

 MANNING

CSS

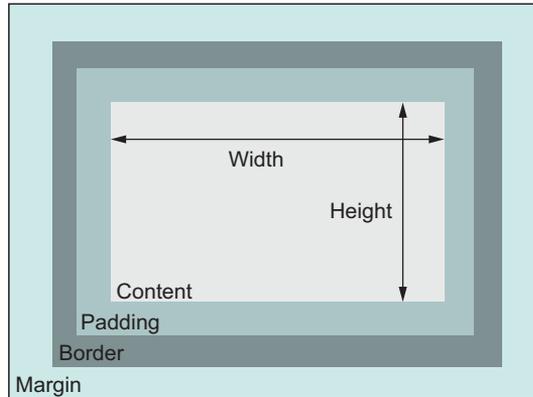
IN DEPTH

Keith J. Grant
FOREWORD BY Chris Coyier

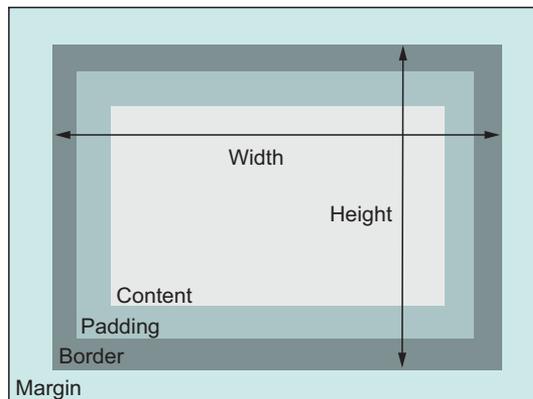


Box model and border-box sizing

The box model refers to the composition of elements on a page. When you specify the height or width of an element, you're setting the content size—any padding, border, and margin will be added to that.



Applying `border-box` to an element changes the box model to a more predictable behavior. Setting height or width will control the size of the entire element, including its padding and border.



See chapter 3 for information on applying border-box sizing to the entire page, as well as other important concepts including:

- Centering content
- Creating columns of equal height
- Controlling spacing between elements

CSS in Depth

CSS in Depth

KEITH J. GRANT



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2018 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Kristen Watterson
Review editor: Aleksandar Dragosavljević
Technical development editor: Robin Dewson
Project editor: Kevin Sullivan
Copy editor: Frances Buran
Proofreader: Elizabeth Martin
Technical proofreader: Birnou Sébarte
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617293450

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 23 22 21 20 19 18

brief contents

PART 1	REVIEWING THE FUNDAMENTALS.....	1
	1 ■ Cascade, specificity, and inheritance	3
	2 ■ Working with relative units	28
	3 ■ Mastering the box model	55
PART 2	MASTERING LAYOUT.....	85
	4 ■ Making sense of floats	87
	5 ■ Flexbox	116
	6 ■ Grid layout	144
	7 ■ Positioning and stacking contexts	177
	8 ■ Responsive design	201
PART 3	CSS AT SCALE	231
	9 ■ Modular CSS	233
	10 ■ Pattern libraries	253

PART 4 ADVANCED TOPICS.....277

- 11 ■ Backgrounds, shadows, and blend modes 279
- 12 ■ Contrast, color, and spacing 300
- 13 ■ Typography 329
- 14 ■ Transitions 353
- 15 ■ Transforms 370
- 16 ■ Animations 396

contents

foreword xv
preface xvii
acknowledgments xix
about this book xxi

PART 1 REVIEWING THE FUNDAMENTALS1

1 *Cascade, specificity, and inheritance* 3

1.1 The cascade 4

Understanding stylesheet origin 8 ■ *Understanding specificity* 10 ■ *Understanding source order* 15
Two rules of thumb 17

1.2 Inheritance 18

1.3 Special values 20

Using the inherit keyword 21 ■ *Using the initial keyword* 22

1.4 Shorthand properties 23

Beware shorthands silently overriding other styles 23
Understanding the order of shorthand values 24

- ## 2 Working with relative units 28
- 2.1 The power of relative values 29
 - The struggle for pixel-perfect design* 29
 - *The end of the pixel-perfect web* 29
 - 2.2 Ems and rems 31
 - Using ems to define font-size* 32
 - *Using rems for font-size* 36
 - 2.3 Stop thinking in pixels 37
 - Setting a sane default font size* 39
 - *Making the panel responsive* 40
 - *Resizing a single component* 41
 - 2.4 Viewport-relative units 43
 - Using vw for font size* 45
 - *Using calc() for font size* 45
 - 2.5 Unitless numbers and line-height 46
 - 2.6 Custom properties (aka CSS variables) 48
 - Changing custom properties dynamically* 50
 - *Changing custom properties with JavaScript* 53
 - *Experimenting with custom properties* 54
- ## 3 Mastering the box model 55
- 3.1 Difficulties with element width 56
 - Avoiding magic numbers* 59
 - *Adjusting the box model* 59
 - Using universal border-box sizing* 61
 - *Adding a gutter between columns* 62
 - 3.2 Difficulties with element height 64
 - Controlling overflow behavior* 64
 - *Applying alternatives to percentage-based heights* 65
 - *Using min-height and max-height* 70
 - *Vertically centering content* 71
 - 3.3 Negative margins 73
 - 3.4 Collapsed margins 74
 - Collapsing between text* 74
 - *Collapsing multiple margins* 75
 - Collapsing outside a container* 76
 - 3.5 Spacing elements within a container 77
 - Considering changing content* 79
 - *Creating a more general solution: the lobotomized owl selector* 80

PART 2 MASTERING LAYOUT85

4 Making sense of floats 87

- 4.1 The purpose of floats 88
- 4.2 Container collapsing and the clearfix 93
 - Understanding container collapsing* 94 ■ *Understanding the clearfix* 96
- 4.3 Unexpected “float catching” 99
- 4.4 Media objects and block formatting contexts 102
 - Establishing a block formatting context* 103 ■ *Using a block formatting context for media object layouts* 104
- 4.5 Grid systems 105
 - Understanding a grid system* 106 ■ *Building a grid system* 107
 - Adding gutters* 111

5 Flexbox 116

- 5.1 Flexbox principles 117
 - Building a basic flexbox menu* 120 ■ *Adding padding and spacing* 123
- 5.2 Flex item sizes 124
 - Using the flex-basis property* 126 ■ *Using flex-grow* 127
 - Using flex-shrink* 128 ■ *Some practical uses* 129
- 5.3 Flex direction 130
 - Changing the flex direction* 132 ■ *Styling the login form* 133
- 5.4 Alignment, spacing, and other details 135
 - Understanding flex container properties* 135 ■ *Understanding flex item properties* 139 ■ *Using alignment properties* 140
- 5.5 A couple of things to be aware of 142
 - Flexbugs* 142 ■ *Full-page layout* 142

6 Grid layout 144

- 6.1 Web layout is here 145
 - Building a basic grid* 146
- 6.2 Anatomy of a grid 148
 - Numbering grid lines* 153 ■ *Working together with flexbox* 155
- 6.3 Alternate syntaxes 158
 - Naming grid lines* 158 ■ *Naming grid areas* 160

- 6.4 Explicit and implicit grid 162
 - Adding variety* 166
 - *Adjusting grid items to fill the grid track* 169
- 6.5 Feature queries 172
- 6.6 Alignment 175

7 **Positioning and stacking contexts** 177

- 7.1 Fixed positioning 178
 - Creating a modal dialog with fixed positioning* 178
 - Controlling the size of positioned elements* 182
- 7.2 Absolute positioning 182
 - Absolutely positioning the Close button* 182
 - *Positioning a pseudo-element* 183
- 7.3 Relative positioning 185
 - Creating a dropdown menu* 186
 - *Creating a CSS triangle* 188
- 7.4 Stacking contexts and z-index 190
 - Understanding the rendering process and stacking order* 191
 - Manipulating stacking order with z-index* 193
 - *Understanding stacking contexts* 194
- 7.5 Sticky positioning 197

8 **Responsive design** 201

- 8.1 Mobile first 202
 - Creating a mobile menu* 209
 - *Adding the viewport meta tag* 213
- 8.2 Media queries 214
 - Understanding types of media query* 215
 - *Adding breakpoints to the page* 217
 - *Adding responsive columns* 221
- 8.3 Fluid layouts 223
 - Adding styles for a large viewport* 224
 - *Dealing with tables* 226
- 8.4 Responsive images 227
 - Using multiple images for different viewport sizes* 227
 - Using srcset to serve the correct image* 228

PART 3 CSS AT SCALE.....231

9 Modular CSS 233

- 9.1 Base styles: laying the groundwork 234
- 9.2 A simple module 235
 - Variations of a module* 237 ■ *Modules with multiple elements* 241
- 9.3 Modules composed into larger structures 243
 - Dividing multiple responsibilities among modules* 244
 - Naming modules* 248
- 9.4 Utility classes 250
- 9.5 CSS methodologies 251

10 Pattern libraries 253

- 10.1 Introduction to KSS 254
 - Setting up KSS* 255 ■ *Writing KSS documentation* 257
 - Documenting module variants* 261 ■ *Creating an overview page* 264 ■ *Documenting modules that require JavaScript* 264
 - Organizing the pattern library into sections* 266
- 10.2 Shifting the way you build CSS 269
 - Using a CSS First workflow* 269 ■ *Using a pattern library as an API* 270

PART 4 ADVANCED TOPICS.....277

11 Backgrounds, shadows, and blend modes 279

- 11.1 Gradients 280
 - Using multiple color stops* 283 ■ *Using radial gradients* 285
- 11.2 Shadows 287
 - Defining depth with gradients and shadows* 288 ■ *Creating elements with a flat design* 289 ■ *Creating buttons with a more modern look* 290
- 11.3 Blend modes 291
 - Tinting an image* 294 ■ *Understanding types of blend mode* 295 ■ *Adding texture to an image* 296
 - Using mix blend modes* 298

12 Contrast, color, and spacing 300

- 12.1 Contrast is king 302
 - Establishing patterns* 303 ■ *Implementing the design* 304
- 12.2 Color 306
 - Understanding color notations* 312 ■ *Adding new colors to a palette* 316 ■ *Considering contrast for font colors* 318
- 12.3 Spacing 320
 - Using ems vs. px* 320 ■ *Factoring in line height* 323
 - Spacing inline elements* 326

13 Typography 329

- 13.1 Web fonts 331
- 13.2 Google fonts 332
- 13.3 How @font-face works 336
 - Font formats and fallbacks* 337 ■ *Multiple variants of the same typeface* 338
- 13.4 Adjusting space for readability 339
 - Body copy spacing* 340 ■ *Headings, small elements, and spacing* 342
- 13.5 The dreaded FOUT and FOIT 346
 - Using Font Face Observer* 348 ■ *Falling back to system fonts* 349
 - Getting ready for font-display* 351

14 Transitions 353

- 14.1 From here to there 354
- 14.2 Timing functions 356
 - Understanding Bézier curves* 357 ■ *Steps* 360
- 14.3 Non-animatable properties 361
 - Properties that cannot be animated* 364 ■ *Fading in and out* 365
- 14.4 Transitioning to auto height 367

15 Transforms 370

- 15.1 Rotate, translate, scale, and skew 371
 - Changing the transform origin* 374 ■ *Applying multiple transforms* 375

- 15.2 Transforms in motion 375
 - Scaling up the icon* 381
 - *Creating “fly in” labels* 383
 - Staggering the transitions* 386
- 15.3 Animation performance 387
 - Looking at the rendering pipeline* 387
- 15.4 Three-dimensional (3D) transforms 389
 - Controlling perspective* 390
 - *Implementing advanced 3D transforms* 393

16 Animations 396

- 16.1 Keyframes 397
 - 16.2 Animating 3D transforms 400
 - Building the layout without animations* 400
 - *Adding animation to the layout* 405
 - 16.3 Animation delay and fill mode 407
 - 16.4 Conveying meaning through animation 409
 - Responding to user interaction* 409
 - *Drawing the user’s attention* 413
 - 16.5 One final piece of advice 416
- appendix A* *Selectors reference* 417
- appendix B* *Preprocessors* 422
- index* 435

foreword

“A minute to learn . . . A lifetime to master.” That phrase might feel a little trite these days, but I still like it. It was popularized in modern times by being the tagline for the board game *Othello*. In *Othello*, players take turns placing white or black pieces onto a grid. If, for example, a white piece is played trapping a row of black pieces between two white, all the black pieces are flipped and the row becomes entirely white.

Like *Othello*, it isn’t particularly hard to learn the rules of CSS. You write a selector that attempts to match elements, then you write key/value pairs that style those elements. Even folks just starting out don’t have much trouble figuring out that basic syntax. The trick to getting *good* at CSS, as in *Othello*, is knowing exactly *when* to do *what*.

CSS is one of the languages of the web, but it isn’t quite in the same wheelhouse as programming. CSS has little in the way of logic and loops. Math is limited to a single function. Only recently have variables become a possibility. Rarely do you need to consider security. CSS is closer to painting than Python. You’re free to do what you like with CSS. It won’t spit out any errors at you or fail to compile.

The journey to getting good at CSS involves learning everything CSS is capable of. The more you know, the more natural it starts to feel. The more you practice, the more easily your brain will reach for that perfect layout and spacing method. The more you read, the more confident you’ll feel in tackling any design.

Really good CSS developers aren’t deterred by any design. Every job becomes an opportunity to get clever, a puzzle to be solved. Really good CSS developers have that full and wide spectrum of knowledge of what CSS is capable of. This book is part of

your journey to being that really good CSS developer. You'll gain the spectrum of knowledge necessary to getting there.

If you'll permit one more metaphor, despite CSS going on a couple of decades old, it's a bit like the Wild Wild West. You can do just about whatever you want to do, as long as it's doing what you want. There aren't any hard and fast rules. But because you're all on your own, with no great metrics to tell you if you're doing a good job—or not—you'll need to be extra careful. Tiny changes can have huge effects. A stylesheet can grow and grow and become unwieldy. You can start to get scared of your own styles!

Keith covers a lot of ground in the book, and every bit of it will help you become a better CSS developer and tame this Wild Wild West. You'll deep dive into the language itself, learning what CSS is capable of. Then, just as importantly, you'll learn about ideas *around* the language that level you up in other ways. You'll be better at writing code that lasts and is understandable and performant.

Even seasoned developers will benefit. If you find yourself reading about something that you already know, you'll firm up your skills, affirm your knowledge, and find little “oooo” bits that surprise you and extend that base.

CHRIS COYER
Co-founder, CodePen

preface

CSS was proposed in 1994 and first implemented (partially) by Internet Explorer 3 in 1996. It was somewhere around that time I discovered the wonderful View Source button and realized all the secrets of a web page were there for me to decipher in plain text. I taught myself HTML and CSS by playing in a text editor and seeing what worked. It was a fun excuse to spend as much time as possible on the internet.

In the meantime, I needed to find a real career. I went on to earn a degree in Computer Science. Little did I know that the two would come crashing together in the 2000s as the concept of “web developer” emerged.

I’ve been in tune with CSS since the very beginning. Even when I’m working, it’s play. I’ve worked on the back end and the front end, yet I’ve always found myself to be the resident CSS expert on every team I’ve been on. It’s often the most neglected part of the web stack. But once you’ve been on a project with clean CSS, you never want to do without it again. After seeing it in action, even seasoned web developers ask, “How do I learn CSS?”

There isn’t one concise, straightforward answer to this question. It’s not a matter of learning one or two quick tips. Rather, you need to understand all the disparate parts of the language and how they can fit together. Some books make a good beginner-level introduction to CSS, but many developers already have a basic understanding. Some books teach a lot of useful tricks but assume the reader has mastery over the language.

At the same time, the rate of change in CSS is accelerating. Responsive design is now the de facto standard. Web fonts are commonplace. In 2016, we saw the rise of flexbox, and 2017 began the rise of something called grid layout. Blend modes, box

shadows, transformations, transitions, and animations are all new to the scene. As more and more browsers become *evergreen*, automatically updating to the newest version, new features will continue to roll out. There is a lot to keep up with.

Whether you are relatively new to the industry or have been at it a while but need to advance or update your CSS skills, I have written this book to bring you up to speed. Everything in this book is here for one of three reasons:

- 1 *It's essential.* There are many fundamentals of the language that, sadly, many developers don't fully understand. This includes the cascade, the behavior of floats, and positioning. I'll take a deep look at them, explaining how they work.
- 2 *It's new.* A lot of new features have emerged in the last few years, or are just emerging now. I will cover the latest improvements to CSS and a few things that are just around the corner. This is a forward-thinking book. I will point out backward compatibility issues where relevant, but I am unabashedly optimistic about the present and future of cross-browser development.
- 3 *It's not covered in most CSS books.* The world of CSS is huge. There are important best practices and common approaches in the modern world of web application development. These are not strictly part of the CSS language, but rather part of its culture. And they are vital for modern web development.

So, how do you learn CSS? This book is an attempt to answer that question, for the people who know they need it most.

acknowledgments

It takes an incredible amount of work to produce a book. I believe this is a great book—and hope you’ll agree—but it wouldn’t be nearly as strong as it is without the help of a number of people along the way.

First and foremost, I’d like to thank my wife, Courtney. You have been supportive and encouraging through the entire process. You have carried the burden of this book with me. You even provided editorial support in a number of key places. I could not have done this without you.

I’d like to acknowledge my boss, Mark Eagle, and the rest of my team at Intercontinental Exchange. Thank you for encouraging me on the way and allowing me to slip away to write on countless lunch breaks.

Thanks to my acquisitions editor, Greg Wild, who found my pathetic first drafts online and reached out to me. And thanks to Manning’s publisher, Marjan Bace, who saw the potential in this idea. There’s always a risk associated in green-lighting a book, particularly with a new author. Thank you for taking that chance.

A good book can’t exist without an editor. Thanks to Kristen Watterson for your commitment to quality. This is a much better book because of your input. And thanks to my technical editor, Robin Dewson, for your patience and insight throughout this long process.

Thanks to Birnou Sebarté and Louis Lazaris for giving the book a final, thorough technical proofread. Thanks to Chris Coyier for your willingness to write my foreword.

I’d also like to thank the technical reviewers and friends who took the time to read through my drafts at various stages and offer feedback: Adam Rackis, Al Pezewski,

Amit Lamba, Anto Aravinth, Brian Gaines, Dico Goldoni, Giancarlo Massari, Goetz Heller, Harsh Raval, James Anaipakos, Jeffrey Lim, Jim Arthur, Matthew Halverson, Mitchell Robles, Jr., Nitin Varma, Patrick Goetz, Phily Austria, Pierfrancesco D'Orsogna, Rafael Cassemiro Freire, Rafael Freire, Sachin Singhi, Tanya Wilke, Trent Whiteley, and William E. Wheeler. Your feedback offered valuable early insight into how the book would be received by developers of all skill levels.

Finally, I'd like to offer enormous gratitude to the good people on the W3C CSS Working Group for your work on the CSS specifications. You work through a lot of really tough problems so that we developers don't have to. Thanks for your continued efforts to make CSS, and the web as a whole, better.

about this book

The world of CSS is maturing. More and more web developers in the industry are realizing that while they “know” CSS, they don’t know it as deeply as they probably should. In recent years, the language has evolved, so even those developers who were once adept at CSS may find a whole new set of skills to catch up on. This book aims to meet both these needs: providing a deep mastery of the language, and bringing you up to speed on recent developments and new features of CSS.

This book is titled *CSS in Depth*, but it is also a book of *breadth*. Where concepts are difficult or commonly misunderstood, I will explain in detail how they work and why they behave the way they do. In other chapters, I may not exhaust the topic, but I will give you enough knowledge to work effectively with it and point you in the right direction if you wish to further your knowledge. In all, this book will fill in your blind spots.

Some of the topics could warrant entire books on their own: animation, typography, even flexbox and grid layout. My goal is to flesh out your knowledge, help you bolster your weak spots, and give you a love for the language.

Who should read this book

First and foremost, this book is for developers who are tired of fighting with CSS and are ready to really understand how it works. You may be a beginner, or you may have fifteen years of experience.

I expect you to have a cursory understanding of HTML, CSS, and—in a few places—JavaScript. As long as you’re familiar with the basic syntax of CSS, you’ll probably be able to follow along with this book. But it’s primarily written for developers

who have spent time with CSS, run into walls, and come out frustrated. In the places where I use JavaScript, I have kept it as simple as possible, so as long as you can follow along with a few short code snippets, you should be in good shape.

If instead you're a designer looking to move into the world of web design, I suspect you too will learn a lot from *CSS in Depth*—though I haven't written it with you particularly in mind. The book may also provide some insight into the perspective of the developers you'll be working with.

How this book is organized

The book is 16 chapters long, divided into four parts. In part 1, “Reviewing the fundamentals,” we'll go back to the basics, with a focus on some details you likely missed the first time around:

- Chapter 1 covers the cascade and inheritance. These concepts control which styles are applied to which elements on the page.
- Chapter 2 discusses relative units, with an emphasis on *em* and *rem*. Relative units are versatile and important tools in CSS, and this chapter will get you familiar with working with them.
- Chapter 3 covers the box model. This involves controlling the size of elements on the page and the amount of space between them.

In part 2, “Mastering layout,” I'll walk you through the key tools for laying out elements on the page:

- Chapter 4 dives into using floats for layout. We'll build a multicolumn page and look at taming some of the quirky aspects of floats.
- Chapter 5 teaches flexbox, which is a fairly new layout method. It begins with the fundamental concepts and moves on to practical layout examples.
- Chapter 6 introduces the brand-new layout tool, grid. It makes possible layouts that have been impossible in CSS until now.
- Chapter 7 goes deep into positioning using the `position` property: absolute positioning, fixed positioning, and more. This is an area that gets a lot of developers in trouble, and a solid understanding is essential.
- Chapter 8 covers responsive design. We'll look at three key principles to building websites that work on a wide array of screen sizes and device types.

In part 3, “CSS at scale,” we'll look at some more recent best practices. It's one thing to make the elements look how you want on the page. It's another thing to organize your code so it can be understood and maintained into the future as your web app grows and evolves. These chapters will teach you some important techniques for managing your code:

- Chapter 9 teaches how to organize your CSS in a modular way, so that your code is reusable and maintainable.
- Chapter 10 walks you through building a *pattern library*. This is a vital part of using and maintaining CSS on a team.

In part 4, “Advanced topics,” I’ll acquaint you with the world of design. We’ll look at important considerations when working with a designer, and how to do a bit of the design work yourself—because sometimes you will need to:

- Chapter 11 discusses shadows, gradients, and blend modes. These work together to build an elegant user interface.
- Chapter 12 shows how to work with contrast, color, and space. Attention to these details goes a long way toward making a good design a great one.
- Chapter 13 is about web typography: using online font files to bring unique personality to your site or app.
- Chapter 14 brings motion to the page with transitions, changing the shape, color, or size of an element on the page.
- Chapter 15 covers transforms, which are a vital tool to use in conjunction with transitions and animations. This chapter also discusses performance implications of motion on the page.
- Chapter 16 discusses keyframe animations. You’ll learn how to use complex motion to communicate meaning to the user.

There are also two appendices:

- Appendix A is a reference of all the types of CSS selector.
- Appendix B is an introduction to preprocessors. If you’re not already familiar with preprocessors, you might want to start with this appendix first.

I’ve put a lot of effort into the progression of the topics in this book. I start with absolute essentials you have to know. From there, the topics build upon one another. In many places, I refer to earlier concepts and work to tie them together when relevant. While I’ve included helpful reference material in places, I encourage you to read the chapters in order.

Code conventions and repository

This book contains many examples of source code, both in numbered listings and inline with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes code is also **in bold** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; I’ve added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (⇒). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

CSS is meant to be paired with HTML; I always provide a code listing for the HTML and another for the CSS. In most chapters, I reuse the same HTML for multiple

CSS listings. I guide you through editing a stylesheet in many stages, and I've tried to make it clear how I expect you to edit your stylesheet from one CSS listing to the next.

Source code for the listings in this book available in a git repository at <https://github.com/CSSInDepth/css-in-depth> and on the publisher's website at <https://www.manning.com/books/css-in-depth>. At first glance, it may appear that some listings are missing—because working examples require both HTML and CSS, I've put most listings in an HTML file, using `<style>` tags for the CSS. This means that both an HTML listing and a CSS listing are combined in one file in the repository.

For example, in chapter 1, listing 1.1 is HTML code and listing 1.2 is CSS that is meant to be applied to that HTML. I have included both in the repository in a file named `listing-1.2.html`. Changes are made to this CSS in listing 1.3; these are included in `listing-1.3.html`, along with the corresponding HTML from listing 1.1.

Browser versions

Cross-browser testing is an important part of web development. Most of the code in this book is supported in IE 10 and 11, Microsoft Edge, Chrome, Firefox, Safari, Opera, and most mobile browsers. Newer features will not work in all of these browsers; I indicate when this is the case.

Just because a feature is not supported in a particular browser doesn't mean you can't use it. You can often provide *fallback* behavior for the older browsers as an acceptable compromise. I show examples of this in many places.

If you're following along with the code examples on your computer, I recommend you use the latest version of Firefox or Chrome.

Note to print book readers

Many graphics in this book are meant to be viewed in color. The eBook versions display the color graphics, so they should be referred to as you read. To get your free eBook in PDF, ePub, and Kindle formats, go to <https://www.manning.com/freebook> and follow the instructions to complete your pBook registration.

Book forum

Purchase of *CSS in Depth* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://forums.manning.com/forums/css-in-depth>. You can also learn more about Manning's forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author challenging questions lest his interest stray! The forum

and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Keith J. Grant is currently a senior web developer at Intercontinental Exchange, Inc. (ICE), where he wrote and maintains the CSS, both for the corporate and for the New York Stock Exchange websites. He has 11 years of professional experience building and maintaining web applications and websites using HTML, CSS, and JavaScript. He is self-taught in HTML and CSS, and he has several more years of informal experience working with the technology.

His manager brought him onto the website team expressly for his expertise in CSS, when ICE needed to implement a redesign of the websites. CSS enables companies to brand their sites in unique and creative ways, and to offer complex web applications with intricate layouts.

Though Keith has primarily been a JavaScript developer, he has become a very important CSS instructor at every company he's worked for.

About the cover illustration

The figure on the cover of *CSS in Depth* is taken from a nineteenth-century collection of works by many artists, edited by Louis Curmer and published in Paris in 1841. The title of the collection is *Les Français peints par eux-mêmes*, which translates as *The French People Painted by Themselves*. Each illustration is finely drawn and colored by hand, and the rich variety of drawings in the collection reminds us vividly of how culturally apart the world's regions, towns, villages, and neighborhoods were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by pictures from collections such as this one.

Part 1

Reviewing the fundamentals

Part 1 takes a deep look into the most essential parts of CSS—the cascade, relative units, and the box model. These fundamentals, covered in these first three chapters, control what styles are applied to the elements on the page and how the sizes of those elements are determined. A comprehensive understanding of these subjects is foundational to everything else in this book and beyond.

Cascade, specificity, and inheritance

This chapter covers

- The four parts that make up the cascade
- The difference between the cascade and inheritance
- How to control which styles apply to which elements
- Common misunderstandings about shorthand declarations

CSS is unlike a lot of things in the world of software development. It's not a programming language, strictly speaking, but it does require abstract thought. It's not purely a design tool, but it does require some creativity. It provides a deceptively simple declarative syntax, but if you've worked with it on any large projects, you know it can grow into unwieldy complexity.

When you need to learn to do something in conventional programming, you can usually figure out what to search for (for example, "How do I find items of type x in an array?"). With CSS, it's not always easy to distill the problem down to a single question. Even when you can, the answer is often "it depends." The best way to

accomplish something is often contingent on your particular constraints and how precisely you'll want to handle various edge cases.

While it's helpful to know some “tricks” or useful recipes you can follow, mastering CSS requires an understanding of the principles that make those practices possible. This book is full of examples, but it is primarily a book of principles.

Part 1 begins with the most fundamental principles of the language: the cascade, the box model, and the wide array of unit types available. Most web developers know about the cascade and the box model. They know about pixel units and may have heard that they “should use ems instead.” The truth is, there's a lot to these topics, and a cursory understanding of them gets you only so far. If you're ever to master CSS, you must first know the fundamentals, and know them deeply.

I know you're excited to start learning the latest and greatest CSS has to offer. That is the exciting stuff. But first, we'll go back to the fundamentals. I'll quickly review the basics, which you're likely already familiar with, and then dive deep into each topic. My aim is to strengthen the foundation on which the rest of your CSS is built.

In this chapter, we begin with the *C* in *CSS*—the cascade. I'll articulate how it works, then show you how to work with it practically. We then look at a related topic, inheritance. I'll follow that with a look at shorthand properties and some common misunderstandings surrounding them.

Together, these topics are all about applying the styles you want to the elements you want. There's a lot of “gotchas” here that often trip up developers. A good understanding of these topics will give you better control over making your CSS do what you want it to do. With any luck, you'll also better appreciate and even enjoy working with CSS.

1.1 *The cascade*

Fundamentally, CSS is about declaring rules: Under various conditions, we want certain things to happen. If this class is added to that element, apply these styles. If element *X* is a child of element *Y*, apply those styles. The browser then takes these rules, figures out which ones apply where, and uses them to render the page.

When you look at small examples, this process is usually straightforward. But as your stylesheet grows, or the number of pages you apply it to increases, your code can become complex surprisingly quickly. There are often several ways to accomplish the same thing in CSS. Depending on which solution you use, you may get wildly different results when the structure of the HTML changes, or when the styles are applied to different pages. A key part of CSS development comes down to writing rules in such a way that they're predictable.

The first step toward this is understanding how exactly the browser makes sense of your rules. Each rule may be straightforward on its own, but what happens when two rules provide conflicting information about how to style an element? You may find one of your rules doesn't do what you expect because another rule conflicts with it. Predicting how rules behave requires an understanding of the cascade.

To illustrate, you'll build a basic page header like one you might see at the top of a web page (figure 1.1). It has the website title atop a series of teal navigational links. The last link is colored orange to make it stand out as a sort of featured link.

NOTE TO PRINT BOOK READERS Many graphics in this book are meant to be viewed in color. The eBook versions display the color graphics, so they should be referred to as you read. To get your free eBook in PDF, ePub, and Kindle formats, go to <https://www.manning.com/books/css-in-depth> to register your print book.

As you build this page header, you'll probably be familiar with most of the CSS involved. This will allow us to focus on aspects of CSS you might take for granted or only partially understand.

Wombat Coffee Roasters



Figure 1.1 Page heading and navigation links

To begin, create an HTML document and a stylesheet named `styles.css`. Add the code in listing 1.1 to the HTML.

NOTE A repository containing all code listings in this book is available for download at <https://github.com/CSSInDepth/css-in-depth>. The repository has all CSS embedded with the corresponding HTML in a series of HTML files.

Listing 1.1 Markup for the page header

```
<!doctype html>
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <header class="page-header">
    <h1 id="page-title" class="title">Wombat Coffee Roasters</h1>
    <nav>
      <ul id="main-nav" class="nav">
        <li><a href="/">Home</a></li>
        <li><a href="/coffees">Coffees</a></li>
        <li><a href="/brewers">Brewers</a></li>
        <li><a href="/specials" class="featured">Specials</a></li>
      </ul>
    </nav>
  </header>
</body>
```

Diagram annotations:

- Page title: points to the `<h1 id="page-title" class="title">Wombat Coffee Roasters</h1>` line.
- List of navigation links: points to the `<ul id="main-nav" class="nav">` line.
- Featured link: points to the `Specials` line.

When two or more rules target the same element on your page, the rules may provide conflicting declarations. The next listing shows how this is possible. It shows three rulesets, each specifying a different font style for the page title. The title can't have three different fonts at one time. Which one will it be? Add this to your CSS file to see.

Listing 1.2 Conflicting declarations

```
h1 {
  font-family: serif;
}

#page-title {
  font-family: sans-serif;
}

.title {
  font-family: monospace;
}
```

← Tag (or type) selector

← ID selector

← Class selector

Rulesets with conflicting declarations can appear one after the other, or they can be scattered throughout your stylesheet. Either way, given your HTML, they all target the same element.

All three rulesets attempt to set a different font family to this heading. Which one will win? To determine the answer, the browser follows a set of rules, so the result is predictable. In this case, the rules dictate that the second declaration, which has an ID selector, wins; the title will have a sans-serif font (figure 1.2).

The *cascade* is the name for this set of rules. It determines how conflicts are resolved, and it's a fundamental part of how the language works. Although most experienced developers have a general sense of the cascade, parts of it are sometimes misunderstood.

Wombat Coffee Roasters

- [Home](#)
- [Coffees](#)
- [Brewers](#)
- [Specials](#)

Figure 1.2 The ID selector wins over the other rulesets, producing a sans-serif font for the title.

Let's unpack the cascade. When declarations conflict, the cascade considers three things to resolve the difference:

- 1 *Stylesheet origin*—Where the styles come from. Your styles are applied in conjunction with the browser's default styles.
- 2 *Selector specificity*—Which selectors take precedence over which.
- 3 *Source order*—Order in which styles are declared in the stylesheet.

The rules of the cascade are considered in this order. Figure 1.3 shows how they're applied at a higher level.

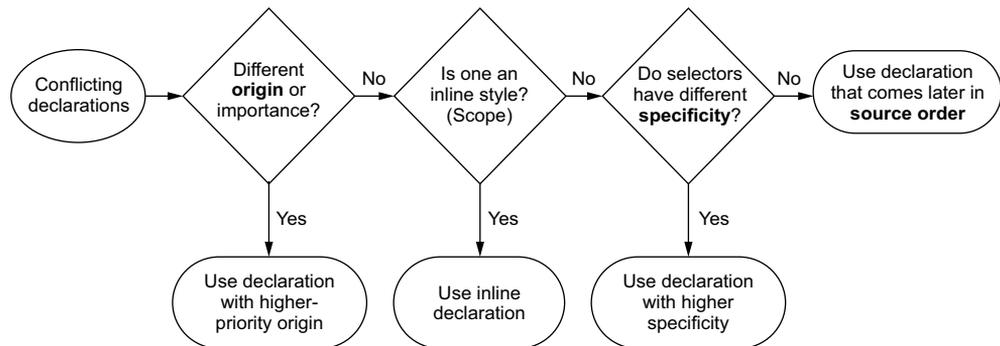


Figure 1.3 High-level flowchart of the cascade showing declaration precedence

These rules allow browsers to behave predictably when resolving any ambiguity in the CSS. Let's step through them one at a time.

A quick review of terminology

Depending on where you learned CSS, you may or may not be familiar with all the names of the various parts of CSS syntax. I won't belabor the point, but because I'll be using these terms throughout the book, it's best to be clear about what they mean.

Following is a line of CSS. This is called a *declaration*. This declaration is made up of a *property* (color) and a *value* (black):

```
color: black;
```

Properties aren't to be confused with *attributes*, which are part of the HTML syntax. For example, in the element ``, href is an attribute of the a tag.

A group of declarations inside curly braces is called a *declaration block*. A declaration block is preceded by a *selector* (in this case, body):

```
body {
  color: black;
  font-family: Helvetica;
}
```

Together, the selector and declaration block are called a *ruleset*. A ruleset is also called a *rule*—although, it's my observation that *rule* is rarely used so precisely and is usually used in the plural to refer to a broader set of styles.

Finally, *at-rules* are language constructs beginning with an “at” symbol, such as `@import` rules or `@media` queries.

1.1.1 Understanding stylesheet origin

The stylesheets you add to your web page aren't the only ones the browser applies. There are different types, or origins, of stylesheets. Yours are called *author* styles; there are also user agent styles, which are the browser's default styles. User agent styles have lower priority, so your styles override them.

NOTE Some browsers let users define a *user stylesheet*. This is considered a third origin, with a priority between user agent and author styles. User styles are rarely used and beyond your control, so I've left them out for simplicity.

User agent styles vary slightly from browser to browser, but generally they do the same things: headings (<h1> through <h6>) and paragraphs (<p>) are given a top and bottom margin, lists (and) are given a left padding, and link colors and default font sizes are set.

USER AGENT STYLES

Let's look again at the example page (figure 1.4). The title is sans-serif because of the styles you added. A number of other things are determined by the user agent styles: the list has a left padding and a `list-style-type` of `disc` to produce the bullets. Links are blue and underlined. The heading and the list have top and bottom margins.

Wombat Coffee Roasters

- [Home](#)
- [Coffees](#)
- [Brewers](#)
- [Specials](#)

Figure 1.4 User agent styles set defaults for your web page header.

After user agent styles are considered, the browser applies your styles—the author styles. This allows declarations you specify to override those set by the user agent stylesheet. If you link to several stylesheets in your HTML, they all have the same origin: the author.

The user agent styles set things you typically want, so they don't do anything entirely unexpected. When you don't like what they do to a certain property, set your own value in your stylesheet. Let's do that now. You can override some of the user agent styles that aren't what you want so your page will look like figure 1.5.

Wombat Coffee Roasters



Figure 1.5 Author styles override user agent styles because they have higher priority.

In the following listing, I've removed the conflicting font-family declarations from the earlier example and added new ones to set colors and override the user agent margins and the list padding and bullets. Edit your stylesheet to match these changes.

Listing 1.3 Overriding user agent styles

```

h1 {
  color: #2f4f4f;
  margin-bottom: 10px;
}

#main-nav {
  margin-top: 10px;
  list-style: none;
  padding-left: 0;
}

#main-nav li {
  display: inline-block;
}

#main-nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

```

← Reduces the margins

← Removes user agent list styles

← Makes list items appear side by side rather than stacked

Provides a button-like appearance for the navigational links

If you've worked with CSS for long, you're probably used to overriding user agent styles. When you do, you're using the origin part of the cascade. Your styles will always override the user agent styles because the origins are different.

NOTE You may notice I used ID selectors in this code. There are reasons to avoid doing this, which I'll cover in a bit.

IMPORTANT DECLARATIONS

There's an exception to the style origin rules: declarations that are marked as *important*. A declaration can be marked important by adding `!important` to the end of the declaration, before the semicolon:

```
color: red !important;
```

Declarations marked `!important` are treated as a higher-priority origin, so the overall order of preference, in decreasing order, is this:

- 1 Author important
- 2 Author
- 3 User agent

The cascade independently resolves conflicts for every property of every element on the page. For instance, if you set a bold font on a paragraph, the top and bottom margin from the user agent stylesheet still applies (unless you explicitly override them). The concept of style origin will come into play when we get to transitions and animations because they introduce more origins to this list. The important annotation is an interesting quirk of CSS, which we'll come back to again shortly.

1.1.2 Understanding specificity

If conflicting declarations can't be resolved based on their origin, the browser next tries to resolve them by looking at their *specificity*. Understanding specificity is essential. You can go a long way without an understanding of stylesheet origin because 99% of the styles on your website come from the same origin. But if you don't understand specificity, it will bite you. Sadly, it's often a missed concept.

The browser evaluates specificity in two parts: styles applied inline in the HTML and styles applied using a selector.

INLINE STYLES

If you use an HTML `style` attribute to apply styles, the declarations are applied only to that element. These are, in effect, “scoped” declarations, which override any declarations applied from your stylesheet or a `<style>` tag. Inline styles have no selector because they are applied directly to the element they target.

In your page, you want the featured Specials link in the navigation menu to be orange, as shown in figure 1.6. I'll evaluate several ways you can accomplish this, beginning with inline styles in listing 1.4.

Wombat Coffee Roasters



Figure 1.6 Applying inline styles overrides the styles applied using selectors.

To see this in your browser, edit your page to match the code given here. (You'll undo this change in a moment.)

Listing 1.4 Inline styles overriding declarations applied elsewhere

```
<li>
  <a href="/specials" class="featured"
    style="background-color: orange;">
    Specials
  </a>
</li>
```

← Inline style applied via the style attribute

To override inline declarations in your stylesheet, you'll need to add an `!important` to the declaration, shifting it into a higher-priority origin. If the inline styles are marked important, then nothing can override them. It's preferable to do this from within the stylesheet. Undo this change, and we'll look at better approaches.

SELECTOR SPECIFICITY

The second part of specificity is determined by the selectors. For instance, a selector with two class names has a higher specificity than a selector with only one. If one declaration sets a background to orange, but another with higher specificity sets it to teal, the browser applies the teal color.

To illustrate, let's see what happens when we try to turn the featured link orange with a simple class selector. Update the final part of your stylesheet so it matches the code given here.

Listing 1.5 Selectors with different specificities

```
#main-nav a {  
  color: white;  
  background-color: #13a4a4;  
  padding: 5px;  
  border-radius: 2px;  
  text-decoration: none;  
}  
  
.featured {  
  background-color: orange;  
}
```

Higher specificity selector

Teal background color

The orange background declaration won't override the teal due to selector specificity.

It doesn't work! All the links remain teal. Why? The first selector here is more specific than the second. It's made up of an ID and a tag name, whereas the second is made up of a class name. There's more to this than merely seeing which selector is longer, however.

Different types of selectors also have different specificities. An ID selector has a higher specificity than a class selector, for example. In fact, a single ID has a higher specificity than a selector with any number of classes. Similarly, a class selector has a higher specificity than a tag selector (also called a *type selector*).

The exact rules of specificity are:

- If a selector has more IDs, it wins (that is, it's more specific).
- If that results in a tie, the selector with the most classes wins.
- If that results in a tie, the selector with the most tag names wins.

Consider the selectors shown in the following listing (but don't add them to your page). These are written in order of increasing specificity.

Listing 1.6 Selectors with increasing specificities

```

html body header h1 {      ← ① Four tags
  color: blue;
}

body header.page-header h1 {  ← ② Three tags and
  color: orange;              one class
}

.page-header .title {        ← ③ Two classes
  color: green;
}

#page-title {                ← ④ One ID
  color: red;
}

```

The most specific selector here is ④, with one ID, so its color declaration of red is applied to the title. The next specific is ③, with two class names. This would be applied if the ID selector ④ were absent. Selector ③ has a higher specificity than selector ②, despite its length: two classes are more specific than one class. Finally, ① is the least specific, with four element types (that is, tag names) but no IDs or classes.

NOTE Pseudo-class selectors (for example, `:hover`) and attribute selectors (for example, `[type="input"]`) each have the same specificity as a class selector. The universal selector (`*`) and combinators (`>`, `+`, `~`) have no effect on specificity.

If you add a declaration to your CSS and it seems to have no effect, often it's because a more specific rule is overriding it. Many times developers write selectors using IDs, without realizing this creates a higher specificity, one that is hard to override later. If you need to override a style applied using an ID, you have to use another ID.

It's a simple concept, but if you don't understand specificity, you can drive yourself mad trying to figure out why one rule works and another doesn't.

A NOTATION FOR SPECIFICITY

A common way to indicate specificity is in a number form, often with commas between each number. For example, "1,2,2" indicates a specificity of one ID, two classes, and two tags. IDs having the highest priority are listed first, followed by classes, then tags.

The selector `#page-header #page-title` has two IDs, no classes, and no tags. We can say this has a specificity of 2,0,0. The selector `ul li`, with two tags but no IDs or classes, has a specificity of 0,0,2. Table 1.1 shows the selectors from listing 1.6.

Table 1.1 Various selectors and their corresponding specificities

Selector	IDs	Classes	Tags	Notation
html body header h1	0	0	4	0,0,4
body header.page-header h1	0	1	3	0,1,3
.page-header .title	0	2	0	0,2,0
#page-title	1	0	0	1,0,0

It now becomes a matter of comparing the numbers to determine which selector is more specific. A specificity of 1,0,0 takes precedence over a specificity of 0,2,2 and even over 0,10,0 (although I don't recommend ever writing selectors as long as one with 10 classes), because the first number (IDs) is of the higher priority.

Occasionally, people use a four-number notation with a 0 or 1 in the most significant digit to represent whether a declaration is applied via inline styles. In this case, an inline style has a specificity of 1,0,0,0. This would override styles applied via selectors, which could be indicated as having specificities of 0,1,2,0 (one ID and two classes) or something similar.

SPECIFICITY CONSIDERATIONS

When you tried to apply the orange background using the `.featured` selector, it didn't work. The selector `#main-nav a` has an ID that overrides the class selector (specificities 1,0,1 and 0,1,0). To correct this, you have some options to consider. Let's look at several possible fixes.

The quickest fix is to add an `!important` to the declaration you want to favor. Change the declaration to match that given here.

Listing 1.7 Possible fix one

```
#main-nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

.featured {
  background-color: orange !important;
}
```

Makes the declaration
important; it's now a
higher-priority origin.

This works because the `!important` annotation raises the declaration to a higher-priority origin. Sure, it's easy, but it's also a naive fix. It may do the trick now, but it can cause you problems down the road. If you start adding `!important` to multiple declarations, what happens when you need to trump something already set to important? When you give several declarations an `!important`, then the origins match and the

regular specificity rules apply. This ultimately will leave you back where you started; once you introduce an `!important`, more are likely to follow.

Let's find a better way. Instead of trying to get around the rules of selector specificity, let's try to make them work for us. What if you raised the specificity of your selector? Update the rulesets in your CSS to match this listing.

Listing 1.8 Possible fix two

```
#main-nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

#main-nav .featured {
  background-color: orange;
}
```

← Specificity remains 1,0,1

← Increases the specificity to 1,1,0

← The !important annotation is no longer necessary.

This fix also works. Now, your selector has one ID and one class, giving it a specificity of 1,1,0, which is higher than `#main-nav a` (a specificity of 1,0,1), so the background color orange is applied to the element.

You can still make this better, though. Instead of *raising* the specificity of the second selector, let's see if we can *lower* the specificity of the first. The element has a class as well: `<ul id="main-nav" class="nav">`, so you can change your CSS to target the element by its class name rather than its ID. Change `#main-nav` to `.nav` in your selectors as shown here.

Listing 1.9 Possible fix three

```
.nav {
  margin-top: 10px;
  list-style: none;
  padding-left: 0;
}

.nav li {
  display: inline-block;
}

.nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}
```

← Changes "#main-nav" to ".nav" throughout stylesheet

← Lowers the first specificity (0,1,1)

```
.nav .featured {
  background-color: orange;
}
```

← Increases the second specificity (0,2,0)

You've brought the specificity of the selectors down. The orange background is high enough to override the teal.

As you can see from these examples, specificity tends to become a sort of arms race. This is particularly the case with large projects. It is generally best to keep specificity low when you can, so when you need to override something, your options are open.

1.1.3 Understanding source order

The third and final step to resolving the cascade is source order. If the origin and the specificity are the same, then the declaration that appears later in the stylesheet—or appears in a stylesheet included later on the page—takes precedence.

This means you can manipulate the source order to style your featured link. If you make the two conflicting selectors equal in specificity, then whichever appears last wins. Let's consider the fourth option shown in the following listing.

Listing 1.10 Possible fix four

```
.nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

a.featured {
  background-color: orange;
}
```

← Makes the specificities equal (0,1,1)

In this solution, the specificities are equal. Source order determines which declaration is applied to your link, resulting in an orange featured button.

This addresses your problem but, potentially, also introduces a new one: although a featured button inside the nav looks correct, what happens if you want to use the featured class on another link elsewhere on the page, outside of your nav? You'll get an odd blend of styles: the orange background, but not the text color, padding, or border radius of the navigational links (figure 1.7).

Wombat Coffee Roasters

Home Coffees Brewers **Specials**

Be sure to check out [our specials](#).

Figure 1.7 The featured class outside the nav declaration produces odd results.

Listing 1.11 shows the markup that creates this behavior. There’s now an element targeted only by the second selector, but not the first, which produces an undesirable result. You’ll have to decide whether you want this orange button style to work outside of the nav, and if you do, you’ll need to make sure all the desired styles apply to it as well.

Listing 1.11 Featured link outside of nav

```
<header class="page-header">
  <h1 id="page-title" class="title">Wombat Coffee Roasters</h1>
  <nav>
    <ul id="main-nav" class="nav">
      <li><a href="/">Home</a></li>
      <li><a href="/coffees">Coffees</a></li>
      <li><a href="/brewers">Brewers</a></li>
      <li><a href="/specials" class="featured">Specials</a></li>
    </ul>
  </nav>
</header>
<main>
  <p>
    Be sure to check out
    <a href="/specials" class="featured">our specials</a>.
  </p>
</main>
```

Featured link outside nav will be partially styled

With no other information about your needs on this site, I’d be inclined to stick with fix number three (listing 1.9). Ideally on your website, you’ll be able to make some educated guesses about your needs elsewhere. Perhaps you know that you are likely to need a featured link in other places. In that case, perhaps fix four (listing 1.10) would be what you want, with the addition of styles to support the featured class elsewhere on the page.

Very often in CSS, as I said earlier, the best answer is “it depends.” There are many paths to the same end result. It’s worth considering several options and thinking about the ramifications of each. When facing a styling problem, I often tackle it in two phases: First figure out what declarations will get it looking right. Second, think through the possible ways to structure the selectors and choose the one that best fits your needs.

LINK STYLES AND SOURCE ORDER

When you began studying CSS, you may have learned that your selectors for styling links should go in a certain order. That’s because source order affects the cascade. This listing shows styles for links on a page in the “correct” order.

