++) {  
  
    }  
})
```

The "tags" variable is an array (or list) of tags we've picked. We're starting our loop at zero, as computers count from there. We're making sure that the loop is only for the amount of tags we have in the tags array and we're adding one to the loop each time. Next let's pick each individual tag within the loop itself.


```

$(document).on("scroll", function () {
    var pageTop = $(document).scrollTop()
    var pageBottom = pageTop + $(window).height()
    var tags = $("section")

    for (var i = 0; i < tags.length; i++) {
        var tag = $(tags[i])
    }
})

```

Remember that variables by their very name are changeable. Each time we run the loop, the new "tag" variable will be each individual tag to use. What do we want to do with each individual tag? We want to use jQuery to check its top position and if it's visually higher up the page than the "pageBottom" variable, add a CSS class to it.

To get a tag's position, we can use jQuery. This is why we'll have the tag variable wrapped with a dollar and brackets, so that we can use jQuery's ".position()". This will give us back an object that contains its top and its left position. We want to just use the top, so we can select it by using ".position()" then ".top" — no brackets after that as we're just picking something from an object.

```
$(document).on("scroll", function () {  
    var pageTop = $(document).scrollTop()  
    var pageBottom = pageTop + $(window).height()  
    var tags = $("section")  
  
    for (var i = 0; i < tags.length; i++) {  
        var tag = $(tags[i])  
  
        if ($(tag).position().top < pageBottom) {  
            $(tag).addClass("visible")  
        }  
    }  
})
```

Currently this will be a one-way effect: as soon as the tag is within the viewable area, it adds the "visible" class to it. If it goes outside the viewable area, it doesn't remove the tag to fade it back out. We could easily add this with an "else" condition to say if it goes back outside the viewable area, fade out again.

```
$(document).on("scroll", function () {  
    var pageTop = $(document).scrollTop()  
    var pageBottom = pageTop + $(window).height()  
    var tags = $("section")  
  
    for (var i = 0; i < tags.length; i++) {  
        var tag = $(tags[i])  
  
        if ($(tag).position().top < pageBottom) {  
            $(tag).addClass("visible")  
        } else {  
            $(tag).removeClass("visible")  
        }  
    }  
})
```

A common alternative to this technique is to not just fade in the tag but to make it look like it's moving upwards too. As this is the visual styling of how it's fading it, we just need to alter our CSS styles to make it look different when that class is added or removed.

```
section {  
    opacity: 0;  
    transform: translate(0, 20px);  
    transition: all 1s;  
}  
  
section.visible {  
    opacity: 1;  
    transform: translate(0, 0);  
}
```

We're using the CSS transform rule rather than the top rule as it's a little smoother plus we want to clash with any other top rules we might have. We're starting the sections with no opacity and 20px further down the page compare to the final position. Then when we add the "visible" class, we are making it full opacity and moving it to its correct position.

This is a really powerful web design technique that looks a lot more complicated than it actually is. In reality it's only around 15 lines of Javascript with 9 lines of CSS. Sometimes the most visually stunning web design techniques can be done in under 100 and even under 25 lines of code.

Counting numbers in Javascript

Earlier in the guide, we briefly touched on the concept of arrays — lists for holding data. Let's look at an example:

```
var colors = ["red", "white", "black"]
```

How do we get data out of our array? How do I pick what the first color is?

In computing, all counting starts at zero. The first item in the list is number 0 in computers. Think of it like scrolling down the page — the first pixel on the page is actually 0 pixels down. The second pixel is 1 pixel down.

To get the first color out of the colors array we can do:

```
var color = colors[0]
```

As the array has three items, to get the last color out of our array, we can add:

```
var color = colors[2]
```

We have counted 0, then 1, then 2 to get the 3rd item.

What if we want to know about how items are in the array? We can just get Javascript to count:

```
var total = colors.length
```

Here, the “total” variable would be equal to 3, not 2 — when we’re counting each item within the array, we start counting from 0, but when counting the size of the array, we count like normal humans.

Let’s say we didn’t know how many items are in the array. How do we get the last item?

```
var last = colors.length - 1
```

```
var color = colors[last]
```

The last variable is 2 (3 minus 1). We can then use that variable to get the last color. We could shorten this to:

```
var color = colors[colors.length - 1]
```

Where is this useful? Let’s say we want to cycle through these colors to change the background. We want to start at the first one, then loop through those every five seconds:

```
var num = 0  
var colors = ["red", "white", "black"]
```

We're starting at 0 (not 1) with the variable `num`, and saying what the colors are. Next we need to set an interval to run the change every 5,000 milliseconds:

```
var num = 0
var colors = ["red", "white", "black"]

var interval = setInterval(function () {}, 5000)
```

Next let's open up those curly brackets:

```
var num = 0
var colors = ["red", "white", "black"]

var interval = setInterval(function () {

}, 5000)
```

We want to increase the `num` in the interval, find the new color and change the background to that new color:

```
var num = 0
var colors = ["red", "white", "black"]

var interval = setInterval(function () {
    num = num + 1

    $("body").css("background-color", colors[num])
}, 5000)
```

This will work for the first three, but we don't have a fourth and fifth color! Let's make sure we reset the num if it's too big:

```
var num = 0
var colors = ["red", "white", "black"]

var interval = setInterval(function () {
  if (num < colors.length - 1) {
    num = num + 1
  } else {
    num = 0
  }

  $("body").css("background-color", colors[num])
}, 5000)
```

Let's check if the variable num is less than the length of colors and minus one (as we'll be adding one), if it is less, add one, if it isn't, make num go back to the start (0).

Getting data out of objects

Just like we did with arrays, let's go back and look at objects. These are the data structure we could use to describe one thing — it could be anything from a user to a pop band, as long as it's a singular thing.

Let's look at an example user object:

```
var user = {  
  name: "Rik Lomas",  
  username: "riklomas",  
  age: 32,  
  isAdmin: false  
}
```

We make an object with curly brackets, then give each part of the description of that object a “key” (think of it like an HTML class attribute), then a “value” for each key.

The curly brackets are one of the more confusing parts of Javascript. We use them both in statements (function, if, for, etc), and in describing an object. We just have to be hyper-aware of looking out for the difference. Statements will always have round brackets in front of their curly brackets, whereas objects just have curly brackets.

How do we get our user's age from the object? We pick the object's variable name, use a “.” then the name of the key:

```
user.age
```

...as the variable is “user” and the part of the user we want is the age key. To pick the user’s username, we would do:

```
user.username
```

Objects don't need the white space between each line, but it's easier to read if the object is large. On smaller objects, you might see it all in one line:

```
var color = { title: "Red", hex: "#ff4141" }
```

To get the hex value out of this color variable, we would do:

```
color.hex
```

To add one more level of complexity, we might have an array of objects. Let's take for instance a set of three colors:

```
var colors = [  
  { title: "Red", hex: "#ff4141" },  
  { title: "White", hex: "#f8f8f8" },  
  { title: "Black", hex: "#111111" }  
]
```

Usually we'd have our array all on one line. The white space doesn't matter, whatever is the most readable for us as coders.

How do we get the hex value “#ff4141” from our colors variable? Well, it’s in the first object in the array (number 0 so we need to get colors[0] first), then we want the hex key from that object:

```
colors[0].hex
```

Why is this even useful? Good question. Let’s take our previous random colors example using “setInterval()”:

```
var num = 0
var colors = ["red", "white", "black"]

var interval = setInterval(function () {
  if (num < colors.length - 1) {
    num = num + 1
  } else {
    num = 0
  }

  $("body").css("background-color", colors[num])
}, 5000)
```

We might want to have more changes within the loop. For instance, we might want to have some text changes as well as a color change. We can switch our plain array with an array of objects and add in the changes to our CSS, plus add in an HTML change:

```
var num = 0

var colors = [
  { title: "Red", hex: "#ff4141" },
  { title: "White", hex: "#f8f8f8" },
  { title: "Black", hex: "#111111" }
]

var interval = setInterval(function () {
  if (num < colors.length - 1) {
    num = num + 1
  } else {
    num = 0
  }

  $("body").css("background-color", colors[num].hex)

  $("div.current-color").html(colors[num].title)
}, 5000)
```

We're still counting over an array, but each item in the array has a lot more detail about it.

jQuery prepend, append, before and after

There are plenty of examples of where we might want to not change the internal HTML of something but add more HTML to it inside. So far, we've been using "html()" in jQuery to change all of the HTML inside tags. There are some more jQuery functions we can use to alter our HTML.

Let's take some starting HTML that looks like this:

```
<header>
  <h1>Lawrence Gosset</h1>
</header>
```

First of all we'll look at putting some extra content within the h1 tag. The first thing we'll add is some HTML at the end of Lawrence's name:

```
$("h1").append(" is the best")
```

Notice the space, otherwise it would be directly after the last "t" in Lawrence's name. This will make our HTML look like:

```
<header>
  <h1>Lawrence Gosset is the best</h1>
</header>
```

If we wanted to add content before Lawrence's name, we can use "prepend()":

```
$("#h1").prepend("<span>This is</span> ")
```

Notice that I've added some extra HTML that wasn't there before. We can do this to all of our HTML changes, including in "html()". This will turn our code into:

```
<header>
  <h1>
    <span>This is</span> Lawrence Gosset
  </h1>
</header>
```

What if we don't want our new content to be in the HTML tag, but before or after the tag instead? jQuery has you covered!

To place content after the tag, we can use:

```
$("#h1").after("<h2>Coder and teacher</h2>")
```

To turn our HTML into:

```
<header>
  <h1>Lawrence Gosset</h1>
  <h2>Coder and teacher</h2>
</header>
```

To add content before we can use the "before()" jQuery function:

```
$("#h1").before("<p>Hi there, I am...</p>")
```

To get:

```
<header>
  <p>Hi there, I am...</p>
  <h1>Lawrence Gosset</h1>
</header>
```

Remember this will add content to any of the selected tags! For instance if we have a navigation:

```
<nav>
  <a href="about.html">About</a>
  <a href="blog.html">Blog</a>
  <a href="contact.html">Contact</a>
</nav>
```

If we had in our jQuery:

```
$("#nav a").append(" - New!")
```

Our HTML would look like:

```
<nav>
  <a href="about.html">About - New!</a>
  <a href="blog.html">Blog - New!</a>
  <a href="contact.html">Contact - New!</a>
</nav>
```

If you want to be more specific, add an HTML class to the tags you like, just like when filtering by class in CSS.

“for” loops and jQuery’s HTML functions

Just having these jQuery HTML functions as standard is pretty useful when you combine them with Javascript’s “for” loops.

For instance, you might have some data in your Javascript that you may want to display in your HTML. When this data is in an array, this is where a combination of “for” loops and jQuery’s HTML functions comes in handy.

Let’s say we have a shopping list:

```
var shopping = ["sausage", "eggs", "bacon"]
```

We may want to go through our list and add it to our HTML. Our standard HTML may look like:

```
<h3>My shopping list</h3>
```

```
<div class="list">
```

```
</div>
```


Currently our `div` tag is empty and we want to fill it with three paragraph tags based on our data.

In our code, we can use a “for” loop to go through the data one by one. Firstly we need to start at 0 (“sausage”), then go to 2 (“bacon”). We may not know how many things are in our list, so we really want to go less than the total length of the list (0 to less than 3, not including 3).

```
var shopping = ["sausage", "eggs", "bacon"]

for () {}
```

In our for loop, we want to start at 0. We'll give it a variable name of “i”, just for short

```
var shopping = ["sausage", "eggs", "bacon"]

for (var i = 0) {}
```

Next we need to go up to 3 but not include 3 (less than 3, not less than and equal to 3):

```
var shopping = ["sausage", "eggs", "bacon"]

for (var i = 0; i < shopping.length) {}
```

Next, we need to tell the variable `i` to go up by 1 in each loop. We could use `i = i + 1` but a shortcut is `i++`:

```
var shopping = ["sausage", "eggs", "bacon"]

for (var i = 0; i < shopping.length; i++) {}
```

Next we can open our curly brackets to give ourselves some room to code in:

```
var shopping = ["sausage", "eggs", "bacon"]

for (var i = 0; i < shopping.length; i++) {

}
```

Next, we need to get each shopping item one by one. The first would be `shopping[0]`, the second would be `shopping[1]` and last would be `shopping[2]`. We already have the variable “`i`” going up one by one starting at 0 so we could use that inside our loop:

```
var shopping = ["sausage", "eggs", "bacon"]

for (var i = 0; i < shopping.length; i++) {
    var item = shopping[i]
}
```

Using this item variable, we can add our “sausage”, “eggs” and “bacon” to the div with the class of “list” in our HTML:

```
var shopping = ["sausage", "eggs", "bacon"]
```

```
for (var i = 0; i < shopping.length; i++) {  
    var item = shopping[i]  
  
    $("div.list").append(item)  
}
```

This will give us in our list:

```
<div class="list">  
    sausageeggsbacon  
</div>
```

Let's make each item be in its own paragraph instead by altering our Javascript. We know that the `<p>` and `</p>` tags will stay the same, so we can make them into strings but the variable changes (hence the name variable):

```
var shopping = ["sausage", "eggs", "bacon"]  
  
for (var i = 0; i < shopping.length; i++) {  
    var item = shopping[i]  
  
    $("div.list").append("<p>" + item + "</p>")  
}
```

At the moment we have our data in their own variable, but sometimes the data may come from other sources. We may want to pull in data after our page has loaded from other sources. To do that, we need to use Ajax.

1

Ajax

6

During the mid 1990s, Microsoft's Internet Explorer started to become the most used web browser.

By the end of 2001, it accounted for more than 90 percent of all web access. During that time, Microsoft had a monopoly on what it could add to the browser, so it started adding a ton of experimental features it thought could be useful for coders. Most of these weren't adopted by most coders and just sat in the background unused.

This didn't stop other browsers adding similar features in case coders started to get the impression those browsers were outdated and slow to react. Still, most of the experimental features remained unused.

Meanwhile, during the development of Gmail and Google Maps, Google found it needed a way to pull in data from its back-end servers. It wanted users to be able to drag on an image on Google Maps to pull in the next portion of the map and it wanted users to read their next email without having to reload.

Luckily Google found an experimental features which did just this — one with the un-catchy name of XMLHttpRequest. Bit of a mouthful, but it did the job. It gave coders a way to pull in extra data to the front-end from the back-end without reloading the page.

Over time, XMLHttpRequest was rebranded with the acronym AJAX — Asynchronous Javascript and XML — reflecting the idea that Javascript

could pull in XML data. As a lot of coders were not pulling XML, but HTML and JSON data (we'll come on to JSON), the acronym was simplified to a word: Ajax.

Nothing to do with the kitchen cleaner or the Dutch soccer team of the same name! Just think of it as pulling data from the back-end without reloading the page.

Ajax as a term gets overhyped. A lot of non-coders assume Ajax is anything that Javascript can do, such as animations or dragging around the page. It's not, it's just data transfer. Not that exciting but pretty useful.

Get data with Ajax

How do we get data from a server without reloading the page? First we'll do this with two HTML files. The first will be something like an `index.html` page:

```
<h3>My shopping list</h3>
```

```
<div class="list">
```

```
</div>
```

The second will be my data itself. This could be generated by a server to be changeable, based on a user's account, time of day etc., or just unchanging HTML. We'll put this in `list.html`:

```
<p>Sausages</p>
<p>Eggs</p>
<p>Bacon</p>
```

We want to put the contents of `data.html` into our `index.html`. First, we'd only be adding our script tags into our `index.html`:

```
<h3>My shopping list</h3>

<div class="list">

</div>

<script src="jquery.js"></script>
<script src="list.js"></script>
```

We've been adding that by default in most of the examples, but we just want to keep the `list.html` contents really simple.

We're going to load this content as soon as the page loads, but it may take a few seconds — we don't know if our shopping list is three items or 300 and it depends on our internet speed too. So let's make it look like it's loading:

```
$("div.list").html("Loading")
```

Great, our user knows that we're loading in some data. But what data? We want to grab the data from `list.html`.

Remember our forms? We had two options for them — “get” and “post” — and most of the time we wanted to post our form, because our user had some information to give us. In this case, we don't have any extra information to give to the back-end, so we're going to “get” our data. To do this we can use the jQuery function “`get()`”.

As Ajax doesn't run on a particular tag, we don't need to select anything. We're still using jQuery, though, so we want to add a dollar sign:

```
$("#div.list").html("Loading")
```

```
$.get()
```

What URL are we getting using Ajax? The `list.html` data. This isn't a special word in Javascript, so we need to add it as a string:

```
$("#div.list").html("Loading")
```

```
$.get("list.html")
```

What do we want to do after we've grabbed the data? Well, we want to run a function:

```
$("#div.list").html("Loading")
```

```
$.get("list.html", function () {})
```


Some data will be passed to the function. It'll be the three paragraphs in my `list.html` file, but it could be 300 paragraphs, and we need to handle them in Javascript. Let's add an argument to our function — we'll call it `data` but it could be anything:

```
$("#div.list").html("Loading")

$.get("list.html", function (data) {})
```

Next, we can expand our curly brackets to make it easier to read:

```
$("#div.list").html("Loading")

$.get("list.html", function (data) {

})
```

Inside our curly brackets, we can now use the `data` variable to add to our `div` tag:

```
$("#div.list").html("Loading")

$.get("list.html", function (data) {
    $("#div.list").html(data)
})
```

We're passing the data that the Ajax “get” request has loaded in — this could take a few milliseconds or a few seconds — and once it has, put the HTML content in the `div` tag.

Send data with Ajax

Ajax has two different options: we can “get” data to pull it from server, or we can “post” data to push information to the server. So if we were to pull tweets into our page, we’d be using “get()”. If we were to send a sign up form to our server without reloading the page, we’d be using a new jQuery function called “post()”.

Let’s take a newsletter sign up form that looks like this in HTML:

```
<form action="signup.html" method="post">
  <input type="email" name="email" placeholder="Enter
email">

  <input type="submit" value="Join newsletter">
</form>
```

Usually when we fill in this form, the data would go to be processed by the signup.html file with the data being one field (email) that the user fills in.

For this, we want to wait when the user has submitted the form and stop it going to the next page, as we’re submitting using Ajax instead. We’ll grab the data from the form and post it using Ajax.

First let’s add an event to the form and stop it going to the next page:

```
$("#form").on("submit", function (event) {
  event.preventDefault()
})
```

The next thing we want to do is add our new “post()” jQuery function into our event. Our “get()” function took two arguments (the URL and the function), but our “post()” function takes three — the URL, the data to send, and the function after it’s finished. Let’s add the first argument:

```
$("#form").on("submit", function (event) {  
    $.post("signup.html")  
  
    event.preventDefault()  
})
```

Next we want to add in the form’s data. There’s a jQuery shortcut to do this. We want to get this form’s information turned into a Javascript object using “serialize()”. We can use `$(this)` as we are talking about the form (remember the context special word, `this`):

```
$("#form").on("submit", function (event) {  
    var formdata = $(this).serialize()  
  
    $.post("signup.html", formdata)  
  
    event.preventDefault()  
})
```

Next, we can add our function:

```
$("#form").on("submit", function (event) {  
    var formdata = $(this).serialize()  
  
    $.post("signup.html", formdata, function (d) {})  
  
    event.preventDefault()  
})
```

We can separate our curly brackets:

```
$("#form").on("submit", function (event) {  
    var formdata = $(this).serialize()  
  
    $.post("signup.html", formdata, function (data) {  
  
    })  
  
    event.preventDefault()  
})
```

We may get some HTML back from the post request, but we don't have to use it. If we did, we can just use the data variable we're using as an argument. We could fade in a hidden thanks message for instance:

```
$("#form").on("submit", function (event) {  
    var formdata = $(this).serialize()
```

```
$.post("signup.html", formdata, function (data) {  
    $("div.thanks").fadeIn()  
})  
  
event.preventDefault()  
})
```

JSON

So far when we've been using Ajax, we've been mainly passing back HTML from our back-end, but often back-end servers don't hand out HTML, they hand out a different data format. One called JSON, or Javascript Object Notation. This is basically just Javascript data saved in a file — it can't do functions or events, it's just string, numbers, booleans, arrays and objects.

It should look pretty familiar based on what we've seen so far. We could have a JSON file called `list.json` instead of `list.html` that we're getting with Ajax. The whole file may look like:

```
["sausages", "eggs", "bacon"]
```

And that would be it, but we could have something more complex. Let's take a sample “tweets” file called `tweets.json`:

```
{
  username: "@riklomas",
  name: "Rik Lomas",
  tweets: [
    "Interesting thread from...",
    "What a funny cat photo",
    "@superhi_ hahaha, you're so funny",
    "@gosseti woah, seriously?"
  ]
}
```

There could be even more data in there too — when the tweets were sent, the URL to their individual Twitter page, and my profile photo on Twitter, for example. But we'll keep it simple for now.

Our back-end code will keep the content up to date, but our front-end code doesn't really care what the content is!

How do we get our tweets.json file into our HTML? We use Ajax! We want to “get” the file and put it in some HTML.

Our HTML might look like this:

```
<div class="tweets">

</div>
```

The first thing we'd want to do is make it look like it was loading the tweets:

```
$("div.tweets").html("Loading...")
```

Next we want to grab the JSON file:

```
$("div.tweets").html("Loading...")

$.get("tweets.json")
```

We need to process the tweets using a function with the JSON passed into it. Again, we'll call the argument “data”, but this could be “json” or “file”, whatever we like:

```

$("div.tweets").html("Loading...")

$.get("tweets.json", function (data) {})

```

Let's separate the curly brackets:

```

$("div.tweets").html("Loading...")

$.get("tweets.json", function (data) {

})

```

The new variable "data" will be this large Javascript object of tweets which we can use. Let's put the object's username key in first to give it a title:

```

$("div.tweets").html("Loading...")

$.get("tweets.json", function (data) {
    $("div.tweets").append("<h3>" + data.username +
        "</h3>")
})

```

Underneath that, let's loop over each tweet and add them to the end of the div tag one by one:

```

$("div.tweets").html("Loading...")

```



```
$.get("tweets.json", function (data) {  
    $("div.tweets").append("<h3>" + data.username +  
    "</h3>")  
  
    for (var i = 0; i < data.tweets.length; i++) {  
        $("div.tweets").append("<p>" + data.tweets[i] +  
        "</p>")  
    }  
})
```

We are looping over the tweets array (`data.tweets`), then putting a paragraph at the end with it in. Watch out for the strings added to a variable added to a string. They can get confusing!

Animation using Javascript

As creative coders, we may want to be able to add some kind of constant animation to our web pages. We've talked about CSS transitions, CSS animations using keyframes, and also repeating intervals in Javascript using `setInterval()`, but we can use one more tool to control animation on our page.

First we need to talk about recursion — the idea of something repeating itself. If you Google the word “recursion”, it will say underneath the result “did you mean recursion?” — if you click on that link, it will take you to the same page where it will say “did you mean recursion?”. If you click on that link, it will take you to the same page where it will say “did you mean recursion?”. If you click on that ... you get the idea. We're looping around on ourselves again and again.

With some animations, we want to do this kind of thing. Repeat and check, repeat and check, repeat and so on.

We could do this with `setInterval()`, where we loop the same code over and over. This would work, but it may not be very smooth when we set our interval to be a very small number such as 30 milliseconds. Instead we can use a Javascript function called `requestAnimationFrame`.

Let's say we have a div tag in our HTML:

```
<div class="box">  
  I am a box, move me  
</div>
```

Our accompanying CSS may look like:

```
div.box {  
  position: fixed;  
  top: 0;  
  left: 0;  
  width: 200px;  
  height: 200px;  
  background-color: red;  
}
```

We could make an animation that moves this box back and forth across the page.

Say we want to move the box's left position. As we're moving it in the "x" direction across the page, let's make a variable called *x* and start it at 0:

```
var x = 0
```

Next we want to run a function to add one to this on every frame of our animation. Let's call this function "step":

```
var x = 0

var step = function () {
  x = x + 1
}
```

We can also change our box's position in this step function:

```
var x = 0

var step = function () {
  x = x + 1

  $("div.box").css("left", x + "px")
}
```

Now, we can use the new “requestAnimationFrame()” function to run this function called step:

```
var x = 0

var step = function () {
  x = x + 1

  $("div.box").css("left", x + "px")
}

requestAnimationFrame(step)
```

This will just run the code once, when the page loads. We need to loop this step function on each frame on the animation:

```
var x = 0

var step = function () {
  x = x + 1

  $("div.box").css("left", x + "px")

  requestAnimationFrame(step)
}

requestAnimationFrame(step)
```

Here we are using “requestAnimationFrame()” within the step function to run the function again. After the first one, it’ll run it a second time. After the second time, it’ll run a third time. It’ll keep going and going.

What if we want to stop it after a certain point? Say we get to 500 and want to stop? We can just put an “if()” statement around it:

```
var x = 0

var step = function () {
  x = x + 1

  $("div.box").css("left", x + "px")

  if (x < 500) {
    requestAnimationFrame(step)
  }
}

requestAnimationFrame(step)
```

Here we are saying if the `x` variable is less than 500, keep doing the loop. If not, it's finished, nothing else to do!

The “`requestAnimationFrame()`” function runs as quickly as it can, usually between 30 and 60 frames per second. Make sure you're not making big changes or movements in your loop function. For example, if you're flicking between background colors, don't flash between black and white very quickly at say 30 times a second as you may have photo-sensitive epileptic users.

Mouse movements

For some animations, you may want to follow your mouse around the page (if your user is on a desktop). We can use the “mousemove” event to track where a user’s mouse pointer is within our page.

Because we’re following the mouse around the page, not the browser, we will attach our event to the “document” special word:

```
$(document).on("mousemove", function () {})
```

We’re also going to track the event in our code by passing in an argument. We’ll call it “event” but it can be any name:

```
$(document).on("mousemove", function (event) {})
```

Then we can separate out the curly brackets:

```
$(document).on("mousemove", function (event) {  
  
  })
```

Inside our curly brackets, we can use the event argument to find out where the mouse position is, by using the “pageX” and “pageY” on the event information (x is across, y is down):

```
$(document).on("mousemove", function (event) {  
    $("div.box").css("left", event.pageX)  
  
    $("div.box").css("top", event.pageY)  
})
```

At the moment, the `div` tag we're asking to follow the mouse will attach instantly. If we want to slow its position to not snap, we can add a transition to our CSS:

```
div.box {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100px;  
    height: 100px;  
    background-color: red;  
    transition: all 1s;  
}
```

Touch events

On a touch device, the “mousemove” event will be completely ignored for the obvious reason — there’s no mouse! We can still track where our user touches, to track where the box should go.

We'll keep our `mousemove` event in here, but we need to add another event to track the different event — not a `mousemove` but a `touchmove` instead:

```
$(document).on("mousemove", function (event) {  
    $("div.box").css("left", event.pageX)  
  
    $("div.box").css("top", event.pageY)  
})  
  
$(document).on("touchmove", function (event) {  
  
})
```

With touch events, we can have multiple touches. We could be tracking two fingers that pinch or drag. In our example, we just want to track the first finger and where it is on the page. The event will have a `touches` array in it where we just want to access the first touch (aka finger number 1):

```
$(document).on("mousemove", function (event) {  
    $("div.box").css("left", event.pageX)  
  
    $("div.box").css("top", event.pageY)  
})
```

```
$(document).on("touchmove", function (event) {  
    $("#div.box").css("left", event.touches[0].pageX)  
  
    $("#div.box").css("top", event.touches[0].pageY)  
})
```

With both of these events, we can handle both the mouse and the touch version of the site so it'll work on all types of browser.

Lightboxes

One popular way to display content in web pages is through pop-out lightboxes (sometimes called modals), which take content, fill the user's browser with that content, and dim the rest of the web page's content.

The technique to add a lightbox should feel pretty familiar using a combination of the techniques we've learned so far — we want to place the content over any other content using CSS's fixed positioning, we want to show and hide that content using Javascript, and we want to change the content depending on what our user has clicked on.

As we are adding an extra feature to our web pages, we need to start by adding some extra HTML to add the container for the lightbox. We're going to do this right at the end of the page, just before our script tags — they need to be before the script tags so we can use them in our scripts:

```
<div class="lightbox">
  <div class="lightbox-background"></div>

  <div class="lightbox-content">
    Content will go here
  </div>
</div>
```

We're using div tags here as we're just using generic tags — they don't really mean anything in terms of structure or for search engines.

We have an outer div tag called “lightbox” that we’ll use to show and hide the inner tags.

We then have two inner tags — one to style the background, then one to contain the content. We’re keeping them as separate tags because our user might want to click the background to remove the lightbox.

In our CSS, we want to start by hiding the lightbox by default:

```
div.lightbox {  
    display: none;  
}
```

Next we want to style up the background. We’re going to make the tag fill the whole page with a black fill which would be around 80 percent opacity. As we want to cover any part of the page our user might be seeing, we will make its position fixed:

```
div.lightbox {  
    display: none;  
}
```

```
div.lightbox-background {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
    height: 100%;  
    background-color: #000000;  
    opacity: 0.8;  
}
```

Next we want to style up the content holder. We can use the centering position that we covered in a previous chapter by moving the top left corner into the middle, then shifting it back up and across by half its width and height:

```
div.lightbox {  
    display: none;  
}  
  
div.lightbox-background {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
    height: 100%;  
    background-color: #000000;  
    opacity: 0.8;  
}
```

```
div.lightbox-content {  
  position: fixed;  
  top: 50%;  
  left: 50%;  
  transform: translate(-50%, -50%);  
  background-color: white;  
}
```

At the moment, we can't see the lightbox because of the first style. If we want to tweak how it looks, we can just remove the “display: none;” rule in the first style, then add it back once we're happy.

Next, we need something to put inside the lightbox! Let's say we want to add a gallery of smaller images and show them larger. We might have some HTML that looks like the following:

```
<a href="image-1.html">  
    
</a>  
  
<a href="image-2.html">  
    
</a>
```

When a user clicks on the link to the next page, instead of going to the next page, let's change it so it takes the content inside the link and put it in the lightbox.

In our Javascript, we want to select these links and stop them going to the next page first of all:

```
$("#a").on("click", function (event) {  
    event.preventDefault()  
})
```

Next, we need to show the lightbox when we click on the links. We'll use jQuery's "fadeIn()" function to fade the whole of the lightbox (background and content) into the page:

```
$("#a").on("click", function (event) {  
    $("#div.lightbox").fadeIn()  
  
    event.preventDefault()  
})
```

Awesome! We can see the lightbox. So how do we remove it? We can't do it within a link click event, because we're not clicking on the link to remove the lightbox. We want to hide the lightbox when we click its background — a completely separate event:

```
$("#a").on("click", function (event) {  
    $("#div.lightbox").fadeIn()  
  
    var content = $(this).html()  
  
    $("#div.lightbox-content").html(content)  
  
    event.preventDefault()  
})  
  
$("#div.lightbox-background").on("click", function () {  
    $("#div.lightbox").fadeOut()  
})
```

Great, we can fade in and fade out the lightbox using two different events.

So let's change the content in the lightbox. We want to find the link that we're talking about (the one the user has just clicked), then get its internal HTML (the image tag) and replace that content with the current lightbox content tag:

```
$("#a").on("click", function (event) {  
    $("#div.lightbox").fadeIn()  
  
    event.preventDefault()  
})
```



```
$("#div.lightbox-background").on("click", function () {  
    $("#div.lightbox").fadeOut()  
})
```

We are taking the HTML from the special word (“this”) and saving it to a variable called “content”. We’re then replacing the HTML of the lightbox content div tag with the “content” variable.

Pop over blockers

If you’ve used the web for more than 10 minutes, you may have noticed those annoying pop overs that appear a few seconds after you’ve visited a page, that are trying to get you to sign up for a newsletter or share the site on social media (before you even know what the site is about!). They use the same technique as the lightbox to do this. They just add one extra bit of Javascript to the page — a delay to show the lightbox rather than on a user’s click. The Javascript may look a bit more like:

```
setTimeout(function () {  
    $("#div.lightbox").fadeIn()  
}, 5000)  
  
$("#div.lightbox-background").on("click", function () {  
    $("#div.lightbox").fadeOut()  
})
```

In this case, we're using a `setTimeout()` to delay the pop over by five seconds (5,000 milliseconds). The content of the lightbox would more than likely be preset inside the HTML rather than change on click — even simpler than our normal lightbox Javascript!

Of course, you should use this version carefully as you don't want to annoy your users. There are better ways to get people to share and sign up than these blockers — having great content that your users enjoy is usually the best start.

jQuery plug-ins

Sometimes it could save you time and effort to install a jQuery plug-in rather than write your own code. A jQuery plug-in is a specialized bit of code, written by another coder and given out free. Generally they're made to do very specific tasks, such as slide shows, lightboxes or menus. Think of them as other files of code that extend jQuery to make your life easier.

To find jQuery plug-ins, usually it's a matter of Googling to get a good solution — for instance searching for “jQuery plugin lightbox” would give you a variety of results that could work for what you want.

How do we add a jQuery plug-in to our websites? Most of the time it's a matter of following the instructions that are given to you. There's no one right way to work with other people's code — we're at their mercy. Usually it's a matter of downloading their files and adding them to our own sites, then seeing how our code should interact with the plug-in.

When should we use a plug-in over writing our own code? It really depends. Personally, I prefer to write 95 percent of my own Javascript code because I like having full control over how the site works and how the user interacts with it. I'm a control freak and I'm particular about what I'm adding.

Think of being a coder as being like a chef. Occasionally we might use someone's recipe, but mostly we want to use our own ideas because we want to get the taste exactly right.

Having said that, there's no shame in using other people's jQuery plug-ins. If it saves you hours or days of work, do it, and spend time tweaking it to look perfect.

But how do you know beforehand if a piece of code is going to take days of work or minutes? The best way is to look at the rest of the Web. If you see common techniques that appear on a variety of websites such as lightboxes, parallax and slideshows, chances are that it's going to be fairly easy to create, so you may want to try and make it yourself. If it's a rarer technique, it could take longer to make, and it might be worth looking for a plug-in.

Fixing your own code

Earlier in the guide, we talked about the process of learning to code and about the common mistakes that even professionals make all the time. I talked about how I was once stuck on a code problem for two days. I was working as a freelancer on a sports website where users could book instructors, and while I was making their booking system, the code just stopped working. Code by its nature is logical, so there must have been something I had written that was the problem.

Over two days, I looked through all the code, tweaking and removing lines of code until I focused down to just one line. The line of code was some simple Javascript that looked like this:

```
var state = "confirmed"
```

I was saying that the user had booked a lesson and it was confirmed, saved as a variable called state. Looking at that code, I was confused. There didn't seem to be any problem with it. So why is the code still not working?

Around 100 lines of code further down, I had a simple if statement that was using the state variable to check whether the booking was confirmed:

```
if (state === "completed") {  
  
}
```

That code looked fine too, no errors in the Javascript code, no missing pieces.

Then it hit. I was using two different words for the same thing — confirmed and completed. I switched completed to confirmed and voila, everything worked as it should.

Two days to find this tiny problem. Two days of looking over and over and over at the same pieces of code.

I could pretend and say something here like “oh haha that’s just coding”, or “I felt like a genius once I’d fixed the mistake”. But seriously, it does make you feel like giving up and becoming a woodworker. Coding is a fun activity but it can also be annoying — battling through the frustration is a big part of learning to code. A lot of my previous students find this the hardest part. Just keep going.

Comments

We’ll start by not fixing our code but explaining to ourselves what we’re doing when we’re writing code by using comments.

Comments are pieces of code that don’t get shown to the user but let us write about what we’re doing. They are especially useful if we’re working as part of a team, or if we’re working on a site for the long term. If we come back to the site in 15 months’ time, would we still know why we wrote the code in this way?

How we add a comment depends on what language we're adding the comment in.

Let's start with HTML. A comment in HTML starts with `<!--` then we write the comment then finish with `-->` to stop the comment.

```
<section class="intro">
  <!-- this will contain the banner -->
</section>
```

The words “this will contain the banner” do not get shown to the user, only we coders can see them.

For CSS, a comment starts with `/*` and finishes with `*/`:

```
header {
  /* make the header sticky to the top */
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;

  /* background is blue */
  background-color: #2727e6;
}
```

In this CSS, we have two comments within the header style. We don't need to be inside the style, they could be anywhere in the code.

For Javascript, we start a comment with two forward slashes `//`. There's no end for this comment, it just goes to the end of the line, so you can't put any usable code after the two forward slashes in this line:

```
// when a user clicks, show a light box
$("a").on("click", function (event) {
    // fade in lightbox over 500 milliseconds
    $("div.lightbox").fadeIn(500)

    // stop going to the next page
    event.preventDefault()
})
```

Here we have three Javascript comments that label pretty much every line of code.

Think of comments as if they are computer-to-human translations that we're adding in. They're useful to add so we're very clear about what we're doing at each stage of our code.

Indenting code

In our code, white space doesn't matter. Any look and feel is added with CSS, so we can add as many or as few spaces and line breaks as we like.

However, to make your code more readable in the future, it's a good idea to make it as clean as possible. A common technique is to indent your code by pressing Tab when you're inside curly brackets or other tags.

For instance, this is perfectly valid code but it's hard to read:

```
<section class="intro"><div>
<h1>Welcome</h1>
</div></section>
```

To clean it up, we can move the div tag on to its own line, and indent every time we're inside another tag by pressing Tab:

```
<section class="intro">
  <div>
    <h1>Welcome</h1>
  </div>
</section>
```

Same code but cleaner.

The same thing for CSS would look like:

```
@media (max-width: 800px) {  
  body {  
    font-size: 12px;  
  }  
  section {  
    padding: 20px  
  }  
}
```

It works but it's hard to see where code starts and stops. With a little bit of pressing Tab and Return, we can clean it up:

```
@media (max-width: 800px) {  
  body {  
    font-size: 12px;  
  }  
  
  section {  
    padding: 20px;  
  }  
}
```

For Javascript, it's particularly important to have clean code. Something that looks like the following is hard to read:

```
$("#a").on("click", function (event) {  
  $("#div").fadeIn(500)  
  $("#div").html("Hello")  
  event.preventDefault()  
})
```

A cleaned up version with a bit of breathing room may look like:

```
$("#a").on("click", function (event) {  
    $("#div").fadeIn(500)  
  
    $("#div").html("Hello")  
  
    event.preventDefault()  
})
```

Fixing bugs

The word “bug” to mean a fault in computers predates computers themselves. It was originally coined by Thomas Edison, inventor of devices such as the light bulb, the gramophone and the movie camera. He said bugs were little faults and difficulties in his inventions.

The popularity of the word bug in computers started with mechanical computers in the 1940s — one of the first computer scientists, Grace Hopper, found a moth stuck inside the mechanics of a computer which was stopping it from working. She took the moth out and added it to her log book, commenting on it being funny how in this case the little fault stopping the computer from working was literally a bug.

The process of fixing code is to remove bugs, literally or figuratively — we call this “debugging”. In our code, it’s not as simple as taking out a

moth, so what does it involve? There's no perfect way to debug code but let's talk about a few options.

Counting starts and ends

The first way is to make sure our code matches up on both sides. For instance, if we're in HTML, we need to make sure there's the same number of opening and closing tags:

```
<div>  
</div>  
</div>
```

For instance, here there's only one opening tag but two closing tags, so we'd need to get rid of one of them to fix the code. Watch out for your single tags that don't have a closing tag, such as the image tag. They can throw your counting!

In our CSS, we might have some code that looks like:

```
@media (max-width: 800px) {  
  body {  
    font-size: 12px;  
  }  
  
  section {  
    padding: 20px;  
  }  
}
```

Let's go through our code. We have an opening round bracket, then a matching close round bracket, so that's fine. Next we have three opening curly brackets but only two closing curly brackets. The first open curly bracket needs to match the very last one. The body style has an open and close curly bracket, but the section style only has one open and no close curly bracket, so we need to close our section style properly.

In Javascript, there are even more curly and round brackets, so we have to be extra careful:

```
$("#a").on("click", function () {}  
  $("#div").addClass("highlight")  
})
```

There are two mistakes here. There are five open round brackets but six close round brackets. The extra one is right at the end of the first line, as we want the “on()” function to go around the function.

The second mistake is with the quote marks. We have seven of them, but there should be either six or eight as each opening quote

should have a close quote. The quotes around “a” are fine. The quotes around “click” are fine. The quotes around “div” are fine. The word “highlight” is missing a close quote mark.

Adding backgrounds and borders

One of my favorite things to do when fixing layout issues is to add either strange backgrounds or borders to tags. I like to add in bright red borders or bright yellow backgrounds to see where our issue may lie. Think of it being like a doctor testing for symptoms.

Let's take for example a sticky bar that doesn't stretch across the screen:

```
header {  
  position: fixed;  
  top: 0;  
  left: 0;  
}
```

To see what the header is doing, we could add in a background color or a border to see how big it is:

```
header {  
    position: fixed;  
    top: 0;  
    left: 0;  
    background-color: yellow;  
}
```

We might notice that the header just fits to its content, so it would be a sign that we need to give it some width:

```
header {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
}
```

Inspect element

There are sets of tools built into every web browser that are specially for coders to debug their code.

In Google Chrome, we can just go to our menu bar, go to View and there are a few options under Developer Tools. In Safari, we have to turn on Developer Tools by going to Preferences, then we'll see a Developer option in the menu bar.

A quick way to debug is to right click on the area you want to fix and click “Inspect Element”. This will pop open the Developer Tools that let you see what’s going on.

We can play around with the tags and styles on our web site to get it right.

However, it won’t save any changes. We have to remember what we change and alter our code accordingly. The reason it doesn’t save is because you can right click and go to Inspect Element on any web site — you wouldn’t want other people editing your code!

Javascript console

Within Developer Tools there is one called Javascript Console, a stream of messages about the Javascript on our page. Usually it’ll be empty, unless there’s a mistake or we tell it to say something.

If there’s a mistake, it will give you a rough idea of where you’re going wrong. It will give you an error message that may be quite technical but will also tell you which file and what line number (on some browsers, it might look like “scroll.js:33” which would mean the scroll.js file on line 33).

The error may not be exactly on this line but it is usually around there.

We can also put some test messages in our Javascript code by using “`console.log()`”. For instance, we might want to see how many pixels down the page we are on scroll, but not put it directly into our web page for our users to see, so we could hide it in the Javascript console:

```
$(document).on("scroll", function () {  
    var pixels = $(document).scrollTop()  
  
    console.log("we have scrolled by... " + pixels)  
})
```

In our Javascript console, as we scroll, we'll see a message saying “we have scrolled by... 1”, then “we have scrolled by... 2”, etc. It's hidden from our users but useful for us coders.

Sometimes, we might want to test our sites on our iPhones and iPads using Developer Tools. The simple way to do this is to plug your device into a Mac, then in your menu bar under Developer Tools, you'll see your plugged-in device as an option to use over the Safari desktop browser.

Google

Guess what? Coders don't keep all this information in their brains. They use Google to find answers all the time. If you're having a problem or can't remember something, the best way is to Google it! Copy and

paste in your error message. Look up what “CSS overflow” is and how it works.

There’s no shame in looking things up. I do it all the time and so does every other coder.

Stack Overflow

One of the sites you see pop up in Google search results for code questions is Stack Overflow, which is a question and answer community for coders. If you get stuck and can’t find a relevant answer, sign up and post a question. Tag it with the right languages and hopefully someone will answer.

Other advanced coders answer questions for free because they want to give back to the community (just like open source code). The Stack Overflow community is very welcoming to new users.

Practice, practice, practice

The best way to learn how to fix bugs is to keep encountering them. Don’t give up, keep at it and try and try again. It’s easy to get disillusioned when you get stuck on a problem but push your way through.

Integration with back-ends

All the code we've written so far has been for the visual and interactivity side of web sites — the front-end. How do we make this work with the other side, the back-end?

We might be using a hosted service like Wordpress, Squarespace, Shopify or Tumblr to serve up our content, or we might be working with a coder who has a custom back-end for the site. This back-end could be built in one of many languages: PHP, Python, Ruby and Elixir are just a few.

The first thing we need to think about is, what does the back-end actually do? Earlier in the guide, we talked about our web site being like a restaurant where our waiting staff took a customer's order to give to the kitchen to cook. The kitchen would then cook the order, give it to a waiter, who then delivers it to our customer.

Would the interior design of the restaurant change depending on whether the customer ordered beef or salmon? No. Would the way the tables and chairs worked change if the customer didn't order a dessert? No.

The only thing that changes depending on the customer's order is what's being consumed by that customer.

In a similar vein, on our websites, the only thing that changes depending on our user is the content. On Facebook, if you log in as a different person, the content is different, but everyone still gets the same layout, the same typography, the same color scheme and the same user interactions.

The back-end does not change how the style and the user actions work. The only thing it changes is the content, so the only part the back-end code will touch is HTML, not CSS or Javascript.

We are turning our HTML content from being unchanged and static to being changeable, dynamic and dependent on a variety of things — the user who's logged in, the time of day, the score in a soccer match, the amount of likes on your photo, etc.

The HTML doesn't care about the content itself, it's just delivering the content given to it by the back-end. It's common to call HTML used with a back-end a “template”, as it now has changeable content.

The way that the back-end interacts with the HTML depends heavily on the back-end we're using. For instance, the way that Shopify will interact with HTML is a lot different from the way Ruby on Rails will interact. The idea is the same but the technique changes.

For instance, Shopify uses a set of custom HTML-style tags that it calls Liquid. So we might have a page that shows the product we're selling, but the HTML doesn't care what product we're selling as that's the back-end's problem.

Shopify's tags would look a little like:

```
<h1>{{ product.title }}</h1>
```

```
<p>{{ product.description }}</p>
```

```
<p class="price">{{ product.price }}</p>
```

What Shopify's back-end will find is its own tags, and replace them with the correct content, so our users see the right content based on what they asked for.

How do we find what Shopify can do and what its special tags are? I Googled “Shopify template tags” and it was the top result.

For Ruby, our tags would look a little different:

```
<h1><%= @product.title %></h1>
```

```
<p><%= @product.description %></p>
```

```
<p class="price"><%= @product.price %></p>
```

Instead of being surrounded by two curly brackets each side, the Ruby way is to use “<%=” at the start and “%>” at the end.

On custom back-ends that don't use a third party service, we'd have to work with the other coder to work out what tags to use and what content goes in there.

The one thing that does get easier with back-end code interacting with HTML is being able to repeat HTML for different content. For example, we might have 21 related products, but we only need to make a loop (similar to the Javascript “for” loop) and add the content in the middle. In Shopify, we might want to loop through all the tags for the product and put each of them in a div tag:

```
<h3>Tags</h3>

{% for tags in product.tags %}
  <div>
    {{ tag }}
  </div>
{% endfor %}
```

On one product we might have just two tags, and on another we might have 50, but our HTML shouldn't care. Here we are looping over a product's tags one by one — if we have two tags, we'll have two div tags with the right content in each, if we have 50 tags, we have 50 div tags — even though we've only put one div tag in our HTML. It's a template for what content will go inside there.

All you need to remember is that each back-end is different, just like every restaurant's kitchen is different, as they try to achieve different tasks. Wordpress is great for blogging and Shopify is great for e-commerce, but they're not great for starting a world-changing startup because they're built to be generic and for one task only.

So if you want to be the next Mark Zuckerberg or Bill Gates, the next step would be to learn back-end as well as front-end coding. For most

coders, it's fine to stick with platforms like Shopify, Wordpress and Squarespace and learn to alter them — just pick the right tool for the right job.

Idiot or genius?

One of the most important things to remember when learning to code is there are essentially two modes: idiot or genius, with nothing in between. Unfortunately those feelings will probably never go away. Coders old and new have these two moods and it's totally natural. It's hard to believe that a coder of 10 or 20 years experience would feel like an idiot but it's true. We do.

Why does this happen? When you're coding, you're essentially in a creative state of mind. You're creating brand new sections, features or even sites that have never been created in this way before. You're in the flow of creativity, you get a feeling of achievement. Everything is going well and you're so good at this coding malarkey. It feels unstoppable and you lose track of time. Three hours of productive coding can speed by in what feels like 30 minutes.

Then. You. Get. Stuck.

Why is this code not working? It should be so simple. But it's just not working.

Computers are dumb. They just carry out the instructions you give them. So if it's going wrong, it's not the computer's fault. It's yours. And that can make you feel really dumb. It's quite incredible how

quickly the high of productive coding can turn into a low of confusion and anger.

Usually it's a tiny mistake that can be easily solved. Sometimes you might fix it and you don't really know what you did but it worked. But as soon as it works again, the high comes back. I fixed the code. I did it. It was me.

I. Am. A. Genius.

Code is built on the foundation of the binary system. A light that goes on or off. Ones and zeroes. If this happens, do this, if it doesn't, do something else.

Learning to code is also binary. It's two moods: genius mode on or genius mode off.

The end of the beginning

We wrote this guide because we felt others already out there ranged from terrible to just okay. We wanted to write something that was an opinionated approach to learning HTML, CSS and Javascript. The SuperHi team has taught hundreds of people to code and what's in this guide covers pretty much everything you need to know on a day-to-day basis, even though there will be coders who will tell you that we're wrong and that they know better.

When you tell the average person that you've learning to code, often their eyes will glaze over and their brain will start thinking about lunch. Even the word "code" often brings up the mental image of either the nerds of The Big Bang Theory or the renegade hoodies of Mr Robot.

We wanted to show that coding is just a tool, not some puzzle to be solved. HTML, CSS and Javascript are simply design tools — they are different from Photoshop, Illustrator and Sketch because we have to type out by hand, rather than drag and drop, because of the uncertainty of the canvas we're given.

Unlike designing a poster or a book, we have to factor in different screen sizes, different browsers and different devices used to access our web pages. A poster design doesn't have to worry about the person looking at it. A magazine isn't interactive and can't change when we turn the pages.

This is why I love making web sites. I love the changability and the uncertainty of the web. I love the interactivity of our pages changing on scroll. I love the fact people from across the world who I've never met can view what I've created instantly and as soon as I've published.

We wanted to challenge the idea that learning to code is hard. It's not hard, but it's not easy, and it takes time to get good. Don't expect to be the world's best coder after just reading this book.

Think of it as like a book on weightlifting. We've shown you how to lift the weights and not make mistakes, but in practice, you still need to start off on the lighter weights to be able to progress to the heavier

ones. You can't lift 300 pounds on day one and you'll get frustrated if you try.

Similarly, you won't be able to make the sites on Site Inspire or Awwwards on day one, but you'll be able to see how the integral parts are put together.

Keep the vision in mind. Keep trying and failing. You will get stuck. It won't be quick, but over time you'll get better and better at coding. Keep persevering and don't give up. Don't be afraid to ask for help. Don't be afraid to ask "dumb questions" — there are no dumb questions.

Now is the time to take this and apply to your own projects.

Time to code now.

—Rik Lomas, founder of SuperHi, New York, 2017