Listing 1.12 Link styles

```
a:link {
  color: blue;
  text-decoration: none;
}
```

```
a:visited {
  color: purple;
}

a:hover {
  text-decoration: underline;
}

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

The cascade is the reason this order matters: given the same specificity, later styles override earlier styles. If two or more of these states are true of one element at the same time, the last one can override the others. If the user hovers over a visited link, the hover styles take precedence. If the user activates the link (that is, clicks it) while hovering over it, the active styles take precedence.

A helpful mnemonic to remember this order is LoVe/HATe—link, visited, hover, active. Note that if you change one of the selectors to have a different specificity than the others, this will break down and you may get unexpected results.

CASCADED VALUES

The browser follows these three steps—origin, specificity, and source order to resolve every property for every element on the page. A declaration that “wins” the cascade is called a *cascaded value*. There’s at most one cascaded value per property per element. A particular paragraph (<p>) on the page can have a top margin and a bottom margin, but it can’t have two different top margins or two different bottom margins. If the CSS specifies different values for one property, the cascade will choose only one when rendering the element. This is the cascaded value.



cascaded value—A value for a particular property applied to an element as a result of the cascade.

If a property is never specified for an element, it has no cascaded value for that property. The same paragraph, for instance, may not have a border or padding specified.

1.1.4 Two rules of thumb

As you may know, there are two common rules of thumb for working with the cascade. Because these can be helpful, here’s a reminder:

- 1 *Don’t use IDs in your selector.* Even one ID ratchets up the specificity a lot. When you need to override the selector, you often don’t have another meaningful ID you can use, so you wind up having to copy the original selector and add another class to distinguish it from the one you are trying to override.

- 2 *Don't use !important.* This is even more difficult to override than an ID, and once you use it, you'll need to add it every time you want to override the original declaration—and then you still have to deal with the specificity.

These two rules can be good advice, but don't cling to them forever. There are exceptions where they can be okay, but never use them in a knee-jerk reaction to win a specificity battle.

An important note about importance

If you're creating a JavaScript module for distribution (such as an NPM package), I strongly urge you not to apply styles inline via JavaScript if it can be avoided. If you do, you're forcing developers using your package to either accept your styles exactly or use `!important` for every property they want to change.

Instead, include a stylesheet in your package. If your component needs to make style changes dynamically, it's almost always preferable to use JavaScript to add and remove classes to the elements. Then users can use your stylesheet, and they have the option to edit it however they like without battling specificity.

A series of practical methodologies has emerged in the last few years to help with managing selector specificity. We'll look at those in detail in chapter 9. There I'll talk more about dealing with specificity, including one place that `!important` is okay. But now that you're clear on how the cascade behaves, we can press on.

1.2 Inheritance

There's one last way that an element can receive styles—*inheritance*. The cascade is frequently conflated with the concept of inheritance. Although the two topics are related, you should understand each individually.

If an element has no cascaded value for a given property, it may inherit one from an ancestor element. It's common to apply a `font-family` to the `<body>` element. All the ancestor elements within will then inherit this font; you don't have to apply it explicitly to each element on the page. Figure 1.8 shows how inheritance flows down the DOM tree.

Not all properties are inherited, however. By default, only certain ones are. In general, these are the properties you'll *want* to be inherited. They are primarily properties pertaining to text: `color`, `font`, `font-family`, `font-size`, `font-weight`, `font-variant`, `font-style`, `line-height`, `letter-spacing`, `text-align`, `text-indent`, `text-transform`, `white-space`, and `word-spacing`.

A few others inherit as well, such as the list properties: `list-style`, `list-style-type`, `list-style-position`, and `list-style-image`. The table border properties, `border-collapse` and `border-spacing`, are also inherited; note that these control border behavior of tables, not the more commonly used properties for specifying

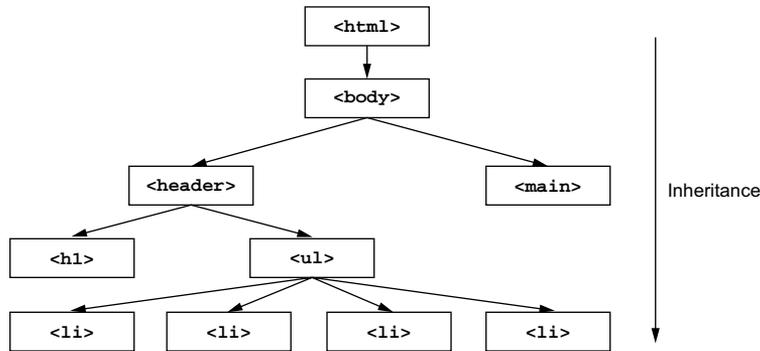


Figure 1.8 Inherited properties are passed down the DOM tree from parent nodes to their descendants.

borders for non-table elements. (We wouldn't want a `<div>` passing its border down to every descendant element.) This is not quite a comprehensive list, but very nearly.

You can use inheritance in your favor on your page by applying a font to the body element, allowing its descendant elements to inherit that value (figure 1.9).

Wombat Coffee Roasters

Home Coffees Brewers **Specials**

Figure 1.9 Apply a `font-family` to the body and let all descendant elements inherit the same value.

Add this code to the top of your stylesheet to apply this principle to your page.

Listing 1.13 Applying `font-family` to a parent element

```
body {
  font-family: sans-serif;
}
```

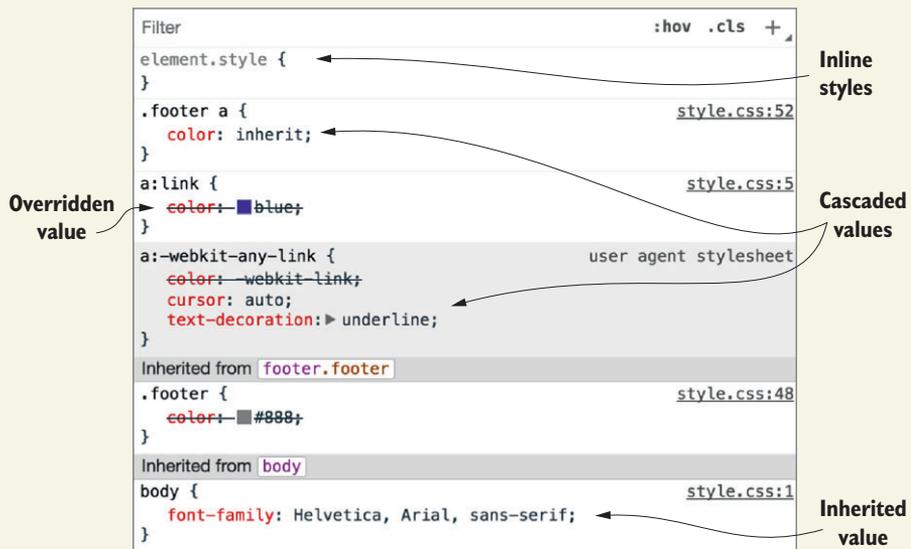
← An inherited property will be applied to descendant elements as well.

This is applied to the entire page by adding it to the body. But you can also target a specific element on the page, and it will only inherit to its descendant elements. The inheritance will pass from element to element until it's overridden by a cascaded value.

Use your DevTools!

A complicated nest of values inheriting and overriding one another can quickly become difficult to keep track of. If you're not already familiar with your browser's developer tools, get in the habit of using them.

DevTools provide visibility into exactly which rules are applying to which elements and why. The cascade and inheritance are abstract concepts; DevTools are the best way I know to get my bearings. Open them by right-clicking an element and choosing Inspect or Inspect Element from the context menu. Here's an example of what you'll see.



The style inspector shows every selector targeting the inspected element, ordered by specificity. Below that are all inherited properties. This shows all of the cascade and inheritance for the element at a glance.

There are lots of subtle features to help you make sense of what's happening with an element's styles. Styles closer to the top override those below. Overridden styles are crossed out. The stylesheet and line number for each ruleset are shown on the right, so you can find them in your source code. This tells you exactly which element inherited which styles and where they originated. You can also type in the Filter box at the top to hide all but a certain set of declarations.

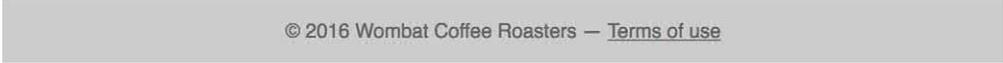
1.3 Special values

There are two special values that you can apply to any property to help manipulate the cascade: `inherit` and `initial`. Let's take a look at these.

1.3.1 Using the `inherit` keyword

Sometimes, you'll want inheritance to take place when a cascaded value is preventing it. To do this, you can use the keyword `inherit`. You can override another value with this, and it will cause the element to inherit that value from its parent.

Suppose you add a light gray footer to your page. In the footer, there may be some links, but you don't want them to stand out too much because the footer's isn't an important part of the page. So you'll make the links in the footer dark gray (figure 1.10).



© 2016 Wombat Coffee Roasters — [Terms of use](#)

Figure 1.10 The Terms of Use link when it inherits the gray text color

Add this markup to the end of your page. A normal page would have more content between this and the header, but this will serve the purpose.

Listing 1.14 Footer with a link

```
<footer class="footer">
  &copy; 2016 Wombat Coffee Roasters &mdash;
  <a href="/terms-of-use">Terms of use</a>
</footer>
```

Typically, you'll have a font color set for all links on the page (and if you don't, the user agent styles sets one), and that color is applied to the Terms of Use link as well. To make the link in the footer gray, you'll need to override it. Add this code to your stylesheet to do that.

Listing 1.15 The `inherit` value

```
a:link {
  color: blue;
}
...
.footer {
  color: #666;
  background-color: #ccc;
  padding: 15px 0;
  text-align: center;
  font-size: 14px;
}
.footer a {
  color: inherit;
  text-decoration: underline;
}
```

Global link color for the page

Footer text color set to gray

Inherits font color from the footer

The third ruleset here overrides the blue link color, giving the link in the footer a cascaded value of `inherit`. Thus, it inherits the color from its parent, `<footer>`.

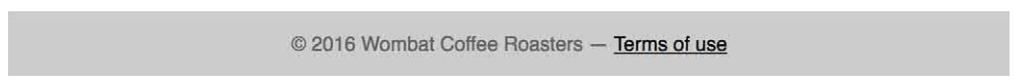
The benefit here is that the footer link will change along with the rest of the footer should anything alter it. (Editing the second ruleset can do this, or another style elsewhere could override it.) If, for example, the footer text on certain pages is a darker gray, then the link will change to match.

You can also use the `inherit` keyword to force inheritance of a property not normally inherited, such as `border` or `padding`. There are few practical uses for this, but you'll see one useful case in chapter 3 when we look at box sizing.

1.3.2 Using the `initial` keyword

Sometimes you'll find you have styles applied to an element that you want to undo. You can do this by specifying the keyword `initial`. Every CSS property has an initial, or default, value. If you assign the value `initial` to that property, then it effectively resets to its default value. It's like a hard-reset of that value. Figure 1.11 shows how your footer renders if you give it a value of `initial` rather than `inherit`.

WARNING The `initial` keyword isn't supported in any version of Internet Explorer or Opera Mini. It works in all other major browsers, including Edge, Microsoft's successor to IE11.



© 2016 Wombat Coffee Roasters — [Terms of use](#)

Figure 1.11 The initial value for the color property is black.

The CSS in figure 1.11 is shown in the following listing. Because `black` is the initial value for the `color` property in most browsers, `color: initial` is equivalent to `color: black`.

Listing 1.16 The initial value

```
.footer a {  
  color: initial;  
  text-decoration: underline;  
}
```

The benefit of this is you don't have to think about it as much. If you want to remove a border from an element, set `border: initial`. If you want to restore an element to its default width, set `width: initial`.

You may be in the habit of using the value `auto` to do this sort of reset. In fact, you can use `width: auto` to achieve the same result. This is because the default value of `width` is `auto`.

It's important to note, however, that `auto` isn't the default value for all properties. It's not even valid for many properties; for example, `border-width: auto` and `padding: auto` are invalid and therefore have no effect. You could take the time to dig up the initial value for these, but it's often easier to use `initial`.

NOTE Declaring `display: initial` is equivalent to `display: inline`. It won't evaluate to `display: block` regardless of what type of element you apply it to. That's because `initial` resets to the initial value for the property, not the element; `inline` is the default value for the `display` property.

1.4 Shorthand properties

Shorthand properties are properties that let you set the values of several other properties at one time. For example, `font` is a shorthand property that lets you set several font properties. This declaration specifies `font-style`, `font-weight`, `font-size`, `line-height`, and `font-family`:

```
font: italic bold 18px/1.2 "Helvetica", "Arial", sans-serif;
```

Similarly,

- `background` is a shorthand property for multiple background properties: `background-color`, `background-image`, `background-size`, `background-repeat`, `background-position`, `background-origin`, `background-clip`, and `background-attachment`.
- `border` is a shorthand for `border-width`, `border-style`, and `border-color`, which are each in turn shorthand properties as well.
- `border-width` is shorthand for the top, right, bottom, and left border widths.

Shorthand properties are useful for keeping your code succinct and clear, but a few quirks about them aren't readily apparent.

1.4.1 Beware shorthands silently overriding other styles

Most shorthand properties let you omit certain values and only specify the bits you're concerned with. It's important to know, however, that doing this still sets the omitted values; they'll be set implicitly to their initial value. This can silently override styles you specify elsewhere. If, for example, you were to use the shorthand `font` property for the page title without specifying a `font-weight`, a font weight of `normal` would still be set (figure 1.12).

Wombat Coffee Roasters

Figure 1.12 Shorthand properties will set omitted values to their initial value.

Add the code from this listing to your stylesheet for an example of how this works.

Listing 1.17 Shorthand property specifying all associated values

```
h1 {
  font-weight: bold;
}

.title {
  font: 32px Helvetica, Arial, sans-serif;
}
```

At first glance, it may seem that `<h1 class="title">` would result in a bold heading, but it doesn't. These styles are equivalent to this code.

Listing 1.18 Expanded equivalent to the shorthand in listing 1.17

```
h1 {
  font-weight: bold;
}

.title {
  font-style: normal;
  font-variant: normal;
  font-weight: normal;
  font-stretch: normal;
  line-height: normal;
  font-size: 32px;
  font-family: Helvetica, Arial, sans-serif;
}
```

Initial values
of these
properties

This means that applying these styles to `<h1>` results in a normal font weight, not bold. It can also override other font styles that would otherwise be inherited from an ancestor element. Of all the shorthand properties, `font` is the most egregious for causing problems, because it sets such a wide array of properties. For this reason, I avoid using it except to set general styles on the `<body>` element. You can still encounter this problem with other shorthand properties, so be aware of this possibility.

1.4.2 Understanding the order of shorthand values

Shorthand properties try to be lenient when it comes to the order of the values you specify. You can set `border: 1px solid black` or `border: black 1px solid` and either will work. That's because it's clear to the browser which value specifies the width, which specifies the color, and which specifies the border style.

But there are many properties where the values can be more ambiguous. In these cases, the order of the values is significant. It's important to understand this order for the shorthand properties you use.

TOP, RIGHT, BOTTOM, LEFT

Shorthand property order particularly trips up developers when it comes to properties like `margin` and `padding`, or some of the border properties that specify values for each of the four sides of an element. For these properties, the values are in clockwise order, beginning at the top.

Remembering this order can keep you out of trouble. In fact, the word *TRouBLE* is an mnemonic you can use to remember the order: top, right, bottom, left.

You can use this mnemonic to set padding on the four sides of an element. The links shown in figure 1.13 have a top padding of 10 px, right padding of 15 px, bottom padding of 0, and left padding of 5 px. This looks uneven, but it illustrates the principle.



Figure 1.13 Elements with various paddings on each side

This listing shows the CSS for these links.

Listing 1.19 Specifying padding on each side of an element

```
.nav a {
  color: white;
  background-color: #13a4a4;
  padding: 10px 15px 0 5px;
  border-radius: 2px;
  text-decoration: none;
}
```

← Top, right, bottom, and left padding

Properties whose values follow this pattern also support truncated notations. If the declaration ends before one of the four sides is given a value, that side takes its value from the opposite side. Specify three values, and the left and right side will both use the second one. Specify two values, and the top and bottom will use the first one. If you specify only one value, it will apply to all four sides. Thus, the following declarations are all equivalent:

```
padding: 1em 2em;
padding: 1em 2em 1em;
padding: 1em 2em 1em 2em;
```

These are equivalent to one another as well:

```
padding: 1em;
padding: 1em 1em;
padding: 1em 1em 1em;
padding: 1em 1em 1em 1em;
```

For many developers, the most problematic of these is when three values are given. Remember, this specifies the top, right, and bottom. Because no left value is given, it will take the same value as the right; the second value will be applied to both the left

and right sides. Thus, `padding: 10px 15px 0` applies 15 px padding to both the left and right sides, whereas the top padding is 10 px and the bottom padding is 0.

Most often, however, you'll need two values. On smaller elements in particular, it's often better to have more padding on the sides than on the top and bottom. This approach looks good on buttons or, in your page, navigational links (figure 1.14).



Figure 1.14 Many elements look better with more horizontal padding.

Update your stylesheet to match this listing. It uses the property shorthand to apply the vertical padding first, then the horizontal.

Listing 1.20 Specifying two padding values

```
.nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px 15px;
  border-radius: 2px;
  text-decoration: none;
}
```

← Top and bottom padding, then left and right padding

Because so many common properties follow this pattern, it's worth committing this order to memory.

HORIZONTAL, VERTICAL

The TRouBLE mnemonic only applies to properties that apply individually to all four sides of the box. Other properties only support up to two values. These include properties like `background-position`, `box-shadow`, and `text-shadow` (although these aren't shorthand properties, strictly speaking). Compared to the four-value properties like `padding`, the order of these values is reversed. Whereas `padding: 1em 2em` specifies the vertical top/bottom values first, followed by the horizontal right/left values, `background-position: 25% 75%` specifies the horizontal right/left values first, followed by the vertical top/bottom values.

Although it seems counter-intuitive that these are opposite, the reason for this is straightforward: the two values represent a Cartesian grid. Cartesian grid measurements are typically given in the order *x*, *y* (horizontal and then vertical). If, for example, you wanted to apply a shadow like the one shown in figure 1.15, you'd specify the *x* (horizontal) value first.



Figure 1.15 Box shadow positioned at 10px 2px

The styles for this element are given here.

Listing 1.21 Box-shadow specifies x value then y value

```
.nav .featured {  
  background-color: orange;  
  box-shadow: 10px 2px #6f9090;  
}
```



Shadow offset 10px to the right and 2px down

The first (larger) value applies to the horizontal offset, whereas the second (smaller) value applies to the vertical.

If you're working with a property that specifies two measurements from a corner, think "Cartesian grid." If you're working with one that specifies measurements for each side all the way around an element, think "clock."

Summary

- Keep selector specificity under control.
- Don't confuse cascade with inheritance.
- Certain properties are inherited, including those for text, lists, and table borders.
- Don't confuse `initial` and `auto` values.
- Stay out of TRouBLE with shorthand properties.

Working with relative units

This chapter covers

- The versatility of relative units
- How to use ems and rems, without letting them drive you mad
- Using viewport-relative units
- An introduction to CSS variables

When it comes to specifying values, CSS provides a wide array of options to choose from. One of the most familiar, and probably easiest to work with, is pixels. These are known as *absolute* units; that is, 5 px always means the same thing. Other units, such as em and rem, are not absolute, but *relative*. The value of relative units changes, based on external factors; for example, the meaning of 2 em changes depending on which element (and sometimes even which property) you're using it on. Naturally, this makes relative units more difficult to work with.

Developers, even experienced CSS developers, often dislike working with relative units, the notorious em included. The way the value of an em can change makes it seem unpredictable and less clear-cut than the pixel. In this chapter, I'll remove the mystery surrounding relative units. First, I'll explain the unique value they bring to CSS, then I'll help you make sense of them. I'll explain how they

work, and I'll show you how to tame their seemingly unpredictable nature. You can make relative values work for you, and wielded correctly, they'll make your code simpler, more versatile, and easier to work with.

2.1 The power of relative values

CSS brings a *late-binding* of styles to the web page: The content and its styles aren't pulled together until after the authoring of both is complete. This adds a level of complexity to the design process that doesn't exist in other types of graphic design, but it also provides more power—one stylesheet can be applied to hundreds, even thousands, of pages. Furthermore, the final rendering of the page can be altered directly by the user, who, for example, can change the default font size or resize the browser window.

In early computer application development (as well as in traditional publishing), developers (or publishers) knew the exact constraints of their medium. A particular program window might be 400 px wide by 300 px tall, or a page could be 4 in. wide by 6½ in. tall. Consequently, when developers set about laying out the application's buttons and text, they knew exactly how big they could make those elements and exactly how much space that would leave them to work with for other items on the screen. On the web, this is not the case.

2.1.1 The struggle for pixel-perfect design

In the web environment, the user can have their browser window set to any number of sizes, and the CSS has to apply to it. Furthermore, users can resize the page after it's opened, and the CSS needs to adjust to new constraints. This means that styles can't be applied when you create your page; the browser must calculate those when the page is rendered onscreen.

This adds a layer of abstraction to CSS. We can't style an element according to an ideal context; we need to specify rules that'll work in any context where that element could be placed. With today's web, your page will need to render on a 4-in. phone screen as well as on a 30-in. monitor.

For a long time, designers mitigated this complexity by focusing on “pixel-perfect” designs. They'd create a tightly defined container, often a centered column around 800 px wide. Then, within these constraints, they'd go about designing more or less like their predecessors did with native applications or print publications.

2.1.2 The end of the pixel-perfect web

As technology improved and manufacturers introduced higher-resolution monitors, the pixel-perfect approach slowly started to break down. In the early 2000s, there was a lot of discussion on whether we developers could safely design for displays 1,024 px wide instead of 800 px wide. Then, we'd have the same conversation all over again for 1,280 px. We had to make judgment calls. Was it better to make our site too wide for older computers or too narrow for new ones?

When smartphones emerged, developers were forced to stop pretending everyone could have the same experience on their sites. Whether we loved it or hated it, we had to abandon columns of some known number of pixels, and begin thinking about *responsive* design. We could no longer hide from the abstraction that comes with CSS. We had to embrace it.



responsive—In CSS, this refers to styles that “respond” differently, based on the size of the browser window. This entails intentional consideration for mobile, tablet, or desktop screens of any size. We’ll take a good look at responsive design in chapter 8, but in this chapter, I’ll lay some important groundwork before we get there.

Added abstraction means additional complexity. If I give an element a width of 800 px, how will that look in a smaller window? How will a horizontal menu look if it doesn’t all fit on one line? As you write your CSS, you need to be able to think simultaneously in specifics, as well as in generalities. When you’ve multiple ways to solve a particular problem, you’ll need to favor the solution that works more generally under multiple and different circumstances.

Relative units are one of the tools CSS provides to work at this level of abstraction. Instead of setting a font size at 14 px, you can set it to scale proportionally to the size of the window. Or, you can set the size of everything on the page relative to the base font size, and then resize the entire page with a single line of code. Let’s take a look at what CSS provides to make this sort of approach possible.

Pixels, points, and picas

CSS supports several absolute length units, the most common of which, and the most basic, is the pixel (px). Less common absolute units are mm (millimeter), cm (centimeter), in. (inch), pt (point—typographic term for 1/72nd of an inch), and pc (pica—typographic term for 12 points). Any of these units can be translated directly to another if you want to work out the math: 1 in. = 25.4 mm = 2.54 cm = 6 pc = 72 pt = 96 px. Therefore, 16 px is the same as 12 pt ($16 / 96 \times 72$). Designers are often more familiar with the use of points, where developers are more accustomed to pixels, so you may have to do some translation between the two when communicating with a designer.

Pixel is a slightly misleading name—a CSS pixel does not strictly equate to a monitor’s pixel. This is notably the case on high-resolution (“retina”) displays. Although the CSS measurements can be scaled a bit, depending on the browser, the operating system, and the hardware, 96 px is usually in the ballpark of 1 physical inch onscreen, though this can vary on certain devices or with a user’s resolution settings.

2.2 Ems and rems

Ems, the most common relative length unit, are a measure used in typography, referring to a specified font size. In CSS, 1 em means the font size of the current element; its exact value varies depending on the element you're applying it to. Figure 2.1 shows a div with 1 em of padding.

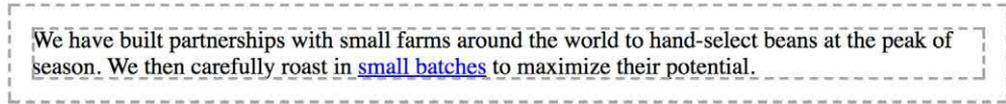


Figure 2.1 Element with 1 em padding (dashed lines added to illustrate padding)

The code to produce this is shown in the next listing. The ruleset specifies a font size of 16 px, which becomes the element's local definition for 1 em. Then the code uses ems to specify the padding of the element. Add this to a new stylesheet, and put some text in a `<div class="padded">` to see it in your browser.

Listing 2.1 Applying ems to padding

```
.padded {
  font-size: 16px;
  padding: 1em;
}
```

← Sets padding on all sides equal to font-size

This padding has a specified value of 1em. This is multiplied by the font size, producing a rendered padding of 16 px. This is important: Values declared using relative units are evaluated by the browser to an absolute value, called the *computed value*.

In this example, editing the padding to 2 em would produce a computed value of 32 px. If another selector targets the same element and overrides it with a different font size, it'll change the local meaning of em, and the computed padding will change to reflect that.

Using ems can be convenient when setting properties like padding, height, width, or border-radius because these will scale evenly with the element if it inherits different font sizes, or if the user changes the font settings.

Figure 2.2 shows two differently sized boxes. The font size, padding, and border radius in each is not the same.



Figure 2.2 Elements with a relatively sized padding and border radius

You can define the styles for these boxes by specifying the padding and border radius using ems. By giving each a padding and border radius of 1 em, you can specify a different font size for each element, and the other properties will scale along with the font.

In your HTML, create two boxes as shown next. Add the `box-small` and `box-large` classes to each, respectively, as size modifiers.

Listing 2.2 Applying ems to different elements (HTML)

```
<span class="box box-small">Small</span>
<span class="box box-large">Large</span>
```

Now, add the styles shown next to your stylesheet. This defines a box using ems. It also defines small and large modifiers, each specifying a different font size.

Listing 2.3 Applying ems applied to different elements (CSS)

```
.box {
  padding: 1em;
  border-radius: 1em;
  background-color: lightgray;
}

.box-small {
  font-size: 12px;
}

.box-large {
  font-size: 18px;
}
```

← Different font sizes,
which will define the
elements' em size

This is a powerful feature of ems. You can define the size of an element and then scale the entire thing up or down with a single declaration that changes the font size. You'll build another example of this in a bit, but first, let's talk about ems and font sizes.

2.2.1 Using ems to define font-size

When it comes to the `font-size` property, ems behave a little differently. As I said, ems are defined by the current element's font size. But, if you declare `font-size: 1.2em`, what does that mean? A font size can't equal 1.2 times itself. Instead, `font-size` ems are derived from the *inherited* font size.

For a basic example, see figure 2.3. This shows two bits of text, each at a different font size. You'll define these using ems in listing 2.4.

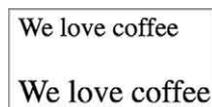


Figure 2.3 Two different font sizes using ems

Change your page to match the following listing. The first line of text is inside the `<body>` tag, so it'll render at the body's font size. The second part, the slogan, inherits that font size.

Listing 2.4 Relative font-size markup

```
<body>
  We love coffee
  <p class="slogan">We love coffee</p>
</body>
```

← The slogan inherits its font size from `<body>`.

The CSS in the next listing specifies the body's font size. I've used pixels here for clarity. Next, you'll use ems to scale up the size of the slogan.

Listing 2.5 Applying ems to font-size

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

.slogan {
  font-size: 1.2em;
}
```

Calculates to 1.2 times the element's inherited font size

The slogan's specified font size is 1.2 em. To determine the calculated pixel value, you'll need to refer to the inherited font size of 16 px: 16 times 1.2 equals 19.2, so the calculated font size is 19.2 px.

TIP If you know the pixel-based font size you'd like, but want to specify the declaration in ems, here's a simple formula: divide the desired pixel size by the parent (inherited) pixel size. For example, if you want a 10 px font and your element is inheriting a 12 px font, $10 / 12 = 0.8333$ em. If you want a 16 px font and the parent font is 12 px, $16 / 12 = 1.3333$ em. We'll do this calculation several times throughout this chapter.

It's helpful to know that, for most browsers, the default font size is 16 px. Technically, it's the keyword value `medium` that calculates to 16 px.

EMS FOR FONT SIZE TOGETHER WITH EMS FOR OTHER PROPERTIES

You've now defined ems for font-size (based on an inherited font size). And, you've defined ems for other properties like padding and border-radius (based on the current element's font size). What makes ems tricky is when you use them for both font size and any other properties on the same element. When you do this, the browser must calculate the font size first, and then it uses that value to calculate the other values. Both properties can have the same declared value, but they'll have different computed values.

In the previous example, we calculated the font size to be 19.2 px (16 px inherited font size times 1.2 em). Figure 2.4 shows the same slogan element, but with an added

padding of 1.2 em and a gray background to make the padding size more apparent. This padding is a bit larger than the font size, even though both have the same declared value.



Figure 2.4 Element with 1.2 em font and 1.2 em padding

What's happening here is the paragraph inherits a font size of 16 px from the body, producing a calculated font size of 19.2 px. This means that 19.2 px is now the local value for an em, and that value is used to calculate the padding. The CSS for this is shown next. Update your stylesheet to see this in your test page.

Listing 2.6 Applying ems to font-size and padding

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

.slogan {
  font-size: 1.2em;
  padding: 1.2em;
  background-color: #ccc;
}
```

Evaluates to
19.2 px
 ←
 Evaluates to
23.04 px
 ←

In this example, padding has a specified value of 1.2 em. This multiplied by 19.2 px (the current element's font size) produces a calculated value of 23.04 px. Even though font-size and padding have the same specified value, their calculated values are different.

THE SHRINKING FONT PROBLEM

Ems can produce unexpected results when you use them to specify the font sizes of multiple nested elements. To know the exact value for each element, you'll need to know its inherited font size, which, if defined on the parent element in ems, requires you to know the parent element's inherited size, and so on up the tree.

This becomes quickly apparent when you use ems for the font size of lists and then nest lists several levels deep. Almost every web developer at some point in their career loads their page to find something resembling figure 2.5. The text is shrinking! This is exactly the sort of problem that leaves developers dreading the use of ems.

- Top level
 - Second level
 - Third level
 - Fourth level
 - Fifth level

Figure 2.5 Nested lists with shrinking text

Shrinking text occurs when you nest lists several levels deep and apply an em-based font size to each level. Listings 2.7 and 2.8 provide an example of this by setting the font size of unordered lists to .8 em. The selector targets every on the page; so when these lists inherit their font size from other lists, the ems compound.

Listing 2.7 Applying ems to a list

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

ul {
  font-size: .8em;
}
```

Listing 2.8 Nested lists

```
<ul>
  <li>Top level
    <ul>
      <li>Second level
        <ul>
          <li>Third level
            <ul>
              <li>Fourth level
                <ul>
                  <li>Fifth level</li>
                </ul>
              </li>
            </ul>
          </li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

This list is nested inside the first one, inheriting its font size . . .

. . . and this one is nested inside of that, inheriting the second list's font size . . .

. . . and so on

Each list has a font size 0.8 times that of its parent. This means the first list has a font size of 12.8 px, but the next one down is 10.24 px (12.8 px × 0.8), and the third level is 8.192 px, and so on. Similarly, if you specified a size larger than 1 em, the text would continually grow instead. What we want is to specify the font at the top level, then maintain the same font size all the way down, as in figure 2.6.

- Top level
 - Second level
 - Third level
 - Fourth level
 - Fifth level

Figure 2.6 Nested lists with corrected text

One way you can accomplish this is with the code in listing 2.9. This sets the font size of the first list to .8 em as before (listing 2.7). The second selector in the listing then targets all unordered lists within an unordered list—all of them except the top level. The nested lists now have a font size equal to their parents, as shown in figure 2.6.

Listing 2.9 Correcting the shrinking text

```
ul {
  font-size: .8em;
}

ul ul {
  font-size: 1em;
}
```

Lists within lists should have the same font size as their parent.

This fixes the problem, though it's not ideal; you're setting a value and then immediately overriding it with another rule. It would be nicer if you could avoid overriding rules by inching up the specificity of the selectors.

By now, it should be clear that ems can get away from you if you're not careful. They're nice for padding, margins, and element sizing, but when it comes to font size, they can get complicated. Thankfully, there is a better option—rems.

2.2.2 Using rems for font-size

When the browser parses an HTML document, it creates a representation in memory of all the elements on the page. This representation is called the *DOM* (Document Object Model). It's a tree structure, where each element is represented by a node. The `<html>` element is the top-level (or root) node. Beneath it are its child nodes, `<head>` and `<body>`. And beneath those are their children, then their children, and so on.

The root node is the ancestor of all other elements in the document. It has a special pseudo-class selector (`:root`) that you can use to target it. This is equivalent to using the type selector `html` with the specificity of a class rather than a tag.

Rem is short for root em. Instead of being relative to the current element, rems are relative to the root element. No matter where you apply it in the document, 1.2 rem has the same computed value: 1.2 times the font size of the root element. The following listing establishes the root font size and then uses rems to define the font size for unordered lists relative to that.

Listing 2.10 Specifying font size using rems

```
:root {
  font-size: 1em;
}

ul {
  font-size: .8rem;
}
```

← The `:root` pseudo-class is equivalent to the `HTML` type selector.

← Uses the browser's default size (16 px)

In this example, the root font size is the browser's default of 16 px (an em on the root element is relative to the browser's default). Unordered lists have a specified font size of .8 rem, which calculates to 12.8 px. Because this is relative to the root, the font size will remain constant, even if you nest lists.

Accessibility: use relative units for font size

Some browsers provide two ways for the user to customize the size of text: zoom and a default font size. By pressing Ctrl-plus (+) or Ctrl-minus (-), the user can zoom the page up or down. This visually scales all fonts and images and generally makes everything on the page larger or smaller. In some browsers, this change is only applied to the current tab and is temporary, meaning it doesn't get carried over to new tabs.

Setting a default font size is a bit different. Not only is it harder to find where to set this (usually in the browser's settings page), but changes at this level remain permanent, until the user returns and changes the value again. The catch is that this setting does *not* resize fonts defined using pixels or other absolute units. Because a default font size is vital to some users, particularly those who are vision-impaired, you should always specify font sizes with relative units or percentages.

Rems simplify a lot of the complexities involved with ems. In fact, they offer a good middle ground between pixels and ems by providing the benefits of relative units, but are easier to work with. Does this mean you should use rems everywhere and abandon the other options? No.

In CSS, again, the answer is often, "it depends." Rems are but one tool in your tool bag. An important part of mastering CSS is learning when to use which tool. My default is to use rems for font sizes, pixels for borders, and ems for most other measures, especially paddings, margins, and border radius (though I favor the use of percentages for container widths when necessary).

This way, font sizes are predictable, but you'll still get the power of ems scaling your padding and margins, should other factors alter the font size of an element. Pixels make sense for borders, particularly when you want a nice fine line. These are my go-to units for the various properties, but again, they're tools, and in some circumstances, a different tool does the job better.

TIP When in doubt, use rems for font size, pixels for borders, and ems for most other properties.

2.3 Stop thinking in pixels

One pattern, or rather, antipattern, that has been common for the past several years is to reset the font size at the page's root to .625 em or 62.5%.

Listing 2.11 Antipattern: globally resetting font-size to 10 px

```
html {  
  font-size: .625em;  
}
```

I don't recommend this. This takes the browser's default font size, 16 px, and scales it down to 10 px. This practice simplifies the math: If your designer tells you to make the font 14 px, you can easily divide by 10 in your head and type 1.4 rem, all while still using relative units.

Initially, this may be convenient, but there are two problems with this approach. First, it forces you to write a lot of duplicate styles. Ten pixels is too small for most text, so you'll have to override it throughout the page. You'll find yourself setting paragraphs to 1.4 rem and asides to 1.4 rem and nav links to 1.4 rem and so on. This introduces more places for error, more points of contact in your code when it needs to change, and increases the size of your stylesheet.

The second problem is that when you do this, you're still thinking in pixels. You might type 1.4 rem into your code, but in your mind, you're still thinking "14 pixels." On a responsive web, you should get comfortable with "fuzzy" values. It doesn't matter how many pixels 1.2 em evaluates to; all you need to know is that it's a bit bigger than the inherited font size. And, if it doesn't look how you want it onscreen, change it. This takes some trial and error, but in reality, so does working with pixels. (In chapter 13, we'll look at additional concrete rules to refine this approach.)

When working with ems, it's easy to get bogged down obsessing over exactly how many pixels things will evaluate to, especially font sizes. You'll drive yourself mad dividing and multiplying em values as you go. Instead, I challenge you to get into the habit of using ems first. If you're accustomed to using pixels, using em values may take practice, but it's worth it.

This isn't to say you'll never have to work with pixels. If you're working with a designer, you'll probably need to talk in some concrete pixel numbers, and that's okay. At the beginning of a project, you'll need to establish a base font size (and often a few common sizes for headings and footnotes). Absolute values are easier to use when discussing the size of things.

Converting to rems will involve arithmetic, so keep a calculator handy. (I press Command-Space on my Mac, and type the equation into Spotlight.) Putting a root font size in place defines a rem. From that point on, working in pixels should be the exception, not the norm.

I'll continue to mention pixels throughout this chapter. This will help me reiterate why the relative units behave the way they do, as well as help you get accustomed to the calculation of ems. After this chapter, I'll primarily discuss font sizes using relative units.

2.3.1 Setting a sane default font size

Let's say you want your default font size to be 14 px. Instead of setting a 10 px default then overriding it throughout the page, set that value at the root. The desired value divided by the inherited value—in this case, the browser's default—is 14/16, which equals 0.875.

Add the following listing to the top of a new stylesheet, as you'll be building on it. This sets the default font at the root (<html>).

Listing 2.12 Setting the true default font size

```
:root {
  font-size: 0.875em;
}
```

Or use the HTML selector
14/16 (desired px / inherited px) equals .875

Now your desired font size is applied to the whole page. You won't need to specify it elsewhere. You'll only need to change it in places where the design deviates from this, such as headings.

Let's create the panel shown in figure 2.7. You'll build this panel based on the 14 px font size, using relative measurements.

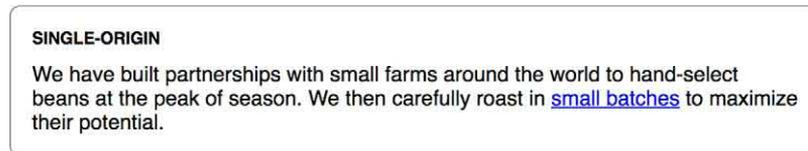


Figure 2.7 Panel with relative units and an inherited font size

The markup for this is shown here. Add this to your page.

Listing 2.13 Markup for a panel

```
<div class="panel">
  <h2>Single-origin</h2>
  <div class="panel-body">
    We have built partnerships with small farms around the world to
    hand-select beans at the peak of season. We then carefully roast
    in <a href="/batch-size">small batches</a> to maximize their
    potential.
  </div>
</div>
```

The next listing shows the styles. You'll use ems for the padding and border radius, rem for the font size of the heading, and px for the border. Add these to your stylesheet.

Listing 2.14 Panel with relative units

```

.panel {
  padding: 1em;
  border-radius: 0.5em;
  border: 1px solid #999;
}

.panel > h2 {
  margin-top: 0;
  font-size: 0.8rem;
  font-weight: bold;
  text-transform: uppercase;
}

```

Uses ems for padding and border radius

Uses 1 px for a thin border

Removes extra space from the panel top; more on this in chapter 3

Styles the heading font using rems for font size

This code puts a thin border around the panel and styles the heading. I opted for a header that is smaller, but bold and all caps. (You can make this larger or a different typeface if your design calls for it.)

The `>` in the second selector is a *direct descendant combinator*. It targets an `h2` that's a child element of a `.panel` element. See appendix A for a complete reference of selectors and combinators.

In listing 2.13, I added a `panel-body` class to the main body of the panel for clarity, but you'll notice you didn't need to use it in your CSS. Because this element already inherits the root font size, it already appears how you want it to look.

2.3.2 Making the panel responsive

Let's take this a bit further. You can use some *media queries* to change the base font size, depending on the screen size. This'll make the panel render at different sizes based on the size of the user's screen (shown in figure 2.8).



media query—An `@media` rule used to specify styles that will be applied only to certain screen sizes or media types (for example, print or screen). This is a key component of responsive design. See listing 2.15 for an example; I'll cover this in greater depth in chapter 8.

SINGLE-ORIGIN

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in [small batches](#) to maximize their potential.

SINGLE-ORIGIN

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in [small batches](#) to maximize their potential.

SINGLE-ORIGIN

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in [small batches](#) to maximize their potential.

Figure 2.8 Responsive panel on different screen sizes: 300 px (top left), 800 px (top right), and 1,440 px (bottom)

To see this result, edit this portion of your stylesheet to match this listing.

Listing 2.15 Responsive base font-size

<pre>:root { font-size: 0.75em; }</pre>	Applies to all screens, but is overridden for larger screens
<pre>@media (min-width: 800px) { :root { font-size: 0.875em; } }</pre>	Applies only to screens 800 px and wider, overriding the original value
<pre>@media (min-width: 1200px) { :root { font-size: 1em; } }</pre>	Applies only to screens 1,200 px and larger, overriding both values

This first ruleset specifies a small default font size. This is the font size that we want to apply on smaller screens. Then you used media queries to override that value with incrementally larger font sizes on screens with a width of 800 px and 1,200 px or more.

By applying these font sizes at the root on your page, you've responsively redefined the meaning of em and rem throughout the entire page. This means that the panel is now responsive, even though you made no changes to it directly. On a small screen, such as a smartphone, the font will be rendered smaller (12 px); likewise, the padding and border radius will be smaller to match. And, on larger screens more than 800 px and 1,200 px wide, the component scales up to a 14 px and 16 px font size, respectively. Resize your browser window to watch these changes take place.

If you are disciplined enough to style your entire page in relative units like this, the entire page will scale up and down based on the viewport size. This can be a huge part of your responsive strategy. These two media queries near the top of your stylesheet can eliminate the need for dozens of media queries throughout the rest of your CSS. But it doesn't work if you define your values in pixels.

Similarly, if your boss or your client decides the fonts on the site you built are too small or too large, you can change them globally by only touching one line of code. The change will ripple throughout the rest of your page, effortlessly.

2.3.3 Resizing a single component

You can also use ems to scale an individual component on the page. Sometimes you might need a larger version of the same part of your interface on certain parts of the page. Let's do this with our panel. You'll add a large class to the panel: `<div class="panel large">`.

Figure 2.9 shows both the normal and the large panel for comparison. The effect is similar to the responsive panels, but both sizes can be used simultaneously on the same page.



Figure 2.9 A normal panel and a large panel defined on the same page

Let's make a small change to the way you defined the panel's font sizes. You'll still use relative units, but you'll adjust what they're relative to. First, add the declaration `font-size: 1rem` to the parent element of each panel. This means each panel will establish a predictable font size for itself, no matter where it's placed on the page.

Second, redefine the heading's font size using ems rather than rems to make it relative to the parent's font size you just established at 1 rem. The code for this is next. Update your stylesheet to match.

Listing 2.16 Creating a larger version of the panel

```
.panel {
  font-size: 1rem;
  padding: 1em;
  border: 1px solid #999;
  border-radius: 0.5em;
}

.panel > h2 {
  margin-top: 0;
  font-size: 0.8em;
  font-weight: bold;
  text-transform: uppercase;
}
```

← Establishes a predictable font size for the component

← Uses ems to make other fonts relative to the established parent font size

This change has no effect on the appearance of the panel, but now it sets you up to make the larger version of the panel with a single line of CSS. All you have to do is override the parent element's 1 rem with another value. Because all the component's measurements are relative to this, overriding it will resize the entire panel. Add the CSS in the next listing to your stylesheet to define a larger version.

Listing 2.17 Scaling the entire panel with one declaration

```
.panel.large {
  font-size: 1.2rem;
}
```

← Compound selector targets elements with both panel and large classes

Now, you can use `class="panel"` for a normal panel and `class="panel large"` for a larger one. Similarly, you could define a smaller version of the panel by setting a smaller font size. If the panel were a more complicated component, with multiple font sizes or paddings, it'd still only take this one declaration to resize it, as long as everything inside is defined using ems.

2.4 Viewport-relative units

You've learned that ems and rems are defined relative to `font-size`, but these aren't the only type of relative units. There are also *viewport-relative units* for defining lengths relative to the browser's viewport.



viewport—The framed area in the browser window where the web page is visible. This excludes the browser's address bar, toolbars, and status bar, if present.

If you're not familiar with viewport-relative units, here is a brief explanation.

- *vh*—1/100th of the viewport height
- *vw*—1/100th of the viewport width
- *vmin*—1/100th of the smaller dimension, height or width (IE9 supports `vm` instead of `vmin`)
- *vmax*—1/100th of the larger dimension, height or width (not supported in IE or, at the time of writing, Edge)

For example, 50 *vw* is equal to half the width of the viewport, and 25 *vh* equals 25% of the viewport's height. *vmin* is based on which of the two (height or width) is smaller. This is helpful for ensuring that an element will fit on the screen regardless of its orientation: If the screen is landscape, it'll be based on the height; if portrait, it's based on the width.

Figure 2.10 shows a square element as it appears in several viewports with different screen sizes. It's defined with both a height and a width of 90 *vmin*, which equals 90% of the smaller of the two dimensions—90% of the height on landscape screens, or 90% of the width on portrait.

Listing 2.18 shows the styles for this element. It produces a large square that always fits in the viewport no matter how the browser is sized. You can add a `<div class="square">` to your page to see this.

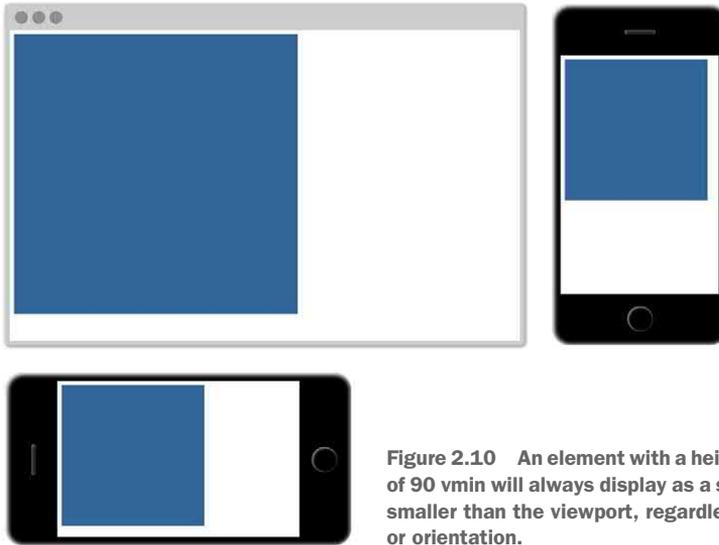


Figure 2.10 An element with a height and width of 90 vmin will always display as a square a little smaller than the viewport, regardless of its size or orientation.

Listing 2.18 Square element sized using vmin

```
.square {  
  width: 90vmin;  
  height: 90vmin;  
  background-color: #369;  
}
```

The viewport-relative lengths are great for things like making a large hero image fill the screen. Your image can be inside a long container, but setting the image height to 100 vh, makes it exactly the height of the viewport.

NOTE Viewport-relative units are a newer feature for most browsers, so there are a few odd bugs when you use them in more exotic combinations with other styles. See “Known Issues” at <http://caniuse.com/#feat=viewport-units> for a list.

CSS3

Several of the unit types in this chapter weren’t in earlier versions of CSS (rems and viewport-relative units, in particular). They were added amid a series of changes to the language, which is often called CSS3.

In the late 1990s and early 2000s, after initial work on the CSS specification, little changed for a long time. The W3C (World Wide Web Consortium) published the CSS

2 specification in May 1998. Shortly thereafter, work began on version 2.1 to correct issues and bugs in version 2. Work on CSS 2.1 continued for many years, with few significant additions to the language. It was not finalized as a Proposed Recommendation until April 2011. By this point, browsers had already implemented most of the CSS 2.1 changes, and were well on their way to adding newer features under the moniker CSS3.

The “3” is an informal version number; there’s no CSS3 specification. Instead, the specification was broken up into individual modules, each independently versioned. The specification for backgrounds and borders is now separate from the one for box models, and from the one for cascading and inheritance. This allows the W3C to make new revisions to one area of CSS without unnecessarily updating areas that are not changing. Many of these specifications remain at version 3 (now called *level 3*), but some, such as the selectors specification, are at level 4 and others, such as a flexbox, are at level 1.

As these changes were introduced, we saw an explosion of new features rolling out in browsers from 2009 through 2013. Notable additions at this time included rems and viewport-relative units, as well as new selectors, media queries, web fonts, rounded borders, animations, transitions, transformations, and different ways to specify colors. And, new features are steadily emerging each year.

This means we’re no longer working with one particular version of CSS. It’s a living standard. Each browser is continually adding support for new features. Developers work with those changes and adapt to them. There won’t be a CSS4, except perhaps as a more generic marketing term. Although this book covers CSS3 features, I don’t necessarily call them out as such because, as far as the web is concerned, it’s all CSS.

2.4.1 Using vw for font size

One application for viewport-relative units that may not be immediately obvious is font size. In fact, I find this use more practical than applying vh and vw to element heights or widths.

Consider what would happen if you applied `font-size: 2vw` to an element. On a desktop monitor at 1,200 px, this evaluates to 24 px (2% of 1,200). On a tablet with a screen width of 768 px, it evaluates to about 15 px (2% of 768). And, the nice thing is, the element scales smoothly between the two sizes. This means there’re no sudden breakpoint changes; it transitions incrementally as the viewport size changes.

Unfortunately, 24 px is a bit too large on a big screen. And worse, it scales all the way down to 7.5 px on an iPhone 6. What would be nice is this scaling effect, but with the extremes a little less severe. You can achieve this with CSS’s `calc()` function.

2.4.2 Using calc() for font size

The `calc()` function lets you do basic arithmetic with two or more values. This is particularly useful for combining values that are measured in different units. This function supports addition (+), subtraction (-), multiplication (*) and division (/). The

addition and subtraction operators must be surrounded by whitespace, so I suggest getting in the habit of always adding a space before and after each operator; for example, `calc(1em + 10px)`.

You'll use `calc()` in the next listing to combine ems with vw units. Remove the previous base font size (and the related media queries) from your stylesheet. Add this in its place.

Listing 2.19 Using `calc()` to define `font-size` in ems and vw units

```
:root {  
  font-size: calc(0.5em + 1vw);  
}
```

Now, open the page and slowly resize your browser. You'll see the font scale smoothly as you do. The 0.5 em here operates as a sort of minimum font size, and the 1 vw adds a responsive scalar. This'll give you a base font size that scales from 11.75 px on an iPhone 6 up to 20 px in a 1,200 px browser window. You can adjust these values to your liking.

You've now accomplished a large piece of your responsive strategy without a single media query. Instead of three or four hard-coded breakpoints, everything on your page will scale fluidly according to the viewport.

2.5 Unitless numbers and line-height

Some properties allow for *unitless* values (that is, a number with no specified unit). Properties that support this include `line-height`, `z-index`, and `font-weight` (700 is equivalent to bold; 400 is equivalent to normal, and so on). You can also use the unitless value 0 anywhere a length unit (such as px, em, or rem) is required because, in these cases, the unit does not matter—0 px equals 0% equals 0 em.

WARNING A unitless 0 can only be used for *length* values and percentages, such as in paddings, borders, and widths. It can't be used for angular values, such as degrees or time-based values like seconds.

The `line-height` property is unusual in that it accepts both units and unitless values. You should typically use unitless numbers because they're inherited differently. Let's put text into the page and see how this behaves. Add the code in the following listing to your stylesheet.

Listing 2.20 Inherited `line-height` markup

```
<body>  
  <p class="about-us">  
    We have built partnerships with small farms around the world to  
    hand-select beans at the peak of season. We then carefully roast in  
    small batches to maximize their potential.  
  </p>  
</body>
```

You'll specify a line height for the body element and allow it to be inherited by the rest of the document. This will work as expected, no matter what you do to the font sizes in the page (figure 2.11).

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in small batches to maximize their potential.

Figure 2.11 Unitless line height is recalculated for each descendant element.

Add listing 2.21 to your stylesheet for these styles. The paragraph inherits a line height of 1.2. Because the font size is 32 px (2 em × 16 px, the browser's default), the line height is calculated locally to 38.4 px (32 px × 1.2). This will leave an appropriate amount of space between lines of text.

Listing 2.21 Line height with a unitless number

```
body {
  line-height: 1.2;
}

.about-us {
  font-size: 2em;
}
```

← Descendant elements inherit the unitless value.

If instead you specify the line height using a unit, you may encounter unexpected results, like that shown in figure 2.12. The lines of text overlap one another. Listing 2.22 shows the CSS that generated the overlap.

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in small batches to maximize their potential.

Figure 2.12 Overlapping lines due to an inherited line-height

Listing 2.22 Line height with units results in unexpected output

```
body {
  line-height: 1.2em;
}

.about-us {
  font-size: 2em;
}
```

← Descendant elements inherit the calculated value (19.2 px).

← Evaluates to 32 px

These results are due to a peculiar quirk of inheritance: when an element has a value defined using a *length* (px, em, rem, and so forth), its computed value is inherited by child elements. When units such as ems are specified for a line height, their value is calculated, and that calculated value is passed down to any inheriting children. With the `line-height` property, this can cause unexpected results if the child element has a different font size, like the overlapping text.



length—The formal name for a CSS value that denotes a distance measurement. It’s a number followed by a unit, such as 5 px. Length comes in two flavors: absolute and relative. Percentages are similar to lengths, but strictly speaking, they’re not considered lengths.

When you use a unitless number, that declared value is inherited, meaning its computed value is recalculated for each inheriting child element. This will almost always be the result you want. Using a unitless number lets you set the line height on the body and then forget about it for the rest of the page, unless there are particular places where you want to make an exception.

2.6 *Custom properties (aka CSS variables)*

In 2015, a long-awaited CSS specification titled *Custom Properties for Cascading Variables* was published as a Candidate Recommendation. This specification introduced the concept of variables to the language, which enabled a new level of dynamic, context-based styles. You can declare a variable and assign it a value; then you can reference this value throughout your stylesheet. You can use this to reduce repetition in your stylesheet, as well as some other beneficial applications as you’ll see shortly.

At the time of writing, support for custom properties has rolled out in all major browsers except IE. For up-to-date support information on lesser-known browsers, check “Can I Use” at <http://caniuse.com/#feat=css-variables>.

NOTE If you happen to use a CSS preprocessor that supports its own variables, such as Sass (syntactically awesome stylesheets) or Less, you may be tempted to disregard CSS variables. Don’t. The new CSS variables are different in nature and are far more versatile than anything a preprocessor can accomplish. I tend to refer to them as “custom properties” rather than variables to emphasize this distinction.

To define a custom property, you declare it much like any other CSS property. Listing 2.23 is an example of a variable declaration. Start a fresh page and stylesheet, and add this CSS.

Listing 2.23 Defining a custom property

```
:root {
  --main-font: Helvetica, Arial, sans-serif;
}
```

This listing defines a variable named `--main-font`, and sets its value to a set of common sans-serif fonts. The name must begin with two hyphens (`--`) to distinguish it from CSS properties, followed by whatever name you’d like to use.

Variables must be declared inside a declaration block. I’ve used the `:root` selector here, which sets the variable for the whole page—I’ll explain this shortly.

By itself, this variable declaration doesn’t do anything until we use it. Let’s apply it to a paragraph to produce a result like that in figure 2.13.

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in small batches to maximize their potential.

Figure 2.13 Simple paragraph using a variable’s sans-serif font

A function called `var()` allows the use of variables. You’ll use this function to reference the `--main-font` variable just defined. Add the ruleset shown in the following listing to put the variable to use.

Listing 2.24 Using a custom property

```
:root {
  --main-font: Helvetica, Arial, sans-serif;
}

p {
  font-family: var(--main-font);
}
```

Sets the font family for paragraphs to Helvetica, Arial, sans-serif

Custom properties let you define a value in one place, as a “single source of truth,” and reuse that value throughout the stylesheet. This is particularly useful for recurring values like colors. The next listing adds a `brand-color` custom property. You can use this variable dozens of times throughout your stylesheet, but if you want to change it, you only have to edit it in one place.

Listing 2.25 Using custom properties for colors

```
:root {
  --main-font: Helvetica, Arial, sans-serif;
  --brand-color: #369;
}
```

Defines a blue brand-color variable

```
p {
  font-family: var(--main-font);
  color: var(--brand-color);
}
```

The `var()` function accepts a second parameter, which specifies a fallback value. If the variable specified in the first parameter is not defined, then the second value is used instead.

Listing 2.26 Providing fallback values

```
:root {
  --main-font: Helvetica, Arial, sans-serif;
  --brand-color: #369;
}

p {
  font-family: var(--main-font, sans-serif);
  color: var(--secondary-color, blue);
}
```

This listing specifies fallback values in two different declarations. In the first, `--main-font` is defined as `Helvetica, Arial, sans-serif`, so this value is used. In the second, `--secondary-color` is an undefined variable, so the fallback value `blue` is used.

NOTE If a `var()` function evaluates to an invalid value, the property will be set to its initial value. For example, if the variable in `padding: var(--brand-color)` evaluates to a color, it would be an invalid padding value. In that case, the padding would be set to `0` instead.

2.6.1 Changing custom properties dynamically

In the examples so far, custom properties are merely a nice convenience; they can save you from a lot of repetition in your code. But what makes them particularly interesting is that the declarations of custom properties cascade and inherit: You can define the same variable inside multiple selectors, and the variable will have a different value for various parts of the page.

You can define a variable as `black`, for example, and then redefine it as `white` inside a particular container. Then, any styles based on that variable will dynamically resolve to `black` if they are outside the container and to `white` if inside. Let's use this to achieve a result like that shown in figure 2.14.

This panel is similar to the one you saw earlier (figure 2.7). The HTML for this is shown in listing 2.27. It has two instances of the panel: one inside the body and one inside a dark section. Update your HTML to match this.

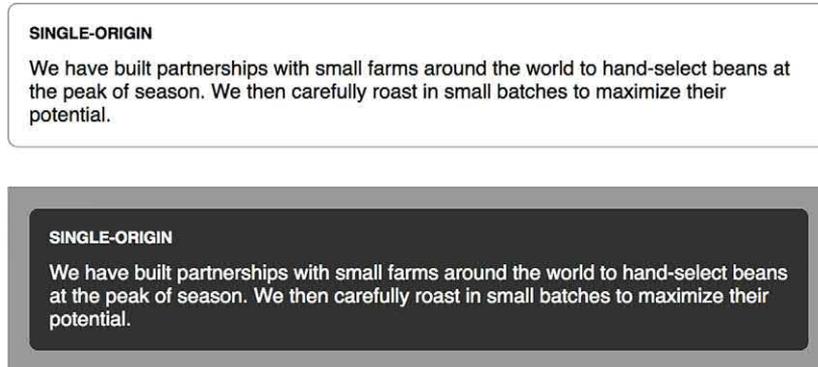


Figure 2.14 Custom properties produce different colored panels based on local variable values.

Listing 2.27 Two panels in different contexts on the page

```

<body>
  <div class="panel">
    <h2>Single-origin</h2>
    <div class="body">
      We have built partnerships with small farms
      around the world to hand-select beans at the
      peak of season. We then careful roast in
      small batches to maximize their potential.
    </div>
  </div>

  <aside class="dark">
    <div class="panel">
      <h2>Single-origin</h2>
      <div class="body">
        We have built partnerships with small farms
        around the world to hand-select beans at the
        peak of season. We then careful roast in
        small batches to maximize their potential.
      </div>
    </div>
  </aside>
</body>

```

← A regular panel on the page

The second panel inside a dark container

Let's redefine the panel to use variables for text and background color. Add the next listing to your stylesheet. This sets the background color to white and the text to black. I'll explain how this works before you add styles for the dark variant.

Listing 2.28 Using variables to define the panel colors

```

:root {
  --main-bg: #fff;
  --main-color: #000;
}

.panel {
  font-size: 1rem;
  padding: 1em;
  border: 1px solid #999;
  border-radius: 0.5em;
  background-color: var(--main-bg);
  color: var(--main-color);
}

.panel > h2 {
  margin-top: 0;
  font-size: 0.8em;
  font-weight: bold;
  text-transform: uppercase;
}

```

Defines background and text color variables as white and black, respectively

Uses the variables in the panel's styles

Again, you've defined the variables inside a ruleset with the `:root` selector. This is significant because it means these values are set for everything in the root element (the entire page). When a descendant element of the root uses the variables, these are the values they'll resolve to.

You have two panels, but they still look the same. Now, let's define the variables again, but this time with a different selector. The next listing provides styles for the dark container. It sets a dark gray background on the container, as well as a little padding and margin. It also redefines both variables. Add this to your stylesheet.

Listing 2.29 Styling the dark container

```

.dark {
  margin-top: 2em;
  padding: 1em;
  background-color: #999;
  --main-bg: #333;
  --main-color: #fff;
}

```

Redefines the `--main-bg` and `--main-color` variables within the scope of the container

Puts a margin between the dark container and the preceding panel

Applies a dark gray background to the dark container

Reload the page, and the second panel will have a dark background and white text. This is because when the panel uses these variables, they'll resolve to the values defined on the dark container, rather than on the root. Notice you didn't have to restyle the panel, or apply any additional classes.

In this example, you’ve defined custom properties twice: first on the root (where `--main-color` is black), and then on the dark container (where `--main-color` is white). The custom properties behave as a sort of scoped variable because the values are inherited by descendant elements. Inside the dark container, `--main-color` is white; elsewhere on the page, it’s black.

2.6.2 Changing custom properties with JavaScript

Custom properties can also be accessed and manipulated live in the browser using JavaScript. Because this isn’t a book on JavaScript, I’ll show you enough to get acquainted with the concept. I’ll leave it to you to integrate this into your JavaScript projects.

The following listing shows how to access a property on an element. It adds a script to the page, which logs the value of the root element’s `--main-bg` property.

Listing 2.30 Accessing a custom property in JavaScript

```
<script type="text/javascript">
  var rootElement = document.documentElement;
  var styles = getComputedStyle(rootElement);
  var mainColor = styles.getPropertyValue('--main-bg');
  console.log(String(mainColor).trim());
</script>
```

Gets the styles object for an element

Gets the `--main-bg` value from the styles object

Ensures `mainColor` is a String and trims whitespace; logs “#fff”

Because you can specify new values for custom properties on the fly, you can use JavaScript to set a new value for `--main-bg` dynamically. If you set it to a light blue, it’ll appear as shown in figure 2.15.

SINGLE-ORIGIN

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in small batches to maximize their potential.

Figure 2.15 JavaScript can set the panel’s background by changing the `--main-bg` variable.

The code in the next listing sets a new value to `--main-bg` on the root element. Add this at the end of the `<script>` tag.

Listing 2.31 Setting a custom property in JavaScript

```
var rootElement = document.documentElement;
rootElement.style.setProperty('--main-bg', '#cdf');
```

Sets `--main-bg` to a light blue on the root element

If you run this script, any elements inheriting the `--main-bg` property will update to use this new value. On your page, this changes the background of the first panel to light blue. The second panel remains unchanged, as it's still inheriting the property from the dark container.

With this technique, you can use JavaScript to re-theme your site, live in the browser. Or, you could highlight certain parts of the page or make any number of other on-the-fly changes. Using only a few lines of JavaScript, you can make changes that'll affect a large number of elements on the page.

2.6.3 *Experimenting with custom properties*

Custom properties are a whole new area of CSS that developers are just beginning to explore. Because browser support has been limited, it hasn't yet seen much "prime-time" use. I'm sure that over time, you'll see best practices and novel uses emerge. This is something to keep your eye on. Experiment with custom properties and see what you can come up with.

Be aware that any declaration using `var()` will be ignored by old browsers that don't understand it. Provide a fallback behavior for those browsers when possible:

```
color: black;
color: var(--main-color);
```

This will not always be possible, however, given the dynamic nature of custom properties. Keep an eye on browser support at <http://caniuse.com>.

Summary

- Embrace the use of relative units, allowing the page's structure to determine the meaning of your styles.
- Favor the use of rems for font size, but selectively use ems for simple scaling of components on the page.
- You can make your entire page scale responsively without any media queries.
- Use unitless values when specifying line height.
- You can start getting familiar with one of CSS's newest features, custom properties.

Mastering the box model

This chapter covers

- Practical advice for element sizing
- Vertical centering
- Columns of equal height
- Negative margins and margin collapsing
- Consistent spacing of components on the page

When it comes to laying out elements on the page, you'll find a lot of things going on. On a complex site, you may have floats, absolutely positioned elements, and other elements of various sizes. You may also have some layouts using newer CSS constructs, such as a flexbox or a grid layout. You have a lot of things to keep track of, and learning everything involved with layout can be overwhelming.

We'll spend several chapters taking a close look at several layout techniques. Before we get to those, it's important to have a solid grasp on the fundamentals of how the browser sizes and positions elements. The more advanced topics of layout are built atop concepts like document flow and the box model; these are the basic rules that determine the position and size of elements on the page.

In this chapter, you'll build a two-column page layout. You may be familiar with this as a classic beginner exercise for CSS, but I'll guide you through it in a way that

highlights several, often-overlooked nuances of layout. We'll look at some of the edge cases of the box model, and I'll give you practical advice for sizing and aligning elements. We'll also tackle two of the most notorious problems in CSS: vertical centering and equal-height columns.

3.1 *Difficulties with element width*

In this chapter, you'll build a simple page with a header at the top and two columns beneath. By the end of the chapter, your page will look like the one shown in figure 3.1. I've intentionally made the page design a bit “blocky,” so you can readily see the size and position of all the elements.

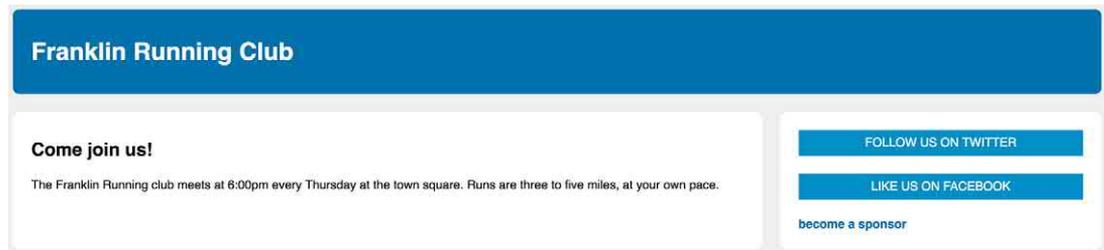


Figure 3.1 Two-column page layout with header

Start a new page and an empty stylesheet, and then link them. Add the markup shown next to your page. Your page will have a header, as well as a main element and a sidebar element that will make up the two columns on your page. A container wraps the two columns.

Listing 3.1 HTML for page with a two-column layout

```
<body>
  <header>
    <h1>Franklin Running Club</h1>
  </header>
  <div class="container">
    <main class="main">
      <h2>Come join us!</h2>
      <p>
        The Franklin Running club meets at 6:00pm every Thursday
        at the town square. Runs are three to five miles, at your
        own pace.
      </p>
    </main>
    <aside class="sidebar">
      <div class="widget"></div>
      <div class="widget"></div>
    </aside>
  </div>
</body>
```

Let's begin with some of the obvious styles. You'll set the font for the page, then background colors for the page and each of the main containers. This will help you see the position and size of each as you go. After you do this, your page will look like the one shown in figure 3.2.



Figure 3.2 Three main containers with background colors

For some site designs, the background color of several containers might be transparent. When this is the case, it might be helpful to temporarily apply a background color to the container until you get it sized and positioned accordingly.

The styles for this are shown in listing 3.2. Currently, the sidebar is empty so, by default, it has no height. You'll add padding to give it some height. The other containers will need padding eventually, but we'll come back to that. For now, add this code to your stylesheet.

Listing 3.2 Applying font and colors

```
body {
  background-color: #eee;
  font-family: Helvetica, Arial, sans-serif;
}

header {
  color: #fff;
  background-color: #0072b0;
  border-radius: .5em;
}

main {
  display: block;
}

.main {
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  padding: 1.5em;
  background-color: #fff;
}
```

Fixes IE bug

Adds padding to the sidebar

```
border-radius: .5em;
}
```

NOTE IE has a bug where `<main>` elements are rendered inline by default, rather than as blocks. We corrected that here by adding a `display: block` declaration.

Next, let's put your two columns in place. To begin, you'll use a float-based layout. You'll float the main and the sidebar to the left and give them widths of 70% and 30%, respectively. Update your stylesheet to match the CSS shown here.

Listing 3.3 Aligning two columns

```
.main {
  float: left;
  width: 70%;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  float: left;
  width: 30%;
  padding: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

← Floats the main column left and sets the width at 70%

← Floats the sidebar left and sets the width at 30%

You can see the result in figure 3.3, but it's not quite what you wanted.

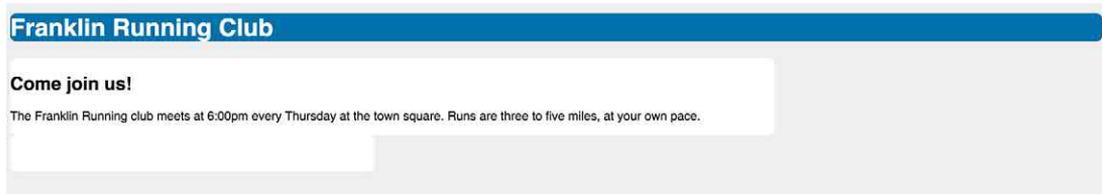


Figure 3.3 Main and sidebar columns with widths of 70% and 30%, respectively

Instead of the two columns sitting side by side, they line wrapped. Even though you specified widths of 70% and 30%, the columns took up more than 100% of the available space. That's because of the default behavior of the box model (figure 3.4). When you set the width or height of an element, you're specifying the width or height of its content; any padding, border, and margins are then added to that width.

This behavior means that an element with a 300 px width, a 10 px padding, and a 1 px border has a rendered width of 322 px (width plus left and right padding plus left and right border). This gets even more confusing when the units aren't all the same.

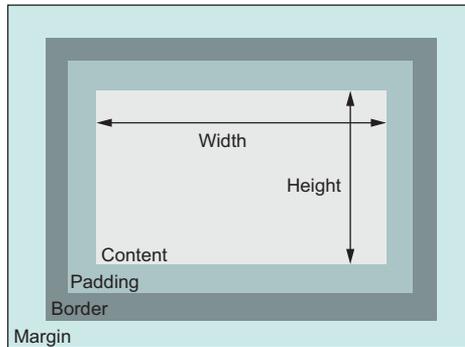


Figure 3.4 The default box model

In the example (listing 3.3), the sidebar has a width of 30% plus 1.5 em left and right padding; the main container has only a width of 70%. This brings the total of the two columns to 100% plus 3 ems. In order to fit, the containers have to wrap.

3.1.1 Avoiding magic numbers

The naive fix is to reduce the width of one of the columns (the sidebar, for example). On my screen, a width of 26% for the sidebar works, but this is unreliable. The 26% is known as a *magic number*. Instead of using a desired value, I found it by making haphazard changes to my styles until I got the result I wanted.

For programming in general, magic numbers aren't desirable. It's often hard to explain why a magic number works. If you don't understand where the number comes from, you won't understand how it will behave under different circumstances. My screen is 1440 px wide, so on smaller viewports, the sidebar will still line-wrap. Although there's a place for trial and error in CSS, typically that's for choices that are stylistic in nature and not for forcing things to fit into position.

One alternative to this magic number is to let the browser do the math. In this case, the columns are 3 em too wide (due to padding), so you can use the `calc()` function to reduce the width by exactly that much. A sidebar width of `calc(30% - 3em)` gives you exactly what you need. But, there's still a better way.

3.1.2 Adjusting the box model

Because of the problems you've just encountered, the default box model isn't what you'll typically want to use. Instead, you'll want your specified widths to include the padding and borders. CSS allows you to adjust the box model behavior with its `box-sizing` property.

By default, `box-sizing` is set to the value of `content-box`. This means that any height or width you specify only sets the size of the content box. You can assign a value of `border-box` to the box sizing instead. That way, the height and width properties set the combined size of the content, padding, and border, which is exactly what you want in this example.

Figure 3.5 shows the box model with box sizing set to border-box. With this model, padding doesn't make an element wider; it makes the inner content narrower. It also does the same for height.

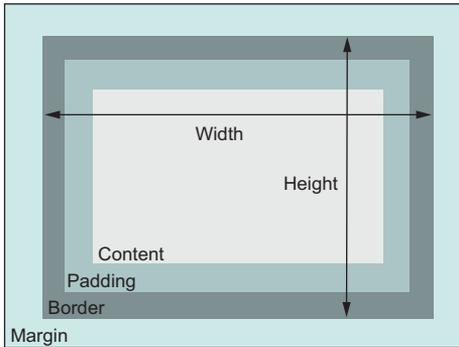


Figure 3.5 The box model with box sizing set to border-box

If you update these elements to use border box sizing, they'll fit on the same line, regardless of the left and right padding (figure 3.6).



Figure 3.6 Columns with an adjusted box model now fit side by side.

To adjust the box model for the two elements, main and sidebar, update your stylesheet to match this listing.

Listing 3.4 Floating columns with a corrected box model

```
.main {
  box-sizing: border-box;
  float: left;
  width: 70%;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  box-sizing: border-box;
  float: left;
  width: 30%;
  padding: 1.5em;
}
```

←
←
Changes the box model to border-box sizing

```
background-color: #fff;
border-radius: .5em;
}
```

Using `box-sizing: border-box`, the two elements add up to an even 100% width. Their widths of 70% and 30% are now inclusive of their padding, so they fit on the same line.

3.1.3 Using universal border-box sizing

You have made box sizing more intuitive for these two elements, but you'll surely run into other elements with the same problem. It would be nice to fix it once, universally for all elements, so you won't have to think about this adjustment again. You can do this with the universal selector (`*`), which targets all elements on the page as the following listing shows. I've added selectors to target every pseudo-element on the page as well. Put this code at the top of your stylesheet.

Listing 3.5 Universal border-box fix

```
*,
::before,
::after {
  box-sizing: border-box;
}
```

← Applies border box sizing to all elements and pseudo-elements on the page

After applying this to the page, height and width will always specify the actual height and width of an element. Padding won't change them.

NOTE Adding this snippet near the beginning of your stylesheet has become common practice.

If, however, you add third-party components with their own CSS to your page, you may see some broken layouts for those components, especially if their CSS wasn't written with this fix in mind. Because the universal border-box fix targets every element in the component with the universal selector, correcting this can be problematic. You would need to target every element inside the component to revert to the content-box sizing.

You can make this easier with a slightly modified version of the fix and inheritance. Update this portion of your stylesheet to match the following listing.

Listing 3.6 More robust universal border-box fix

```
:root {
  box-sizing: border-box;
}
```

← Applies border box sizing to the root element

```
*,
::before,
::after {
```

```

    box-sizing: inherit;
  }

```

← Tells all other elements and pseudo-elements to inherit their box sizing

Box sizing isn't normally an inherited property, but by using the `inherit` keyword, you can force it to be. With the version shown here, you can convert a third-party component into a `content-box` when necessary by targeting its top-level container. Then all elements inside the component will inherit the box sizing:

```

.third-party-component {
  box-sizing: content-box;
}

```

Now, every element on your site will have a more predictable box model. I recommend that you add listing 3.6 to your CSS every time you start a new site; it'll save you a lot of trouble in the long run. It can be a little problematic in an existing stylesheet, however, especially if you've already written lots of styles based on the default `content-box` model. If you do add this to an existing project, be sure to give it a thorough review for any resulting bugs.

NOTE From this point on, every example in this book will assume that this `border-box` fix is at the beginning of your stylesheet.

3.1.4 Adding a gutter between columns

It's often more visually appealing to have a small gap (or *gutter*) between columns. You can sometimes achieve this by adding padding to one column; but in some cases, this approach doesn't work. If both columns have a background color or border, as with your example page, you'll want the gutter to appear between the two elements' borders (figure 3.7). Note the gray space between the two white backgrounds. You can achieve this look in a handful of ways. Let's look at a couple of them, shown in listings 3.7 and 3.8.

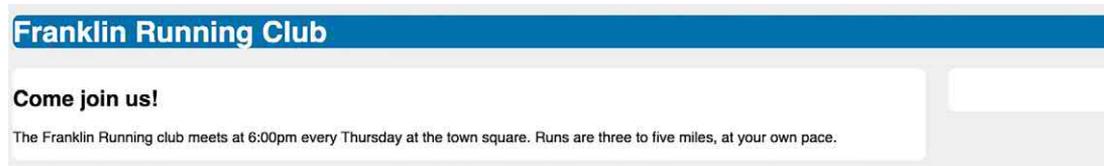


Figure 3.7 Gutter added between the two columns

First, you can add a margin to one of the columns and adjust the widths of your elements to account for the added space. Listing 3.7 shows how to subtract 1% from the sidebar column width and move it to the margin. Update your CSS to match.

Listing 3.7 Percent-based gutter margin

```
.main {
  float: left;
  width: 70%;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  float: left;
  width: 29%;
  margin-left: 1%;
  padding: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

Subtracts 1% from the width ...

... and adds it as a margin for the gutter

This adds a gutter, but its width is based on the outer container’s width—the percentage is relative to the full width of the parent. What if you want to specify the gutter in units other than a percentage? (I prefer an em-based gutter, which I find more consistent.) You can accomplish this with `calc()`.

Instead of moving 1% from the width into the margin, you can move 1.5 em. This listing shows how `calc()` makes this possible. Change your CSS again to match this listing.

Listing 3.8 Using `calc()` to subtract the gutter from the width

```
.main {
  float: left;
  width: 70%;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  float: left;
  width: calc(30% - 1.5em);
  margin-left: 1.5em;
  padding: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

Subtracts 1.5 em from the width ...

... and adds it as a margin

Not only does this allow you to use ems rather than percentages for the gutter, but it also has the benefit of being a little more explicit in code. When reviewing the code later, it might not be apparent where a specific percentage comes from, but `30% - 1.5em` provides a clue that you’re doing something based on 30%.

3.2 Difficulties with element height

Working with element height is different than working with element width. The border-box fixes you've made thus far still apply and can be helpful; but, typically it's best to avoid setting explicit heights on elements. Normal document flow is designed to work with a constrained width and an unlimited height. Contents fill the width of the viewport and then line wrap as necessary. Because of this, the height of a container is organically determined by its contents, not by the container itself.



Normal document flow refers to the default layout behavior of elements on the page. Inline elements flow along with the text of the page, from left to right, line wrapping when they reach the edge of their container. Block-level elements fall on individual lines, with a line break above and below.

3.2.1 Controlling overflow behavior

When you explicitly set an element's height, you run the risk of its contents *overflowing* the container. This happens when the content doesn't fit the specified constraint and renders outside of the parent element. Figure 3.8 shows this behavior. Document flow doesn't account for overflow, and any content below the container will render over the top of the overflowing content.

We'll be running the Polar Bear 5k together on December 14th. Meet us at the town square at 7:00am to carpool. Wear blue!

Figure 3.8 Content overflowing its container

You can control the exact behavior of the overflowing content with the `overflow` property, which supports four values:

- `visible` (default value)—All content is visible, even when it overflows the container's edges.
- `hidden`—Content that overflows the container's padding edge is clipped and won't be visible.
- `scroll`—Scrollbars are added to the container so the user can scroll to see the remaining content. On some operating systems, both horizontal and vertical scrollbars are added, even if all the content is visible. In this case, the scrollbars will be disabled (grayed).
- `auto`—Scrollbars are added to the container only if the contents overflow.

Typically, I prefer `auto` rather than `scroll` because, in most cases, I don't want the scrollbars to appear except when necessary. Figure 3.9 shows four containers with these overflow settings.



Figure 3.9 Overflow from left to right: visible, hidden, scroll, and auto

Be judicious with the use of scrollbars. Browsers insert a scrollbar for scrolling the page, and adding nested scrollable areas inside your page can be frustrating to users. If a user is using a mouse scroll wheel to scroll down the page, and their cursor reaches a smaller scrollable area, their scroll wheel will stop scrolling the page and will scroll the smaller box instead.

Horizontal overflow

It's possible for content to overflow horizontally, not just vertically. One typical situation is when a long URL appears in a narrow container. The same rules apply here as with vertical overflow.

You can control only horizontal overflow using the `overflow-x` property, or vertical overflow with `overflow-y`. These properties support the same values as the `overflow` property. Explicitly setting both `x` and `y` to different values, however, tends to have unpredictable results.

3.2.2 Applying alternatives to percentage-based heights

Specifying height using a percentage is problematic. Percentage refers to the size of an element's containing block; the height of that container, however, is typically determined by the height of its children. This produces a circular definition that the browser can't resolve, so it'll ignore the declaration. For percentage-based heights to work, the parent must have an explicitly defined height.

One reason people try to use percentage-based heights is to make a container fill the screen. A better approach is to use the viewport-relative `vh` units, which you reviewed in chapter 2. A height of `100vh` is exactly the height of the viewport. The most common use, though, is to create columns of equal height. This too can be solved without a percentage.

COLUMNS OF EQUAL HEIGHT

The columns-of-equal-height problem is one weakness that has plagued CSS from the beginning. In the early 2000s, CSS supplanted the use of HTML tables for laying out content. At the time, tables were the only way to produce two columns of equal

height, or, more specifically, columns of the same height without explicitly specifying the height. You could easily set all columns to a height of 500 px or some other arbitrary value. But if you allowed the columns to determine their heights naturally, each element would evaluate to a different height, based on its content. This became a simple use case fraught with frustration.

It took some creative hacks to work around the issue. As CSS evolved, solutions involving pseudo-elements or negative margins emerged. If you're still using any of these complicated methods, it's time to fix that. Modern browsers make it much easier—they support CSS tables. For example, IE8+ supports `display: table`, and IE10+ allows for a flexible box, or flexbox, both of which, by default, produce equal-height columns.

NOTE When I say *modern browsers*, I mean recent versions of auto-updating, (*evergreen*) browsers. These include Chrome, Firefox, Edge, Opera, and, in most cases, Safari. Internet Explorer is the biggest concern; if I say something is supported in IE10+, that generally implies the evergreen browsers support it as well.

A number of common designs call for equal-height columns. Your two-column page is a great example. It'd look more polished if you were to align the heights of the main column and the sidebar (figure 3.10). As the content in either column grows, each column will grow as needed so the bottoms are always flush.

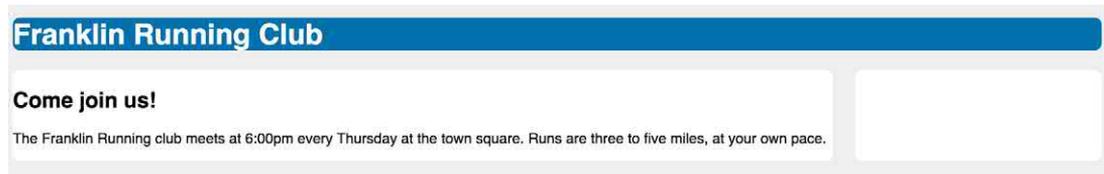


Figure 3.10 Columns of equal height

You could accomplish this by setting an arbitrary height on both columns, but what value would you choose? Too big, and you'll have a large empty space at the bottom of your containers; too small, and you'll have overflow to deal with.

The best solution is for the columns to size themselves naturally, and then extend the shorter column so that its height is equal to the height of the taller one. I'll show you how to do this using both CSS-based table layouts and a flexbox.

CSS TABLE LAYOUTS

First, you'll use a CSS-based table layout. Instead of using floats, you'll make the container a `display: table` and each column a `display: table-cell`. Update your styles

to match listing 3.9. (You may notice there's no `table-row` element. With CSS tables, the inclusion of a row element isn't as strict a requirement as it is with HTML tables.)

Listing 3.9 Equal-height columns using a CSS-based table layout

```
.container {
  display: table;
  width: 100%;
}

.main {
  display: table-cell;
  width: 70%;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  display: table-cell;
  width: 30%;
  margin-left: 1.5em;
  padding: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

Makes the container layout resemble a table

① Makes the table fill its container's width

Makes the column layout mimic table cells

② Margin no longer works

By default, an element with a `table` display value won't expand to a 100% width like a block element will, so you'll have to declare the width explicitly ①. This code gets you close, but now the gutter is missing. That's because margins ② can't be applied to `table-cell` elements. You'll have to make more changes to get this exactly how you want it.

To define space between cells of a table, you can use the `border-spacing` property of the table element. This property accepts two length values: one for horizontal spacing and one for vertical spacing. (You can also specify only one value to apply to both.) You could add `border-spacing: 1.5em 0` to your container, but this has a peculiar side effect: that value is also applied to the outside edges of the table. Now your two columns no longer align with the header on the left and right edges (figure 3.11).

Franklin Running Club

Come join us!

The Franklin Running club meets at 6:00pm every Thursday at the town square. Runs are three to five miles, at your own pace.

Figure 3.11 The `border-spacing` applies between table cells and affects the outside edges.

You can fix this with the clever use of a *negative* margin, but that needs to go on a new container that wraps around the whole table. Here's how. Add a `<div class="wrapper">` around the container and apply a left and right margin of -1.5 em to counteract the 1.5 em of the border spacing on the sidebars. This portion of your stylesheet should look like this.

Listing 3.10 Table-based columns with a corrected gutter

```
.wrapper {
  margin-left: -1.5em;
  margin-right: -1.5em;
}

.container {
  display: table;
  width: 100%;
  border-spacing: 1.5em 0;
}

.main {
  display: table-cell;
  width: 70%;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  display: table-cell;
  width: 30%;
  padding: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

← Adds a new wrapper element with negative margins

← Applies horizontal border spacing between table cells

Instead of positive margins pushing in the edges of the container, the negative margin pulls the edges out. Combined with `border-spacing`, the outside column edges now align with the edges of the `<body>` (the containing box for the wrapper). You now have the layout you want: two columns with equal height, a 1.5 em gutter, and outside edges that align with the header (figure 3.12).



Figure 3.12 Columns of equal height working correctly

Negative margins have some interesting uses that we'll look at in a bit.

Tables for layout?

If you've worked in web development for a while, you've likely heard that it's bad practice to use HTML tables for layouts. Many website designers in the early 2000s laid out their sites using `<table>` elements. It was often easier to lay out pages using tables instead of fighting with floats (the only viable alternative at the time). Eventually, there was a lot of backlash against the use of tables for layouts because doing so meant using non-semantic HTML. Instead of the HTML tags representing content, they were doing the work of layout—something CSS should be responsible for.

Browsers now support table display for all sorts of elements other than `<table>`, so you can enjoy the benefits of table layouts and maintain semantic markup. It's not a "holy grail" solution, however. The HTML table attributes `colspan` and `rowspan` have no equivalent, and floats, flexboxes, and inline-blocks can layout content in ways that tables can't.

FLEXBOX

Accomplishing a two-column layout with equal-height columns can also be done with a flexbox shown in listing 3.11. Notably, a flexbox doesn't necessitate the use of an extra div wrapper. By default, using a flexbox produces elements of equal height; you won't have to worry about negative margins.

TIP Favor the use of a flexbox instead of a table layout if you aren't actively supporting IE9 or older.

Remove the div wrapper you've added to the table layout, and update your stylesheet to match the following listing. If you're new to flexbox, this will be a gentle introduction.

Listing 3.11 Equal-height columns using a flexbox

```
.container {
  display: flex;
}

.main {
  width: 70%;
  background-color: #fff;
  border-radius: 0.5em;
}

.sidebar {
  width: 30%;
  padding: 1.5em;
  margin-left: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

← Applies flex display to the container

← Items inside the flex container don't need specified display or float properties.

← Margins work as before with floats

By applying `display: flex` to the container, it becomes a *flex container*. Its child elements will become the same height by default. You can set widths and margins on the items; even though this would add up to more than a 100%, the flexbox sorts it out. This listing renders pixel-for-pixel the same as the table layout. It doesn't need the extra wrapper, and the CSS is a bit simpler.

A flexbox provides a lot of options, which I'll dive into in chapter 5. This example shows all that you need to build your first flexbox-based layout. (IE10 requires some vendor-prefixed properties as well; I'll address those in chapter 5.)

WARNING Never explicitly set the height of an element unless you have no other choice. Always seek an alternative approach first. Setting a height invariably leads to further complications.

3.2.3 Using `min-height` and `max-height`

Two properties that can be immensely helpful are `min-height` and `max-height`. Instead of explicitly defining a height, you can use these properties to specify a minimum or maximum value, allowing the element to size naturally within those bounds.

Suppose you want to place your hero image behind a larger paragraph of text, and you're concerned about it overflowing the container. Instead of setting an explicit height, you can specify a minimum height with `min-height`. This means the element will be at least as high as you specify, and if the content doesn't fit, the browser will allow the element to grow naturally to prevent overflow.

Figure 3.13 shows three elements. The element on the left has no `min-height`, so its height is determined naturally, while each of the other two has a `min-height` of 3 em. The element in the middle would have a natural height shorter than that, but the `min-height` value has brought it to a height of 3 em. The element on the right has enough content that it has exceeded 3 em, and the container has grown naturally to contain the content.

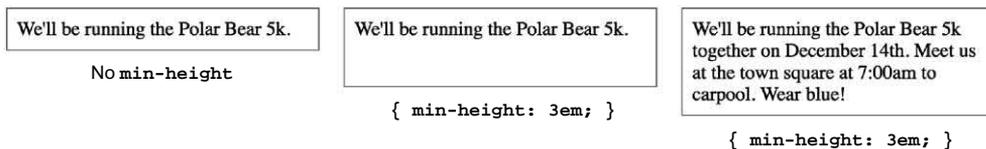


Figure 3.13 Three elements: one with no specified height, and two elements with a 3 em `min-height`

Likewise, `max-height` allows an element to size naturally, up to a point. If that size is reached, the element doesn't become any taller, and the contents will overflow. Similar properties `min-width` and `max-width` constrain an element's width.

3.2.4 Vertically centering content

Vertical centering in CSS is another notorious problem. Historically there have been several ways to achieve vertical centering, with each one working only under certain circumstances. With CSS, the answer to a problem is often “it depends,” and that can certainly be the case here.

Why doesn't vertical-align work?

Developers are often frustrated when they apply `vertical-align: middle` to a block element, expecting it to center the contents of the block. Instead, this declaration is ignored by the browser.

A `vertical-align` declaration only affects inline and table-cell elements. With inline elements, it controls alignment among other elements on the same line. You can use it to control how an inline image aligns with the neighboring text, for example.

With table-cell elements, `vertical-align` controls the alignment of the contents within the cell. If a CSS table layout works for your page, then you can accomplish vertical centering with `vertical-align`.

A lot of the trouble comes from setting the height of a container at a constant value, and then attempting to center a dynamically sized piece of content inside it. When possible, try to achieve your desired effect by allowing the browser to determine heights naturally.

Here's the simplest way to vertically center in CSS—give a container equal top and bottom padding, and let both the container and its contents determine their height naturally (figure 3.14). Listing 3.12 shows the code for this. You can temporarily add this code to your stylesheet to view it on your page (be sure to remove it afterward, as it's not part of your design).



Franklin Running Club

Figure 3.14 Using padding to vertically center the contents

Listing 3.12 Using padding to vertically center contents

```
header {  
  padding-top: 4em;  
  padding-bottom: 4em;  
  color: #fff;
```

Equal top and bottom padding vertically centers an element's content without a fixed height.

```
background-color: #0072b0;  
border-radius: .5em;  
}
```

This approach works whether the content inside the container is inline, block, or of any other display value. Sometimes, however, you may need to set a certain height on the container, or you don't have the option of using padding because you want another child in the container near the top or bottom.

This is also a common problem that arises with columns of equal height, particularly if you use an older technique with floats. Fortunately, both CSS tables and flex-boxes make centering easy. (If you use one of the older techniques, you'll have to find another way to center content.) For help dealing with various scenarios, see the following sidebar.

Guide to vertical centering

The best approach to centering contents inside a container may depend on a number of factors based on your particular scenario. To help you decide, step through these questions until you encounter a solution that you can use. Some of these techniques will be covered in later chapters, so I've directed you to where you can find the answer.

- *Can you use a natural height container?* Apply an equal top and bottom padding to the container to center its contents.
- *Do you need a specific height container, or do you need to avoid using padding?* Use `display: table-cell` and `vertical-align: middle` on your container.
- *Can you use flexbox?* If you don't need to support IE9, you can center your content with flexbox. See chapter 5.
- *Is the inner content only one line of text?* Set a tall line height equal to the desired container height. This will force the container to grow to contain the line height. If the contents aren't inline, you may have to set them to `inline-block`.
- *Do you know the height of both the container and the inner content?* Center the contents with *absolute* positioning. See chapter 7. (I only recommend this when all approaches mentioned here fail.)
- *What if you don't know the height of the inner element?* Use absolute positioning in conjunction with a transform. See chapter 15 for an example. (Again, I only recommend this if you've ruled out all other options.)

When in doubt, see <http://howtocenterincss.com>. It's a great resource. You can fill in several options based on your exact scenario, and it will produce the code you can use for vertical centering.

3.3 Negative margins

Unlike padding and border width, you can assign a negative value to margins. This has some peculiar uses, such as allowing elements to overlap or stretch wider than their containers.

The exact behavior of a negative margin depends on which side of the element you apply it to. You can see this illustrated in figure 3.15. If applied to the left or top, the negative margin moves the element leftward or upward, respectively. This can cause the element to overlap another element preceding it in the document flow. If applied to the right or bottom side, a negative margin doesn't shift the element; instead, it pulls in any succeeding element. Giving an element a negative bottom margin is not unlike giving the element(s) beneath it a negative top margin.

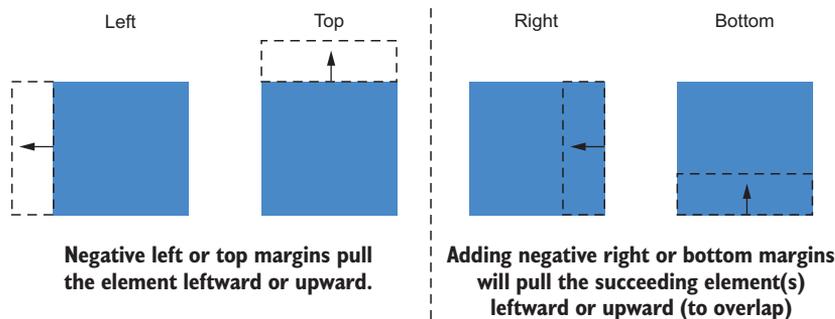


Figure 3.15 Behavior of negative margins

When a block element doesn't have a specified width, it naturally fills the width of its container. A negative right margin, however, can change this: As long as no width is specified, it pulls the edge of the element to the right, bringing it outside the container. Join this with an equal negative left margin, and both sides of the element will be extended outside the container. This quirk is what allowed you to resize the table layout in figure 3.12 to fill the `<body>` width, despite the border spacing.

WARNING Using negative margins to overlap elements can render some elements unclickable if they're moved beneath other elements.

Negative margins may not be something you use often, but they're useful in some circumstances. In particular, they come in handy when building column layouts. Be sure not to use them too frequently, though, or you may quickly find you lose track of what's happening on the page.

3.4 Collapsed margins

Take another look at your page. Notice something strange going on with the margins? You haven't applied any margin to the header or the container, yet there's a gap between them (figure 3.16). Why is that gap there?

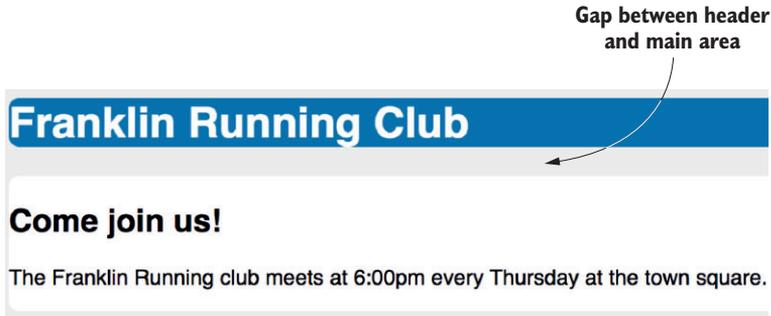


Figure 3.16 Gap caused by the margins collapsing

When top and/or bottom margins are adjoining, they overlap, combining to form a single margin. This is referred to as *collapsing*. The space below the header in figure 3.16 is the result of collapsed margins. Let's look at how this works.

3.4.1 Collapsing between text

The main reason for collapsed margins has to do with the spacing of blocks of text. Paragraphs (`<p>`), by default, have a 1 em top margin and a 1 em bottom margin. This is applied by the user agent stylesheet. But when you stack two paragraphs, one after the other, their margins don't add up to a gap of 2 em. Instead they collapse, overlapping to produce only 1 em of space between the two paragraphs.

You can see this sort of collapsed margin in the left column on the page. The title ("Come join us!") in an `<h2>` has a bottom margin of 0.83 em, which collapses with the top margin of the following paragraph. The margins of each are illustrated in figure 3.17. Note how the margins of each element occupy the same space on the page.



Figure 3.17 Outlined margins of the heading (left) and paragraph (right)

The size of the collapsed margin is equal to the largest of the joined margins. In this case, the heading has a bottom margin of 19.92 px (24 px font size × 0.83 em), and the paragraph has a top margin of 16 px (16 px font size × 1 em margin). The larger of these, 19.92 px, is the amount of space rendered between the two elements.

3.4.2 Collapsing multiple margins

Elements don't have to be adjacent siblings for their margins to collapse. Even if you wrap the paragraph inside an extra `div`, as in the next listing, the visual result will be the same. In the absence of any other CSS interfering, all the adjacent top and bottom margins will collapse.

Listing 3.13 Paragraph wrapped in a `div`, with the same result

```
<main class="main">
  <h2>Come join us!</h2>
  <div>
    <p>
      The Franklin Running club meets at 6:00pm
      every Thursday at the town square. Runs
      are three to five miles, at your own pace.
    </p>
  </div>
</main>
```

Margins still collapse even with the paragraph wrapped inside another `div`.

In this case, there are three different margins collapsing together: the bottom margin of the `<h2>`, the top margin of the `<div>`, and the top margin of the `<p>`. The computed values of these are 19.92 px, 0 px, and 16 px, respectively, so the space between the elements is still 19.92 px, the largest of the three. In fact, you can nest the paragraph inside several `div`s, and it will still render the same—all the margins collapse together.

In short, any adjacent top and bottom margins will collapse together. If you add an empty, unstyled `div` (one with no height, border, or padding) to the page, its own top and bottom margins will collapse.

NOTE Margin collapsing only occurs with top and bottom margins. Left and right margins don't collapse.

Collapsed margins act as a sort of “personal space bubble.” If two people standing at a bus stop are each comfortable with 3 feet of personal space between, they'll happily stand 3 feet apart. They don't need to stand 6 feet apart to both be satisfied.

This behavior typically means you can style margins on various elements without much concern for what might appear above or below them. If you apply a bottom margin of 1.5 em to the headings, you can expect the same spacing following the headings, whether the next element is a `<p>` with a top margin of 1 em or a `div` with no top margin. The collapsed margin between the elements only appears larger if the following element requires more space.

3.4.3 Collapsing outside a container

The way three consecutive margins collapse might catch you off guard. An element's margin collapsing outside its container typically produces an undesirable effect if the container has a background.

Take another look at the gap below the header in figure 3.16. The page title is a `<h1>`, with a 0.67 em (21.44 px) bottom margin applied by the user agent styles. That title is inside a `<header>` with no margins. The bottom margins of both elements are adjacent, so they collapse, resulting in a 21.44 px bottom margin on the header. The same thing happens with the top margins of the two elements as well.

This is a little strange. In this case, you want the `<h1>`'s margin to stay inside the `<header>`. Margins don't always collapse exactly to the spot where you want. Fortunately, there are a number of ways to prevent this. In fact, you've already fixed it for the main section of the page; notice that the margin above "Come join us!" doesn't collapse upward outside of its container. That's because the margins of flexbox items don't collapse, and you laid out that part of the page using a flexbox.

Padding provides another solution. If you add top and bottom padding to the header, the margins inside it won't collapse to the outside. While you're at it, let's update the header so it looks like figure 3.18, and apply left and right padding as well. To do so, update your stylesheet to match listing 3.14. You'll notice this now means there's no margin between the header and the main content. We'll come back to address that shortly.



Figure 3.18 Adding padding to the header prevents margin collapsing

Listing 3.14 Applying padding to the header

```
header {
  padding: 1em 1.5em;
  color: #fff;
  background-color: #0072b0;
  border-radius: .5em;
}
```

Here are ways to prevent margins from collapsing:

- Applying `overflow: auto` (or any value other than `visible`) to the container prevents margins inside the container from collapsing with those outside the container. This is often the least intrusive solution.
- Adding a border or padding between two margins stops them from collapsing.

- Margins won't collapse to the outside of a container that is floated, that is an inline block, or that has an absolute or fixed position.
- When using a flexbox, margins won't collapse between elements that are part of the flex layout. This is also the case with grid layout (chapter 6).
- Elements with a `table-cell` display don't have a margin, so they won't collapse. This also applies to `table-row` and most other table display types. Exceptions are `table`, `table-inline`, and `table-caption`.

Many of these change the layout behavior of the element, though, so you probably won't want to apply them unless they produce the layout you're looking for.

3.5 Spacing elements within a container

The interplay between the padding of a container and the margins of its content can be tricky to work with. Let's put a few items in your sidebar and work through problems that might arise. In the end, I'll show you a useful technique that can greatly simplify things.

You'll add two buttons that link to social media pages and another, less important link to the sidebar. Your goal is for the sidebar to look like figure 3.19.



Figure 3.19 The sidebar with properly spaced contents

Let's start with the two social links. Add them to your sidebar as shown in the following listing. The `button-link` class will be a good target for your CSS selector.

Listing 3.15 Adding two social buttons to the sidebar

```
<aside class="sidebar">
  <a href="/twitter" class="button-link">
    follow us on Twitter
  </a>
  <a href="/facebook" class="button-link">
    like us on Facebook
  </a>
</aside>
```

Next, you'll apply styles for the buttons' general appearance. You'll make them block elements so they'll fill the width of the container, and each will appear on its own line. Add this CSS to your stylesheet.

Listing 3.16 Setting size, fonts, and colors for the sidebar buttons

```
.button-link {
  display: block;
  padding: 0.5em;
  color: #fff;
  background-color: #0090C9;
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}
```

← The block element fills the available width and puts each link on its own line.

Now the links are styled correctly, but you still need to figure out the spacing between them. Without margins, they'll stack directly atop one another, as they do now. You have options: you could give them separate top and bottom margins or both, where margin collapsing would occur between the two buttons.

No matter which approach you choose, however, you'll still encounter a problem: the margin needs to work in conjunction with the sidebar's padding. If you add `margin-top: 1.5em`, you'll get the result shown in figure 3.20.



Figure 3.20 The top margin adds spacing to the container's padding.

Now you'll have extra space at the top of the container. The first button's top margin plus the container's top padding produce spacing that's uneven with the other three sides of the container.

You can fix this in a number of ways. Listing 3.17 shows one of the simpler fixes. It uses the adjacent sibling combinator (+) to target only `button-link`s that immediately follow other `button-link`s as siblings under the same parent element. Now the margin only appears between two buttons.

Listing 3.17 Using an adjacent sibling combinator to apply a margin between buttons

```
.button-link {
  display: block;
  padding: .5em;
  color: #fff;
  background-color: #0090C9;
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}
```

```
.button-link + .button-link {
  margin-top: 1.5em;
}
```

Only apply a top margin to button-links that immediately follow another button-link

This appears to work (figure 3.21). The first button no longer has a top margin, so the spacing is even.

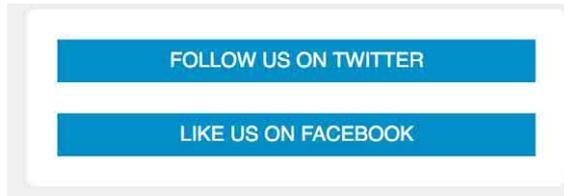


Figure 3.21 Even spacing is applied around the buttons.

3.5.1 Considering changing content

You're on the right track, but the spacing problem arises again as soon as you add more content to the sidebar. Add the third link to your page, as shown in the following listing. This one has the class `sponsor-link` so you can apply different styles to the link.

Listing 3.18 Adding a different type of link to the sidebar

```
<aside class="sidebar">
  <a href="/twitter" class="button-link">
    follow us on Twitter
  </a>
  <a href="/facebook" class="button-link">
    like us on Facebook
  </a>
  <a href="/sponsors" class="sponsor-link">
    become a sponsor
  </a>
</aside>
```

Adds a different type of link to the sidebar

You'll style this one, but again, you'll have to address the question of the spacing between it and the other buttons. Figure 3.22 shows how the link will look *before* you fix the margin.



Figure 3.22 Spacing is off between the second button and the bottom link.

The styles for this are shown in the next listing. Add these to your stylesheet. You're probably tempted to add a top margin to the link as well; hold off on that for now. I'll show you an interesting alternative next.

Listing 3.19 Adding styles for the sponsor link

```
.sponsor-link {  
  display: block;  
  color: #0072b0;  
  font-weight: bold;  
  text-decoration: none;  
}
```

You could add a top margin, and it would look right. But consider this: HTML has a nasty habit of changing. At some point, whether next month or next year, something in this sidebar will need to be moved or replaced. Maybe the sponsorship link will need to be moved to the top of the sidebar. Or, maybe you'll need to add a widget to sign up for an email newsletter.

Every time things change, you'll have to revisit the question of these margins. You'll need to make sure that there's space between each item, but no extraneous space at the top (or bottom) of the container.

3.5.2 *Creating a more general solution: the lobotomized owl selector*

Web designer Heydon Pickering once said margins are “like applying glue to one side of an object before you've determined whether you actually want to stick it to something or what that something might be.” Instead of fixing margins for the current page contents, let's fix it in a way that works no matter how the page gets restructured. You'll do this with something Pickering calls a *lobotomized owl selector*. It looks like this: * + *.

That's a universal selector (*) that targets all elements, followed by an adjacent sibling combinator (+), followed by another universal selector. It earns its name because it resembles the vacant stare of an owl. The lobotomized owl is not unlike the selector you used earlier: `.social-button + .social-button`. Except, instead of targeting buttons that immediately follow other buttons, it targets any element that immediately follows any other element. That is, it selects all elements on the page that aren't the first child of their parent.

Let's use the lobotomized owl to add top margins to elements throughout your page. Doing so will evenly space each item in your sidebar. It'll also target the main container because a sibling immediately follows the header, providing the space you want there as well. The result is shown in figure 3.23.

Add listing 3.20 near the top of your stylesheet. I've included `body` at the beginning of the selector. This restricts the selector to only target items inside the body. If you use the lobotomized owl by itself, it will target the `<body>` element because it's an adjacent sibling of the `<head>` element.

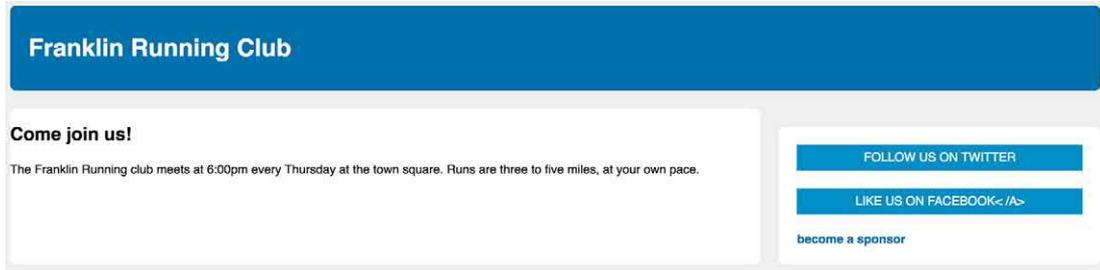


Figure 3.23 All adjacent sibling elements now have a top margin

Listing 3.20 Globally space stacked items with the lobotomized owl

```
body * + * {
  margin-top: 1.5em;
}
```

NOTE You might be worried about the performance implications of the universal selector (*). In IE6, it was incredibly slow, so developers avoided using it. Today, this is no longer a concern because modern browsers handle it well. Furthermore, using it in the lobotomized owl potentially reduces the number of selectors in your stylesheet, because it globally fixes most elements. In fact, it might be more performant, depending on the particulars of your stylesheet.

The lobotomized owl top margin has one unwanted side effect on the sidebar. Because the sidebar is an adjacent sibling of the main column, it too receives a top margin. You'll have to revert that to zero. You'll also have to add padding to the main columns because you haven't done so yet. Update the corresponding portion of your stylesheet to match the listing shown here.

Listing 3.21 Final touches

```
.main {
  width: 70%;
  padding: 1em 1.5em;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  width: 30%;
  padding: 1.5em;
  margin-top: 0;
  margin-left: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

← Adds padding to the main column

← Removes the top margin applied by the lobotomized owl

These are the final touches for your page. It should now look like figure 3.24.

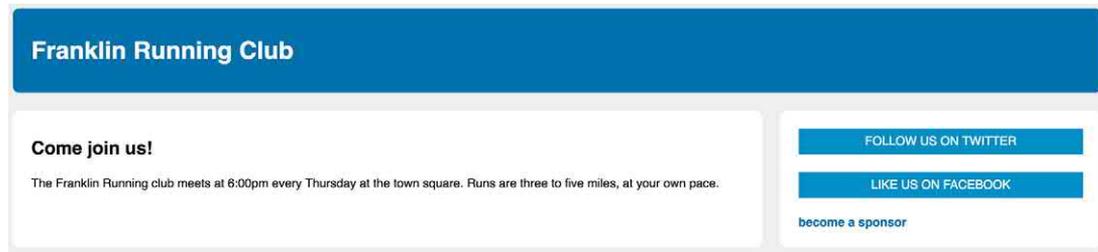


Figure 3.24 Final page with a two-column layout

Using the lobotomized owl like this is a tradeoff. It simplifies many margins throughout your page, but you'll have to override it in places where you don't want it to apply. This will generally only be in places where you've elements side by side, as with multi-column layouts. Depending on your design, you'll also need to set the desired margins on paragraphs and headings.

I'll use the lobotomized owl in more examples in the next few chapters to help you get a feel for the tradeoffs involved. The lobotomized owl may not be the correct solution for every project, and it's difficult to add to an existing project without breaking the layout, but consider it the next time you start a new website or web application.

The full stylesheet is given here.

Listing 3.22 Final stylesheet

```
:root {
  box-sizing: border-box;
}

*,
::before,
::after {
  box-sizing: inherit;
}

body {
  background-color: #eee;
  font-family: Helvetica, Arial, sans-serif;
}

body * + * {
  margin-top: 1.5em;
}

header {
  padding: 1em 1.5em;
  color: #fff;
```

```
    background-color: #0072b0;
    border-radius: .5em;
}

.container {
  display: flex;
}

.main {
  width: 70%;
  padding: 1em 1.5em;
  background-color: #fff;
  border-radius: .5em;
}

.sidebar {
  width: 30%;
  padding: 1.5em;
  margin-top: 0;
  margin-left: 1.5em;
  background-color: #fff;
  border-radius: .5em;
}

.button-link {
  display: block;
  padding: .5em;
  color: #fff;
  background-color: #0090C9;
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}

.sponsor-link {
  display: block;
  color: #0072b0;
  font-weight: bold;
  text-decoration: none;
}
```

Summary

- Always use a universal border-box fix for predictable element sizing.
- Avoid explicitly setting the height of an element to avoid overflow issues.
- Use modern layout techniques like `display: table` or a flexbox to produce columns of equal height or to vertically center content.
- If your margins behave oddly, take steps to prevent margins from collapsing.
- Consider using the lobotomized owl selector on your page to globally apply margins between stacked elements.

Part 2

Mastering layout

CSS provides several tools you can use to control the layout of a web page. In part 2 (chapters 4–8), we’ll look at the most important of these tools, from floats to flexbox to positioning. None of these tools is intrinsically better than another, but rather they each accomplish something a little different. I’ll show you how they each work so you can use this understanding to mix and match them on a page to achieve the result you need.

Making sense of floats



This chapter covers

- How floats work and how to avoid common pitfalls
- Container collapsing and the clearfix
- The media object and double container pattern
- Block formatting contexts
- How to build and understand a grid system

At the end of part 1, we covered some fundamental concepts of element sizing and spacing. Throughout part 2, we'll build on these concepts by looking closer at the primary methods for laying out the page. We'll look at the three most important methods to alter document flow—floats, flexbox, and grid layout. Then we'll look at positioning, which is used primarily for stacking elements in front of one another. The flexbox and grid layouts are both new to CSS and are proving to be essential tools. Although floats and positioning are not new, they're often misunderstood.

In this chapter, we'll first look at floats. They're the oldest method for laying out a web page, and for many years were the only way. They're a little odd, however. Making sense of floats begins with an understanding of their original purpose,

which is where we'll start. I'll show you how to deal with some of their quirks, including a tool called a clearfix. This will put some context to their behavior.

As we go, you'll also learn about two patterns that you might often see in page layouts: the double container pattern and the media object. To wrap up, you'll put your knowledge to work to build a grid system, which is a versatile tool for structuring a page.

4.1 *The purpose of floats*

Although floats were not originally intended to construct page layouts, they have served that job well. In order to make sense of floats, however, we must first bear in mind their original purpose.

A *float* pulls an element (often an image) to one side of its container, allowing the document flow to wrap around it (figure 4.1). This layout is common in newspapers and magazines, so floats were added to CSS to achieve this effect.

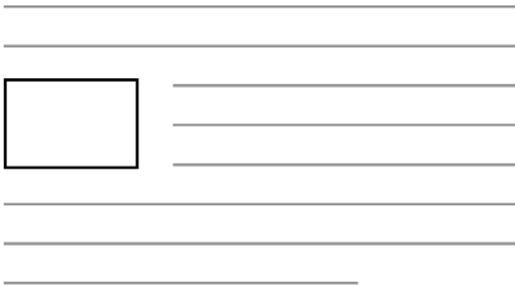


Figure 4.1 Lines of text wrap around floated elements

This illustration shows an element pulled to the left, but you can also float an element to the right. A floated element is removed from the normal document flow and pulled to the edge of the container. The document flow then resumes, but it'll wrap around the space where the floated element now resides. If you float multiple elements in the same direction, they'll stack alongside one another, as shown in figure 4.2.

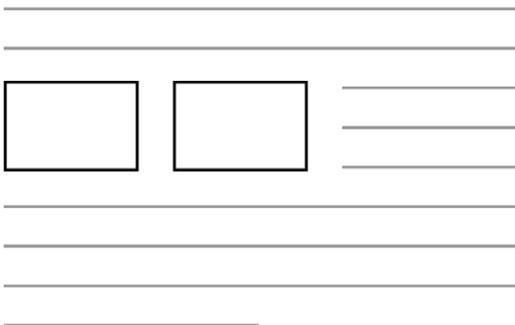


Figure 4.2 Two floated elements stacked alongside one another

If you've been writing CSS for a while, this behavior is probably not new to you. But the important thing to note is this: We don't always use floats in this way, even though it's their original purpose.

In the early days of CSS, developers realized they could use this simple tool to move sections of the page around to build all sorts of layouts. It was not intended to be a page layout tool, but for nearly two decades, we've been using it as such.

We did this because it was our only option. Eventually, the ability to use `display: inline-block` or `display: table` emerged, which offered alternatives, albeit limited ones. Until the addition of the flexbox and grid layouts in the past few years, floats remained our heavy hitter for page layout. Let's take a good look at how they work. As a guide, you'll build the page shown in figure 4.3.

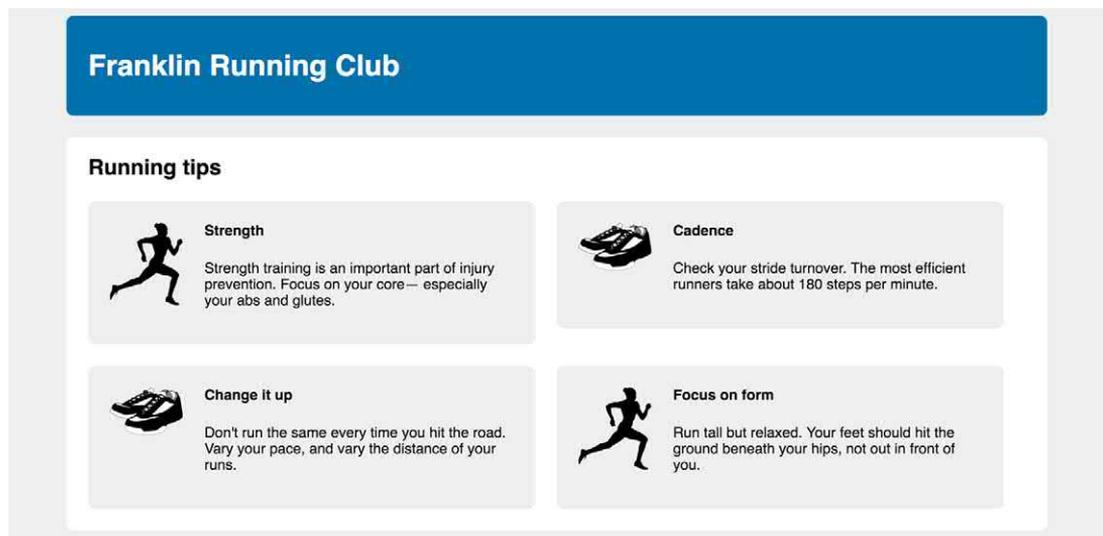


Figure 4.3 Web page with a float-based layout

In the examples in this chapter, you'll use floats to position each of the four gray boxes. Inside the boxes, you'll then float the images beside the text. Create a blank page and link it to a new stylesheet, then add the code in this listing to your page.

Listing 4.1 HTML for a page with a float-based layout

```
<body>
  <div class="container">
    <header>
      <h1>Franklin Running Club</h1>
    </header>
```

Header layout
similar to that
in chapter 3

```

<main class="main clearfix">
  <h2>Running tips</h2>

  <div>
    <div class="media">
      
      <div class="media-body">
        <h4>Strength</h4>
        <p>
          Strength training is an important part of
          injury prevention. Focus on your core&mdash;
          especially your abs and glutes.
        </p>
      </div>
    </div>

    <div class="media">
      
      <div class="media-body">
        <h4>Cadence</h4>
        <p>
          Check your stride turnover. The most efficient
          runners take about 180 steps per minute.
        </p>
      </div>
    </div>

    <div class="media">
      
      <div class="media-body">
        <h4>Change it up</h4>
        <p>
          Don't run the same every time you hit the
          road. Vary your pace, and vary the distance
          of your runs.
        </p>
      </div>
    </div>

    <div class="media">
      
      <div class="media-body">
        <h4>Focus on form</h4>
        <p>
          Run tall but relaxed. Your feet should hit
          the ground beneath your hips, not out in
          front of you.
        </p>
      </div>
    </div>
  </div>
</main>
</div>
</body>

```

← Main element, the white box,
that contains most of the page

← Four media
objects for
each of the
gray boxes

This listing gives you the page structure: a header and a main element that will contain the rest of the page. Inside the main element is the page title, followed by an *anonymous div* (that is, a `div` with no class or ID). This serves to group the four gray media elements, each of which contains an image and a body element.

TIP It's usually easiest to lay out the large regions of a page first, then work your way to the smaller elements within.

Before you start floating elements, you'll put the outer structure of the page in place. Add the next listing to your stylesheet.

Listing 4.2 Base styles for the page

```

:root {
  box-sizing: border-box;
}

*,
::before,
::after {
  box-sizing: inherit;
}

body {
  background-color: #eee;
  font-family: Helvetica, Arial, sans-serif;
}

body * + * {
  margin-top: 1.5em;
}

header {
  padding: 1em 1.5em;
  color: #fff;
  background-color: #0072b0;
  border-radius: .5em;
  margin-bottom: 1.5em;
}

.main {
  padding: 0 1.5em;
  background-color: #fff;
  border-radius: .5em;
}

```

Global border-box fix (from chapter 3)

Lobotomized owl global margins (from chapter 3)

Header colors and padding

Main (white container) colors and padding

This sets some base styles for the page, including a box-sizing fix and lobotomized owl from chapter 3. Next, you'll want to constrain the width of the page contents, shown in figure 4.4. Notice the light gray margins on both sides and how both the header and the main container are equal widths within.

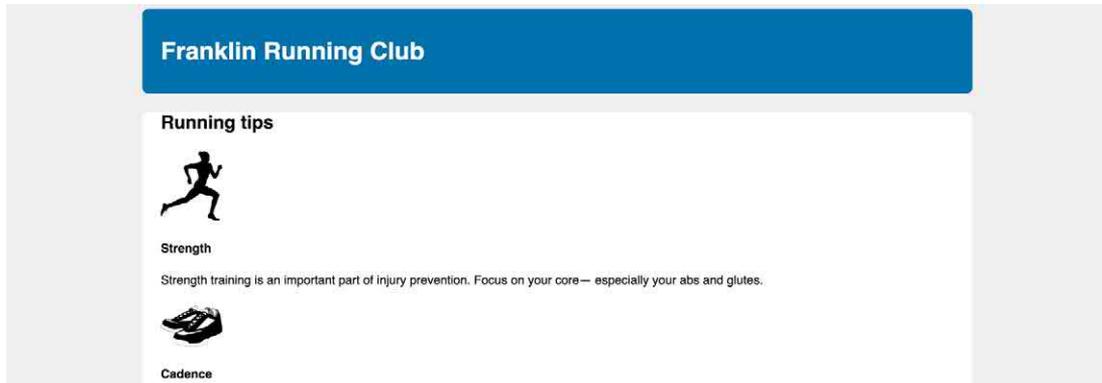


Figure 4.4 Page with constrained width

This layout is common for centering content on a page. You can achieve it by placing your content inside two nested containers and then set margins on the inner container to position it within the outer one (figure 4.5). Web developer Brad Westfall calls this the *double container pattern*.

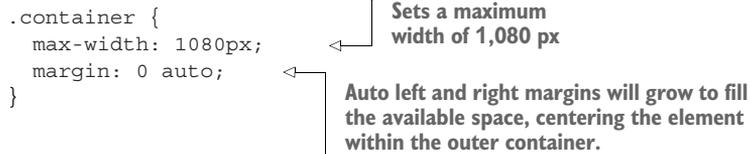


Figure 4.5 The double container pattern

In our example, `<body>` serves as the outer container. By default, this is already 100% of the page width, so you won't have to apply any new styles to it. Inside that, you've wrapped the entire contents of the page in a `<div class="container">`, which serves as the inner container. To that you'll apply a `max-width` and `auto` margins to center the contents. Add this listing to your stylesheet.

Listing 4.3 Styles for the double container

```
.container {  
  max-width: 1080px;  
  margin: 0 auto;  
}
```



Sets a maximum width of 1,080 px

Auto left and right margins will grow to fill the available space, centering the element within the outer container.

By using `max-width` instead of `width`, the element shrinks to below 1080 px if the screen's viewport is smaller than that. That is to say, in smaller viewports, the inner container will fill the screen, but on larger ones, it'll expand to 1080 px. This is important to avoid horizontal scrolling on devices with smaller screens.

Do you still need to know how to use floats?

Flexbox is rapidly supplanting the use of floats for page layout. Its behavior is straightforward and often more predictable for new developers. You might find yourself asking whether you need to know about floats at all. Has CSS moved past this?

With modern browsers, you can certainly go a lot further without floats than you could in the past. You can probably get by without floats altogether. But if you need to support Internet Explorer, you may not want to let go of them just yet. Flexbox is only supported in IE 10 and 11, and even then it has a few bugs. If you don't want to worry about browser bugs or you need to support older browsers, floats could be a better option.

If you're supporting an older codebase, it likely uses floats; you'll need to know how they work in order to maintain it. Additionally, float-based layouts often require less markup, where newer methods require the addition of container elements. If you have limited control over the markup you're styling, floats might be more capable of doing what you need.

And floats are still the only way to move an image to the side of the page and allow text to wrap around it.

4.2 Container collapsing and the clearfix

In the past, browser bugs have plagued the behavior of floats, albeit mostly in IE 6 and 7. It's almost certain you no longer need to support these browsers, so you don't need to worry about those bugs. Now you can trust that browsers will handle floats consistently.

A few behaviors of floats still might catch you off guard, however. These are not bugs, but rather floats behaving precisely how they're supposed to behave. Let's look at how they work and how you can adjust their behavior to achieve the layout you want.

4.2.1 Understanding container collapsing

On your page, let's float the four media boxes to the left. The problems will immediately become apparent (figure 4.6).

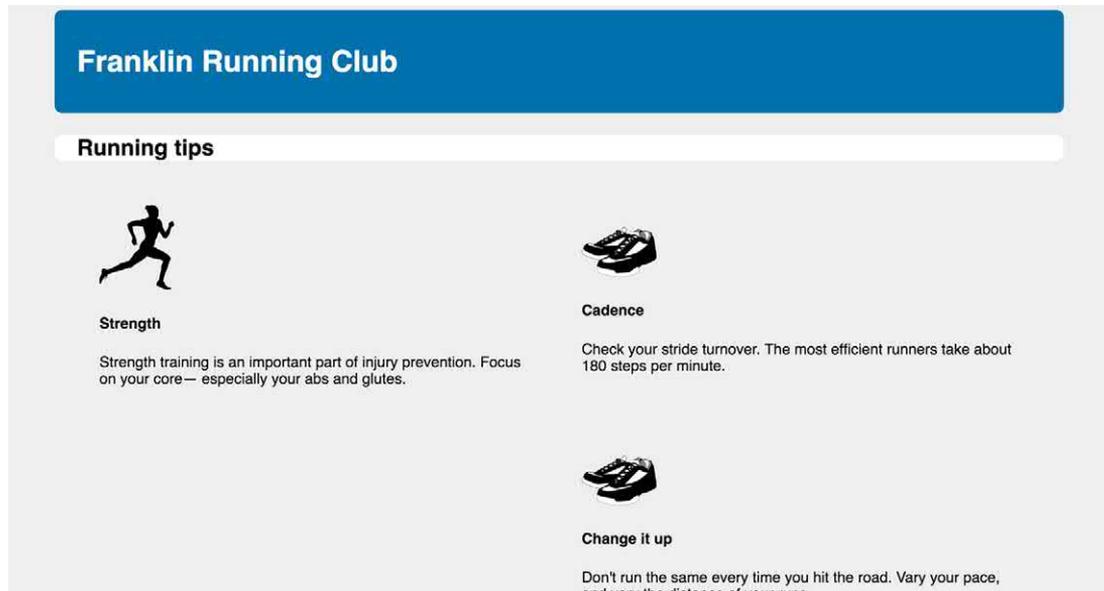


Figure 4.6 Container with its floated descendants

What happened to the white background? We see it behind the page title (“Running tips”), but it stops there instead of extending down to encompass the media boxes. To see this on your page, add the following listing to your stylesheet. Then we’ll look at why this happens and how you can fix it.

Listing 4.4 Floating the four media boxes to the left

```
.media {
  float: left;
  width: 50%;
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}
```

← Floats each media box to the left

← Sets a width to fit two boxes across the page

You’ve set a light gray background on each media box, expecting to see the white background of the container behind (or rather, around) them. Instead, the white background stopped above the top row of media boxes. Why is this?

The problem is that, unlike elements in the normal document flow, floated elements do not add height to their parent elements. This may seem odd, but it goes back to the original purpose of floats.

As you learned near the beginning of this chapter, floats are intended to allow text to wrap around them. When you float an image inside a paragraph, the paragraph does not grow to contain the image. This means, if the image is taller than the text of the paragraph, the next paragraph will start immediately below the text of the first, and the text in both paragraphs will wrap around the float. This is illustrated in figure 4.7.

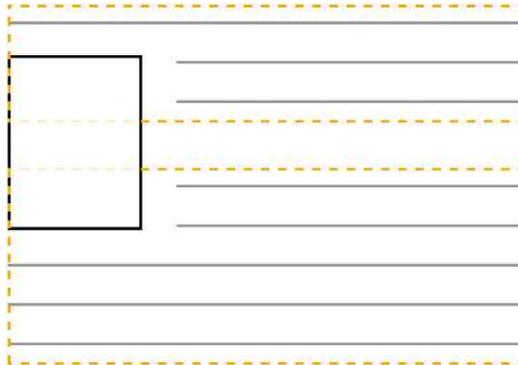


Figure 4.7 The float in one container extends into the next container, allowing text in both containers to wrap around the floated element (containers highlighted with dashed lines).

In your page, everything inside the main element is floated except for the page title, so only the page title contributes height to the container, leaving all the floated media elements extending below the white background of the main. This isn't the behavior we want, so let's fix it. The main element should extend down to contain the gray boxes (shown in figure 4.8).

One way you can correct this is with the float's companion property, `clear`. If you place an element at the end of the main container and use `clear`, it causes the container to expand to the bottom of the floats. The code in the next listing shows, in principle, what we want to do. You can add this to your page temporarily to see how it works.

Listing 4.5 Container extends to contain an element that clears floats

```
<main class="main">
  ...
  <div style="clear: both"></div>
</main>
```

← Adds an empty div with a clear at the end of the main container

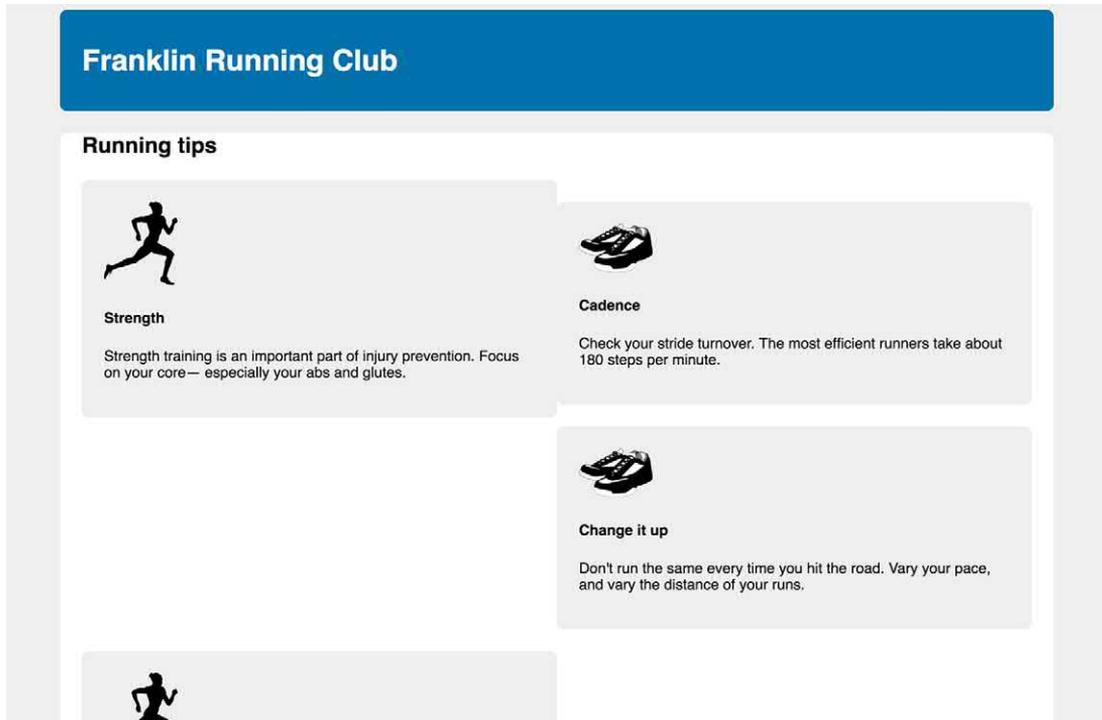


Figure 4.8 Container extended to encompass the floats

The `clear: both` declaration causes this element to move below the bottom of floated elements, rather than beside them. You can give this property the value `left` or `right` to clear only elements floated to the left or right, respectively. Because this empty div itself is not floated, the container will extend to encompass it, thereby containing the floats above it as well.

This sizes the container how you want, but it's rather hacky; you're adding unwanted markup to your HTML to do the work that should be done by the CSS. Go ahead and delete that empty div. Let's look at a way you can accomplish this purely in your CSS.

4.2.2 *Understanding the clearfix*

Instead of adding an extra div to your markup, you'll use a *pseudo-element*. By using the `::after` pseudo-element selector, you can effectively insert an element into the DOM at the end of the container, without adding it to the markup.



pseudo-element—Special selectors that target certain parts of the document. These begin with a double-colon (::) syntax, though most browsers also support a single-colon syntax for backward compatibility. The most common pseudo-elements are ::before and ::after, which are used to insert content at the beginning or end of an element. See appendix A for more information.

Listing 4.6 shows a common approach to the problem of containing floats, called a *clearfix*. (Some developers like to abbreviate the class name to *cf*, which is conveniently also an abbreviation for “contain floats.”) Add this to your stylesheet.

Listing 4.6 Using clearfix to contain floats

```
.clearfix::after {
  display: block;
  content: " ";
  clear: both;
}
```

Targets the pseudo-element at the end of the container

A non-inline display value and a content value cause the pseudo-element to appear in the document.

Makes the pseudo-element clear all floats in the container

It’s important to know that the clearfix is applied to the element that contains the floats; a common mistake is to apply it to the wrong element, such as the floats or the container after the one that contains them.

NOTE The clearfix has gone through dozens of iterations over the years, some more complicated than others. Many versions had nuances to correct various browser bugs. Most of the workarounds are no longer necessary, though this example has one such workaround in place: the space in the content value. An empty string (“”) works as well, but the added space character fixes an obscure bug in older versions of Opera. I tend to leave this fix in because it’s unobtrusive.

One inconsistency with this clearfix remains: Margins of floated elements inside won’t collapse to the outside of the clearfixed container; but, margins of non-floated elements will collapse as normal. You can see this in your page where the heading “Running tips” is pressed directly against the top of the white <main> (figure 4.8); its margin has collapsed out of the container.

Some developers prefer to use a modified version of the clearfix that will contain all margins because it can be slightly more predictable. Adding this version to your page will prevent the top margin of the page title from collapsing outside of the main, as shown in figure 4.9, leaving appropriate spacing above the heading.

For the modified version, update the clearfix in your stylesheet to match this listing.

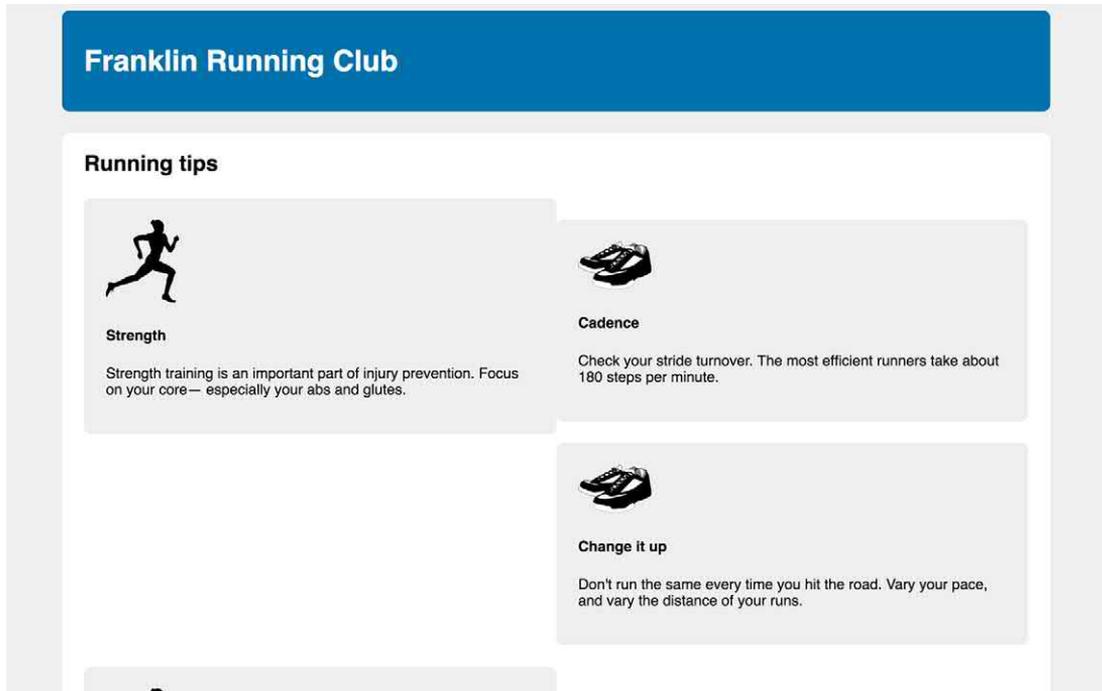


Figure 4.9 A modified clearfix contains all floats and margins; notice the top margin of the heading “Running tips” is now contained inside the white `<main>`.

Listing 4.7 Modifying clearfix to contain all margins

```
.clearfix::before,
.clearfix::after {
  display: table;
  content: " ";
}
```

← Causes both `::before` and `::after` pseudo elements to appear
Prevents margins from collapsing through the pseudo elements

```
.clearfix::after {
  clear: both;
}
```

Only the `::after` pseudo element needs to clear floats.

This version makes use of `display: table` rather than `display: block`. By applying this to both the `::before` and `::after` pseudo-elements, you’ll contain any child elements’ margins at both the top and bottom of the container. See the sidebar “Clearfix and `display: table`” for a more complete explanation of why this works.

TIP This version of the clearfix also doubles as a useful way to prevent margin collapsing where we don’t want it.

Which version of the clearfix you use in your projects is up to you. Some developers make the argument that margin collapsing is a fundamental feature of CSS, so they prefer not to contain margins. But, because neither version contains the margins of floated elements, others prefer the more consistent behavior of the modified version. Each argument has its merit.

Clearfix and `display: table`

Using `display: table` in the clearfix contains margins because of a few peculiarities of CSS. The creation of a `display-table` element (or, in this case, pseudo-element) implicitly creates a table row within the element and a table cell within that. Because margins don't collapse through table-cell elements (as mentioned in chapter 3), they won't collapse through a `display-table` pseudo element either.

It might seem, then, that you could use `display: table-cell` to the same effect. However, the `clear` property only works when applied to block-level elements. A table is a block-level element, but a table cell is not; thus, the `clear` property could not be applied along with `display: table-cell`. Therefore, you need to use `display: table` to clear floats and its implied table cell to contain the margins.

4.3 Unexpected “float catching”

Now that the white container contains the floated media elements on your page, another issue becomes apparent: The four media boxes aren't laying out in two even rows like you want. Instead, the first two boxes (“Strength” and “Cadence”) are in a row as expected, but the third box (“Change it up”) is on the right, beneath the second box. This leaves a large gap below the first box, which happens because the browser places floats as high as possible. See figure 4.10 for a simplified diagram.

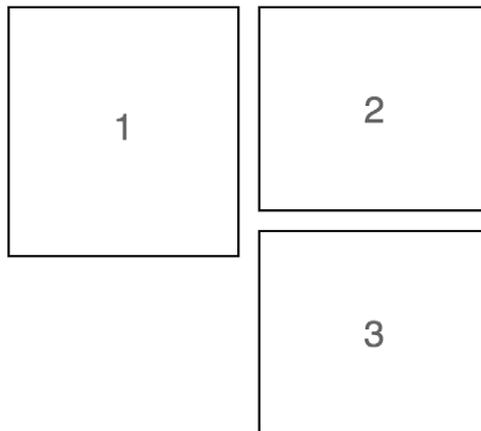


Figure 4.10 Three left-floated boxes: Box 3 doesn't float all the way to the left if box 1 is taller than box 2, instead it floats up against box 1.

Because box 2 is shorter than box 1, there's room for box 3 beneath it. Instead of clearing box 1, box 3 "catches" on it. That is to say, it doesn't float all the way to the left edge, but rather floats against the bottom corner of box 1.

The exact nature of this behavior is dependent on the heights of each of the floated blocks. Even a 1 px difference in element heights can cause this problem. On the other hand, if box 1 is shorter than box 2, there'll be no edge for the third box to catch on, and you won't see the problem until the content changes later, resulting in changed element heights.

By floating a series of elements to one side, you can end up with a wild array of layouts, depending on the heights of each box. Even changing the browser width can alter things as this will affect line wrapping and will change the heights of the elements. What you want to see on your page instead is two floated boxes per row, as in figure 4.11.

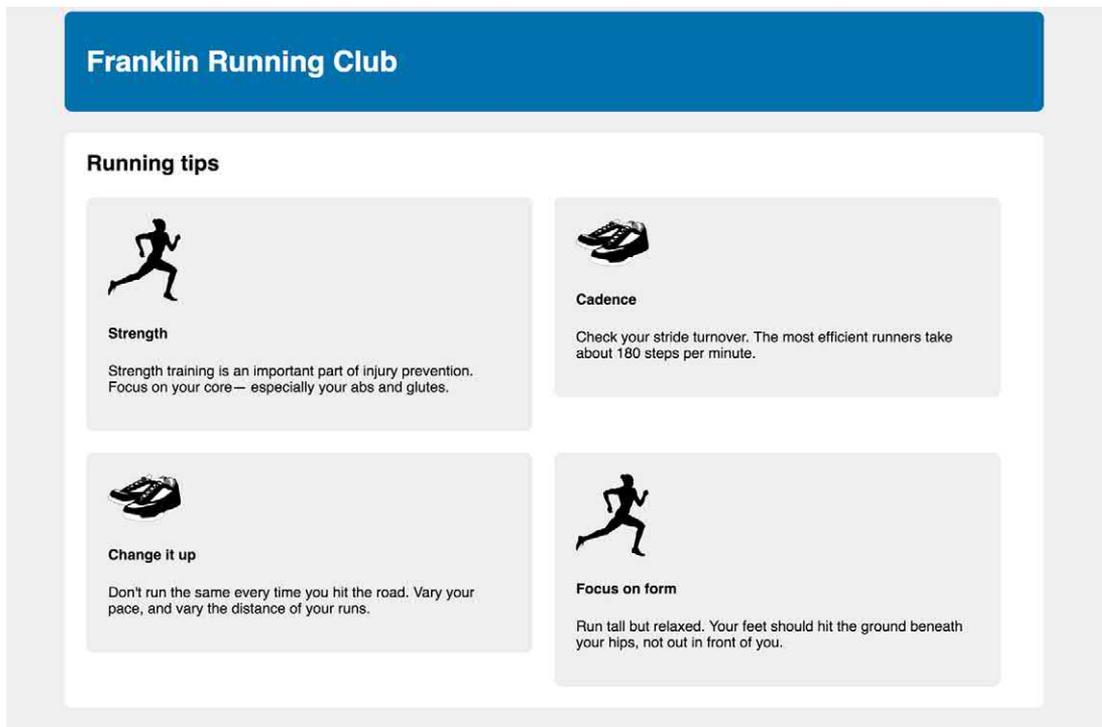


Figure 4.11 Two elements per row: the second row of media elements should clear the elements in the first row.

The fix for this is simple: The third float needs to clear the floats above it. Or, more generally, the first element of each row needs to clear the float above it. Because you know you have two boxes per row, you'll need the odd numbered elements to each

clear the row above. You can target these with the `:nth-child()` pseudo-class selector. Add this ruleset to your stylesheet.

Listing 4.8 Using the `:nth-child()` selector to target every odd media element

```
.media {
  float: left;
  width: 50%;
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}

.media:nth-child(odd) {
  clear: left;
}
```

Each new row clears the row above

This code will work even if you add more elements to the page later. It targets the first, third, fifth elements, and so on. If, instead, you had three items per row, you could target every third with the selector `.media:nth-child(3n+1)`. See appendix A for more on using the `:nth-child` selector.

NOTE This technique for clearing each row only works when you know how many elements are on each row. If the width is defined using something other than a percentage, the number of items can vary, depending on the viewport width. In this case, your best bet is to use a different layout technique such as the flexbox or inline-block elements.

Let’s also add margins to our media elements to provide a gutter between them. The lobotomized owl will also add a top margin to every element except the first. This misaligns the elements in the top row, so you’ll need to reset the top margin on those as well. Update your stylesheet to match the following listing.

Listing 4.9 Adding margins to the media elements

```
.media {
  float: left;
  margin: 0 1.5em 1.5em 0;
  width: calc(50% - 1.5em);
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}

.media:nth-child(odd) {
  clear: left;
}
```

Adds a right and bottom margin to each media element

Subtracts the margin from the width to avoid unwanted line wrapping

By adding right margins to the elements, they’ll no longer fit two to a row, so you’ll have to subtract an equal amount from the element width using `calc()`.

4.4 **Media objects and block formatting contexts**

Now that each of the four gray boxes is laid out, let's look at their contents. In our intended design, we have an image on one side and a block of text beside it (figure 4.12). This is another common pattern in page layouts, which web developer Nicole Sullivan has called the “media object.”

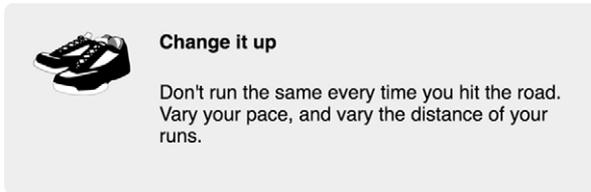


Figure 4.12 The media object pattern: an image on the left and descriptive content on the right

This pattern can be implemented in a number of ways, including the flexbox or table displays, but we'll do it using floats. The markup for one of the media objects in your page looks like this:

```
<div class="media">
  
  <div class="media-body">
    <h4>Change it up</h4>
    <p>
      Don't run the same every time you hit the
      road. Vary your pace, and vary the distance
      of your runs.
    </p>
  </div>
</div>
```

I've added the classes `media-image` and `media-body` to the left and right parts of each media object, which you'll use to position them. You'll start by floating the image to the left. As you can see (figure 4.13), merely floating the image is not enough. When the text is long enough, it wraps around the floated element. This is the normal float behavior, but it's not what we want in this case.

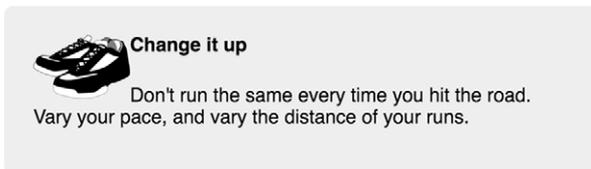


Figure 4.13 Unwanted text wrapping around the floated image

Add listing 4.10 to your stylesheet so that your page matches figure 4.13, then we'll take a look at how you can fix it. This code also removes the top margins from the media body and the title within.

Listing 4.10 Floating the media object image to the left

<pre>.media-image { float: left; }</pre>	<p>Floats image to the left</p>
<pre>.media-body { margin-top: 0; }</pre>	<p>Removes the top margin applied by the lobotomized owl</p>
<pre>.media-body h4 { margin-top: 0; }</pre>	<p>Overrides the top margin applied by user agent styles</p>

To fix the behavior of the text, you'll need to understand a little more about how floats work.

4.4.1 Establishing a block formatting context

If you examine the media body in your browser's DevTools (right-click and select Inspect or Inspect Element), you'll see that its box extends all the way to the left, so it envelops the floated image (figure 4.14, left). The text inside the body wraps around the image, but once it's clear of the bottom of the image, it moves all the way to the left of the box. What we want instead is to position the media body's left edge to the right of the floated image (figure 4.14, right).



Figure 4.14 By default, the text in the media object body wraps around the floated image (left). By giving the body a block formatting context, the text doesn't overlap (right).

To achieve the layout on the right, you'll need to establish something called a block formatting context for the media body. A *block formatting context* (sometimes called a BFC) is a region of the page in which elements are laid out. A block formatting context itself is part of the surrounding document flow, but it isolates its contents from the outside context. This isolation does three things for the element that establishes the BFC:

- 1 It contains the top and bottom margins of all elements within it. They won't collapse with margins of elements outside of the block formatting context.
- 2 It contains all floated elements within it.
- 3 It doesn't overlap with floated elements outside the BFC.

Put simply, the contents inside a block formatting context will not overlap or interact with elements on the outside as you would normally expect. If you apply `clear` to an element, it'll only clear floats within its own BFC. And, if you force an element to have a new BFC, it won't overlap with other BFCs.

You can establish a new block formatting context in several ways. Applying any of the following property values to an element triggers one:

- `float: left` or `float: right`—anything but none
- `overflow: hidden`, `auto`, or `scroll`—anything but visible
- `display: inline-block`, `table-cell`, `table-caption`, `flex`, `inline-flex`, `grid`, or `inline-grid`—these are called *block containers*.
- `position: absolute` or `position: fixed`

NOTE The page's root element also creates a top-level block formatting context for the page.

4.4.2 Using a block formatting context for media object layouts

Once each media-body establishes its own block formatting context, your page will have the layout you want (figure 4.15). The best way to do this is often to set an overflow value, either `hidden` or `auto`.

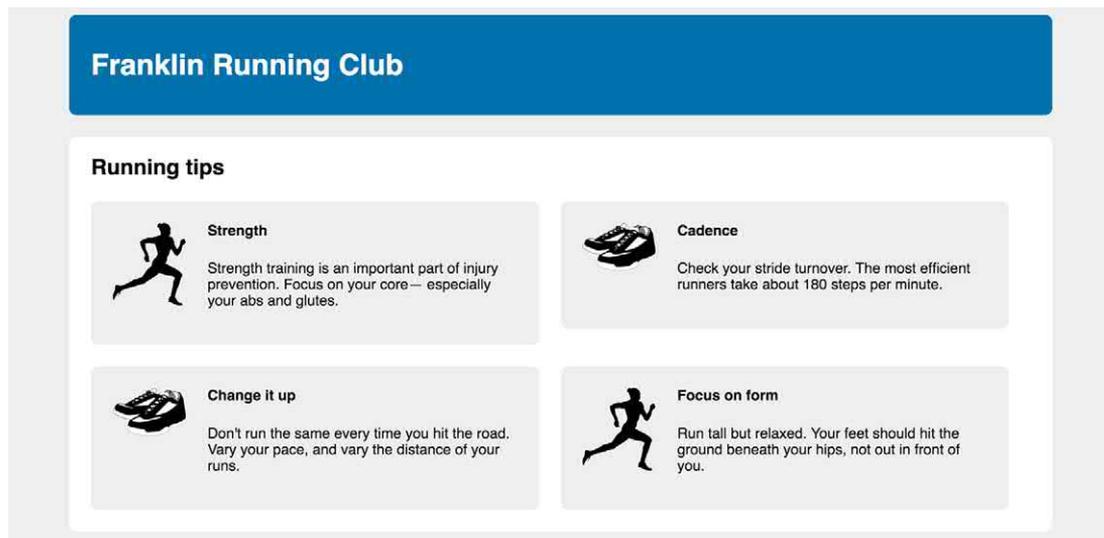


Figure 4.15 Block formatting contexts applied to all media bodies

Let's set the overflow value in your stylesheet. Update the corresponding portion of your stylesheet to match this listing.

Listing 4.11 Adding overflow auto triggers a new block formatting context

```

.media {
  float: left;
  margin: 0 1.5em 1.5em 0;
  width: calc(50% - 1.5em);
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}

.media:nth-child(odd) {
  clear: left;
}

.media-image {
  float: left;
  margin-right: 1.5em;
}

.media-body {
  overflow: auto;
  margin-top: 0;
}

.media-body h4 {
  margin-top: 0;
}

```

← Adds a margin to the image to insert space between it and the body

← Establishes a new block formatting context so the body doesn't overlap the floated image

Using `overflow: auto` for the BFC is generally the simplest approach. You can use instead the other properties mentioned earlier, but some have considerations to take into account: A float or an inline-block will grow to 100% width, so you'd need to restrict the width of the element to prevent it from line wrapping below the float. On the contrary, a table-cell element will only grow enough to contain its contents, so you may need to set a large width to force it to fill the remaining space.

NOTE In some circumstances, the contents from one block formatting context may still overlap the contents of another. This will happen if the contents overflow the container (for example, the content is too wide) or if negative margins pull the contents outside the container.

For more on the media object, read Nicole Sullivan's seminal post about it at <http://mng.bz/6wj3w>. This post gets into a methodology called Object-Oriented CSS (OOCSS), which we'll look at in more depth in chapter 9.

4.5 Grid systems

You've now built your full page layout, but it's not without faults. Most notably, you haven't set yourself up to easily reuse parts of the styles. You've coded the media objects to have a width of 50%, so they're always in rows of two. What if later you want to use the same design in rows of three instead?

One popular way to facilitate code reuse is with the help of a *grid system*. This is a series of class names you can add to your markup to structure portions of the page into rows and columns. It should provide no visual styles, like colors or borders, to the page—it should only set widths and positions of containers. Inside each of these containers, you can add new elements to visually style however you want.

Most of the popular *CSS frameworks* include a grid system of some sort. Details vary from one to another, but usually the general principle is the same: put a row container around one or more column containers. The classes applied to the column containers will each determine their respective widths. Let's build our own grid system so you'll have a good sense of how they work, then you can apply it to your page.



CSS framework—A library of prebuilt CSS code that provides styles for patterns common in web development. These can be useful for rapid prototyping or providing a solid starting point upon which you can build additional styles. Common frameworks include Bootstrap, Foundation, and Pure.

4.5.1 Understanding a grid system

Before you build the grid system, let's look at how you can expect it to behave. A *grid system* is usually defined to hold a certain number of columns in each row; this is usually 12, but that can vary. The child elements of a row may have a width anywhere from one column up to 12 columns wide.

Figure 4.16 illustrates a couple of rows in a 12-column grid. The first row has six, 1-column elements and three, 2-column elements. The next row has a 4-column element followed by an 8-column element. Each set adds up to 12 columns, so they'll fill the width of the row.

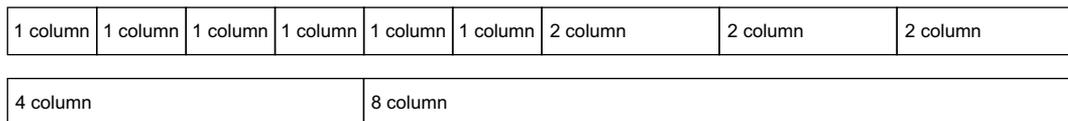


Figure 4.16 Two rows of a 12-column grid system: child elements of each row can be any width from 1 to 12 columns.

Twelve is a good number of columns because it is divisible by two, three, four, and six, which provides a lot of flexibility. This makes it easy to do a three-column layout (three, 4-column elements) or a four-column layout (four, 3-column elements). You can also build asymmetrical layouts, such as a 9-column main element and a 3-column sidebar. Inside each column element, you can place whatever markup you need.

The markup for this example is straightforward. Each row has a row container `div` and inside that you'll place a `div` for each column element with a `column-n` class (where n is the number of columns across the grid):

```
<div class="row">
  <div class="column-4">4 column</div>
  <div class="column-8">8 column</div>
</div>
```

4.5.2 Building a grid system

Let's convert your page so that it uses a grid system. This will prove to be a little more verbose than our previous approach to the page, but the tradeoff for more reusable CSS will be worth it. Edit your HTML to match this listing.

Listing 4.12 HTML restructured to use a grid system

```
<main class="main clearfix">
  <h2>Running tips</h2>
```

```
  <div class="row">
    <div class="column-6">
      <div class="media">
        
        <div class="media-body">
          <h4>Strength</h4>
          <p>
            Strength training is an important part of
            injury prevention. Focus on your core&mdash;
            especially your abs and glutes.
          </p>
        </div>
      </div>
    </div>

    <div class="column-6">
      <div class="media">
        
        <div class="media-body">
          <h4>Cadence</h4>
          <p>
            Check your stride turnover. The most efficient
            runners take about 180 steps per minute.
          </p>
        </div>
      </div>
    </div>
  </div>

  <div class="row">
    <div class="column-6">
      <div class="media">
        
```

Adds a row around each set of two media objects

Adds a column-6 around each media object, placing each media object in its own column

Closes the first row before opening the second

```

<div class="media-body">
  <h4>Change it up</h4>
  <p>
    Don't run the same every time you hit the
    road. Vary your pace, and vary the distance
    of your runs.
  </p>
</div>
</div>
</div>

<div class="column-6">
  <div class="media">
    
    <div class="media-body">
      <h4>Focus on form</h4>
      <p>
        Run tall but relaxed. Your feet should hit
        the ground beneath your hips, not out in
        front of you.
      </p>
    </div>
  </div>
</div>
</div>
</main>

```

← Adds a column-6 around each media object, placing each media object in its own column

This listing gives you a row around each set of two media objects. Inside that you've wrapped each media object within its own 6-column container.

Let's add the styles to lay out the grid. First, you'll define the row class. Add this code to your stylesheet.

Listing 4.13 CSS for the grid rows

```

.row::after {
  content: " ";
  display: block;
  clear: both;
}

```

Replicates the clearfix so the row contains its floated columns

This is nothing more than a clearfix. You add it here so you don't need to add a clearfix class every time you use a row. You'll add more to this in a bit, but fundamentally all the row does is provide a wrapper to contain the columns, and that's what this clearfix does for you.

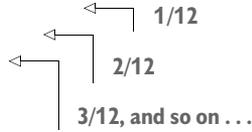
Next, you'll add the initial styles for the columns. This is where the "heavy lifting" takes place, but as you'll see, it's not too complicated. You'll float all the columns to the left and specify widths for each column value. Add this listing to your stylesheet.

Listing 4.14 CSS for the grid columns

```
[class*="column-"] {
  float: left;
}

.column-1 { width: 8.3333%; }
.column-2 { width: 16.6667%; }
.column-3 { width: 25%; }
.column-4 { width: 33.3333%; }
.column-5 { width: 41.6667%; }
.column-6 { width: 50%; }
.column-7 { width: 58.3333%; }
.column-8 { width: 66.6667%; }
.column-9 { width: 75%; }
.column-10 { width: 83.3333%; }
.column-11 { width: 91.6667%; }
.column-12 { width: 100%; }
```

Targets all elements with a class attribute that includes "column-"



The first selector here may be new to you. It's an *attribute selector*, targeting elements based on their class attribute. This allows you to do something a little more complex than what you can do with a normal class selector. The `*=` comparator specifies any values that include the substring specified: any elements with `column-` anywhere within the class attribute. This targets `<div class="column-2">` as well as `<div class="column-6">`. In short, it targets any of your column classes. Now all columns, regardless of their size, will float left. See appendix A for more on attribute selectors.

NOTE This attribute selector casts a wider net than you'll need, as it also targets something like a `<div class="column-header">`. Keep this in mind as you write more styles. For our purposes, it'd be best to consider "column" in a class name as a sort of reserved word from here on out so you don't collide with these rules.

After applying `float: left` to all columns, you then target each one individually, specifying their widths. This may take a little work on a calculator to get each value: the desired number of columns divided by the total number of columns (12). Be sure to keep at least a few decimal points of precision to avoid rounding errors.

At this stage, you've got the basics of a grid system in place. Your page should look like figure 4.17. It looks a bit broken at this point because the media objects still have some styles that are duplicating the work done by the grid system.

You can now simplify the media object. It doesn't need to float left anymore, as the grid column does that for you. It also doesn't need a width; without one, it'll naturally fill 100% of its container. The container is a column-6 element, which is the size you want. You can also remove the margins and the *n*th-child selector that clears each row. Then your page should match figure 4.18.



Figure 4.17 With the grid system applied, the media objects no longer need some of their styles.

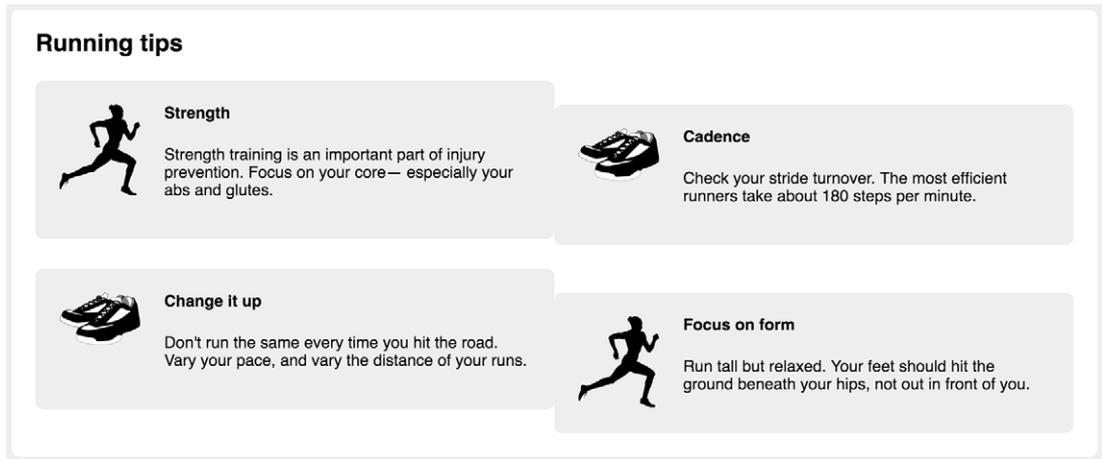


Figure 4.18 Removing position-related properties from the media object allows the grid system to position and size them accordingly.

After deleting these portions, all your styles for the media object should match those in the next listing.

Listing 4.15 Removing positioning and sizing declarations from the media object

```
.media {
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}
```

Deleted float,
margin, and width
declarations

Deleted the `.media:nth-child(odd)` ruleset with the `clear: left` declaration



```
.media-image {
  float: left;
  margin-right: 1.5em;
}

.media-body {
  overflow: auto;
  margin-top: 0;
}

.media-body h4 {
  margin-top: 0;
}
```

Because you removed all margins from the media object, including the bottom margin, there's no longer a gap below the last row of media objects and the bottom of their container. Let's use a padding on the container to bring that back.

Listing 4.16 Adding a bottom padding to the main container

```
.main {
  padding: 0 1.5em 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

← Adds a 1.5 em bottom padding to match the left and right padding

Now you're close to the final design, although there are a few more details to put into place.

4.5.3 Adding gutters

One thing your grid system still lacks is a gutter between each column. Let's add those next, along with a couple other details. After you finish, your page should look like figure 4.19.

You can create gutters by adding left and right padding to each grid column. By adding this to the grid system instead of individual components, like the media object, you'll be able to re-use the grid over and over on other pages without worrying about gutters again.

Because you want a gutter size of 1.5 em, you can divide that in half and then put half on the left and half on the right of each column element. Update your grid styles to match the next listing. This also removes the top margin from all columns, overriding the lobotomized owl again.

Listing 4.17 Adding gutters to the grid system

```
[class*="column-"] {
  float: left;
  padding: 0 0.75em;
  margin-top: 0;
}
```

← Adds .75 em left and right padding to each column element

← Removes top margins from columns

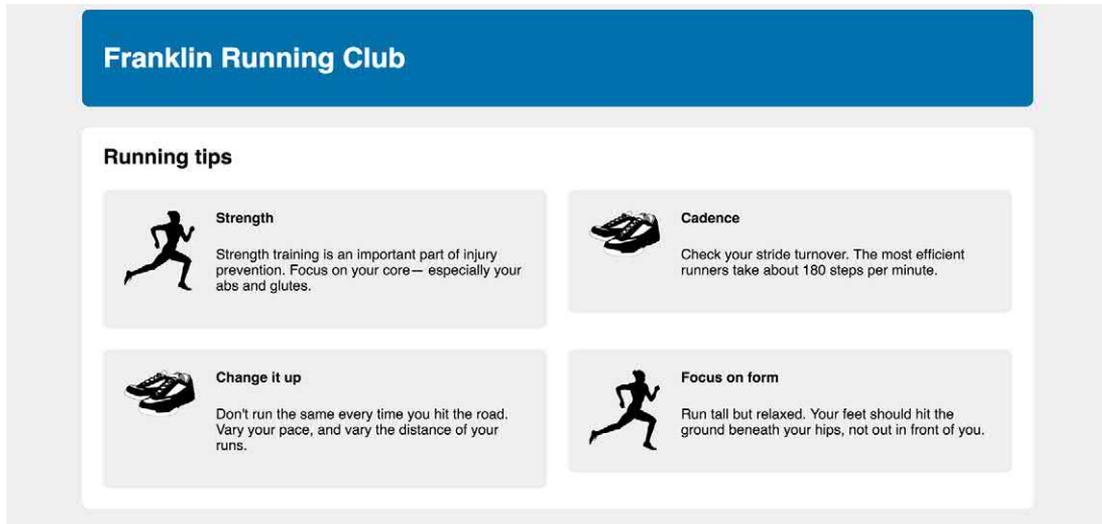


Figure 4.19 Complete page with fully working grid system

Now the grid column elements will always have a nice 1.5 em gutter between them, and things look pretty good. This code does, however, introduce a slight misalignment between a grid column and content outside the grid row. Figure 4.20 shows where this occurs on the page: the left edge of the page title (“Running tips”) should align with the edge of the media object in the first column. Instead, the column padding pushes the gray box of the media object a little to the right.

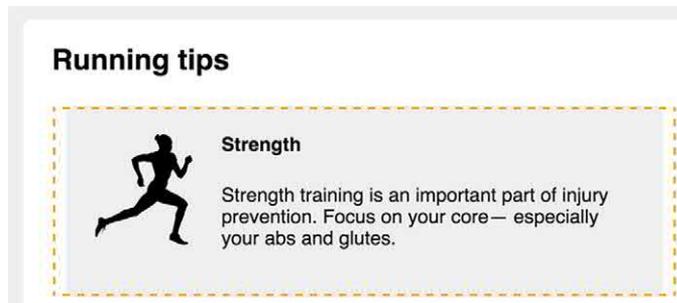


Figure 4.20 The left side of the page title aligns with the edge of the column element (dashed line); instead, it should align with the contents inside the column.

You could fix this by removing the left padding from the first column in each row and right padding from the last column in each row; but instead of applying a bunch of specific rules for that, let’s adjust the width of the row.

You can stretch the row to be a little wider using negative margins. By applying a -0.75 em left margin to the row, the negative margin pulls the row left outside its container. After doing this, the padding of the column will push its contents 0.75 em back to the right, making the first column align with the title (figure 4.21). Applying a negative margin on the right as well will do the same on the right side.

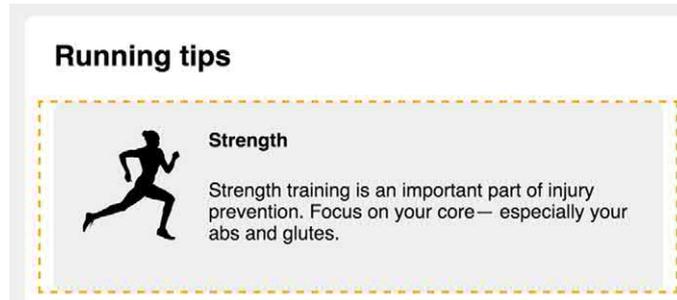


Figure 4.21 Applying a negative margin to the row pulls it left, compensating for the column's padding. Its contents now align with the page title.

The code for this is shown in listing 4.18. No matter where you place a row, it'll now be stretched to 1.5 em wider than its container; padding on the columns will then shift their contents back to align with the edges of the outer container. This is effectively a modified version of the double container pattern, where the row is the inner container inside its wrapper.

Listing 4.18 Adding negative margins to the grid row

```
.row {
  margin-left: -0.75em;
  margin-right: -0.75em;
}
```

You now have a fully working grid system, powered by floats. Whether you use this one, or one you find in a CSS framework, you should now have an understanding of how it does what it does. Your full stylesheet should now match the following listing.

Listing 4.19 Complete stylesheet

```
:root {
  box-sizing: border-box;
}

*,
::before,
::after {
```

```
    box-sizing: inherit;
  }

body {
  background-color: #eee;
  font-family: Helvetica, Arial, sans-serif;
}

body * + * {
  margin-top: 1.5em;
}

.row {
  margin-left: -0.75em;
  margin-right: -0.75em;
}

.row::after {
  content: " ";
  display: block;
  clear: both;
}

[class*="column-"] {
  float: left;
  padding: 0 0.75em;
  margin-top: 0;
}

.column-1 { width: 8.3333%; }
.column-2 { width: 16.6667%; }
.column-3 { width: 25%; }
.column-4 { width: 33.3333%; }
.column-5 { width: 41.6667%; }
.column-6 { width: 50%; }
.column-7 { width: 58.3333%; }
.column-8 { width: 66.6667%; }
.column-9 { width: 75%; }
.column-10 { width: 83.3333%; }
.column-11 { width: 91.6667%; }
.column-12 { width: 100%; }

header {
  padding: 1em 1.5em;
  color: #fff;
  background-color: #0072b0;
  border-radius: .5em;
  margin-bottom: 1.5em;
}

.main {
  padding: 0 1.5em 1.5em;
  background-color: #fff;
  border-radius: .5em;
}
```

```
.container {
  max-width: 1080px;
  margin: 0 auto;
}

.media {
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}

.media-image {
  float: left;
  margin-right: 1.5em;
}

.media-body {
  overflow: auto;
  margin-top: 0;
}

.media-body h4 {
  margin-top: 0;
}

.clearfix::before,
.clearfix::after {
  display: table;
  content: " ";
}

.clearfix::after {
  clear: both;
}
```

You've now laid out a page using floats entirely. They have their quirks, but they get the job done. With a deeper understanding of their behavior, you'll hopefully not find floats too intimidating. As I mentioned earlier, there are more easy-to-understand alternatives to float-based layouts. We'll take a look at those in the next two chapters.

Summary

- Floats exist to allow text to wrap around an element—but that's not often the effect you want.
- Use a clearfix to contain floated elements.
- Understand the three tricks of a block formatting context: containing floats, preventing margin collapse, and preventing document flow from wrapping around a floated element.
- Use the double container pattern to center page contents.
- Use the media object pattern to position descriptive text alongside an image.
- Use a grid system to create a wide array of page layouts.

5 *Flexbox*

This chapter covers

- Flex containers and flex items
- Main axis and cross axis
- Element sizes in flexbox
- Element alignment in flexbox

If you've been in the CSS world in the past few years, you've almost certainly heard someone sing the praises of flexbox. Flexbox—formally Flexible Box Layout—is a new method for laying out elements on the page. It's more predictable and offers far more specific control than floats. It's also a simple solution to the long-standing problems of vertical centering and equal height columns.

Flexbox has been on the horizon for several years, and developers who only need to support cutting-edge browsers have been using it for a little while. But now we've reached a point where it's supported in all major browsers, including partial support in IE10. In fact, it has broader support than even the `border-radius` property (which isn't supported in Opera Mini). If you've been waiting for the right time to learn flexbox, that time has arrived. This chapter will get you acquainted.

If flexbox has one weakness, it's the overwhelming number of options it provides. It introduces 12 new properties to CSS, including some shorthand proper-

ties. That can be a lot to take in at once. When I first started learning flexbox, it felt a bit like drinking from a fire hose, and I had a hard time committing all the new properties to memory. I'm going to take a different approach in teaching you about flexbox—we'll ease into it.

I'll cover a few basic principles of the flexbox layout that you'll need to understand, followed by practical examples. You don't need to learn all 12 new properties in order to use flexbox. I've found that only a few of them do most of the heavy lifting, so we'll focus on those. The rest of the properties provide options for aligning and spacing elements. Near the end of the chapter, I'll explain those and provide a quick reference guide that you can return to when you need it.

5.1 Flexbox principles

Flexbox begins with the familiar `display` property. Applying `display: flex` to an element turns it into a *flex container*, and its direct children turn into *flex items*. By default, flex items align side by side, left to right, all in one row. The flex container fills the available width like a block element, but the flex items may not necessarily fill the width of their flex container. The flex items are all the same height, determined naturally by their contents.

TIP You can also use `display: inline-flex`. This creates a flex container that behaves more like an inline-block element rather than a block. It flows inline with other inline elements, but it won't automatically grow to 100% width. Flex items within it generally behave the same as with `display: flex`. Practically speaking, you won't need to use this very often.

Flexbox is unlike previous `display` values (`inline`, `inline-block`, and so on), which only affect the elements they are applied to. Instead, a flex container asserts control over the layout of the elements within. A flex container and its items are illustrated in figure 5.1.

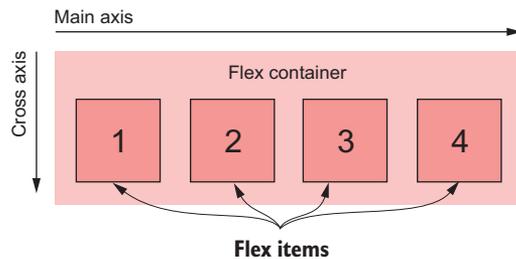


Figure 5.1 A flexbox container and its elements

The items are placed along a line called the *main axis*, which goes from the *main-start* (left) to the *main-end* (right). Perpendicular to the main axis is the *cross axis*. This goes from the *cross-start* (top) to the *cross-end* (bottom). The direction of these axes can be changed; I'll show you how to do this later in the chapter.

NOTE Because flexbox layout is defined in terms of the main axis and cross axis, I'll use the terms *start* and *end* in reference to the axes, rather than *left* and *right* or *top* and *bottom*.

These concepts (flex container, flex items, and the two axes) cover a lot of what you need to know about flexbox. Applying `display: flex` gets you pretty far before you'll need to pick up any of those 12 new properties. To try it out, you'll build the page shown in figure 5.2.

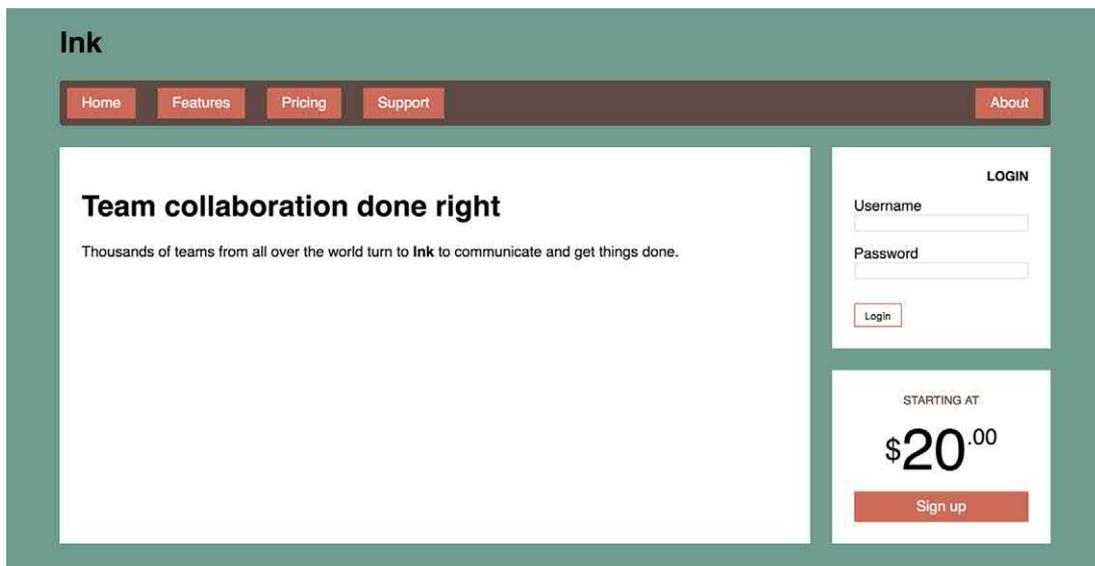


Figure 5.2 Finished page with a flexbox layout

I've structured this page to cover a number of ways to use flexbox. We'll use flexbox for the navigational menu across the top and to lay out the three white boxes and the stylistic \$20.00 text on the bottom right.

Start a new page and link it to a new stylesheet. Then add this markup to your page.

Listing 5.1 Markup for the page

```
<!doctype html>
<head>
  <title>Flexbox example page</title>
  <link href="styles.css" rel="stylesheet"
    type="text/css" />
</head>
<body>
  <div class="container">
    <header>
```

```

    <h1>Ink</h1>
  </header>
  <nav>
    <ul class="site-nav">
      <li><a href="/">Home</a></li>
      <li><a href="/features">Features</a></li>
      <li><a href="/pricing">Pricing</a></li>
      <li><a href="/support">Support</a></li>
      <li class="nav-right">
        <a href="/about">About</a>
      </li>
    </ul>
  </nav>

  <main class="flex">
    <div class="column-main tile">
      <h1>Team collaboration done right</h1>
      <p>Thousands of teams from all over the
      world turn to <b>Ink</b> to communicate
      and get things done.</p>
    </div>

    <div class="column-sidebar">
      <div class="tile">
        <form class="login-form">
          <h3>Login</h3>
          <p>
            <label for="username">Username</label>
            <input id="username" type="text"
              name="username"/>
          </p>
          <p>
            <label for="password">Password</label>
            <input id="password" type="password"
              name="password"/>
          </p>
          <button type="submit">Login</button>
        </form>
      </div>
      <div class="tile centered">
        <small>Starting at</small>
        <div class="cost">
          <span class="cost-currency">$</span>
          <span class="cost-dollars">20</span>
          <span class="cost-cents">.00</span>
        </div>
        <a class="cta-button" href="/pricing">
          Sign up
        </a>
      </div>
    </div>
  </main>
</div>
</body>

```

Navigation menu

Large main tile

← Sidebar containing two stacked tiles

To get your stylesheet started, enter this CSS. (Hopefully, these styles are becoming familiar by now.)

Listing 5.2 Base styles for the page

<pre> :root { box-sizing: border-box; } *, ::before, ::after { box-sizing: inherit; } body { background-color: #709b90; font-family: Helvetica, Arial, sans-serif; } body * + * { margin-top: 1.5em; } .container { max-width: 1080px; margin: 0 auto; } </pre>	<p>Global box-sizing fix (chapter 3)</p>
<pre> body * + * { margin-top: 1.5em; } </pre>	<p>Global margins (chapter 3)</p>
<pre> .container { max-width: 1080px; margin: 0 auto; } </pre>	<p>Double-container to center page contents (chapter 4)</p>
<pre> body { background-color: #709b90; font-family: Helvetica, Arial, sans-serif; } </pre>	<p>Sets green background color and sans-serif font for the page</p>

Now that the page is started, let's start laying out some things with flexbox. You'll start with the navigational menu at the top.

5.1.1 Building a basic flexbox menu

For this example, you'll want the navigational menu to look like figure 5.3. Most of the menu items will align to the left, but you'll pull one over to the right side.

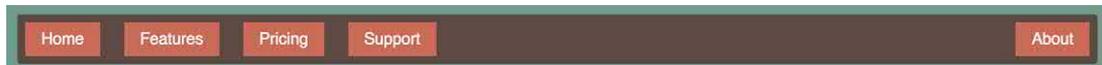


Figure 5.3 Navigational menu with items laid out using flexbox

To build this menu, you should consider which element needs to be the flex container; keep in mind that its child elements will become the flex items. In the case of our page menu, the flex container should be the unordered list (). Its children, the list items (), will be the flex items. Here's what this looks like:

```

<ul class="site-nav">
  <li><a href="/">Home</a></li>
  <li><a href="/features">Features</a></li>

```

```

<li><a href="/pricing">Pricing</a></li>
<li><a href="/support">Support</a></li>
<li class="nav-right"><a href="/about">About</a></li>
</ul>

```

We'll take a few passes at this as I walk you through building this menu step by step. First, you'll apply `display: flex` to the list. You'll also need to override the default list styles from the user agent stylesheet and the lobotomized owl top margins. You'll also apply the colors. Figure 5.4 shows the result for these steps.



Figure 5.4 Menu with flexbox and colors applied

In the markup, you've given the `` a `site-nav` class, which you can then use to target it in the styles. Add these declarations to your stylesheet.

Listing 5.3 Applying flexbox and colors to the menu

```

.site-nav {
  display: flex;
  padding-left: 0;
  list-style-type: none;
  background-color: #5f4b44;
}

```

Makes `site-nav` the flex container and its children the flex items

```

.site-nav > li {
  margin-top: 0;
}

```

Removes the left padding and list bullets in the user agent styles

```

.site-nav > li > a {
  background-color: #cc6b5a;
  color: white;
  text-decoration: none;
}

```

Overrides the lobotomized owl top margin

Removes the underline from link text in the user agent styles

Note that you're working with three levels of elements here: the `site-nav` list (the flex container), the list items (the flex items), and the anchor tags (the links) within them. I've used direct descendant combinators (`>`) to ensure you only target direct child elements. This is probably not strictly necessary; that is, it's unlikely future changes will add a nested list inside the navigational menu, but it doesn't hurt to play it safe. If you're not familiar with this combinator, see appendix A for more information.

Vendor prefixes

If you happen to use flexbox in an older browser, such as IE10 or Safari 8, you'll find that it doesn't work. That's because older browsers require *vendor prefixes* on flexbox attributes. This is how browsers have supported several new CSS features before the specification was stable. Instead of implementing `display: flex`, for example, older versions of Safari implemented `display: -webkit-flex`. You'll need to add this declaration for flexbox to work in Safari 8, followed by the normal one:

```
.site-nav {  
  display: -webkit-flex;  
  display: flex;  
}
```

A browser ignores declarations it doesn't understand, so in Safari 8, the cascaded value will be `-webkit-flex`, which behaves the same as `flex` in more recent versions. The same goes for property names as well as values. For example, you'll need to declare the `flex` property (which I'll cover later in this chapter) like this:

```
-webkit-flex: 1;  
flex: 1;
```

For IE10, it gets even more complicated, as that browser implements an older version of the flexbox specification. To add support for this version, you'll need to know the older property names (for example, `flexbox` instead of `flex`) and then add prefixed versions of those:

```
display: -ms-flexbox;  
display: -webkit-flex;  
display: flex;
```

This is a lot to keep track of, and adds a lot of repetition to your stylesheet. I *strongly* recommend you automate this process with a tool called Autoprefixer, which is available at <https://github.com/postcss/autoprefixer>. This tool parses your CSS and outputs a new file with all the relevant prefixes added where necessary. It'll also convert to the older flexbox standard for IE10 when necessary. It works with a wide array of build tools, so you can incorporate it into whatever workflow you are comfortable with.

For simplicity, I'm leaving prefixes out of the examples in this chapter. The examples will work in all modern browsers, but when it comes to production-ready code, please run your code through Autoprefixer first so it'll work with a broader range of browser versions.

It's important to know, too, that the concept of prefixes is going away. All major browsers have switched (or are in the process of switching) to a new method of supporting upcoming, unstable features behind an "experimental features" option. I'll walk you through this in chapter 6.

5.1.2 Adding padding and spacing

Our menu looks rather scrawny at this point. Let's flesh it out a bit with some padding. You'll add padding to both the container and to the menu links. After this step, your menu will look like figure 5.5.

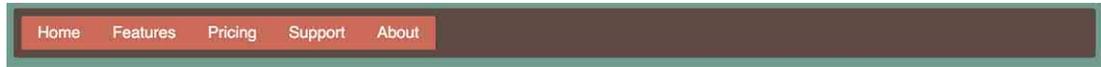


Figure 5.5 Menu with padding and link styles added

If you're not too familiar with building this sort of menu (whether with flexbox or any other layout method), it's important to note how to do this. In the examples, you'll apply the menu item padding to the internal `<a>` elements, not the `` elements. You'll need the entire area that looks like a menu link to behave like a link when the user clicks it. Because the link behavior comes from clicking the `<a>` element, you don't want to turn the `` into a big nice-looking button, but only have a small clickable target area (the `<a>`) inside it.

Update your styles to match those in this listing. This will fill out the padding of the menu.

Listing 5.4 Adding padding to the menu and its links

```
.site-nav {
  display: flex;
  padding: .5em;
  background-color: #5f4b44;
  list-style-type: none;
  border-radius: .2em;
}

.site-nav > li {
  margin-top: 0;
}

.site-nav > li > a {
  display: block;
  padding: .5em 1em;
  background-color: #cc6b5a;
  color: white;
  text-decoration: none;
}
```

← Adds padding to menu, outside of the links

← Makes links block level so they add to the parent elements' height

← Adds padding inside the links

You'll notice you made the links a display block. If they were to remain inline, the height they'd contribute to their parent would be derived from their line height—not their padding and content, which is the behavior you want for this page. You also applied a little more horizontal padding than vertical, which is generally more aesthetically pleasing.

Next, you'll need to add space between the menu items. Regular old margins will do the trick. Even better, flexbox allows you to use `margin: auto` to fill available space between the flex items. You can also use this to move the final menu item to the right side. After applying the margins, the menu will be complete (figure 5.6).



Figure 5.6 Margins apply spacing between flex items

See the styles for this in listing 5.5, where you'll apply a margin between each item, but not to the outside edges. You can achieve this layout by using the `margin-left` property and an adjacent sibling combinator, which is a method similar to the lobotomized owl from chapter 3. You'll also apply an `auto` left margin to the last button, which causes the margin to fill all the available space, pushing the last button all the way to the right. Add this listing to your stylesheet.

Listing 5.5 Using margins to space the items

```
.site-nav > li + li {
  margin-left: 1.5em;
}

.site-nav > .nav-right {
  margin-left: auto;
}
```

← Targets every list item that follows another list item (that is, all but the first)

← Auto margins inside a flexbox will fill the available space.

You applied the `auto` margin to only one element (About). You could apply it to the Support menu item instead to shift both it and the About item to the right.

Margins work well here because you want different spacing between these items. If you wanted equal spacing between the items, the `justify-content` property would be a better approach. I'll come back to this in a bit.

5.2 Flex item sizes

The previous listing used margins for spacing between the flex items. To define their size, you could use the familiar `width` and `height` properties, but flexbox provides more options for sizing and spacing than the familiar `margin`, `width`, and `height` properties alone can accomplish. Let's look at one of the more useful flexbox properties, `flex`.

The `flex` property controls the size of the flex items along the main axis (that is, the width). In listing 5.6, you'll apply a flex layout to the main area of the page, then you'll use the `flex` property to control the size of the columns. Initially, your main area will look like figure 5.7.

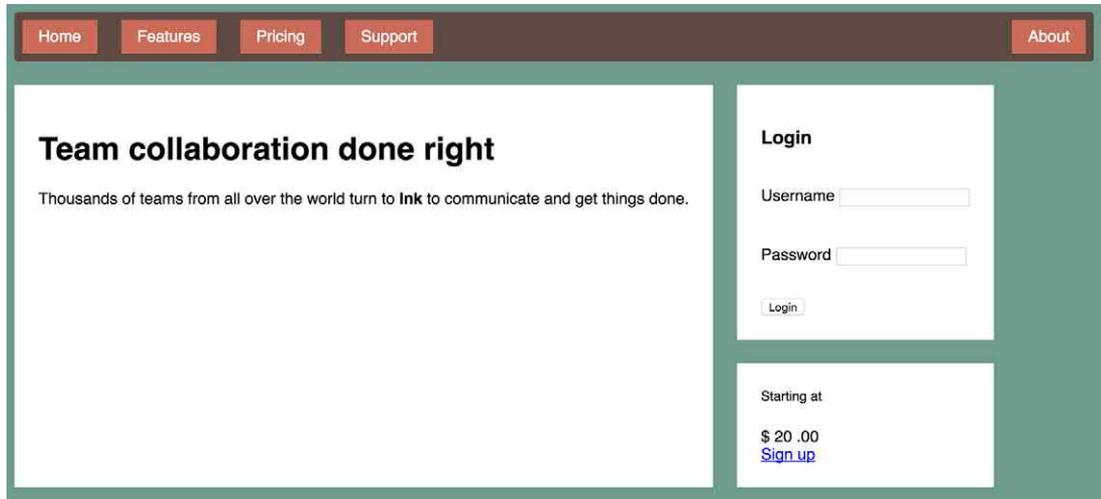


Figure 5.7 Main area with a flex layout applied

Add the styles in the next listing to your stylesheet. This listing provides a white background to the three tiles via the `tile` class and a flex layout to the `<main>` element by targeting the `flex` class.

Listing 5.6 Applying flexbox to the main container

<pre><code>.tile { padding: 1.5em; background-color: #fff; }</code></pre>	<p>Adds a background color and padding to the three tiles</p>
<pre><code>.flex { display: flex; }</code></pre>	<p>Applies a flexbox layout to the main container</p>
<pre><code>.flex > * + * { margin-top: 0; margin-left: 1.5em; }</code></pre>	<p>Removes the top margin and applies space between the flex items</p>

Now your content is divided into two columns: on the left is the larger area for the primary content of the page, and on the right is a login form and a small pricing box. You haven't done anything yet to specify the width of the two columns, so they'll size themselves naturally, based on their content. On my screen (figure 5.7), this means they don't quite fill the width of the available space, though this isn't necessarily the case with a smaller window size.

NOTE When it comes to CSS, it’s important to consider not only the specific content you have on the page now, but also what will happen as that content changes, or as the stylesheet is applied to similar pages. You need to decide how you want things like these two columns to behave under various circumstances.

The `flex` property, which is applied to the flex items, gives you a number of options. Let’s apply the most basic use case first to get familiar with it. You’ll use the `column-main` and `column-sidebar` classes to target the columns, using `flex` to apply widths of two-thirds and one-third. Add the following to your stylesheet.

Listing 5.7 Using the `flex` property to set column widths

```
.column-main {  
  flex: 2;  
}  
  
.column-sidebar {  
  flex: 1;  
}
```

Now the two columns grow to fill the space, so together they are the same width as the nav bar, with the main column twice as wide as the sidebar. Flexbox was kind enough to take care of the math for you. Let’s take a closer look at what’s going on.

The `flex` property is shorthand for three different sizing properties: `flex-grow`, `flex-shrink`, and `flex-basis`. In this listing, you’ve only supplied `flex-grow`, leaving the other two properties to their default values (1 and 0% respectively). So `flex: 2` is equivalent to `flex: 2 1 0%`. These shorthand declarations are generally preferred, but you can also declare the three individually:

```
flex-grow: 2;  
flex-shrink: 1;  
flex-basis: 0%;
```

Let’s look at what these three properties mean, one at a time. We’ll start with `flex-basis`, as the other two are based on it.

5.2.1 Using the *flex-basis* property

The *flex basis* defines a sort of starting point for the size of an element—an initial “main size.” The `flex-basis` property can be set to any value that would apply to width, including values in px, ems, or percentages. Its initial value is `auto`, which means the browser will look to see if the element has a width declared. If so, the browser uses that size; if not, it determines the element’s size naturally by the contents. This means that width will be ignored for elements that have any flex basis other than `auto`. Figure 5.8 illustrates this.



Figure 5.8 Three flex items with a flex basis of 20%, giving each an initial main size (width) of 20%

Once this initial main size is established for each flex item, they may need to grow or shrink in order to fit (or fill) the flex container along the main axis. That’s where `flex-grow` and `flex-shrink` come in.

5.2.2 Using `flex-grow`

Once `flex-basis` is computed for each flex item, they (plus any margins between them) will add up to some width. This width may not necessarily fill the width of the flex container, leaving a remainder (figure 5.8).

The remaining space (or remainder) will be consumed by the flex items based on their `flex-grow` values, which is always specified as a non-negative integer. If an item has a `flex-grow` of 0, it won’t grow past its flex basis. If any items have a non-zero growth factor, those items will grow until all of the remaining space is used up. This means the flex items will fill the width of the container (figure 5.9).

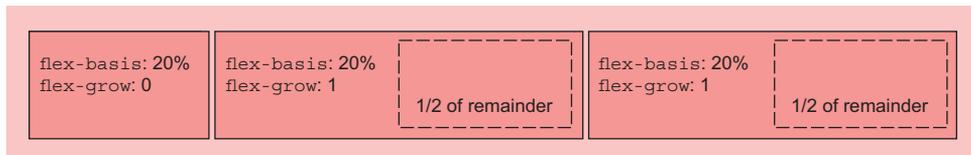


Figure 5.9 Remaining width partitioned evenly among items with equal `flex-grow` values

Declaring a higher `flex-grow` value gives that element more “weight”; it’ll take a larger portion of the remainder. An item with `flex-grow: 2` will grow twice as much as an item with `flex-grow: 1` (figure 5.10).

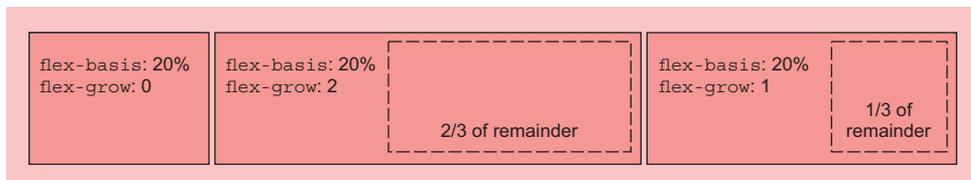


Figure 5.10 Items with a higher `flex-grow` value consume a higher proportion of the remaining available width.

This is what you did on your page. The shorthand declarations `flex: 2` and `flex: 1` set a flex basis of 0%, so 100% of the container's width is the remainder (minus the 1.5 em margin between the two columns). The remainder is then distributed to the two columns: two-thirds to the first column and the remaining third to the second (figure 5.11).

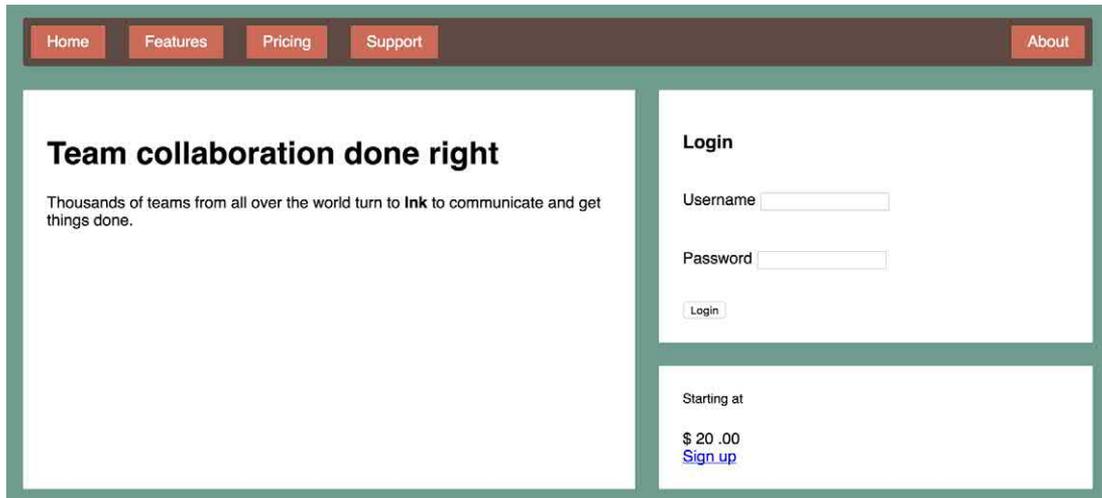


Figure 5.11 The two columns fill the flex container's width.

TIP Favor the use of the shorthand `flex` property instead of individually declaring `flex-grow`, `flex-shrink`, or `flex-basis`. Unlike most shorthand properties, these aren't set to their initial values when omitted. Instead, the shorthand assigns useful default values for any of the three that you omit: `flex-grow` of 1, `flex-shrink` of 1, and a `flex-basis` of 0%. These are most commonly what you'll need.

5.2.3 Using *flex-shrink*

The `flex-shrink` property follows similar principles as `flex-grow`. After determining the initial main size of the flex items, they could exceed the size available in the flex container. Without `flex-shrink`, this would result in an overflow (figure 5.12).



Figure 5.12 Flex items can have an initial size exceeding that of the flex container.

The `flex-shrink` value for each item indicates whether it should shrink to prevent overflow. If an item has a value of `flex-shrink: 0`, it will not shrink. Items with a value greater than 0 will shrink until there is no overflow. An item with a higher value will shrink more than an item with a lower value, proportional to the `flex-shrink` values.

As an alternate approach to your page, you could achieve the two columns sizing by relying on `flex-shrink`. To do this, specify the flex basis for each column using the desired percent (66.67% and 33.33%). The width plus the 1.5 em gutter would overflow by 1.5 em. Give both columns a `flex-shrink` of 1, and 0.75 em is subtracted from the width of each, allowing them to fit in the container. The following listing shows what this code would look like.

Listing 5.8 Using the `flex` property to set widths

```
.column-main {
  flex: 66.67%;
}

```

← Equivalent to flex:
1 1 66.67%

```
.column-sidebar {
  flex: 33.33%;
}

```

← Equivalent to flex:
1 1 33.33%

This is a different approach to effectively get the same result as before (listing 5.7). Either one suits our purposes for this page.

NOTE If you look at the nitty gritty details, there’s a slight discrepancy between the results of listing 5.7 and listing 5.8. The reason for this is a little complicated, but in short, it’s because the `column-main` has padding but the `column-sidebar` doesn’t. The padding changes the way the initial main size of the flex item is determined when `flex-basis` is 0%. Therefore, the `column-main` from listing 5.7 is 3 em wider than that in listing 5.8—the size of its left and right padding. If you need your measurements to be precise, either ensure the paddings are equal or use the flex basis method shown in listing 5.8.

5.2.4 Some practical uses

You can make use of the `flex` property in countless ways. You can define proportional columns using `flex-grow` values or `flex-basis` percentages as you did on your page. You can define fixed width columns and “fluid” columns that scale with the viewport. You can build a grid system, much like the one you built in chapter 4, using flexbox instead of floats. Figure 5.13 illustrates some of layouts you can build with flexbox.

The third example illustrates part of the “Holy Grail” layout. This is a layout that has been notoriously difficult in CSS. The two sidebars are a fixed width, whereas the

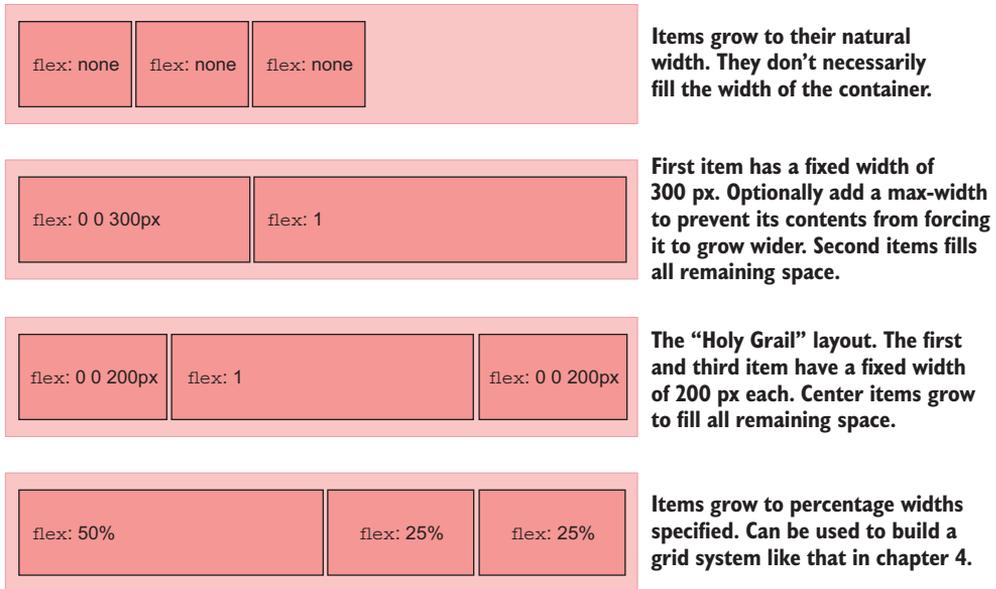


Figure 5.13 Ways you can define item sizes using flex

center column is “fluid,” meaning it will grow to fill the available space. Most notably, all three columns are equal height, as determined by their contents. Although this layout is possible using floats, it requires the use of some obscure and brittle hacks. As you might imagine, you can mix and match these layouts in any number of ways, with any different number of flex items.

5.3 *Flex direction*

Another important option in flexbox is the ability to shift the direction of the axes. The `flex-direction` property, applied to the flex container, controls this. Its initial value (`row`) causes the items to flow left-to-right, as you’ve done. Specifying `flex-direction: column` causes the flex items to stack vertically (top to bottom) instead. Flexbox also supports `row-reverse` to flow items right to left, and `column-reverse` to flow items bottom to top (figure 5.14).

You’ll use this in the right column of the page, where two tiles are stacked atop one another. This may seem unnecessary; after all, the two tiles in our right column are already stacked. Normal block elements behave like this. But there’s a problem with the page layout that isn’t immediately obvious. It shows up if you add more content to the main tile. This is shown in figure 5.15.

Add a few more headings and paragraphs to the `column-main` in your code. You’ll see that the main tile grows beyond the bottom of the tiles on the right. Flexbox is supposed to provide columns of equal height, so why isn’t this working?

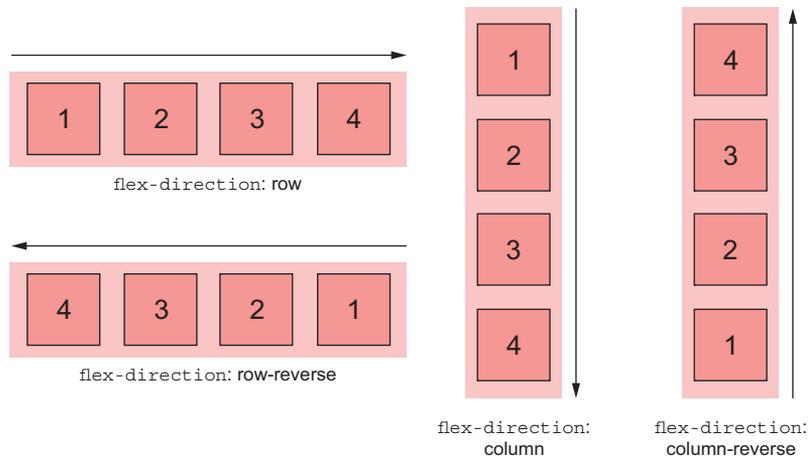


Figure 5.14 Changing the flex direction changes the main axis. The cross axis changes as well, to remain perpendicular to the main axis.

The screenshot shows a web page layout with a main content area on the left and a sidebar on the right. The main content area contains four tiles, each with a heading and a paragraph of text:

- Team collaboration done right**
- Communication around the globe**
- Instant access to your team's documents**
- Intuitive interface**

The sidebar contains a login form and a price display:

- Login** section with fields for Username and Password, and a Login button.
- Starting at \$ 20 .00 with a [Sign up](#) link.

A dashed green outline is drawn around the sidebar area, indicating its height. The main content area is taller than this dashed outline, showing that the main flex item grows beyond the height of the sidebar items.

Figure 5.15 The main tile grows beyond the height of the tiles in the right column (dashed line indicates the size of column-sidebar).

Figure 5.15 shows (by the dashed outline I've added) that the flex items are in fact equal height. The problem is that the tiles inside the right flex item don't grow to fill it.

The ideal layout would be the one shown in figure 5.16. The two tiles on the right grow to fill the column, even when the content on the left is longer. Before flexbox, this effect was impossible to achieve using CSS (though it was possible with a little help from JavaScript).

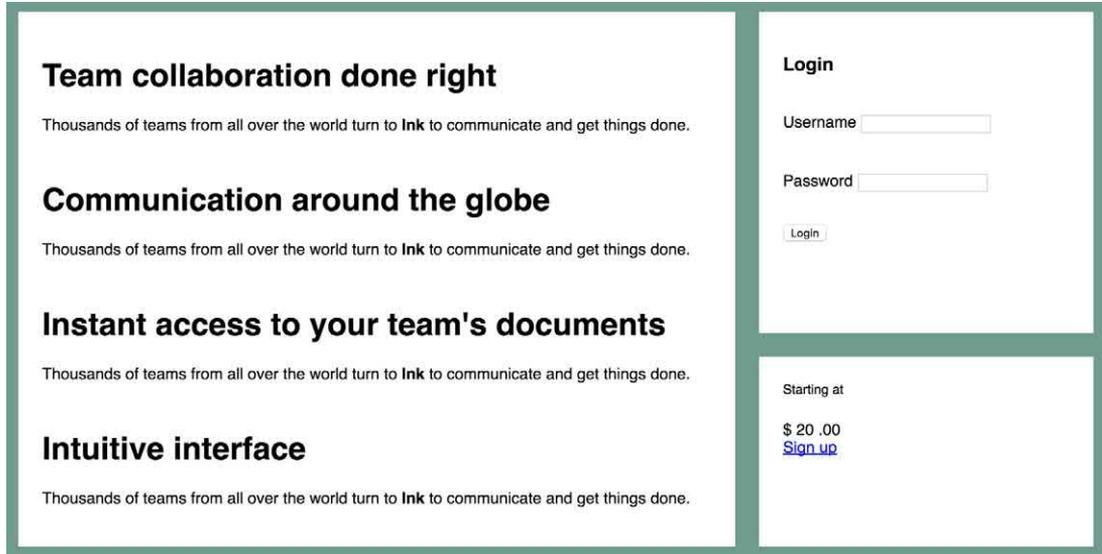


Figure 5.16 The ideal layout: Tiles in the right column align with the large tile on the left.

5.3.1 Changing the flex direction

What you need is for the two columns to grow if necessary to fill the container's height. To do this, turn the right column (the `column-sidebar`) into a flex container with a `flex-direction: column`. Then, apply a non-zero `flex-grow` value to both tiles within. The next listing shows the code for this. Update your stylesheet to match.

Listing 5.9 Creating a flex column on the right

```
.column-sidebar {
  flex: 1;
  display: flex;
  flex-direction: column;
}

.column-sidebar > .tile {
  flex: 1;
}
```

A flex item for the outer flexbox and a flex container for the new inner one

← Applies flex-grow to the items within

You now have *nested flexboxes*. The element `<div class="column-sidebar">` is a flex item for the outer flexbox, and it's the flex container for the inner flexbox. The overall structure of these elements looks like this (with text removed for brevity):

```
<main class="flex">
  <div class="column-main tile">
    ...
  </div>
  <div class="column-sidebar">
    <div class="tile">...</div>
    <div class="tile">...</div>
  </div>
</div>
```

The inner flexbox here has a flex direction of `column`, so the main axis is rotated. It flows from top to bottom (and the cross axis now flows from left to right). This means that for those flex items, `flex-basis`, `flex-grow`, and `flex-shrink` now apply to the element height rather than the width. By specifying `flex: 1`, the height of these items will stretch if necessary to fill the container. Now, regardless of which side is taller, the bottom of the large tile and the bottom of the second smaller tile align.

When working with a vertical flexbox (`column` or `column-reverse`), the same general concepts for rows apply, but there's one difference to keep in mind—in CSS, working with height is fundamentally different than working with widths. A flex container will be 100% the available width, but the height is determined naturally by its contents. This behavior does not change when you rotate the main axis.

The flex container's height is determined by its flex items. They fill it perfectly. In a vertical flexbox, `flex-grow` and `flex-shrink` applied to the items will have no effect unless something else forces the height of the flex container to a specific size. On your page, that “something” is the height derived from the outer flexbox.

5.3.2 Styling the login form

You've now applied the overall layout to the entire page. All that remains is styling the smaller elements in the two tiles on the right: the login form and the signup link. You don't need flexbox for the login form, but for the sake of completeness, I'll walk you through it briefly. Afterwards, the form should look like figure 5.17.

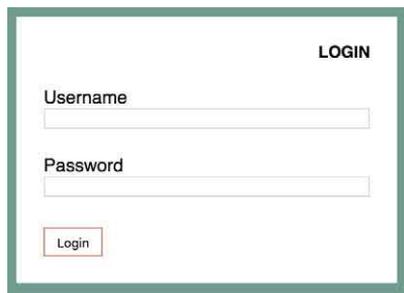


Figure 5.17 Login form

The `<form>` has the class `login-form`, so you'll use that to target it in your CSS. Add the code in this listing to your stylesheet. This will style the login form in three parts: the heading, the input fields, and the button.

Listing 5.10 Login form styles

```
.login-form h3 {
  margin: 0;
  font-size: .9em;
  font-weight: bold;
  text-align: right;
  text-transform: uppercase;
}

.login-form input:not([type=checkbox]):not([type=radio]) {
  display: block;
  width: 100%;
  margin-top: 0;
}

.login-form button {
  margin-top: 1em;
  border: 1px solid #cc6b5a;
  background-color: white;
  padding: .5em 1em;
  cursor: pointer;
}
```

Makes the heading bold, right-aligned, and all caps

Styles all text-like inputs (not checkboxes or radio buttons)

Styles the button

First is the heading, which uses font properties you should be familiar with. You used `text-align` to shift the text to the right and `text-transform` to make the text all uppercase. Notice it wasn't capitalized in the HTML. When capitalization is purely a styling decision, as this is, you would normally capitalize it according to standard grammatical rules in the HTML and use CSS to manipulate it. This way, you can change it in the future without having to retype portions of the HTML in proper caps.

The second ruleset styles the input boxes. The selector here is peculiar, mainly because the `<input>` element is peculiar. The `input` element is used for text inputs and passwords, as well as a number of other HTML5 inputs that look similar, like numbers, emails, and dates. It's also used for form input items that look entirely different; namely, radio buttons and checkboxes.

I've combined the `:not()` pseudo-class with the attribute selectors `[type=checkbox]` and `[type=radio]` (see appendix A for details). This targets all `input` elements except checkboxes and radio buttons. It's a blacklist approach, excluding what I don't want to target. You could alternately use a whitelist approach, using multiple attribute selectors to name every type of input you want to target, but this can get rather long.

NOTE The form for this page only uses a text input and a password input, but it's important for you to consider other markup the CSS could be applied to in the future and try to account for that.

Inside the ruleset, you make the inputs `display block`, so they appear on their own line. You also had to specify a width of 100%. Normally, `display block` elements automatically fill the available width, but `<input>` is a bit different. Its width is determined by the `size` attribute, which indicates roughly the number of characters it should contain without scrolling. This attribute reverts to a default value if not specified. You can force a specific width with the CSS `width` property.

The third ruleset styles the Login button. These styles are mostly straightforward; however, the `cursor` property may be unfamiliar. It controls the appearance of the mouse cursor when the cursor is over the element. The value `pointer` turns the cursor into a hand with a pointing finger, like the default cursor when pointing at links. This communicates to the user that they can click the element. It gives a final detail of polish to the button.

5.4 Alignment, spacing, and other details

You should now have a solid grasp of the most essential parts of flexbox. But as I mentioned earlier, there's a wide array of options that you'll occasionally need. These pertain mostly to the alignment or spacing of flex items within the flex container. You can also enable line wrapping or reorder individual flex items. The properties that control these are all illustrated on the following pages: table 5.1 lists all the properties that may be applied to a flex container, and table 5.2 lists all the properties for flex items.

In general, you'll begin a flexbox with the methods we've already covered:

- Identify a container and its items and use `display: flex` on the container
- If necessary, set the `flex-direction` on the container
- Declare margins and/or `flex` values for the flex items where necessary to control their size

Once you've put elements roughly where they belong, you can add other flexbox properties where necessary. My suggestion is to get familiar with the concepts we've covered thus far. Go ahead and read through the rest of this chapter to get a sense for the other options flexbox supplies, but don't worry about committing them all to memory until you need them. When you find you do need them, return here as a reference. Most of these final options are fairly straightforward, though you'll only need them occasionally.

5.4.1 Understanding flex container properties

Several properties can be applied to a flex container to control the layout of its flex items. The first is `flex-direction`, which I've already covered in section 5.3. Let's look at some others.

Table 5.1 Flex container properties

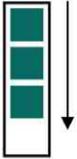
Property	Values (initial values in bold)
<p><code>flex-direction</code></p> <p>This specifies the direction of the main axis. The cross axis will be perpendicular to the main axis.</p>	<p>row </p> <p><code>row-reverse</code> </p> <p><code>column</code>  <code>column-reverse</code> </p>
<p><code>flex-wrap</code></p> <p>This specifies whether flex items will wrap on to a new row inside the flex container (or on to a new column if <code>flex-direction</code> is <code>column</code> or <code>column-reverse</code>).</p>	<p>nowrap </p> <p><code>wrap</code> </p> <p><code>wrap-reverse</code> </p>
<p><code>flex-flow</code></p>	<p>Shorthand for <code><flex-direction> <flex-wrap></code></p>
<p><code>justify-content</code></p> <p>Controls how items are positioned along the main axis.</p>	<p>flex-start </p> <p><code>flex-end</code> </p> <p><code>center</code> </p> <p><code>space-between</code> </p> <p><code>space-around</code> </p>

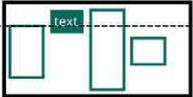
Table 5.1 Flex container properties (continued)

Property	Values (initial values in bold)			
<p><code>align-items</code></p> <p>Controls how items are positioned along the cross axis.</p>	<code>flex-start</code>		stretch	
	<code>flex-end</code>		<code>baseline</code>	
	<code>center</code>			
<p><code>align-content</code></p> <p>If flex-wrap is enabled, this controls the spacing of the flex rows along the cross axis. If items don't wrap, this property is ignored.</p>	<code>flex-start</code>		<code>stretch</code>	
	<code>flex-end</code>		<code>space-between</code>	
	<code>center</code>		<code>space-around</code>	

Table 5.2 Flex item properties

Property	Values
<p><code>flex-grow</code></p> <p>An integer that specifies the “growth factor,” determining how much the item will grow along the main axis to fill unused space</p>	
<p><code>flex-shrink</code></p> <p>An integer that specifies the “shrink factor,” determining how much the item will shrink along the main axis, if needed, to prevent overflow. Ignored if the container has flex wrap enabled.</p>	

Table 5.2 Flex item properties (continued)

Property	Values
<p><code>flex-basis</code></p> <p>Specifies the initial size of the item before <code>flex-grow</code> or <code>flex-shrink</code> is applied.</p>	<length> or <percent>
<p><code>flex</code></p>	Shorthand for: <flex-grow> <flex-shrink> <flex-basis>
<p><code>align-self</code></p> <p>Controls how the item is aligned on the cross axis. This will override the container's <code>align-items</code> value for specific item(s). Ignored if the item has an <code>auto</code> margin set on the cross axis.</p>	<p>auto</p>  <p>center</p>  <p>flex-start</p>  <p>stretch</p>  <p>flex-end</p>  <p>baseline</p> 
<p><code>order</code></p> <p>An integer that moves a flex item to a specific position among its siblings, disregarding source order.</p>	

FLEX-WRAP PROPERTY

The `flex-wrap` property can be used to allow flex items to wrap to a new row (or rows). This can be set to `nowrap` (the initial value), `wrap`, or `wrap-reverse`. When wrapping is enabled, the items don't shrink according to their `flex-shrink` values. Instead, any items that would overflow the flex container wrap onto a new line.

If the flex direction is `column` or `column-reverse`, then `flex-wrap` will allow the flex items to overflow into a new column. However, this only happens if something constrains the height of the container; otherwise, it grows to contain its flex items.

FLEX-FLOW PROPERTY

The `flex-flow` property is shorthand for both `flex-direction` and `flex-wrap`. For example, `flex-flow: column wrap` specifies that the flex items will flow from top to bottom, wrapping onto a new column if necessary.

JUSTIFY-CONTENT PROPERTY

The `justify-content` property controls how the items are spaced along the main axis if they don't fill the size of the container. Supported values include a number of new

keywords: `flex-start`, `flex-end`, `center`, `space-between`, and `space-around`. A value of `flex-start` (the default) stacks the items against the beginning of the main axis—the left side in a normal row direction. There will be no space between them unless the items have margins specified. A value of `flex-end` stacks the items at the end of the main axis and, accordingly, `center` centers them.

The value `space-between` puts the first flex item at the beginning of the main axis, and the last item at the end. Remaining items are positioned evenly between them. The value `space-around` is similar, but it will also add even spacing before the first item and after the last.

Spacing is applied after margins and `flex-grow` values are calculated. This means if any items have a non-zero `flex-grow` value, or any items have an auto margin on the main axis, then `justify-content` has no effect.

ALIGN-ITEMS PROPERTY

Whereas `justify-content` controls item alignment along the main axis, `align-items` adjusts their alignment along the cross axis. The initial value for this is `stretch`, which causes all items to fill the container’s height in a row layout, or width in a column layout. This provides columns of equal height.

The other values allow flex items to size themselves naturally, rather than filling the container size. (This is similar conceptually to the `vertical-align` property.)

- `flex-start` and `flex-end` align the items along the start or end of the cross axis (top or bottom of a row, respectively).
- `center` centers the items.
- `baseline` aligns the items so that the baseline of the first row of text in each flex item is aligned.

The value `baseline` is useful if you want the baseline of a header in one flex item with a large font to line up with the baseline of smaller text in the other flex items.

TIP It’s easy to confuse the names of the properties `justify-content` and `align-items`. I remember them by thinking of styling text: you can “justify” text to spread it horizontally from edge to edge. And, much like `vertical-align`, you can “align” inline items vertically.

ALIGN-CONTENT PROPERTY

If you enable wrapping (using `flex-wrap`), this property controls the spacing of each row inside the flex container along the cross axis. Supported values are `flex-start`, `flex-end`, `center`, `stretch` (the initial value), `space-between`, and `space-around`. These values apply spacing similar to the way described above for `justify-content`.

5.4.2 Understanding flex item properties

I’ve described `flex-grow`, `flex-shrink`, `flex-basis`, and their collective shorthand, `flex` (section 5.2). We’ll next look at two additional properties for flex items: `align-self` and `order`.

ALIGN-SELF PROPERTY

This property controls a flex item's alignment along its container's cross axis. This does the same thing as the flex container property `align-items`, except it lets you align individual flex items differently. Specifying the value `auto` will defer to the container's `align-items` value—this is the initial value. Any other value overrides the container's setting. The `align-self` property supports the same keyword values as `align-items`: `flex-start`, `flex-end`, `center`, `stretch`, and `baseline`.

ORDER PROPERTY

Normally, flex items are laid out in the order they appear in the HTML source. They are stacked along the main axis, beginning at the start of the axis. By using the `order` property, you can change the order the items are stacked. You may specify any integer, positive or negative. If multiple flex items have the same value, they'll appear according to source order.

Initially, all flex items have an order of 0. Specifying a value of -1 to one item will move it to the beginning of the list, and a value of 1 will move it to the end. You can specify order values for each item to rearrange them however you wish. The numbers don't necessarily need to be consecutive.

WARNING Be careful with the use of `order`. Making the visual layout order on the screen drastically different from the source order can harm accessibility of your site. Navigation using the Tab key will still follow the source order in most browsers, which can be confusing. Screen-reading software for visually impaired users will also follow the source order in most cases.

5.4.3 Using alignment properties

Let's use a couple of these properties to finish your page. The final tile has a stylized price and a call-to-action (CTA) button. When you're done, the last step in the page should render like figure 5.18.

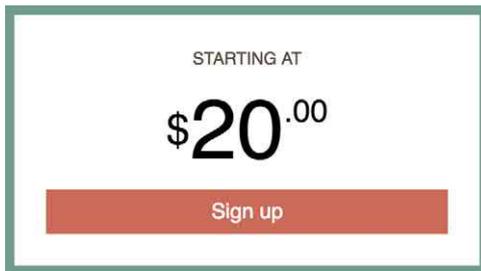


Figure 5.18 Stylized text using flexbox

The markup for this section is already in your page. It's as follows:

```
<div class="tile centered">
  <small>Starting at</small>
```

<https://sanet.cd/blogs/polatebooks/>

```

<div class="cost">
  <span class="cost-currency">$</span>
  <span class="cost-dollars">20</span>
  <span class="cost-cents">.00</span>
</div>
<a class="cta-button" href="/pricing">
  Sign up
</a>
</div>

```

The text \$20.00 is wrapped in a `<div class="cost">`, which you'll use as the flex container. It has three flex items for the three different parts of the text you want to align (\$, 20, and .00). I've chosen spans for these, rather than divs, because these are inline by default. If for some reason the CSS fails to load, or the browser doesn't support flexbox, the text \$20.00 will still appear on one line.

In the next listing, you'll use `justify-content` to horizontally center the flex items within the container. Then you'll use `align-items` and `align-self` to control their vertical alignment. Add this code to your stylesheet.

Listing 5.11 Setting styles for the cost tile

```

.centered {
  text-align: center;
}

.cost {
  display: flex;
  justify-content: center;
  align-items: center;
  line-height: .7;
}

.cost > span {
  margin-top: 0;
}

.cost-currency {
  font-size: 2rem;
}
.cost-dollars {
  font-size: 4rem;
}
.cost-cents {
  font-size: 1.5rem;
  align-self: flex-start;
}

.cta-button {
  display: block;
  background-color: #cc6b5a;
  color: white;
  padding: .5em 1em;
}

```

Centers flex items on both the main and cross axes

Overrides margins from the lobotomized owl

Sets different font sizes for each part of the cost

Overrides align-items for this item, aligning it to the top instead of center

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

This code lays out the flexbox for the stylized \$20.00, as well as defining a centered class to center the rest of the text, and a `cta-button` class for the CTA button.

The one strange declaration here is `line-height: .7`. This is because the line height of the text inside each flex item is what determines the height of each item. This means that the elements had a little more height than the height of the text itself—an em-height includes descenders, which this text doesn't have, so the characters here are actually a little less than 1 em tall. I arrived at this value purely by trial-and-error until the tops of the 20 and .00 aligned visually. See chapter 13 for more on working with text.

5.5 *A couple of things to be aware of*

Flexbox is a huge step forward for CSS. Once you're familiar with it, you might be tempted to start using it for everything on the page. I caution you to trust the normal document flow and only add flexbox where you know you'll need it. There's no reason to avoid it; but don't go crazy treating everything as a nail to its hammer.

5.5.1 *Flexbugs*

The implementation of flexbox isn't perfect in all browsers, especially IE10 and IE11. Flexbox works fine in most cases, but there are some bugs you might encounter if you have the wrong circumstances on your page. Always be sure to test your flexbox implementations in any older browsers that you want to support.

Rather than spend a lot of time discussing bugs you may or may not ever need to deal with, I'll instead refer you to a great resource called Flexbugs. Visit it at <https://github.com/philipwalton/flexbugs>. This is an up-to-date list of all known flexbox browser bugs (14 total at the time of writing). It explains exactly what circumstances cause them and, in most cases, offers a workaround to deal with the bug. If you ever find your flexbox layout behaving strangely in a particular browser, visit this page and see if you've encountered one of these browser bugs.

5.5.2 *Full-page layout*

One of the interesting things about flexbox is how the flex sizes are calculated based on the number of flex items and the amount (and size) of content within them. This can lead to an odd behavior if your page is large or is loaded over a slow connection.

As the browser loads content, it progressively renders it to the screen, even as it continues to download the remainder of the page. Assume you have a three-column layout, built using a flexbox (`flex-direction: row`). If the content for two of these columns loads, the browser might render them before it loads the content for the third column. Then, when the rest of the content loads, the browser recalculates the sizes of each flex item and renders the page again. The user will see a two-column

layout momentarily, then the columns will resize (perhaps drastically), and the third column will appear.

Jake Archibald, a developer advocate for Google Chrome, has written about this at <https://jakearchibald.com/2014/dont-use-flexbox-for-page-layout/>. You can see examples of this happening in that article. One suggestion he gives is to favor grid layout for the full page layout (which I cover in the next chapter).

NOTE This behavior is only an issue with multiple columns in a row. It doesn't occur with multiple rows in a column (`flex-direction: column`) for the main page layout.

Summary

- Use flexbox for versatile, easy-to-control layout of page content.
- Autoprefixer can simplify flexbox support for older browsers.
- Use `flex` to specify almost any imaginable combination of flex item sizes.
- Use nested flexboxes to piece together more complicated layouts and to fill the heights of naturally sized boxes.
- Flexbox automatically creates columns of equal height.
- Use `align-items` or `align-self` to vertically center a flex item inside its flex container.

Grid layout

This chapter covers

- Using CSS' first true layout system—grid
- Understanding grid layout options
- Laying out items on a grid
- Using flexbox and grid together to build a cohesive web page layout

Flexbox has revolutionized the way we do layout on the web, but it's only the beginning. It has a big brother: another new specification called the Grid Layout Module. Together, these two specifications provide a full-featured layout engine for the web like you've never seen before.

In this chapter, I'll show you how you can start learning grid layout today. I'll give you an overview of how it works, then take you through several examples to illustrate the different things grid layout can do. Building a basic grid is simple. It's also powerful enough to enable complex layouts, but doing so requires learning new properties and keywords. This chapter will guide you through them.

The CSS grid lets you define a two-dimensional layout of columns and rows and then place elements within the grid. Some elements may only fill one cell of the grid; others can span multiple columns or rows. The size of the grid can be

defined precisely, or you can allow it to automatically size itself as needed to fit the contents within. You can place items precisely within the grid, or allow them to flow naturally to fill in the gaps. A grid lets you build complex layouts like the one shown in figure 6.1.



Figure 6.1 Boxes in a sample grid layout

6.1 Web layout is here

The emergence of grid layout was not like that of other CSS features, such as flexbox. As browsers implemented early versions of flexbox, they made it available through the use of vendor prefixes. The original intention of vendor prefixes was to allow developers to experiment with the technology before using it in production. This is not how things played out, however.

It took several years to develop the flexbox specification to a stable point. In the meantime, developers got excited about the new feature and started using it, prefixes and all. Then, as the specification evolved, browsers updated their implementations. Developers had to update their code to match, but they also had to leave the old code in place to support older browsers as well. It made for a rough introduction of flexbox into the world.

To prevent this from happening again, browser vendors approached grid layout differently. Instead of implementing it with vendor prefixes, they implemented it as a feature the user must explicitly opt into. Developers could experiment with it to learn how it worked and to report bugs, but as far as your average user was concerned, browser support was effectively zero. At the same time, browsers had almost completely implemented grid layout.

Instead of a long, drawn-out rollout of development and debugging, all major browsers were able to turn on a full-featured, mostly debugged implementation of grid, virtually overnight. In March 2017, they started flipping the switch. In the span of three weeks, Firefox, Chrome, Opera, and Safari all released updates to their browsers, enabling the grid layout. Microsoft Edge followed suit in June 2017. In the span of three months, browser support grew from 0% to almost 70% of users. This is unprecedented in the world of CSS.

Level 1 of the grid specification is stable, and all modern browsers now conform to it. This means grid layout is ready for production use now, as long as you do a little work to ensure a reasonable fallback design. I'll cover this near the end of the chapter.

NOTE Microsoft implemented an early version of grid layout using vendor prefixes. This means IE10 and IE11 have partial support for grid layout using a `-ms-` prefix. To support these browsers, use Autoprefixer as discussed in chapter 5 (see the sidebar “Vendor prefixes”).

Enabling experimental features

Before browsers supported grid layout by default, they allowed developers to enable it. Even though grid is now enabled, it’s important to know how to access other experimental features should you want to learn those in the future.

In Chrome and Opera, this is done by enabling a flag in the browser settings. In Chrome, type `chrome://flags` into your address bar and press Enter. If you use Opera, go to `opera://flags` instead. Then scroll down until you find Experimental Web Platform Features (or use the browser’s search feature) and click Enable.

If you prefer Firefox, you’ll need to download and install either Firefox Developer Edition (<https://www.mozilla.org/en-US/firefox/developer/>) or Firefox Nightly (<https://nightly.mozilla.org/>). If you use Safari, you can install the Safari Technology Preview or the Webkit Nightly Builds edition.

6.1.1 Building a basic grid

Now, let’s create a simple grid layout to make sure it works in your browser. You’ll lay out six boxes in three columns as shown in figure 6.2. The markup for this grid is shown in listing 6.1.



Figure 6.2 A simple grid with three columns and two rows

Create a new page and link it to a new stylesheet. Add the code in the following listing to your page. In the code, I’ve added the letters *a* through *f* so it’s apparent where each element ends up in the grid.

Listing 6.1 A grid with six items

```
<div class="grid">
  <div class="a">a</div>
  <div class="b">b</div>
  <div class="c">c</div>
```

← The grid container

↓ The container’s children become the grid items

```

<div class="d">d</div>
<div class="e">e</div>
<div class="f">f</div>
</div>

```

↑ The container's children become the grid items

As with flexbox, grid layout applies to two levels of the DOM hierarchy. An element with `display: grid` becomes a *grid container*. Its child elements then become *grid items*.

Next, you'll apply a few new properties to define the specifics of the grid. Add the styles from this listing to your stylesheet.

Listing 6.2 Laying out a basic grid

```

.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
  grid-gap: 0.5em;
}

.grid > * {
  background-color: darkgray;
  color: white;
  padding: 2em;
  border-radius: 0.5em;
}

```

Makes the element a grid container

Defines three columns of equal width

Defines two rows of equal height

Applies a gutter between each grid cell

If your browser supports grid layout, this code will render six equal-sized boxes in three columns (figure 6.2). A number of new things are going on here. Let's take a closer look at them.

First, you've applied `display: grid` to define a grid container. The container behaves like a block display element, filling 100% of the available width. Although not shown in this listing, you could also use the value `inline-grid`; in which case, the element will flow inline and will only be as wide as is necessary to contain its children. You'll most likely not use `inline-grid` as often.

Next come the new properties: `grid-template-columns` and `grid-template-rows`. These define the size of each of the columns and rows in the grid. This example uses a new unit, `fr`, which represents each column's (or row's) *fraction unit*. This unit behaves essentially the same as the `flex-grow` factor in flexbox. The declaration `grid-template-columns: 1fr 1fr 1fr` declares three columns with an equal size.

You don't necessarily have to use fraction units for each column or row. You can also use other measures such as `px`, `em`, or percent. Or, you could mix and match. For instance, `grid-template-columns: 300px 1fr` would define a fixed-size column of 300 px followed by a second column that will grow to fill the rest of the available space. A 2 fr column would be twice as wide as a 1 fr column.

Finally, the `grid-gap` property defines the amount of space to add to the gutter between each grid cell. You can optionally provide two values to specify vertical and horizontal spacing individually (for example, `grid-gap: 0.5em 1em`).

I encourage you to experiment with these values to see how they affect the final layout. Add new columns or change their widths. Add or remove grid items. Continue to experiment with the other layouts throughout this chapter. This will be the best way to get the hang of things.

6.2 Anatomy of a grid

It's important to understand the various parts of a grid. I've already mentioned grid containers and grid items, which are the elements that make up the grid. Four other important terms to know are illustrated in figure 6.3.

- *Grid line*—These make up the structure of the grid. A grid line can be vertical or horizontal and lie on either side of a row or column. The `grid-gap`, if defined, lies atop the grid lines.
- *Grid track*—A grid track is the space between two adjacent grid lines. A grid has horizontal tracks (rows) and vertical tracks (columns).
- *Grid cell*—A single space on the grid, where a horizontal grid track and a vertical grid track overlap.
- *Grid area*—A rectangular area on the grid made up by one or more grid cells. The area is between two vertical grid lines and two horizontal grid lines.

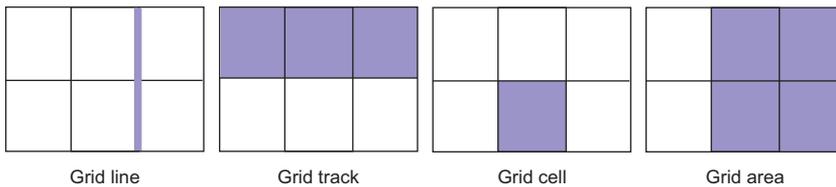


Figure 6.3 The parts of a grid

You'll refer to these parts of the grid as you build grid layouts. For instance, declaring `grid-template-columns: 1fr 1fr 1fr` defines three vertical *grid tracks* of equal width. It also defines four vertical *grid lines*: one down the left edge of the grid, two more between each grid track, and one more along the right edge.

In the previous chapter, you built a page using flexbox. Let's take another look at that design and consider how you could implement it using grid. The design is shown in figure 6.4. I've added dashed lines to indicate the location of each grid cell. Notice that some of the sections span multiple cells—filling a larger *grid area*.

This grid has two columns and four rows. The top two horizontal grid tracks are each dedicated to the page title (Ink) and the main navigational menu. The main area fills the remaining two cells in the first vertical track, and the two sidebar tiles are each placed in one of the remaining cells in the second vertical track.

NOTE Your design doesn't need to fill every cell of the grid. Leave a cell empty where you want to add whitespace.

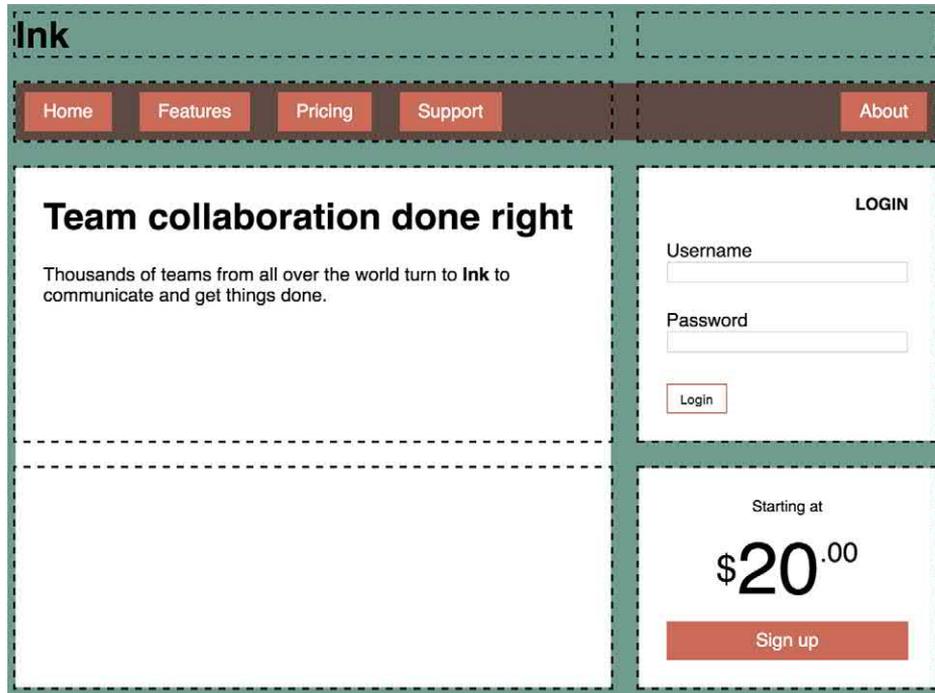


Figure 6.4 Page layout created with grid. The dashed lines are added to indicate location of each grid cell.

It's important to note the use of grid here does not render flexbox useless. As we go through the page, you'll see that flexbox is still an important part of the layout. I'll point out places on the page where it makes sense to use flexbox.

When you built this page using flexbox, you had to nest the elements in a certain way. You used one flexbox to define columns and nested another flexbox inside it to define rows (listing 5.1). To build this layout with grid requires a different HTML structure: You'll need to flatten the HTML. Each item you place on the grid must be a child of the main grid container. The new markup is shown next. Create a new page (or modify your page from chapter 5) to match this listing.

Listing 6.3 HTML structure for a grid layout

```

<body>
  <div class="container">
    <header>
      <h1 class="page-heading">Ink</h1>
    </header>

    <nav>
      <ul class="site-nav">
        <li><a href="/">Home</a></li>

```

The "container" becomes your grid container.

Each grid item must be a child element of the grid container.

```

    <li><a href="/features">Features</a></li>
    <li><a href="/pricing">Pricing</a></li>
    <li><a href="/support">Support</a></li>
    <li class="nav-right">
      <a href="/about">About</a>
    </li>
  </ul>
</nav>

<main class="main tile">
  <h1>Team collaboration done right</h1>
  <p>Thousands of teams from all over the
    world turn to <b>Ink</b> to communicate
    and get things done.</p>
</main>

<div class="sidebar-top tile">
  <form class="login-form">
    <h3>Login</h3>
    <p>
      <label for="username">Username</label>
      <input id="username" type="text"
        name="username"/>
    </p>
    <p>
      <label for="password">Password</label>
      <input id="password" type="password"
        name="password"/>
    </p>
    <button type="submit">Login</button>
  </form>
</div>

<div class="sidebar-bottom tile centered">
  <small>Starting at</small>
  <div class="cost">
    <span class="cost-currency">$</span>
    <span class="cost-dollars">20</span>
    <span class="cost-cents">.00</span>
  </div>
  <a class="cta-button" href="/pricing">
    Sign up
  </a>
</div>
</div>
</body>

```

Each grid item must be a child element of the grid container.

This version of the page has placed each section of the page as a grid item: the header, the menu (nav), the main, and the two sidebars. I've also added the `tile` class to the main and the two sidebars, as this class provides the white background color and the padding that these elements have in common.

Let's apply grid layout to the page, and put each section in place. We'll pull in a lot of styles from the version in chapter 5 momentarily, but first let's get a general shape

of the page in place. (I find it's generally easier to build a page from the outside in.) After building the basic grid, the page will appear as in figure 6.5.

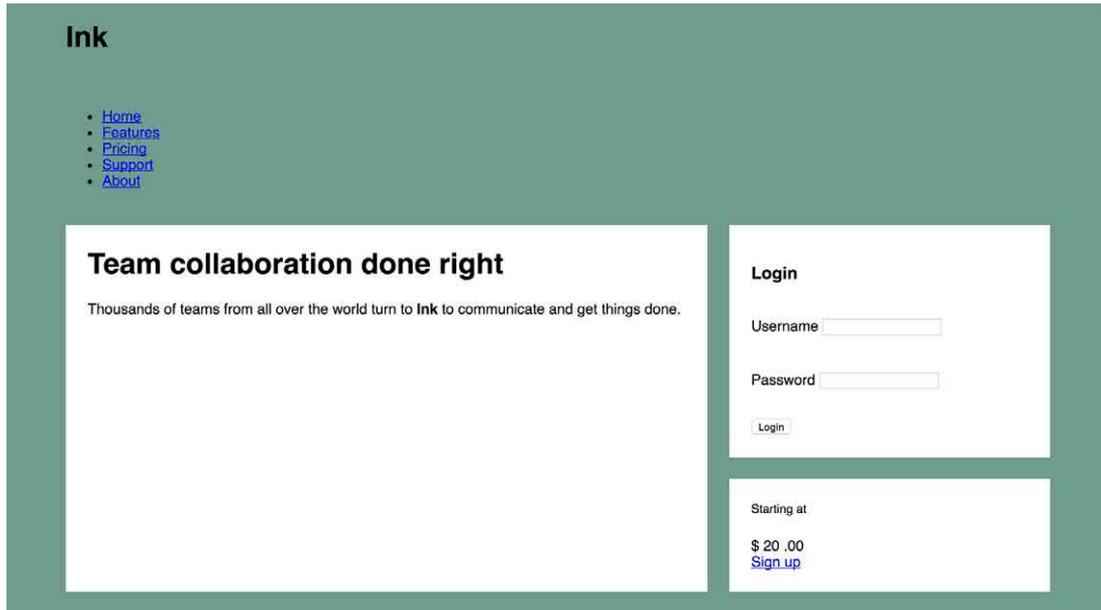


Figure 6.5 Page with basic grid structure in place

Create an empty stylesheet and link to it from the page. Add this listing to the new stylesheet. This code introduces a few new concepts, which I'll walk you through in a bit.

Listing 6.4 Applying a top-level page layout using grid

```

:root {
  box-sizing: border-box;
}

*,
::before,
::after {
  box-sizing: inherit;
}

body {
  background-color: #709b90;
  font-family: Helvetica, Arial, sans-serif;
}

.container {
  display: grid;
  grid-template-columns: 2fr 1fr;

```

← Defines two vertical grid tracks

```

grid-template-rows: repeat(4, auto);
grid-gap: 1.5em;
max-width: 1080px;
margin: 0 auto;
}

header,
nav {
  grid-column: 1 / 3;
  grid-row: span 1;
}

.main {
  grid-column: 1 / 2;
  grid-row: 3 / 5;
}

.sidebar-top {
  grid-column: 2 / 3;
  grid-row: 3 / 4;
}

.sidebar-bottom {
  grid-column: 2 / 3;
  grid-row: 4 / 5;
}

.tile {
  padding: 1.5em;
  background-color: #fff;
}

.tile > :first-child {
  margin-top: 0;
}

.tile * + * {
  margin-top: 1.5em;
}

```

← Defines four horizontal grid tracks of size auto

← Spans from vertical grid line 1 to grid line 3

← Spans exactly one horizontal grid track

← Positions other grid items between various grid lines

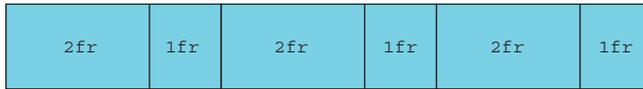
This listing provides a number of new concepts. Let's take them one at a time.

You set the grid container and defined its grid tracks using `grid-template-columns` and `grid-template-rows`. The columns are defined using the fraction units `2fr` and `1fr`, so the first column will grow twice as much as the second. The rows use something new, the `repeat()` function. This function provides a shorthand for declaring multiple grid tracks.

The declaration, `grid-template-rows: repeat(4, auto);` defines four horizontal grid tracks of height `auto`. It's equivalent to `grid-template-rows: auto auto auto auto`. The track size of `auto` will grow as necessary to the size of its content.

You can also define a repeating pattern with the `repeat()` notation. For instance, `repeat(3, 2fr 1fr)` defines six grid tracks by repeating the pattern three times, resulting in `2fr 1fr 2fr 1fr 2fr 1fr`. Figure 6.6 illustrates the resulting columns.

```
grid-template-columns: repeat(3, 2fr 1fr);
```



```
grid-template-columns: 1fr repeat(3, 3fr) 1fr;
```



Figure 6.6 Using the `repeat()` function to define a repeating pattern in a template definition

Or you can use `repeat()` as part of a longer pattern. `grid-template-columns: 1fr repeat(3, 3fr) 1fr`, for instance, defines a 1 fr column followed by three 3 fr columns then another 1 fr column (or `1fr 3fr 3fr 3fr 1fr`). As you can see, the longhand is a bit tricky to parse visually, which is why the `repeat()` shorthand comes in handy.

6.2.1 Numbering grid lines

With the grid tracks defined, the next portion of the code places each grid item into a specific location on the grid. The browser assigns numbers to each grid line in a grid, as shown in figure 6.7. The CSS uses these numbers to indicate where each item should be placed.

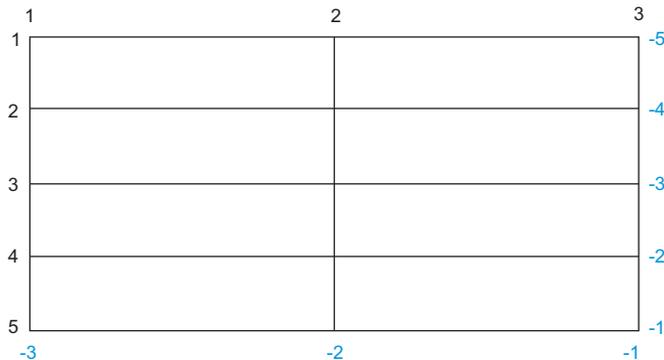


Figure 6.7 Grid lines are numbered beginning with 1 on the top left. Negative numbers refer to the position from the bottom right.

You can use the grid numbers to indicate where to place each grid item using the `grid-column` and `grid-row` properties. If you want a grid item to span from grid line 1 to grid line 3, you'll apply `grid-column: 1 / 3` to the element. Or, you can apply

`grid-row: 3 / 5` to a grid item to make it span from the horizontal grid line 3 to grid line 5. These two properties together specify the grid area you want for an element.

In your page, several grid items are positioned this way:

```
.main {
  grid-column: 1 / 2;
  grid-row: 3 / 5;
}

.sidebar-top {
  grid-column: 2 / 3;
  grid-row: 3 / 4;
}

.sidebar-bottom {
  grid-column: 2 / 3;
  grid-row: 4 / 5;
}
```

This code positions the main in the first column (between grid lines 1 and 2), spanning the third and fourth rows (between grid lines 3 and 5). It places each sidebar tile in the right column (between grid lines 2 and 3), stacked atop each other in the third and fourth rows.

NOTE These properties are in fact shorthand properties: `grid-column` is short for `grid-column-start` and `grid-column-end`; `grid-row` is short for `grid-row-start` and `grid-row-end`. The forward slash is only needed in the shorthand version to separate the two values. The space before and after the slash is optional.

The ruleset that positions the header and nav at the top of the page is a little bit different. Here I've used the same ruleset to target both:

```
header,
nav {
  grid-column: 1 / 3;
  grid-row: span 1;
}
```

This example uses `grid-column` as you've seen previously, making the grid item span the full width of the grid. You can also specify `grid-row` and `grid-column` using a special keyword, `span` (used in this example for `grid-row`). This tells the browser that the item will span one grid track. I didn't specify an explicit row with which to start or end, so the grid item will be placed automatically using the grid item *placement algorithm*. The placement algorithm will position items to fill the first available space on the grid where they fit; in this case, the first and second rows. We'll look closer at auto-placement later in the chapter.

6.2.2 Working together with flexbox

After learning about grid, developers often ask about flexbox. Specifically, are these two competing layout methods? The answer is no; they're complementary. They were largely developed in conjunction. Although there's some overlap in what each can accomplish, they each shine in different scenarios. Choosing between flexbox and grid for a piece of a design is going to come down to your particular needs. The two layout methods have two important distinctions:

- Flexbox is basically one-dimensional, whereas grid is two-dimensional.
- Flexbox works from the content out, whereas grid works from the layout in.

Because flexbox is one-dimensional, it's ideal for rows (or columns) of similar elements. It supports line wrapping using `flex-wrap`, but there's no way to align items in one row with those in the next. Grid, on the contrary, is two-dimensional. It's intended to be used in situations where you want to align items in one track with those in another. This distinction is illustrated in figure 6.8.

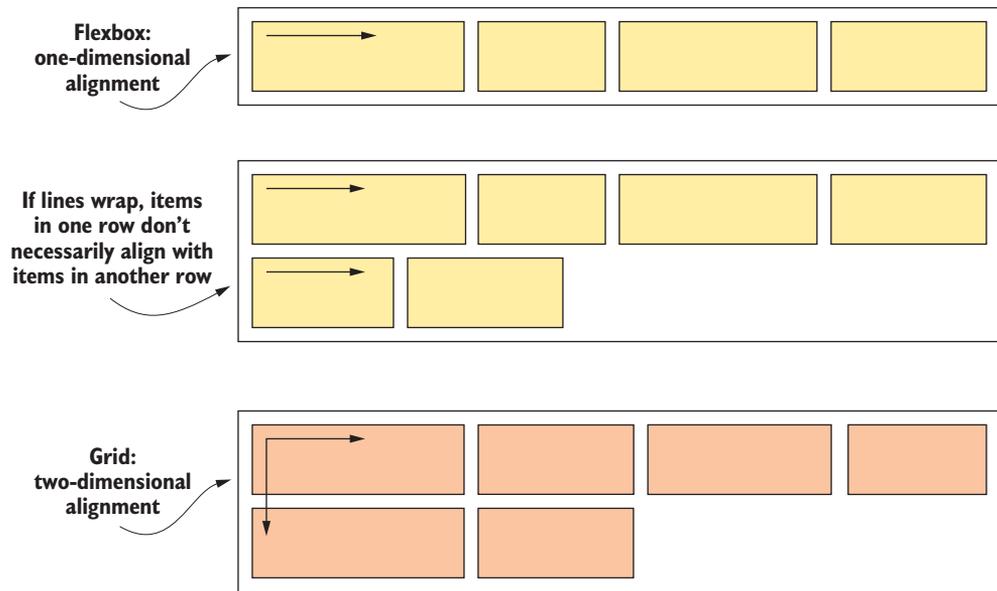


Figure 6.8 Flexbox aligns items in one direction, while grid aligns items in two directions.

The second major distinction, as articulated by CSS WG member Rachel Andrew, is that flexbox works from the content out, whereas a grid works from the layout in. Flexbox lets you arrange a series of items in a row or column, but their sizes don't need to be explicitly set. Instead, the content determines how much space each item needs.

With grid, however, you are first and foremost describing a layout, then placing items into that structure. While the content of each grid item has the ability to influence the

size of its grid track, this will affect the size of the entire track and, therefore, the size of other grid items in the track.

We've positioned the main regions of the page using grid because we want the contents to adhere to the grid as it is defined. But for some other items on the page, such as the navigational menu, we can allow the contents to have a greater influence on the outcome; that is, items with more text can be wider, and items with less text can be narrower. It's also a horizontal (one-dimensional) layout. For these reasons, flexbox is a more appropriate solution for these items. Let's style these items using flexbox to finish the page.

Figure 6.9 shows the page with a top navigational menu that consists of a list of links aligned horizontally. We'll also use flexbox for the stylized pricing number on the lower right. After adding these and a few other styles, we'll arrive at the page's final look and feel.

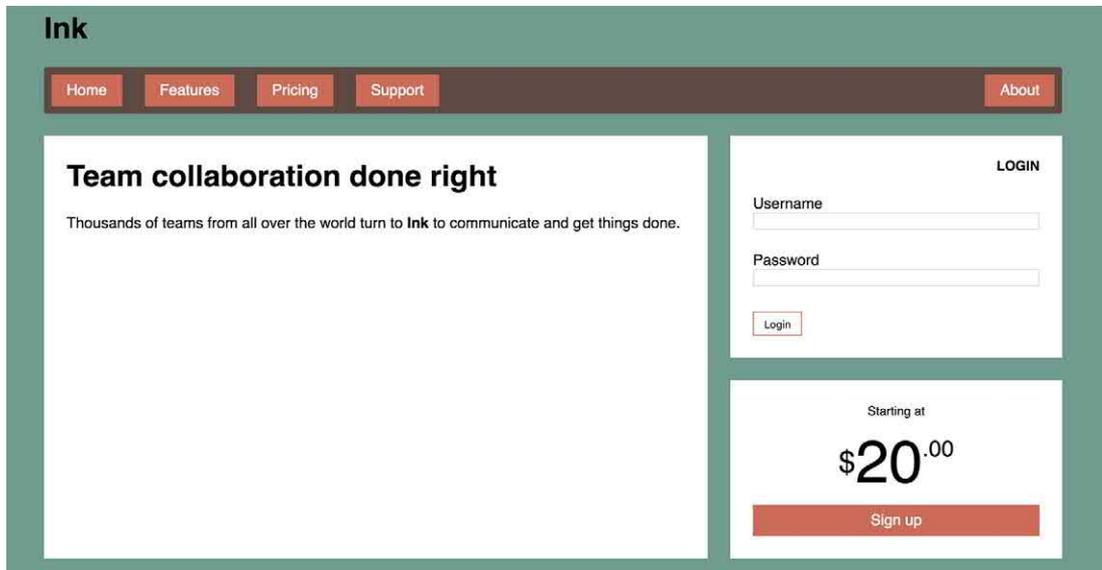


Figure 6.9 Fully styled page

The styles to do this are identical to those from the styles in chapter 5, minus the high-level layout you've applied using grid (listing 6.4). I've repeated them in the next listing. Add this to your stylesheet.

Listing 6.5 Remaining styling for the page

```
.page-heading {
  margin: 0;
}
```

```
.site-nav {  
  display: flex;  
  margin: 0;  
  padding: .5em;  
  background-color: #5f4b44;  
  list-style-type: none;  
  border-radius: .2em;  
}  
  
.site-nav > li {  
  margin-top: 0;  
}  
  
.site-nav > li > a {  
  display: block;  
  padding: .5em 1em;  
  background-color: #cc6b5a;  
  color: white;  
  text-decoration: none;  
}  
  
.site-nav > li + li {  
  margin-left: 1.5em;  
}  
  
.site-nav > .nav-right {  
  margin-left: auto;  
}  
  
.login-form h3 {  
  margin: 0;  
  font-size: .9em;  
  font-weight: bold;  
  text-align: right;  
  text-transform: uppercase;  
}  
  
.login-form input:not([type=checkbox]):not([type=radio]) {  
  display: block;  
  margin-top: 0;  
  width: 100%;  
}  
  
.login-form button {  
  margin-top: 1em;  
  border: 1px solid #cc6b5a;  
  background-color: white;  
  padding: .5em 1em;  
  cursor: pointer;  
}  
  
.centered {  
  text-align: center;  
}
```

← The flexbox menu

```

.cost {
  display: flex;
  justify-content: center;
  align-items: center;
  line-height: .7;
}

.cost > span {
  margin-top: 0;
}

.cost-currency {
  font-size: 2rem;
}
.cost-dollars {
  font-size: 4rem;
}
.cost-cents {
  font-size: 1.5rem;
  align-self: flex-start;
}

.cta-button {
  display: block;
  background-color: #cc6b5a;
  color: white;
  padding: .5em 1em;
  text-decoration: none;
}

```

← Stylized “cost”
using flexbox

When your design calls for an alignment of items in two dimensions, use grid. When you’re only concerned with a one-directional flow, use flexbox. In practice, this will often (but not always) mean grid makes the most sense for a high-level layout of the page, and flexbox makes more sense for certain elements within each grid area. As you continue to work with both, you’ll begin to get a feel for which is appropriate in various instances.

6.3 *Alternate syntaxes*

There are two other alternate syntaxes for laying out grid items: named grid lines and named grid areas. Choosing between them is a matter of preference. In some designs, one syntax may be easier to read and understand than the others. Let’s look at both.

6.3.1 *Naming grid lines*

Sometimes it can be a bit tricky to keep track of all the numbered grid lines, especially when working with a lot of grid tracks. To make this easier, you can name the grid lines and use the names instead of numbers. When declaring grid tracks, place a name in brackets to name a grid line between any two tracks.:

```
grid-template-columns: [start] 2fr [center] 1fr [end];
```

This declaration defines a two-column grid with three vertical grid lines named `start`, `center`, and `end`. You can then reference these names instead of the numbers when placing grid items in your grid. For example:

```
grid-column: start / center;
```

This declaration places a grid item so it spans from grid line 1 (`start`) to grid line 2 (`center`). You can also provide multiple names for the same grid line as shown in this example (I've added line breaks to aid readability):

```
grid-template-columns: [left-start] 2fr
                      [left-end right-start] 1fr
                      [right-end];
```

In this declaration, grid line 2 is named both `left-end` and `right-start`. You can then use either of these names when placing a grid item. This declaration allows for another trick here as well: by naming grid lines `left-start` and `left-end`, you've defined an area called `left` that spans between them. The `-start` and `-end` suffixes act as a sort of keyword defining an area in between. If you apply `grid-column: left` to an element, it'll span from `left-start` to `left-end`.

The CSS in the next listing uses named grid lines to lay out the page. This produces the same result as the approach in listing 6.4. Update this portion of your stylesheet to match.

Listing 6.6 Grid layout using named grid lines

```
.container {
  display: grid;
  grid-template-columns: [left-start] 2fr
                        [left-end right-start] 1fr
                        [right-end];
  grid-template-rows: repeat(4, [row] auto);
  grid-gap: 1.5em;
  max-width: 1080px;
  margin: 0 auto;
}

header,
nav {
  grid-column: left-start / right-end;
  grid-row: span 1;
}

.main {
  grid-column: left;
  grid-row: row 3 / span 2;
}

.sidebar-top {
```

Names each vertical grid line

Names horizontal grid lines "row"

Spans from left-start to left-end

Places the item beginning at the third row grid line and spanning two grid tracks

```

grid-column: right;
grid-row: 3 / 4;
}

.sidebar-bottom {
grid-column: right;
grid-row: 4 / 5;
}

```

←
Spans from right-
start to right-end
←

This example places each item into the appropriate grid columns using the named grid lines. It also declares a named horizontal grid line inside the `repeat()` function. Doing this names each horizontal grid line row (except for the last one). This may seem peculiar, but it's perfectly valid to use the same name repeatedly. You then place the main element so it begins at row 3 (the third grid line named row) and spans two grid tracks from there.

You can use named grid lines in countless ways. How you use them can vary from one grid to the next, depending on the particular structure of each grid. One possible example is shown in figure 6.10.

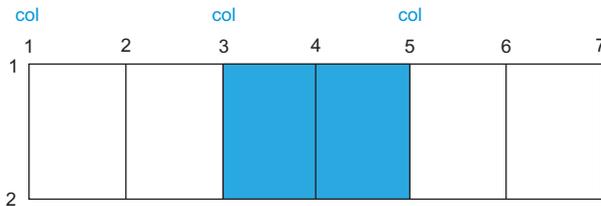


Figure 6.10 Placing a grid item at the second “col” grid line, spanning two tracks (col 2 / span 2)

This scenario presents a repeating pattern of two grid columns, naming the grid line before each pair of grid tracks (`grid-template-columns: repeat(3, [col] 1fr 1fr)`). Then you can use named grid lines to position an item in the second set of columns (`grid-column: col 2 / span 2`).

6.3.2 Naming grid areas

Another approach you can take is to name the grid areas. Instead of counting or naming the grid lines, you can use these named areas to position items in the grid. This is done with the `grid-template` property on the grid container and a `grid-area` property on the grid items.

The code in listing 6.7 shows an example of this. Again, this code produces exactly the same result as the previous layout (listings 6.4 and 6.6). It's an alternate syntax that can be used instead. Update your stylesheet to match these styles.

Listing 6.7 Using named grid areas

```

.container {
  display: grid;
  grid-template-areas: "title title"
                      "nav  nav"
                      "main aside1"
                      "main aside2";
  grid-template-columns: 2fr 1fr;
  grid-template-rows: repeat(4, auto);
  grid-gap: 1.5em;
  max-width: 1080px;
  margin: 0 auto;
}

header {
  grid-area: title;
}

nav {
  grid-area: nav;
}

.main {
  grid-area: main;
}

.sidebar-top {
  grid-area: aside1;
}

.sidebar-bottom {
  grid-area: aside2;
}

```

Assigns each grid cell to a named grid area

Defines grid track sizes as before

Places each grid item into a named grid area

The `grid-template-areas` property lets you draw a visual representation of the grid directly into your CSS, using a sort of “ASCII art” syntax. This declaration provides a series of quoted strings, each one representing a row of the grid, with whitespace between each column.

In this example, the first row is assigned entirely to the grid area `title`. The second row is assigned to `nav`. The left column of the next two rows is assigned to `main`, and each sidebar tile is assigned to `aside1` and `aside2`. Each grid item is then placed into these named areas using the `grid-area` property.

WARNING Each named grid area must form a rectangle. You cannot create more complex shapes like an *L* or a *U*.

You can also leave a cell empty by using a period as its name. For example, this code defines four grid areas surrounding an empty grid cell in the middle:

```

grid-template-areas: "top top right"
                   "left . right"
                   "left bottom bottom";

```

When you build a grid, use whichever syntax is most comfortable for you, given the design: numbered grid lines, named grid lines, or named grid areas. The latter is likely to be a favorite for a lot of developers, and it shines when you know exactly where you want to place each grid item.

6.4 Explicit and implicit grid

In some instances, you may not know exactly where you want to place each item in the grid. Perhaps you're working with a large number of grid items and placing each one explicitly is unwieldy. You might even have an unknown number of items populated by a database. In these cases, it'll probably make more sense to loosely define a grid, and then allow the grid item placement algorithm to fill it for you.

This will require you to rely on an *implicit grid*. When you use the `grid-template-*` properties to define grid tracks, you're creating an *explicit grid*. But grid items can still be placed outside of these explicit tracks; in which case, implicit tracks will be automatically generated, expanding the grid so it contains these elements.

Figure 6.11 illustrates a grid with only one explicit grid track in each direction. When a grid item is placed in the second track (between grid lines two and three), additional tracks are added to include it.

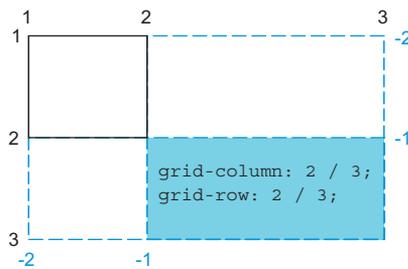


Figure 6.11 If a grid item is placed outside the declared grid tracks, implicit tracks will be added to the grid until it can contain the item.

By default, implicit grid tracks will have a size of `auto`, meaning they'll grow to the size necessary to contain the grid item contents. The properties `grid-auto-columns` and `grid-auto-rows` can be applied to the grid container to specify a different size for all implicit grid tracks (for example, `grid-auto-columns: 1fr`).

NOTE Implicit grid tracks don't change the meaning of negative numbers when referencing grid lines. Negative grid-line numbering still begins at the bottom/right of the explicit grid.

Let's lay out another page using an implicit grid. This will be a photography portfolio, as shown in figure 6.12. For this layout, you'll set grid tracks for the columns, but the grid rows will be implicit. This way, the page isn't structured for any specific number of images; it'll be adaptable for any number of grid items. Any time the images need to wrap onto a new row, another row will be added implicitly.

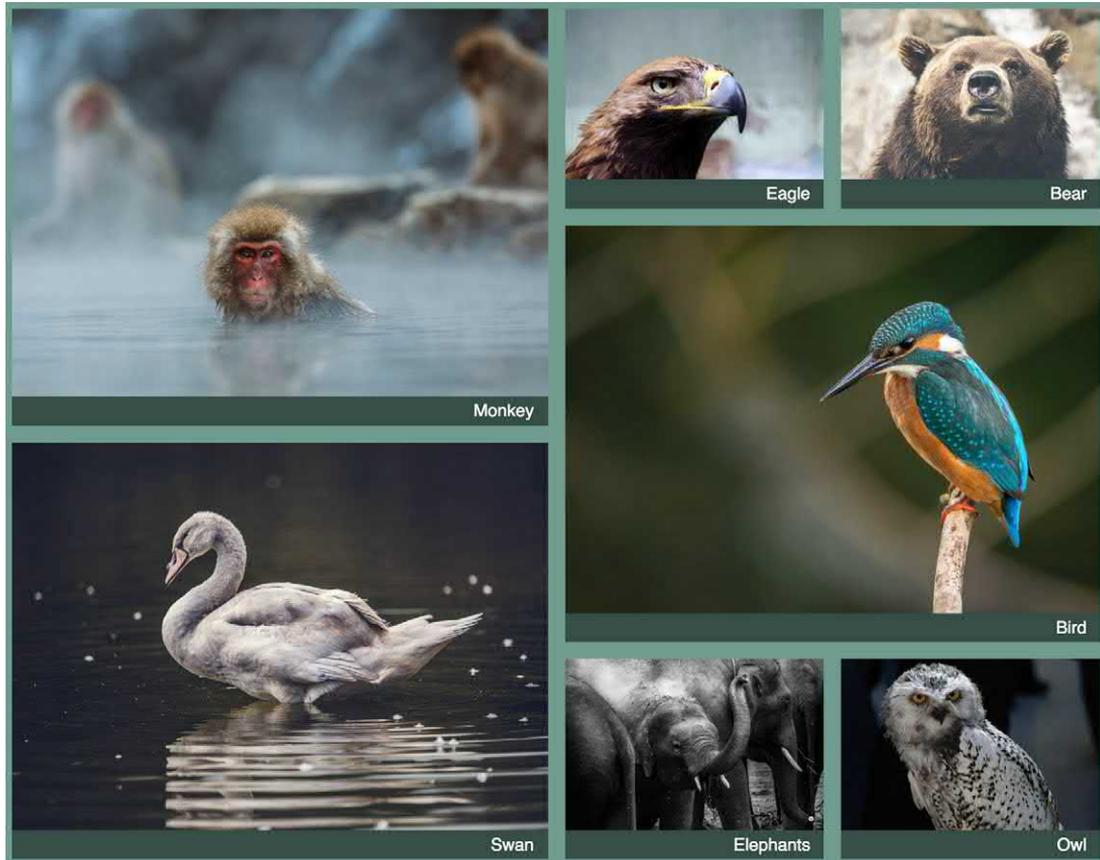


Figure 6.12 A series of photographs laid out in a grid using implicit grid rows

This is a fun layout because it would be difficult to achieve with flexbox or floats. It showcases the unique power of grids.

To build this, you'll need a new page. Create a blank page and a new stylesheet and link them. The markup for this is shown here. Add it to the page.

Listing 6.8 Markup for a portfolio

```

<div class="portfolio">
  <figure class="featured">
    
    <figcaption>Monkey</figcaption>
  </figure>
  <figure>
    
    <figcaption>Eagle</figcaption>
  </figure>

```

Each `<figure>` will be a grid item.

Encloses the image and its caption inside the `<figure>` element

```

<figure class="featured">
  
  <figcaption>Bird</figcaption>
</figure>
<figure>
  
  <figcaption>Bear</figcaption>
</figure>
<figure class="featured">
  
  <figcaption>Swan</figcaption>
</figure>
<figure>
  
  <figcaption>Elephants</figcaption>
</figure>
<figure>
  
  <figcaption>Owl</figcaption>
</figure>
</div>

```

The featured class will make certain images larger.

This markup is a portfolio element (which will be the grid container) and a series of figures (which will be the grid items). Each figure contains an image and a caption. I've added the class `featured` to a few items, which you'll use to make those larger than the other images.

I'll walk you through this in a few phases. First, you'll shape the grid tracks and see the images in a basic grid formation (figure 6.13). After that, you'll enlarge the "featured" images and apply a few other finishing touches.

The styles for this are shown in listing 6.9. It uses `grid-auto-rows` to specify a 1 fr size for all implicit grid rows, so each row will be the same height. It also introduces



Figure 6.13 Images laid out in a basic grid

two new concepts: auto-fill and the `minmax()` function, which I'll explain in a moment. Add these styles to your stylesheet.

Listing 6.9 A grid with implicit grid rows

```
body {
  background-color: #709b90;
  font-family: Helvetica, Arial, sans-serif;
}

.portfolio {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  grid-auto-rows: 1fr;
  grid-gap: 1em;
}

.portfolio > figure {
  margin: 0;
}

.portfolio img {
  max-width: 100%;
}

.portfolio figcaption {
  padding: 0.3em 0.8em;
  background-color: rgba(0, 0, 0, 0.5);
  color: #fff;
  text-align: right;
}
```

Sets a minimum column width of 200 px and auto-fills the grid

Sets an implicit horizontal grid track size of 1 fr

Overrides user agent margins

Sometimes you won't want to set a fixed size on a grid track, but you'll want to constrain it within certain minimum and maximum values. This is where the `minmax()` function comes in. It specifies two values—a minimum size and a maximum size. The browser will ensure the grid track falls between these values. (If the maximum size is smaller than the minimum size, then the maximum is ignored.) By specifying `minmax(200px, 1fr)`, the browser ensures that all tracks are at least 200 px wide.

The `auto-fill` keyword is a special value you can provide for the `repeat()` function. With this set, the browser will place as many tracks onto the grid as it can fit, without violating the restrictions set by the specified size (the `minmax()` value).

Together, `auto-fill` and `minmax(200px, 1fr)` mean your grid will place as many grid columns as the available space can hold, without allowing any of them to be smaller than 200 px. And because no track can be larger than 1 fr (our maximum value), all the grid tracks will be the same size.

In figure 6.13, the viewport has room for four columns of 200 px, so that's how many tracks were added. If the screen is wider, more may fit. If it's narrower, then fewer will be created.

Note that `auto-fill` can also result in some empty grid tracks, if there are not enough grid items to fill them all. If you don't want empty grid tracks, you can use the keyword `auto-fit` instead of `auto-fill`. This causes the non-empty tracks to stretch to fill the available space. See <http://gridbyexample.com/examples/example37/> for an example of the difference.

Whether you use `auto-fill` or `auto-fit` depends on whether you want to ensure you get the expected grid track size or whether you want to make certain the length of the entire grid container is filled. I typically find I prefer `auto-fit`.

6.4.1 *Adding variety*

Next, let's add visual interest to your grid by increasing the size of the featured images (the bird and the swan in this example). Each grid item currently fills a 1 x 1 area on the grid. You'll increase the size of featured images to fill a 2 x 2 grid area. You can target these items with the `featured` class and make them span two grid tracks in each direction.

This introduces a problem, however. Depending on the order of the items, increasing the size for some grid items could result in gaps in the grid. Figure 6.14 illustrates these gaps. The bird in this figure is the third item in the grid. But because it's a larger item, it doesn't fit in the space to the right of the second image, the eagle. Instead, it has dropped down to the next grid track.

When you don't specifically position items on a grid, they are positioned automatically by the grid item placement algorithm. By default, this algorithm places grid



Figure 6.14 Increasing the size of some grid items introduced gaps in the layout where the large items don't fit.

items column by column, row by row, according to the order of the items in the markup. When an item doesn't fit in one row (that is, it spans too many grid tracks), the algorithm moves to the next row, looking for space large enough to accommodate the item. In this case, the bird is moved down to the second row, beneath the eagle.

The Grid Layout Module provides another property, `grid-auto-flow`, that can be used to manipulate the behavior of the placement algorithm. Its initial value, `row`, behaves as I've described. Given the value `column`, it instead places items in the columns first, moving to the next row only after a column is full.

You can also add the keyword `dense` (for example, `grid-auto-flow: column dense`). This causes the algorithm to attempt to fill gaps in the grid, even if it means changing the display order of some grid items. If you apply this to your page, smaller grid items will “backfill” the gaps created by the larger grid items. The result is shown in figure 6.15.

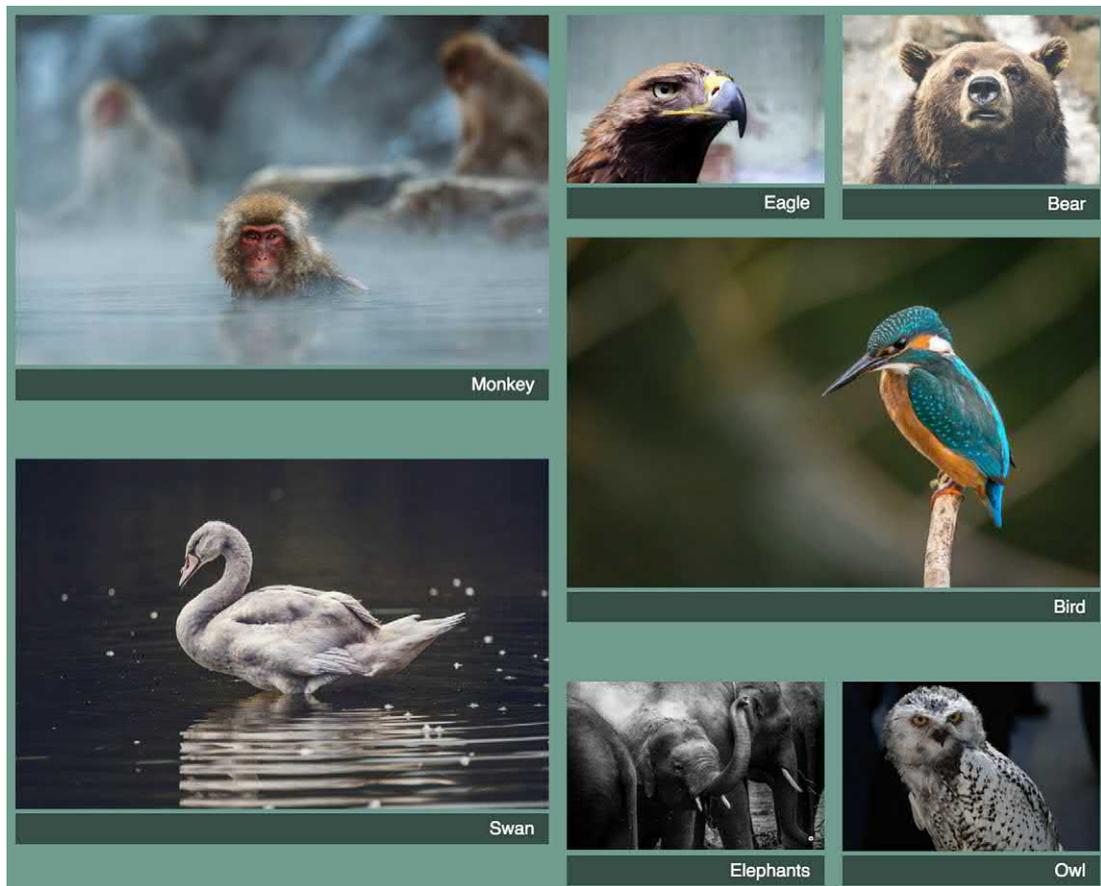


Figure 6.15 Using dense grid auto-flow allows small grid items to backfill gaps in the grid.

With the dense auto-flow option, smaller grid items fill the gaps left by larger items. The source order here is still the monkey, eagle, bird, then bear, but the bear is moved into position before the bird, thus filling the gap.

Add the next listing to your stylesheet. This enlarges featured images to fill two grid tracks in each direction and applies a dense auto-flow.

Listing 6.10 Enlarging featured images

```
.portfolio {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  grid-auto-rows: 1fr;
  grid-gap: 1em;
  grid-auto-flow: dense;
}

.portfolio .featured {
  grid-row: span 2;
  grid-column: span 2;
}
```

← Enables the dense grid placement algorithm

Enlarges featured images to span two grid tracks in each direction

This listing uses the declaration `grid-auto-flow: dense`, which is equivalent to `grid-auto-flow: row dense`. (In the first part of the value, row is implied because it's the initial value.) Then it targets the featured items and sets them to span two grid tracks in each direction. Note this example uses only the `span` keyword and doesn't expressly place any grid items on a specific grid track. This allows the grid item placement algorithm to position the grid items where it sees fit.

Depending on your viewport size, your screen may not match figure 6.12 exactly. That's because you used `auto-fill` to determine the number of vertical grid tracks. A larger screen will have room for more tracks; a smaller screen will have fewer. I took this screenshot with a viewport about 1,000 px wide, producing four grid tracks. Resize your browser width to various sizes to see how the grid automatically responds, filling available space.

Use caution with a dense auto-flow because items may not be displayed in the same order as they appear in the HTML. This can cause some confusion for users navigating via the keyboard (Tab key) or a screen reader as those methods navigate according to source order, not display order.

Subgrids

One limitation of grid is the specific DOM structure required—namely, all grid items must be direct children of the grid container. Thus, it's not possible to align deeply nested elements on the grid.

You can give the grid item `display: grid` to create an inner grid within the outer one. But the grid items of the inner grid will not necessarily align to the grid tracks of the

outer grid. Nor will the size of items in one grid affect the size of the grid tracks in the other grid.

In the future, the solution to this will be *subgrids*. By applying `display: subgrid` to a grid item, it becomes its own inner grid container, with grid tracks that align to the grid tracks of the outer grid. Unfortunately, this feature is not yet implemented in any browser and has been pushed back to the Level 2 version of the specification.

This feature is highly anticipated. Keep an eye out for it.

6.4.2 Adjusting grid items to fill the grid track

You now have a fairly complex layout. You didn't have to do a lot of work to place each item in a precise location, but rather allowed the browser to figure that out for you.

One last issue remains: the larger images aren't completely filling the grid cells, which leaves a small gap beneath them. Ideally, both the top and bottom edges of each grid item should align with others on the same grid track. Our top edges align, but the bottom edges don't as shown in figure 6.16.

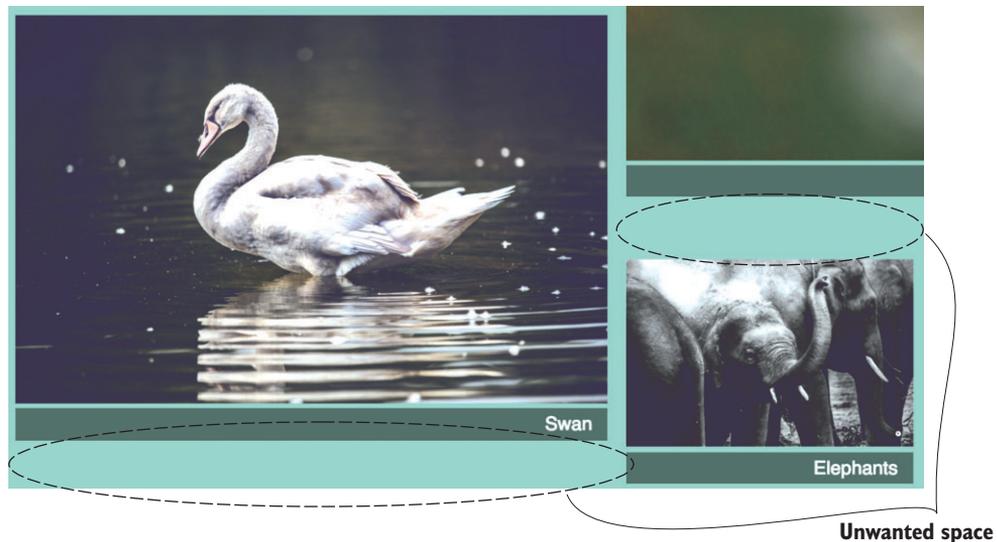


Figure 6.16 Images are not entirely filling the grid cells, leaving an unwanted gap.

Let's fix that gap. If you recall, each grid item is a `<figure>` that contains two child elements—an image and a caption:

```
<figure class="featured">
  
  <figcaption>Monkey</figcaption>
</figure>
```

By default, each grid item is stretched to fill the entire grid area, but its child elements are not stretched to fill it, so the grid area has some unused height. An easy way to fix this is with flexbox. In listing 6.11, you'll make each `<figure>` a flex container with a direction of `column` so items stack vertically, atop one another. You can then apply a flex grow to the image, forcing it to stretch to fill the space.

Stretching an image is problematic, however. This will change its height-to-width ratio, distorting the picture. Fortunately, CSS provides a special property for controlling this behavior, `object-fit`. By default, an `` has an `object-fit` value of `fill`, meaning the entire image will be resized to fill the `` element. You can also set other values to change this.

For example, the `object-fit` property accepts the values `cover` and `contain` (illustrated in figure 6.17). These values tell the browser to resize the image within the rendered box, without distorting its aspect ratio.

- To expand the image to fill the box (resulting in part of the image being cut off), use `cover`.
- To resize the image so that it fits entirely in the box (resulting in empty space within the box), use `contain`.

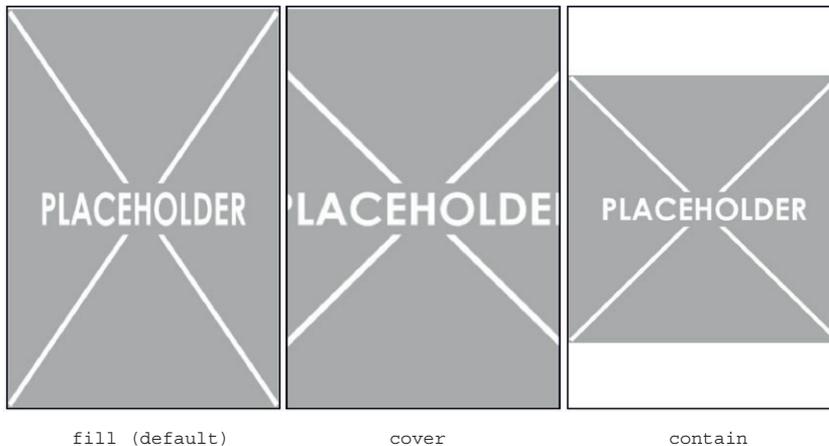


Figure 6.17 Using `object-fit` to control how an image is rendered in its box

There's an important distinction to make here: there is the box (determined by the `` element's height and width), and there is the rendered image. By default, these are the same size. The `object-fit` property lets you manipulate the size of the rendered image within that box, but the size of the box itself remains unchanged.

Because you'll use the `flex-grow` property to stretch the images, you should also apply `object-fit: cover` to prevent the images from being distorted. This will crop off a small bit of the edge of the images, which is a compromise you'll have to make.

The end result is shown in figure 6.18. For a more detailed look at this property, see <https://css-tricks.com/on-object-fit-and-object-position/>.

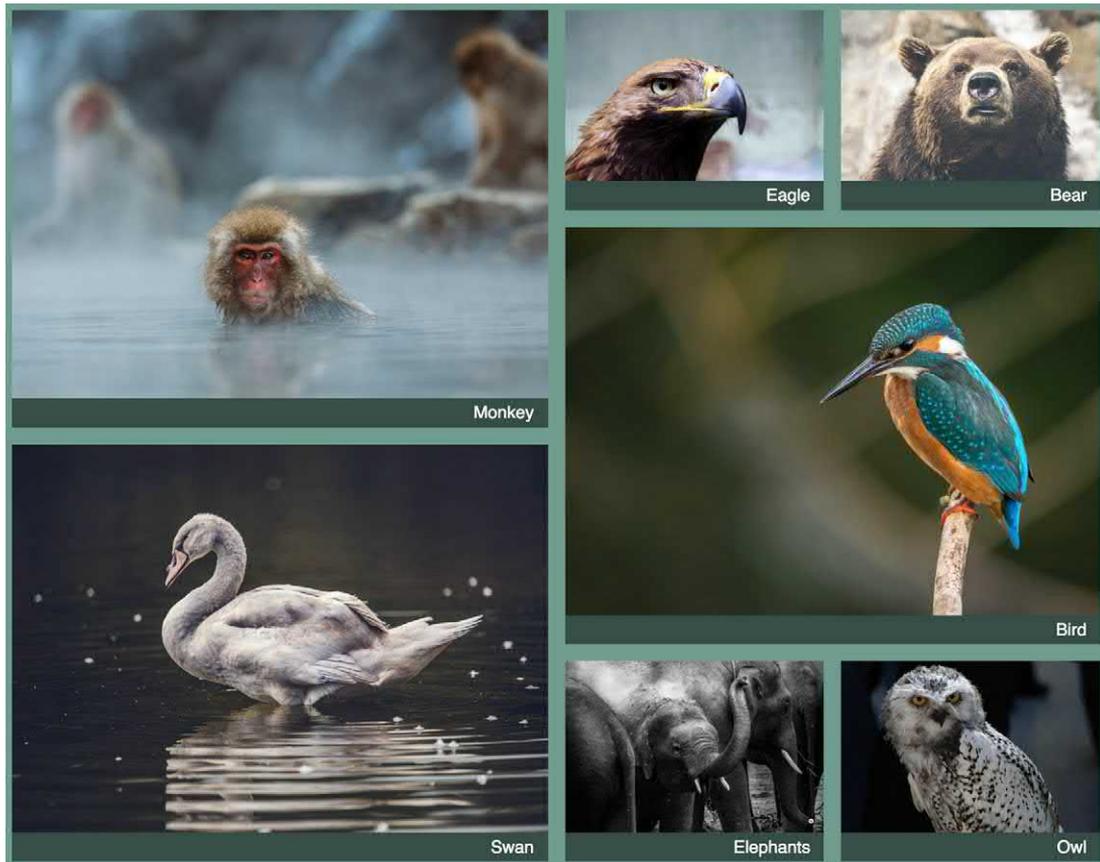


Figure 6.18 All images now fill their grid areas and align cleanly.

Now the top and bottom edges of all the images and their captions align in each grid track. The code for this is shown here. Add it to your stylesheet.

Listing 6.11 Using a column flexbox to stretch images to fill the grid area

```
.portfolio > figure {
  display: flex;
  flex-direction: column;
  margin: 0;
}

.portfolio img {
  flex: 1;
}
```

Makes each grid item a vertical flexbox

Uses flex grow to make the image fill the available space in the flex container

```

object-fit: cover;
max-width: 100%;
}

```

← Allows the image to fill the box without being stretched (cropping edges instead)

This completes the design of your photography portfolio. Everything aligns in a neat grid, and the browser makes decisions for you regarding the number and size of each vertical grid track. Using a dense auto-flow allows the browser to fill in gaps neatly.

6.5 *Feature queries*

Now that you have a general grasp of grid layout, you might be wondering: Do you have to wait until all browsers support grid before you can start using it? The answer is, no. You can start using it today if you want. You'll need to consider what you want the browser to do for your layout if it doesn't support grid and provide those styles as a fallback.

CSS has recently added something called a *feature query* that can help with this. It looks like this:

```

@supports (display: grid) {
  ...
}

```

The `@supports` rule is followed by a declaration in parentheses. If the browser understands the declaration (in this case, it supports grid), it applies any rulesets that appear between the braces. If it doesn't understand this, it won't apply them.

This means you can provide one set of styles using older layout technologies like floats. These will not necessarily be ideal styles (you'll have to make some compromises), but it will get the job done. Then, using a feature query, apply the full-featured layout using grid.

Let's use this in the portfolio. You can provide a more basic layout for older browsers using inline-block elements, then put all the code related to the grid layout inside a feature query. Browsers that don't support grid will render the page as in figure 6.19.

This layout has a couple of compromises: featured images will not be shown at a larger size and columns will be a fixed 300 px rather than stretching to fill the available screen width. Because the figures are displayed `inline-block`, they'll line wrap normally. This allows for more images per row when there is enough screen space for them.

The code for this (including a feature query) is shown in listing 6.12. Update your stylesheet to match.

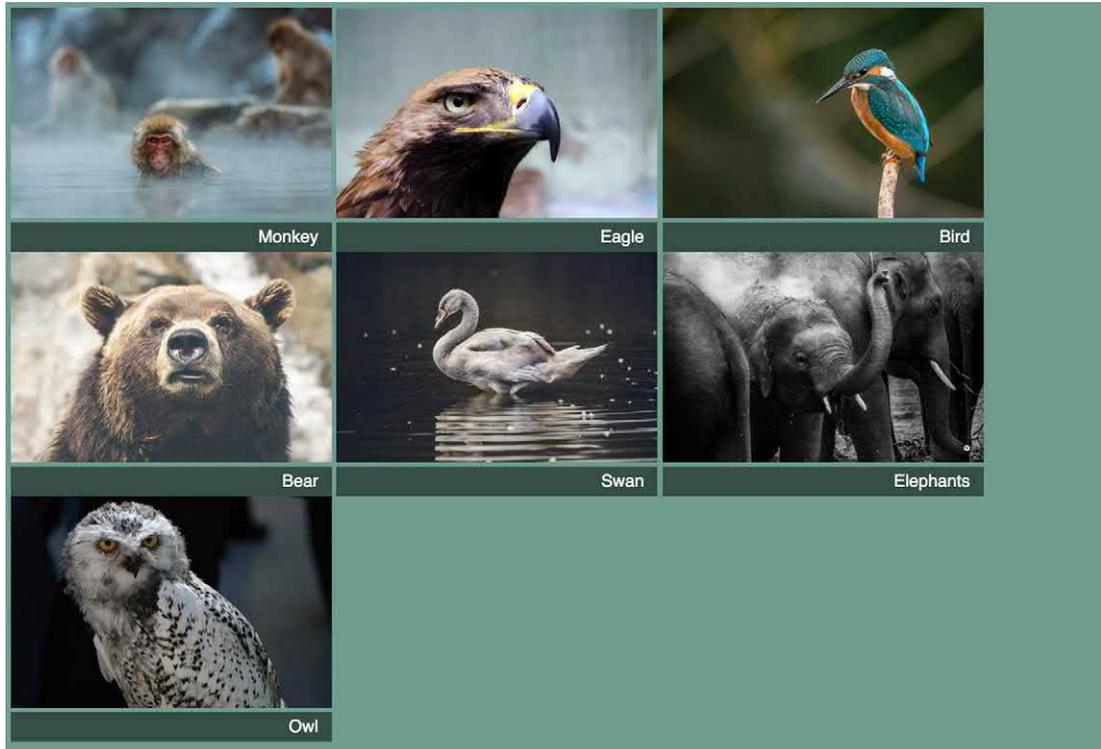


Figure 6.19 Fallback layout for browsers that don't support grid

Listing 6.12 Using a feature query for progressive enhancement

```

.portfolio > figure {
  display: inline-block;
  max-width: 300px;
  margin: 0;
}

.portfolio img {
  max-width: 100%;
  object-fit: cover;
}

.portfolio figcaption {
  padding: 0.3em 0.8em;
  background-color: rgba(0, 0, 0, 0.5);
  color: #fff;
  text-align: right;
}

@supports (display: grid) {
  .portfolio {

```

Uses inline-block for the fallback layout

Feature query for grid support

```

display: grid;
grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
grid-auto-rows: 1fr;
grid-gap: 1em;
grid-auto-flow: dense;
}

.portfolio > figure {
display: flex;
flex-direction: column;
max-width: initial;
}

.portfolio img {
flex: 1;
}

.portfolio .featured {
grid-row: span 2;
grid-column: span 2;
}
}

```

←
Puts all grid layout styles within feature query block

←
Overrides fallback styles

The fallback and other basic styles such as colors are outside of the feature query block, so they'll always apply. If you open the page in a browser that doesn't support grid, you'll see the fallback layout (figure 6.19). All styles relating to the grid-based layout are within the feature query block, so they'll only apply if the browser supports grid.

The `@supports` rule can be used to query for all sorts of CSS features. Use `@supports (display: flex)` to query for flexbox support, or `@supports (mix-blend-mode: overlay)` to query for blend mode support (see chapter 11 for more on blend modes).

WARNING IE doesn't support the `@supports` rule. That browser ignores any rules within the feature query block, regardless of the actual feature support. This is usually okay, as you'll want the older browser to render the fallback layout.

Feature queries may be constructed in a few other ways as well:

- `@supports not (<declaration>)`—Only apply rules in the feature query block if the queried declaration isn't supported
- `@supports (<declaration>)` or `(<declaration>)`—Apply rules if either queried declaration is supported
- `@supports (<declaration>) and (<declaration>)`—Apply rules only if both queried declarations are supported

These can be combined as well to query for more complex situations. The `or` keyword can be useful to query for support using prefixed properties:

```
@supports (display: grid) or (display: -ms-grid)
```

This declaration will target browsers that support the un-prefixed version of the property as well as older versions of MS Edge, which require the `-ms-` prefix. I caution that partial grid support in older versions of Edge is not as robust as modern browsers. Chances are, it's more trouble than it's worth trying to make the prefixed `@supports` query work, and you'll be better off omitting it. This will leave older versions of Edge to render your fallback layout.

6.6 Alignment

The grid module makes use of several of the same alignment properties that flexbox uses, as well as a couple of new ones. I've covered most of these in the previous chapter, but let's look to see how they apply to a grid. If you need more control over various aspects of a grid layout, these properties may come in handy.

CSS provides three justify properties: `justify-content`, `justify-items`, `justify-self`. These properties control horizontal placement. I remember this by thinking about justifying text in a word processor, which spreads out text horizontally.

And there are three alignment properties: `align-content`, `align-items`, `align-self`. These control vertical placement. I remember this by thinking about the `vertical-align` property from table layouts. These properties are each illustrated in figure 6.20.

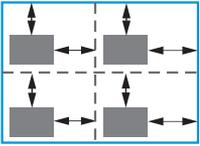
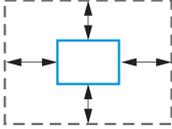
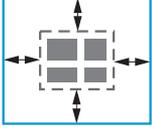
Properties	Applies to	Aligns
<code>justify-items</code> <code>align-items</code>	Grid container	Items within grid areas 
<code>justify-self</code> <code>align-self</code>	Grid item	Item within grid area 
<code>justify-content</code> <code>align-content</code>	Grid container	Grid tracks within container 

Figure 6.20 Alignment properties for a grid

You can use `justify-content` and `align-content` to place the grid tracks horizontally and vertically within the grid container. This becomes necessary if the total size of the grid doesn't fill the size of the grid container. Consider this code:

```
.grid {
  display: grid;
```

```
height: 1200px;
grid-template-rows: repeat(4, 200px);
}
```

It explicitly sets the height of a grid container to 1,200 px, but only defines 800 px worth of horizontal grid tracks. The `align-content` property specifies how to distribute the remaining 400 px of space. Supported values are as follows:

- `start`—Places grid tracks to the top/left of the grid container (use `flex-start` in a flexbox)
- `end`—Places grid tracks to the bottom/right of the grid container (use `flex-end` in a flexbox)
- `center`—Places grid tracks in the middle of the grid container
- `stretch`—Resizes the tracks to fill the size of the grid container
- `space-between`—Evenly distributes remaining space between each grid track (effectively overriding any `grid-gap`)
- `space-around`—Distributes space between each grid track, with a half-sized space on each end
- `space-evenly`—Distributes space between each grid track, with an equal amount of space on each end (not supported in flexbox)

For detailed examples of all these justify/alignment properties, visit <http://gridby-example.com/>. This is an excellent resource. It's a large collection of grid examples put together by Rachel Andrew, a developer and member of the W3C.

Because there's a lot to grid layout, I've taken you through the essential concepts you'll need to know. I encourage you to experiment with grids further. There are far more ways to mix and match these than I could cover in one chapter, so challenge yourself to try new things. When you come across an interesting page layout on the web, see if you can replicate it using grids.

Summary

- Grid is excellent for a high-level layout of the web page (but it is not limited to that).
- You can use grid in conjunction with flexbox for a complete layout system.
- You should use whichever syntax (numbered grid lines, named grid lines, or named grid areas) is the most intuitive for you and the given situation.
- You can use `auto-fill`/`auto-fit` and the implicit grid for layouts with a large or unknown number of grid items.
- You can use feature queries for progressive enhancement.

7

Positioning and stacking contexts

This chapter covers

- The types of element positioning: fixed, relative, and absolute
- Building modal dialogs and dropdown menus
- CSS triangles
- Understanding z-index and stacking contexts
- A new type of positioning: sticky

We've now looked at multiple ways to control the layout of the page, from table display to flexbox to floats. In this chapter, we'll look at one important technique: the `position` property, which you can use to build dropdown menus, modal dialogs, and other essential effects for modern web applications.

Positioning can get complicated. It's a subject where many developers only have a cursory understanding. Without a complete grasp on positioning, and the ramifications involved, it's easy to dig yourself into a hole. You can find yourself with the wrong elements in front of others and correcting the problem isn't always straightforward.

As we look at the various types of positioning, I'll make sure you understand precisely what they do. Then, you'll learn about something called a stacking context,

which is a sort of hidden side effect of positioning. Understanding the stacking context will keep you out of trouble, and if you ever find yourself “out in the weeds” with a page layout, this understanding will give you the tools you need to get back on track.

The initial value of the `position` property is `static`. Everything we’ve done in previous chapters is with static positioning. When you change this value to anything else, the element is said to be *positioned*. An element with static positioning is thus *not positioned*.

The layout methods we’ve covered in previous chapters do various things to manipulate the way document flow behaves. Positioning is different: it removes elements from the document flow entirely. This allows you to place the element somewhere else on the screen. It can place elements in front of or behind one another, thus overlapping one another.

7.1 **Fixed positioning**

Fixed positioning, although not as common as some of the other types of positioning, is probably the simplest to understand, so we’ll start there. Applying `position: fixed` to an element lets you position the element arbitrarily within the viewport. This is done with four companion properties: `top`, `right`, `bottom`, and `left`. The values you assign to these properties specify how far the fixed element should be from each edge of the browser viewport. For example, `top: 3em` on a fixed element means its top edge will be 3 em from the top of the viewport.

By setting these four values, you also implicitly define the width and height of the element. For example, specifying `left: 2em; right: 2em` means the left edge of the element will be 2 em from the left side of the viewport, and the right edge will be 2 em from the right side; thus, the element will be 4 em less than the total viewport width. Likewise with `top`, `bottom`, and the viewport height.

7.1.1 **Creating a modal dialog with fixed positioning**

Let’s use these properties to build the modal dialog box shown in figure 7.1. This dialog will pop up in front of the page content, masking it from view until the dialog is closed.

Typically, you’ll use a modal dialog to require the user to read something or to input something before continuing. For example, this modal (figure 7.1) displays a form where a user can sign up for a newsletter. You’ll initially hide the dialog with `display: none`, and you’ll then use a bit of JavaScript to change its value to `block` in order to open the modal.

Create a new page and add the following listing to the `<body>` element. It places the content inside two containing elements and uses a `<script>` tag with some JavaScript to provide basic functionality.

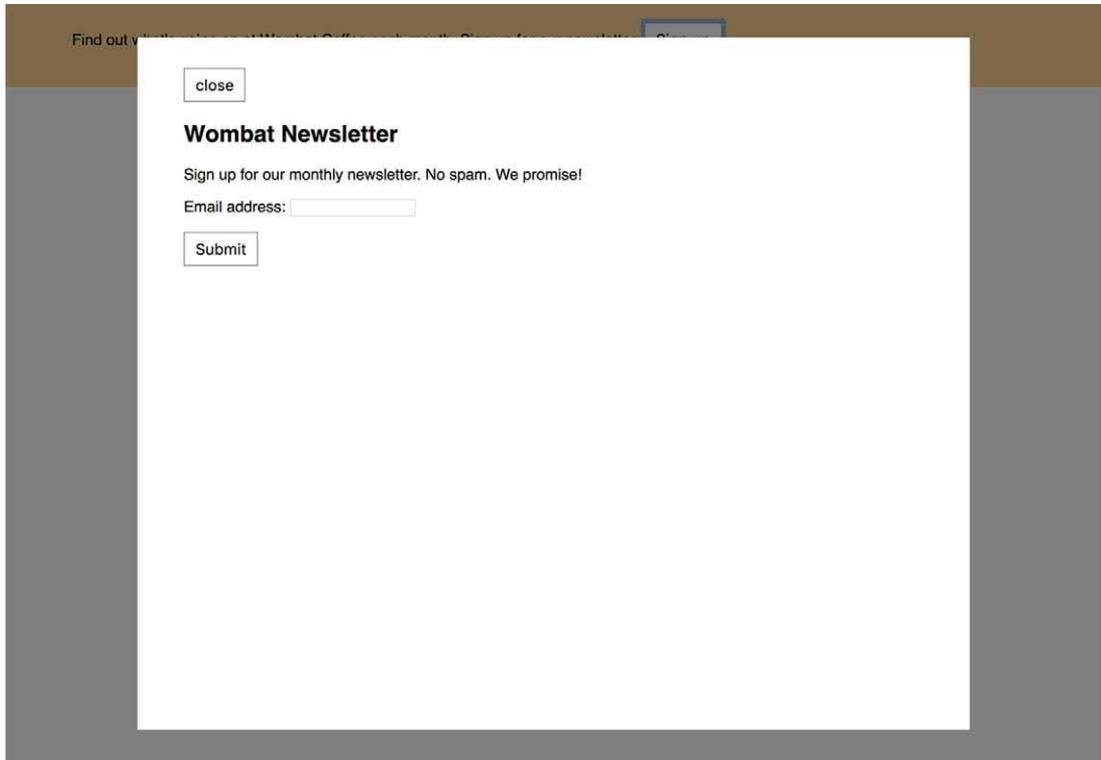


Figure 7.1 A modal dialog box

Listing 7.1 Creating a modal dialog box

```

<header class="top-banner">
  <div class="top-banner-inner">
    <p>Find out what's going on at Wombat Coffee each
      month. Sign up for our newsletter:
      <button id="open">Sign up</button>
    </p>
  </div>
</header>
<div class="modal" id="modal">
  <div class="modal-backdrop"></div>
  <div class="modal-body">
    <button class="modal-close" id="close">close</button>
    <h2>Wombat Newsletter</h2>
    <p>Sign up for our monthly newsletter. No spam.
      We promise!</p>
    <form>
      <p>
        <label for="email">Email address:</label>
        <input type="text" name="email"/>
      </p>
    </form>
  </div>
</div>

```

Button to trigger the modal
 Container for the modal
 A "backdrop" to obscure content behind the modal
 Modal contents

```

        </p>
        <p><button type="submit">Submit</button></p>
    </form>
</div>
</div>

<script type="text/javascript">
    var button = document.getElementById('open');
    var close = document.getElementById('close');
    var modal = document.getElementById('modal');

    button.addEventListener('click', function (event) {
        event.preventDefault();
        modal.style.display = 'block';
    });

    close.addEventListener('click', function (event) {
        event.preventDefault();
        modal.style.display = 'none';
    });
</script>

```

Opens the modal when the user clicks the Sign up button

Closes the modal when the user clicks the Close button

The first element in this listing is a banner across the top of the page. It contains the button that triggers the modal. The second element is the modal dialog. It includes an empty `modal-backdrop`, which you'll use to obscure the rest of the page, drawing the user's focus to the contents of the dialog. The contents are inside a `modal-body` element.

The CSS to implement this is shown next. Add it to your stylesheet. It includes basic styling for the top banner as well as the styles for the modal.

Listing 7.2 Adding modal styles

```

body {
    font-family: Helvetica, Arial, sans-serif;
    min-height: 200vh;
    margin: 0;
}

button {
    padding: 0.5em 0.7em;
    border: 1px solid #8d8d8d;
    background-color: white;
    font-size: 1em;
}

.top-banner {
    padding: 1em 0;
    background-color: #ffd698;
}

.top-banner-inner {
    width: 80%;

```

Forces the page height to enable scrolling (for demo purposes only)

```

max-width: 1000px;
margin: 0 auto;
}

.modal {
  display: none;
}

.modal-backdrop {
  position: fixed;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  background-color: rgba(0, 0, 0, 0.5);
}

.modal-body {
  position: fixed;
  top: 3em;
  bottom: 3em;
  right: 20%;
  left: 20%;
  padding: 2em 3em;
  background-color: white;
  overflow: auto;
}

.modal-close {
  cursor: pointer;
}

```

Hides the modal by default; JavaScript will set `display: block` when it opens the modal.

Uses a semi-transparent backdrop to obscure the rest of the page while the modal is open

Positions the main part of the modal

Allows the modal body to scroll, if necessary

In this CSS, you’ve used fixed positioning twice. First on the `modal-backdrop` with each of the four sides set to 0. This makes the backdrop fill the entire viewport. You’ve given it a background color of `rgba(0, 0, 0, 0.5)`. This color notation specifies red, green, and blue values of 0, which evaluates to black. The fourth value is the “alpha” channel, which specifies its transparency: a value of 0 is completely transparent, and 1 is completely opaque. The value 0.5 is half transparent. This serves to darken all of the page contents behind the element.

The second place you used fixed positioning is in the `modal-body`. You’ve positioned each of its four sides inside the viewport: 3 em from the top and bottom edges and 20% from the left and right sides. You also set a white background color. This makes the modal a white box, centered on the screen. You can scroll the page however you wish, but the backdrop and the modal body remain in place.

Load the page and you’ll see a pale yellow banner across the top of the screen with a button. Click the button to open the positioned modal. Because of fixed positioning, the modal remains in place, even when you scroll the page (the large `min-height` value on the body enables scrolling to illustrate this behavior).

Click the close button at the top of the modal to close it. This button is in an odd location now, but we’ll come back and position it in a bit.

7.1.2 **Controlling the size of positioned elements**

When positioning an element, you're not required to specify values for all four sides. You can specify only the sides you need and then use `width` and/or `height` to help determine its size. You can also allow the element to be sized naturally. Consider these declarations:

```
position: fixed;
top: 1em;
right: 1em;
width: 20%
```

These would affix the element 1 em from the top and right edges of the viewport with a width 20% of the viewport width. By omitting both `bottom` and `height` properties, the element's height will be determined naturally by its contents. This can be used to affix a navigational menu to the screen, for example. It'll remain in place even as the user scrolls down through the content on the page.

Because a fixed element is removed from the document flow, it no longer affects the position of other elements on the page. They'll follow the normal document flow as if it's not there, which means they'll often flow behind the fixed element, hidden from view. This is usually fine with a modal dialog because you want it to be front and center until the user dismisses it.

With something persistent, such as a side navigational menu, you'll need to take care that other content doesn't flow behind it. This is usually easiest to do by adding a margin to the content. For instance, place all your content in a container with a `right-margin: 20%`. This margin will flow behind your fixed element, and the content won't overlap.

7.2 **Absolute positioning**

Fixed positioning, as you've just seen, lets you position an element relative to the viewport. This is called its *containing block*. The declaration `left: 2em`, for example, places the left edge of a positioned element 2 em from the left edge of the containing block.

Absolute positioning works the same way, except it has a different containing block. Instead of its position being based on the viewport, its position is based on the closest positioned ancestor element. As with a fixed element, the properties `top`, `right`, `bottom`, and `left` place the edges of the element within its containing block.

7.2.1 **Absolutely positioning the Close button**

To see how this works, let's reposition the Close button to the top right corner of your modal dialog. After doing so, the modal will look like figure 7.2.

To do this, you'll declare absolute positioning for the Close button. Its parent element is `modal-body`, which has fixed positioning, so it becomes the containing block for the button. Edit your button's styles to match listing 7.3.

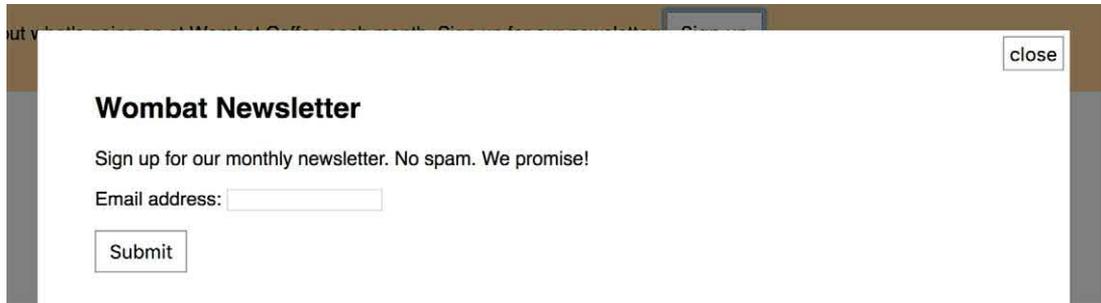


Figure 7.2 Close button positioned at top right corner of modal dialog

Listing 7.3 Absolutely positioned Close button

```
.modal-close {
  position: absolute;
  top: 0.3em;
  right: 0.3em;
  padding: 0.3em;
  cursor: pointer;
}
```

This listing places the button 0.3 em from the top and 0.3 em from the right of the modal-body. Typically, as in this example, the containing block will be the element's parent. In cases where the parent element is not positioned, then the browser will look up the DOM hierarchy at the grandparent, great-grandparent, and so on until a positioned element is found, which is then used as the containing block.

NOTE If none of the element's ancestors are positioned, then the absolutely positioned element will be positioned based on something called the *initial containing block*. This is an area with dimensions equal to the viewport size, anchored at the top of the page.

7.2.2 Positioning a pseudo-element

You've positioned the Close button where you want it, but it's rather spartan. For a Close button such as this, users typically expect to see some graphical indication such as an *x* as in figure 7.3.

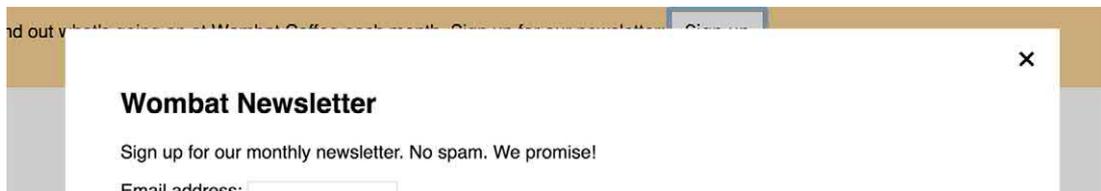


Figure 7.3 Close button replaced with an x

Your first temptation may be to remove the word *close* from your markup and replace it with an *x*, but this would introduce an accessibility problem: Assistive screen readers read the text of the button, so it should give some meaningful indication of the button's purpose. The HTML must make sense on its own before the CSS is applied.

Instead, you can use CSS to hide the word *close* and display an *x*. You'll accomplish this by doing two things. First, you'll push the button's text outside the button and hide the overflow. Second, you'll use the content property to add the *x* to the button's `::after` pseudo-element and absolute positioning to center it within the button. Update you button's styles to match listing 7.4.

TIP Instead of the letter *x*, I recommend the unicode character for the multiplication sign. It's more symmetrical and usually more esthetically pleasing for this purpose. The HTML character `×` will render as this character, but in the CSS content property, you must use the escaped unicode number: `\00D7`.

Listing 7.4 Replacing the Close button with an x

```
.modal-close {
  position: absolute;
  top: 0.3em;
  right: 0.3em;
  padding: 0.3em;
  cursor: pointer;
  font-size: 2em;
  height: 1em;
  width: 1em;
  text-indent: 10em;
  overflow: hidden;
  border: 0;
}

.modal-close::after {
  position: absolute;
  line-height: 0.5;
  top: 0.2em;
  left: 0.1em;
  text-indent: 0;
  content: "\00D7";
}
```

Makes the button a small square

Forces the text to overflow the element and hides it

Adds the unicode character U+00D7 (multiplication sign)

This listing explicitly sets the button size to 1 em square. The `text-indent` property then pushes the text to the right, outside of the element. The exact value doesn't matter as long as it's more than the width of the button. Then, because `text-indent` is an inherited property, you reset it to 0 on the pseudo-class so the *x* isn't also indented.

The pseudo-class is now absolutely positioned. It behaves like a child element of the button, so the button being positioned becomes the containing block for its pseudo-element. The short `line-height` keeps the pseudo-element from being too

tall, and the top and left properties position it in the center of the button. I arrived at the exact values here through some trial and error; I encourage you to experiment with those values in your browser's development tools to see how they affect the positioning.

Absolute positioning is the heavy hitter among the positioning types. It's used often in conjunction with JavaScript for popping up menus, tooltips, and "info" boxes. We'll use it to build a dropdown menu, but to do that, you'll need to take a look at its companion, relative positioning.

7.3 Relative positioning

Relative positioning is probably the least understood positioning type. When you first apply `position: relative` to an element, you won't usually see any visible change on the page. The relatively positioned element, and all elements on the page around it, will remain exactly where they were (though you may see some changes regarding which elements are in front of which. We'll get to that in a bit).

The `top`, `right`, `bottom`, and `left` properties, if applied, will shift the element from its original position, but it won't change the position of any elements around it. See figure 7.4 for an example. It shows four inline-block elements. I've applied three additional properties to the second element: `position: relative`; `top: 1em`; `left: 2em`. As you can see, this has shifted the element from its initial position, but the other elements are unaffected. They still follow normal document flow around the initial position of the shifted element.

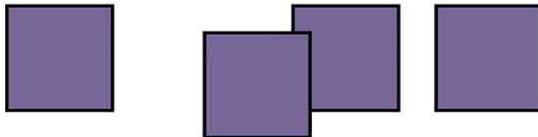


Figure 7.4 The second element shifted using relative positioning

Applying `top: 1em` moves the element down 1 em (away from its original top edge). And `left: 2em` shifts the element right 2 em (away from its original left edge). These shifts may cause that element to overlap elements below or beside it. When positioning, negative values are supported as well, so `bottom: -1em` would shift the element down 1 em, just as `top: 1em` does.

NOTE Unlike fixed and absolute positioning, you cannot use `top`, `right`, `bottom`, and `left` to change the size of a relatively positioned element. Those values will only shift the position of the element up or down, left or right. You can use `top` or `bottom`, but not both together (`bottom` will be ignored); likewise, you can use `left` or `right`, but not both (`right` will be ignored).

Using these properties to adjust the position of a relative element may be useful to nudge an element into place on occasion, but truth be told, this is rarely why you'll

use relative positioning. Far more often, you'll use `position: relative` to establish the containing block for an absolutely positioned element inside it.

7.3.1 Creating a dropdown menu

Let's use relative and absolute positioning to create a dropdown menu. Initially it'll be a simple rectangle, but when the user hovers the mouse over it, it pops open a list of links like those shown in figure 7.5.

Listing 7.5 shows the markup for this menu. Append it to your HTML after the closing `</div>` of the `<div class="modal">` element. This code includes a container that you'll use to center the content and align it with the banner's content. I've also added an `<h1>` below the popup to illustrate how the popup appears in front of other content on the page.



Figure 7.5 Dropdown menu

Listing 7.5 Adding the dropdown menu's HTML

```
<div class="container">
  <nav>
    <div class="dropdown">
      <div class="dropdown-label">Main Menu</div>
      <div class="dropdown-menu">
        <ul class="submenu">
          <li><a href="/">Home</a></li>
          <li><a href="/coffees">Coffees</a></li>
          <li><a href="/brewers">Brewers</a></li>
          <li><a href="/specials">Specials</a></li>
          <li><a href="/about">About us</a></li>
        </ul>
      </div>
    </div>
  </nav>

  <h1>Wombat Coffee Roasters</h1>
</div>
```

Container for the dropdown

Label will always be visible.

Div to show and hide as the menu is opened and closed

The dropdown container includes two children: the label, a gray rectangle that's always visible, and the dropdown menu that will show and hide as the dropdown is opened and closed. The dropdown menu will be absolutely positioned so it doesn't shift around the layout of the page when it's revealed. This means it appears in front of other content when it's visible.

Next, you'll apply relative positioning to the dropdown container. This establishes the containing block for the absolutely positioned menu. Add these styles to your stylesheet.

Listing 7.6 Opening the dropdown menu on hover

```
.container {
  width: 80%;
  max-width: 1000px;
  margin: 1em auto
}

.dropdown {
  display: inline-block;
  position: relative;
}

.dropdown-label {
  padding: .5em 1.5em;
  border: 1px solid #ccc;
  background-color: #eee;
}

.dropdown-menu {
  display: none;
  position: absolute;
  left: 0;
  top: 2.1em;
  min-width: 100%;
  background-color: #eee;
}

.dropdown:hover .dropdown-menu {
  display: block;
}

.submenu {
  padding-left: 0;
  margin: 0;
  list-style-type: none;
  border: 1px solid #999;
}

.submenu > li + li {
  border-top: 1px solid #999;
}

.submenu > li > a {
  display: block;
  padding: .5em 1.5em;
  background-color: #eee;
  color: #369;
  text-decoration: none;
}

.submenu > li > a:hover {
  background-color: #fff;
}
```

Establishes the containing block

Hides the menu initially

Positions the menu below the dropdown menu

Reveals the menu on hover

Now, when you move your mouse pointer over the Main Menu label, the dropdown menu pops open beneath it. Notice that you used the `:hover` state of the whole container to open the menu. This means that as long as the cursor remains over any of its contents—either the `dropdown-label` or the `dropdown-menu`—the menu remains open.

On the absolutely positioned `dropdown-menu`, you used `left: 0` to align its left side with the left side of the dropdown. Then you used `top: 2.1em` to place its top edge beneath the label (with the padding and border, the label is about 2.1 em tall). A `min-width: 100%` ensures it'll be at least as wide as the dropdown container (whose width is determined by `dropdown-label`). You then used the `submenu` class to style the menu inside the dropdown. (If you open the modal dialog at this point, you might notice it appears behind the dropdown menu in an odd way. That's okay; we'll address that issue soon.)

Important concerns when using dropdown menus

For simplicity, the example in listing 7.6 uses a `:hover` pseudo-class to open the menu when the user hovers the mouse cursor over it. This example is not complete. Typically, a more robust approach would be to use JavaScript to add and remove a class that controls whether the menu is opened. This lets you add a slight delay before opening or closing the menu to prevent the user from unintentionally triggering it when mousing quickly past.

Furthermore, although the example here works with a mouse, it won't work on some touchscreen devices (only certain touchscreen devices trigger the `:hover` state on tap). It also doesn't address the accessibility concerns of a screen reader or of keyboard navigation. It would be prudent to augment the dropdown to ensure touchscreen controls work and the menu stays open when the user uses the Tab key to navigate through the links in the menu.

The JavaScript to do this is beyond the scope of this book, but if you're adept at it, you can address these concerns with the code you write there. Alternately, you can use a third-party library that provides dropdown functionality to take care of this for you, then use CSS to customize its appearance to your liking.

7.3.2 *Creating a CSS triangle*

Let's add one finishing touch to your dropdown menu. It works as is, but it may not be immediately apparent to the user that there's more content to discover along with the Main Menu label. Let's add a small down arrow to the label to indicate there's more to explore.

You can use a little trick with borders to draw a triangle that serves as the down-pointing arrow. You'll use the dropdown label's `::after` pseudo-element to draw the triangle, then use absolute positioning to put it on the right side of the label.

Most of the time, when you add borders to an element, you make them thin: usually 1 px or 2 px is enough. But observe what happens when you make a border much

thicker as shown in figure 7.6. I've made each side of the border a unique color to highlight where one edge ends and the next begins.

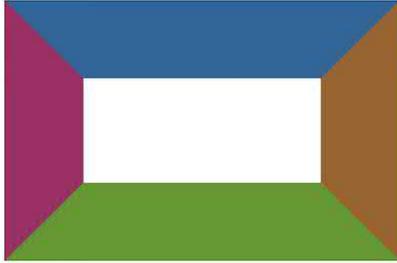


Figure 7.6 An element with thick borders

Notice what happens in the corners where the borders from two edges meet: they form a diagonal edge. Now observe what happens if you shrink the element to a height and width of zero (figure 7.7). The borders all come together and join in the middle.

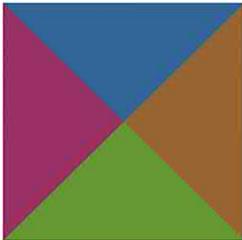


Figure 7.7 When the element has no height or width, each border forms a triangle.

The border for each side of the element forms a triangle. The top border points down; the right border points left, and so on. With this in mind, you can use one border to give you the triangle you need and then set the color of the remaining borders to transparent. An element with transparent left and right borders and a visible top border will look like figure 7.8—a simple triangle.



Figure 7.8 Triangle formed from an element's top border

Let's apply styles to the `dropdown-label::after` pseudo-element to make a triangle and position it using absolute positioning. Add this to your stylesheet.

Listing 7.7 Absolutely positioning a triangle on the dropdown label

```

.dropdown-label {
  padding: 0.5em 2em 0.5em 1.5em;
  border: 1px solid #ccc;
  background-color: #eee;
}

.dropdown-label::after {
  content: "";
  position: absolute;
  right: 1em;
  top: 1em;
  border: 0.3em solid;
  border-color: black transparent transparent;
}

.dropdown:hover .dropdown-label::after {
  top: 0.7em;
  border-color: transparent transparent black;
}

```

← Increases the right padding to add space for the arrow

Positions the element on the right side of the label

Uses the top border to form a down arrow

← On hover, changes to an up arrow

The pseudo-element has no content, so it has no height or width. You then used the `border-color` shorthand property to set the top border black and the side and right borders transparent, giving you the down arrow. An extra bit of padding on the `dropdown-label` makes space on the right, where you can place the triangle. The final result is shown in figure 7.9.

Main Menu ▾

Figure 7.9 Dropdown label with a down arrow

When you open the menu, the arrow reverses directions, pointing upward to indicate the menu can be closed. The slight change in the `top` value (from 1 em to 0.7 em) helps the up arrow to optically appear in the same spot as the down arrow.

Alternately, you can add this arrow with the use of an image or background image, but a few extra lines of CSS can save your users the extraneous request over the network. The result of this is a small finishing touch that can add a lot of polish to your application or website.

You can use this technique to build other complicated shapes, such as trapezoids, hexagons, and stars. For a list of dozens of shapes built in CSS, see <https://css-tricks.com/examples/ShapesOfCSS/>.

7.4 Stacking contexts and z-index

Positioning is useful, but it's important to know the ramifications involved. When you remove an element from the document flow, you become responsible for all the things the document flow normally does for you.

You need to ensure the element doesn't accidentally overflow outside the browser viewport, thus becoming hidden from the user. And, you need to make sure it doesn't inadvertently cover important content you need visible.

Eventually, you'll encounter problems with stacking. When you position multiple elements on the same page, you may run into a scenario where two different positioned elements overlap. You may occasionally be surprised to find the "wrong" one appearing in front of the other. In fact, in this chapter, I've intentionally set up such a scenario to illustrate this.

On the page you've built, click the Sign up button in the page header to open the modal dialog. If you placed the markup for the dropdown after the modal in your HTML, it'll look like figure 7.10. Notice the dropdown you added to the page now appears in front of the modal.

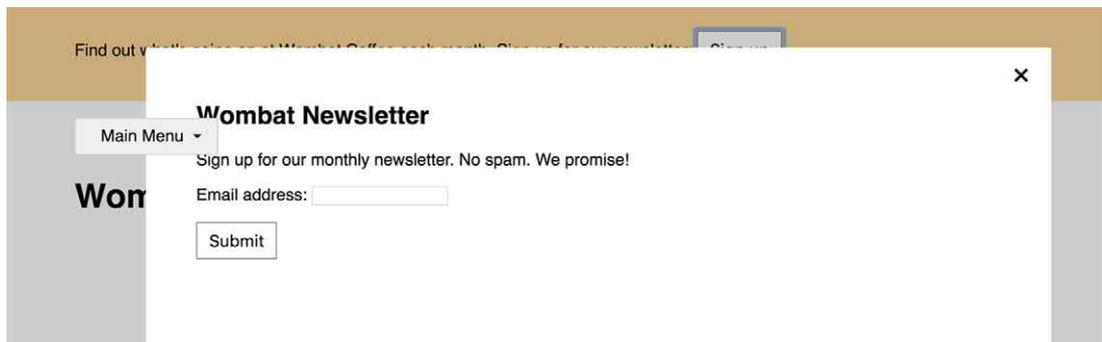


Figure 7.10 The modal dialog incorrectly appears behind the dropdown menu

You can address this problem in a couple of ways. Before we get to that, it's important to understand how the browser determines the stacking order. For that, we need to take a closer look at how the browser renders a page.

7.4.1 Understanding the rendering process and stacking order

As the browser parses HTML into the DOM, it also creates another tree structure called the *render tree*. This represents the visual appearance and position of each element. It's also responsible for determining the order in which the browser will *paint* the elements. This order is important because elements painted later appear in front of elements painted earlier, should they happen to overlap.

Under normal circumstances (that is, before positioning is applied), this order is determined by the order the elements appear in the HTML. Consider the three elements in this bit of markup:

```
<div>one</div>
<div>two</div>
<div>three</div>
```

Their stacking behavior is illustrated in figure 7.11. I've used some negative margins to force them to overlap, but I haven't applied any positioning to them. The elements appearing later in the markup are painted over the top of the previous ones.

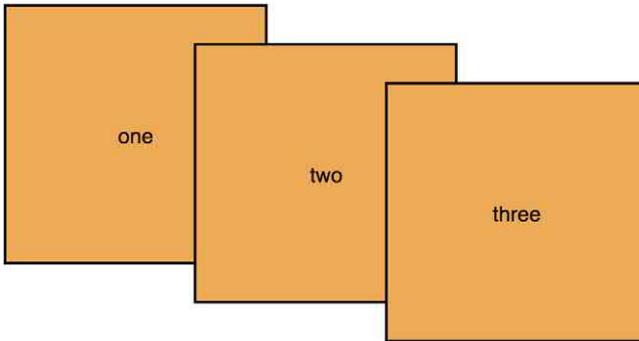


Figure 7.11 Three elements stacking normally—later elements appear in front of earlier ones

This behavior changes when you start positioning elements. The browser first paints all non-positioned elements, then it paints the positioned ones. By default, any positioned element appears in front of any non-positioned elements. In figure 7.12, I've added `position: relative` to the first two elements. This brings them to the front, covering the statically positioned third element, even though the order of the elements in the HTML is unchanged.

Notice that among the positioned elements, the second element still appears in front of the first. Positioned elements are brought to the front, but the source-dependent stacking between them remains unchanged.

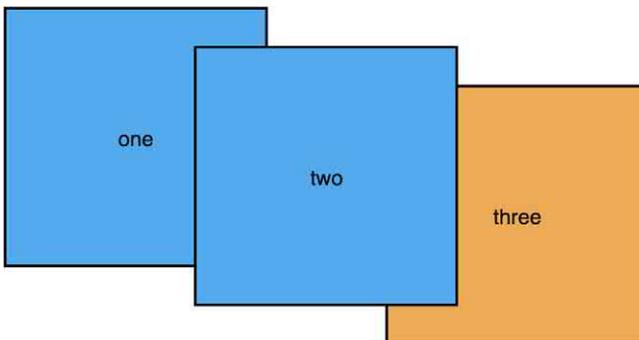


Figure 7.12 Positioned elements are painted in front of static elements.

On your page, this means that both the modal and the dropdown menu appear in front of the static content (which you want), but whichever appears later in your markup displays in front of the other. One way to fix your page would be to move the `<div class="modal">` and all of its contents to somewhere after the dropdown menu.

Typically, modals are added to the end of the page as the last bit of content before the closing `</body>` tag. Most JavaScript libraries for building modal dialogs will do this automatically. Because the modal uses fixed positioning, it doesn't matter where in the markup it appears; it'll always be positioned in the center of the screen.

Moving the element to somewhere else in the markup tends to be harmless for fixed positioning, but this isn't a solution you can normally use for relative or absolute positioning. Relative positioning depends on the document flow, and absolute positioning depends on its positioned ancestor element. We need a way to control their stacking behavior. This is done through a property called `z-index`.

7.4.2 Manipulating stacking order with z-index

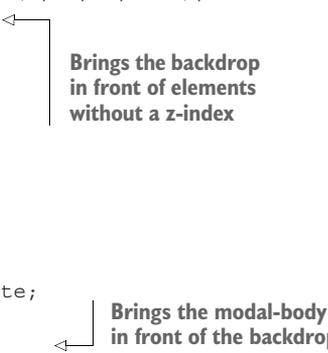
The `z-index` property can be set to any integer (positive or negative). *Z* refers to the depth dimension in a Cartesian X-Y-Z coordinate system. Elements with a higher `z-index` appear in front of elements with a lower `z-index`. Elements with a negative `z-index` appear behind static elements.

Using `z-index` is the second approach you can use to fix your page's stacking problem. This approach gives you more freedom in how you structure the HTML. Let's apply a `z-index` of 1 to the `modal-backdrop` and a `z-index` of 2 to the `modal-body` (this will ensure the body appears in front of the backdrop). Update this portion of your stylesheet to match the next listing.

Listing 7.8 Adding z-indexes to the modal dialog to bring it in front of the dropdown

```
.modal-backdrop {
  position: fixed;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  background-color: rgba(0, 0, 0, 0.5);
  z-index: 1;
}

.modal-body {
  position: fixed;
  top: 3em;
  bottom: 3em;
  right: 20%;
  left: 20%;
  padding: 2em 3em;
  background-color: white;
  overflow: auto;
  z-index: 2;
}
```



Brings the backdrop in front of elements without a z-index

Brings the modal-body in front of the backdrop

`Z-index` seems straightforward, but using it introduces two gotchas. First, `z-index` only works on positioned elements. You cannot manipulate the stacking order of static

elements. Second, applying a z-index to a positioned element establishes something called a stacking context.

7.4.3 Understanding stacking contexts

A *stacking context* consists of an element or a group of elements that are painted together by the browser. One element is the root of the stacking context, so when you add a z-index to a positioned element that element becomes the root of a new stacking context. All of its descendant elements are then part of that stacking context.

Stacking contexts are not to be confused with block formatting contexts (chapter 4). The two are separate concepts, though not necessarily mutually exclusive. Stacking contexts deal with which elements are in front of other elements; block formatting contexts deal with the document flow and whether or not elements will overlap.

The fact that all the elements of the stacking context are painted together has an important ramification: No element outside the stacking context can be stacked between any two elements that are inside it. Put another way, if an element is stacked in front of a stacking context, no element within that stacking context can be brought in front of it. Likewise, if an element on the page is stacked behind the stacking context, no element within that stacking context can be stacked behind that element.

This can be a little tricky to get your head around, so let's put together a scenario to illustrate this. In a fresh HTML page, add this markup.

Listing 7.9 Stacking contexts example

```
<div class="box one positioned">
  one
  <div class="absolute">nested</div>
</div>
<div class="box two positioned">two</div>
<div class="box three">three</div>
```

This markup consists of three boxes, two of which will be positioned and given a z-index of 1. The `absolute` element inside the first box will be positioned and given a z-index of 100. Despite its high z-index, it still appears behind the second box because its parent forms a stacking context behind the second box (figure 7.13).

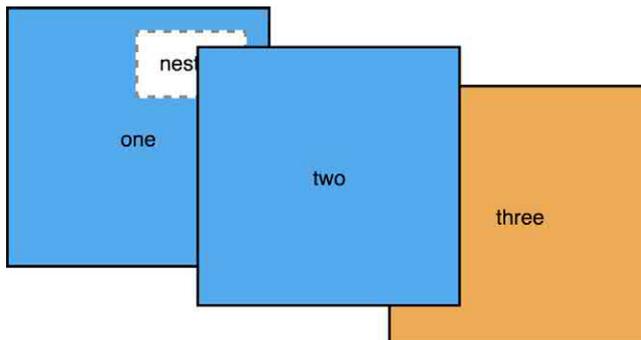


Figure 7.13 The entire stacking context is stacked together relative to other elements on the page.

The styles for this scenario are in listing 7.10. Apply this to your page. Most of these styles are for sizing and coloring to make the exact stacking order plainly visible. The negative margins force the elements to overlap. The only essential declarations are position and z-index for the various elements.

Listing 7.10 Creating the stacking contexts

```
body {
  margin: 40px;
}

.box {
  display: inline-block;
  width: 200px;
  line-height: 200px;
  text-align: center;
  border: 2px solid black;
  background-color: #ea5;
  margin-left: -60px;
  vertical-align: top;
}

.one { margin-left: 0; }
.two { margin-top: 30px; }
.three { margin-top: 60px; }

.positioned {
  position: relative;
  background-color: #5ae;
  z-index: 1;
}

.absolute {
  position: absolute;
  top: 1em;
  right: 1em;
  height: 2em;
  background-color: #fff;
  border: 2px dashed #888;
  z-index: 100;
  line-height: initial;
  padding: 1em;
}
```

Each positioned box establishes a stacking context with z-index of 1.

A z-index only controls the element's stacking position within its stacking context.

Box one, stacked behind box two, is the root of a stacking context. Because of this, the absolutely positioned element within it cannot be made to appear in front of box two, even with a high z-index value. Play around with this example in your browser's developer tools to get a feel for things. Manipulate the z-index of each element to see what happens.

NOTE Adding a z-index to a positioned element is the most notable way a stacking context is created, but a few other properties can create one as well.

An opacity below 1 creates one, as do the transform or filter properties. These fundamentally affect how the element and its children are rendered so they are all painted together. The document root (<html>) also creates a top-level stacking context for the whole page.

All the elements within a stacking context are stacked in this order, from back to front:

- The root element of the stacking context
- Positioned elements with a negative z-index (along with their children)
- Non-positioned elements
- Positioned elements with a z-index of auto (and their children)
- Positioned elements with a positive z-index (and their children)

Use variables to keep track of z-indexes

It's easy for a stylesheet to devolve into a z-index war, with no clear order as to the priority of various components. Without clear guidance, developers adding to the stylesheet might add something like a modal and, for fear of it appearing behind something else on the page, they'll give it a ridiculously high z-index, like 999999. After this happens a few times, it's anybody's guess what the z-index for another new component should be.

If you use a preprocessor such as LESS or SASS (see appendix B), or if you're supporting browsers that all support custom properties (see chapter 2), then use this to your advantage. Put all your z-index values into variables, all in one place. That way you can see at a glance what is supposed to appear in front of what:

```
--z-loading-indicator: 100;  
--z-nav-menu:         200;  
--z-dropdown-menu:   300;  
--z-modal-backdrop:  400;  
--z-modal-body:      410;
```

Use increments of 10 or 100, so you can insert new values in between should the need arise.

If you ever find that z-index isn't behaving how you'd expect, look up the DOM tree at the element's ancestors until you find one that is the root of a stacking context. Then use z-index to bring the entire stacking context forward or backward. Be careful, as you may find multiple stacking contexts nested within one another.

When the page is complicated, it can be difficult to determine exactly which stacking context is responsible for the behavior you see. For this reason, always be cautious when creating a stacking context. Don't create one unless you have a specific reason for it. This is doubly true for elements that encompass large portions of the page. When possible, bring standalone positioned elements like modals to the top level of the DOM, right before the closing </body> tag, so you don't have to break them free from any outer stacking contexts.

Some developers fall into the trap of positioning a large number of elements all over the page. You should fight this urge. The more you use positioning, the more complicated the page becomes, and the harder it is to debug. If you find you're positioning dozens of elements, step back and re-assess the situation. This is particularly important if you find yourself fighting to get the layout to behave the way you want. When your layout can be accomplished using other methods, you should use those methods instead.

If you can get the document flow to do the work for you, rather than explicitly positioning things, the browser will take care of a lot of edge-cases for you. Remember, positioning takes elements out of the document flow. Generally speaking, you should only do this when you need to stack elements in front of one another.

7.5 Sticky positioning

The four main positioning types (static, fixed, absolute, and relative) have been around for a long time, but there's a new type of positioning making its way into browsers: *sticky positioning*. It's sort of a hybrid between relative and fixed positioning: The element scrolls normally with the page until it reaches a specified point on the screen, at which point it will "lock" in place as the user continues to scroll. The common use-case for this is sidebar navigation.

Sticky positioning has been supported in only Firefox for some time. But at the time of writing, this feature is available in Chrome and Edge browsers. Safari supports it with the vendor prefix (`position: -webkit-sticky`). Be sure to check <http://caniuse.com/#feat=css-sticky> for the latest support information. Typically, you'll use fixed or absolute positioning as a fallback behavior for other browsers.

Let's update your page that has the modal dialog and dropdown menu. You'll change it to a two-column layout, adding a sticky-positioned sidebar as the right-hand column. This will look like the screenshot in figure 7.14.

When the page initially loads, the position of the sidebar appears routine. It'll scroll normally with the page—up to a point. As soon as it's about to leave the viewport,



Figure 7.14 The sticky-positioned sidebar is initially positioned normally.

it'll lock into place. It'll then stay there like a fixed-positioned element, remaining on the screen as the rest of the page continues to scroll. After scrolling down the page a little, it'll look like figure 7.15.

Wombat Coffee Roasters

Home
Coffees
Brewers
Specials
About us

Figure 7.15 The sidebar remains “stuck” in place.

Let's restructure your page a bit to define two columns. Edit the container in your HTML to match the next listing. This places your existing content (the dropdown menu and page heading) into a left column and adds a right column with an “affix” menu.

Listing 7.11 Changing to a two-column layout with a sidebar

```

<div class="container">
  <main class="col-main">
    <nav>
      <div class="dropdown">
        <div class="dropdown-label">Main Menu</div>
        <div class="dropdown-menu">
          <ul class="submenu">
            <li><a href="/">Home</a></li>
            <li><a href="/coffees">Coffees</a></li>
            <li><a href="/brewers">Brewers</a></li>
            <li><a href="/specials">Specials</a></li>
            <li><a href="/about">About us</a></li>
          </ul>
        </div>
      </div>
    </nav>
    <h1>Wombat Coffee Roasters</h1>
  </main>

  <aside class="col-sidebar">
    <div class="affix">
      <ul class="submenu">
        <li><a href="/">Home</a></li>
        <li><a href="/coffees">Coffees</a></li>
        <li><a href="/brewers">Brewers</a></li>
        <li><a href="/specials">Specials</a></li>
        <li><a href="/about">About us</a></li>
      </ul>
    </div>
  </aside>

```

Wraps existing content in a col-main for the main column

Adds a second column with an affix element inside it

```

    </div>
  </aside>
</div>

```

Next, you'll update the CSS so the container becomes a flex container to size the two columns. For this demo, you'll repurpose the submenu styles from the dropdown menu, though you could alternately add whatever elements and styles you want into the sidebar. Add these styles to your stylesheet.

Listing 7.12 Creating a two-column layout and sticky-position side menu

```

.container {
  display: flex;
  width: 80%;
  max-width: 1000px;
  margin: 1em auto;
  min-height: 100vh;
}

.col-main {
  flex: 1 80%;
}

.col-sidebar {
  flex: 20%;
}

.affix {
  position: sticky;
  top: 1em;
}

```

← Makes the container a flex container for the two-column layout

← Artificially adds height to the container

← Lays out the two columns

← Applies sticky positioning to the side menu. It will dock 1 em from the top of the viewport.

Most of these changes are setting up the two-column layout. With that established, it only takes two declarations to position the `affix` element. The `top` value sets the location where the element will lock into place: 1 em from the top of the viewport.

A sticky element will always remain within the bounds of its parent element—the `col-sidebar` in this case. As you scroll down the page, the `col-sidebar` will always scroll normally, but the `affix` element will scroll until it locks into place. Continue scrolling far enough and it'll unlock and resume scrolling. This occurs when the bottom edge of the parent reaches the bottom edge of the sticky menu. Note that the parent must be taller than the sticky element in order for it to stick into place, which is why I artificially increased its height by adding a `min-height` to its flex container.

Summary

- Use fixed positioning for modal dialogs.
- Use absolute positioning for dropdown menus, tooltips, and other dynamic interactions.

- Be aware of accessibility concerns when building these features.
- There are two gotchas of z-index: it only works on positioned elements and using it creates a new stacking context.
- Be aware of the potential pitfalls when creating multiple stacking contexts on a page.
- Keep an eye out for better browser support of sticky positioning.

Responsive design

This chapter covers

- Building web pages for multiple devices and screen sizes
- Using media queries to alter your design based on viewport size
- Taking the “mobile first” approach
- Responsive images

In our modern world, the web is everywhere. We use it on our desktop in our office. We lie in bed surfing on our tablet. It’s even on some of our television screens in the living room. And we carry it everywhere with us on our smartphones. The web platform of HTML, CSS, and JavaScript is a universal ecosystem unlike anything that has come before.

This poses an interesting problem for us as web developers: How do we design our site so it’s usable and appealing on any device our users might use to access it? Initially, many developers approached this problem by creating two websites: desktop and mobile. The server then redirected mobile devices from <http://www.wombatcoffee.com> to <http://m.wombatcoffee.com>. This mobile website would usually offer a more minimal experience and a streamlined design for smaller screens.

This approach began to break down as more and more devices emerged on the market. Do you direct a tablet to the mobile website or to the desktop? What about a large “phablet” phone? An iPad Mini? What if a mobile user wants to perform an action you only have available on the desktop version of your site? In the end, a forced dichotomy between desktop and mobile causes more problems than it solves. Plus, you have to maintain an extra website to make it work.

A far better approach is to serve the same HTML and CSS to all your users. By using a few key techniques, you can make your content render differently, based on your user’s browser viewport size (or, occasionally, based on screen resolution). This way, you don’t need two distinct websites. You create one website that works on a smartphone, a tablet, or anything else you throw at it. This approach, popularized by web designer Ethan Marcotte, is called *responsive design*.

As you browse the web, make note of the responsive designs you come across. See how websites respond to different browser widths. Newspaper sites are particularly interesting because they have so much content crammed onto the page. At the time of writing, www.bostonglobe.com/ is a great example, offering a one-, two-, or three-column layout depending on the width of your browser window. Typically, you can resize the width of the browser window and see the page layout respond immediately. This is responsive design at work.

The three key principles to responsive design:

- 1 *A mobile first approach to design.* This means you build the mobile version before you construct the desktop layout.
- 2 *The @media at-rule.* With this rule, you can tailor your styles for viewports of different sizes. This syntax (often called *media queries*) lets you write styles that only apply under certain conditions.
- 3 *The use of fluid layouts.* This approach allows containers to scale to different sizes based on the width of the viewport.

This chapter takes a look at these three principles. Let’s start by building a responsive page, and I’ll unpack each of those as we go. After that, we’ll also take a look images, which require special considerations on responsive sites.

8.1 **Mobile first**

The first principle of responsive design is *mobile first*. As mentioned, this means exactly what it sounds like: You build your mobile layout before you build the desktop. This is the best way to ensure both versions work.

Developing for mobile is an exercise in constraints. Screen space is limited. The network is often slower. The user on a mobile device uses a different set of interactive controls. Typing is possible, but cumbersome. The user can’t hover over elements to trigger effects. If you begin by designing a full interactive website, then try to scale it down to meet these constraints, you’ll often fail.

Instead, the mobile first approach dictates you design your site with these constraints in mind from the beginning. Once your mobile experience works (or is at least planned), you can use “progressive enhancement” to augment the experience for large screen users.

Figure 8.1 shows the page you’re going to build. You guessed it—that’s the mobile design.

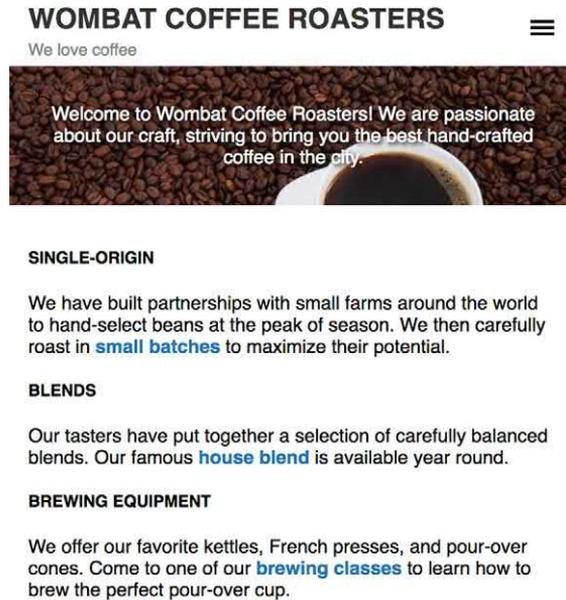


Figure 8.1 Mobile page design

This page has three main components: a header, the hero image with a bit of text superimposed over it, and the main content. You can also show a hidden menu by tapping or clicking the icon on the top right (figure 8.2). This icon, with three horizontal lines, is often called a *hamburger* icon because it resembles the buns and patty of a burger.

A mobile layout is mostly a no-frills design. Apart from the interactive menu, this design is highly focused on the content. On larger screens, you can afford to dedicate a lot of space to things like the header, the hero image, and the menu. But on mobile devices, users are often more task-oriented. They might be out with their friends and want to quickly find store hours or another specific piece of information, like a price or address.

A mobile design is about the content. Consider a desktop design that has an article on one side and sidebar on the other, where the sidebar has links and less important items. You’ll want to ensure the article appears first on the mobile design. This means you’ll want the most important content to appear first in the HTML. Conveniently,

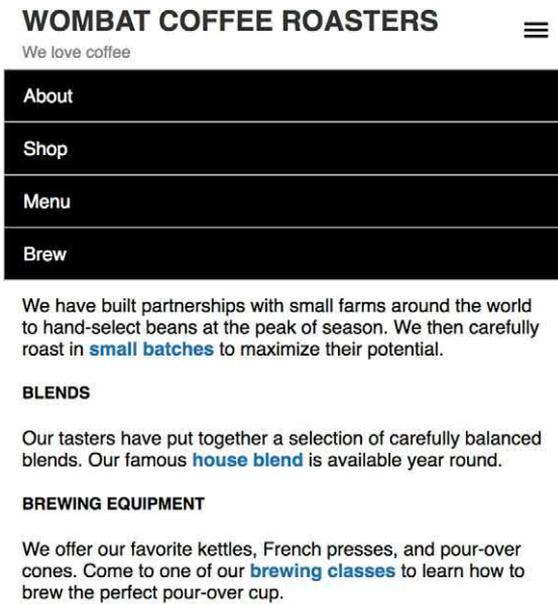


Figure 8.2 Mobile page with the menu opened by clicking or tapping the hamburger icon

this coincides with accessibility concerns: A screen reader gets right to the “good stuff,” or a user navigating via the keyboard gets to the links in the article before those in the sidebar.

That said, this isn’t always a hard and fast rule. You could probably make the argument that your hero image isn’t as important as the content below it. But it’s a striking part of the design; so, in this case, I think it’s worth having that image close to the top of the page. It also contains little content, so it takes minimal effort to navigate past it.

IMPORTANT When writing the HTML for a responsive design, it’s important to ensure it has everything you need for each screen size. You can apply different CSS for each instance, but they must all share the same HTML.

Now, let’s consider our design for larger viewports. You’ll code the mobile layout first, but keeping your overall design in mind will help guide your decisions as you do that. For this exercise, you’ll add a medium and large *breakpoint*. Figure 8.3 shows the medium layout.

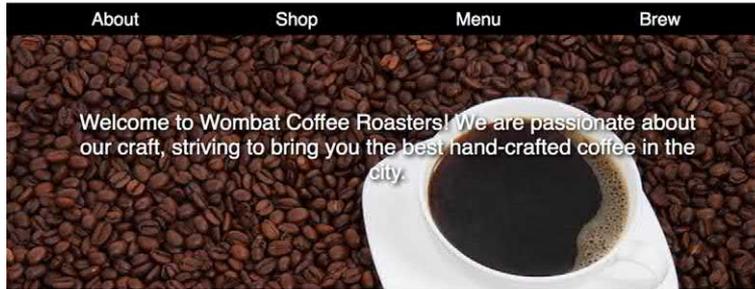


breakpoint—A particular point at which the page styles change to provide the best possible layout for the screen size.

At this viewport size, you’ve a little more space to work with. The header and hero can afford more padding. The menu items can fit beside each other on one line so they

WOMBAT COFFEE ROASTERS

We love coffee



SINGLE-ORIGIN

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in **small batches** to maximize their potential.

BLENDS

Our tasters have put together a selection of carefully balanced blends. Our famous **house blend** is available year round.

BREWING EQUIPMENT

We offer our favorite kettles, French presses, and pour-over cones. Come to one of our **brewing classes** to learn how to brew the perfect pour-over cup.

Figure 8.3 Page on medium-sized viewport

no longer need to be hidden, and the hamburger icon is gone because you don't need it to open the menu. Now the main content can be arranged into three equal-width columns. Most of the elements fill to within 1 em of the sides of the viewport.

The larger viewport will be the same, but you'll increase the margins on the sides of the page and make the hero image even larger. This design is shown in figure 8.4.

WOMBAT COFFEE ROASTERS

We love coffee



SINGLE-ORIGIN

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in **small batches** to maximize their potential.

BLENDS

Our tasters have put together a selection of carefully balanced blends. Our famous **house blend** is available year round.

BREWING EQUIPMENT

We offer our favorite kettles, French presses, and pour-over cones. Come to one of our **brewing classes** to learn how to brew the perfect pour-over cup.

Figure 8.4 Page on large viewport

Because you'll build the mobile design first, it's important to have an idea of how the larger viewports will look as that can influence how you structure the HTML. Create a new web page and stylesheet. Link the stylesheet and add listing 8.1 to the HTML's `<body>`.

The markup looks much the same as that for a non-responsive design. I've made a couple of considerations for the mobile design, which I'll highlight momentarily.

Listing 8.1 Page markup for a responsive design

```

<header id="header" class="page-header">
  <div class="title">
    <h1>Wombat Coffee Roasters</h1>
    <div class="slogan">We love coffee</div>
  </div>
</header>

<nav class="menu" id="main-menu">
  <button class="menu-toggle" id="toggle-menu">
    toggle menu
  </button>
  <div class="menu-dropdown">
    <ul class="nav-menu">
      <li><a href="/about.html">About</a></li>
      <li><a href="/shop.html">Shop</a></li>
      <li><a href="/menu.html">Menu</a></li>
      <li><a href="/brew.html">Brew</a></li>
    </ul>
  </div>
</nav>

<aside id="hero" class="hero">
  Welcome to Wombat Coffee Roasters! We are
  passionate about our craft, striving to bring you
  the best hand-crafted coffee in the city.
</aside>

<main id="main">
  <div class="row">
    <section class="column">
      <h2 class="subtitle">Single-origin</h2>
      <p>We have built partnerships with small farms
      around the world to hand-select beans at the
      peak of season. We then carefully roast in
      <a href="/batch-size.html">small batches</a>
      to maximize their potential.</p>
    </section>
    <section class="column">
      <h2 class="subtitle">Blends</h2>
      <p>Our tasters have put together a selection of
      carefully balanced blends. Our famous
      <a href="/house-blend.html">house blend</a>
      is available year round.</p>
    </section>
    <section class="column">

```

← Adds the hamburger button for a mobile menu

← Main menu that will be hidden by default on mobile devices

← Adds row and columns for medium and large viewports

```

<h2 class="subtitle">Brewing Equipment</h2>
<p>We offer our favorite kettles, French
  presses, and pour-over cones. Come to one of
  our <a href="/classes.html">brewing
  classes</a> to learn how to brew the perfect
  pour-over cup.</p>
</section>
</div>
</main>

```

In this markup, the button to toggle the menu for mobile screens is inside the nav element. The nav-menu is placed where it can meet your needs for both mobile and desktop designs. And the row and column classes are in place to allow for the desktop design. (You may not know this all up front, which is okay.)

Let's begin styling the page. First, you'll add some of the simpler styles like the font, headings, and colors, as shown in figure 8.5. Because we're concerned right now with mobile styles, resize your browser to a narrow size to mimic a mobile device. This helps to show you what the page will look like on a small screen.



Figure 8.5 First set of styles applied

These styles are given in listing 8.2. Add them to your stylesheet to establish border-box sizing, font, and link colors. The listing adds the responsive viewport-based font size you learned about in chapter 2 (section 2.4.1). Then, it defines styles for the header and the main body of the page.

Listing 8.2 Adding initial styles for the page

```

:root {
  box-sizing: border-box;
  font-size: calc(1vw + 0.6em);
}

```

Base font size scales slightly with the viewport.

```

*,
*::before,
*::after {
  box-sizing: inherit;
}

body {
  margin: 0;
  font-family: Helvetica, Arial, sans-serif;
}

a:link {
  color: #1476b8;
  font-weight: bold;
  text-decoration: none;
}
a:visited {
  color: #1430b8;
}
a:hover {
  text-decoration: underline;
}
a:active {
  color: #b81414;
}

```

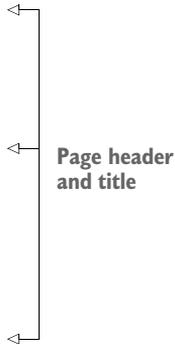
```

.page-header {
  padding: 0.4em 1em;
  background-color: #fff;
}

.title > h1 {
  color: #333;
  text-transform: uppercase;
  font-size: 1.5rem;
  margin: 0.2em 0;
}

.slogan {
  color: #888;
  font-size: 0.875em;
  margin: 0;
}

```



```

.hero {
  padding: 2em 1em;
  text-align: center;
  background-image: url(coffee-beans.jpg);
  background-size: 100%;
  color: #fff;
  text-shadow: 0.1em 0.1em 0.3em #000;
}

```

← Adds the hero image to the page

← A dark text shadow helps light text remain readable in front of complex background.

```

main {
  padding: 1em;
}

.subtitle {
  margin-top: 1.5em;
  margin-bottom: 1.5em;
  font-size: 0.875rem;
  text-transform: uppercase;
}

```

← Main content

These styles are mostly straightforward. They transform the page title and the subtitles in the body to all caps. They add some margins and padding, and adjust font sizes for the various components on the page.

The `text-shadow` property in the hero image might be new to you. It consists of several values that together define a shadow to add behind the text. The first two values are Cartesian coordinates, indicating how far the shadow should shift from the text's position. The values `0.1em 0.1em` shift the shadow slightly right and down. The third value (`0.3em`) indicates how much to blur the shadow. Finally, `#000` defines the color of the shadow.

8.1.1 Creating a mobile menu

At this point, you're left with the most complicated part of the page: the menu. Let's build that now. When you're done, it'll look like figure 8.6.



Figure 8.6 Opened navigational menu on a mobile device

Writing code in any language is often an iterative process, and CSS is no different. For this page, the menu took some careful consideration. I originally tried to place the `<nav>` inside the `<header>` because that's where I wanted the hamburger button to appear. But after I started on the CSS, I realized I should keep the two elements as siblings because this allows them to stack naturally in the desktop layout. Sometimes it takes a few passes over certain parts of the HTML to get it right.

Functionally, this menu is much like the dropdown menu you built in the last chapter (listing 7.6). Initially, you'll hide the menu-dropdown. Then, instead of using a hover effect, you'll add some proper JavaScript functionality. When the user clicks

(or taps) the `menu-toggle`, the dropdown will appear. Clicking a second time hides the menu.

TIP Screen readers use certain HTML5 elements such as `<form>`, `<main>`, `<nav>`, and `<aside>` as *landmarks*. This helps users with poor vision to quickly navigate the page. It's important that you place the button to reveal your menu within the `<nav>` so it's quickly discoverable when the user navigates there. Otherwise, the user would jump to the `<nav>` only to find it empty (the screen reader ignores the dropdown menu when `display: none` is applied).

The controversial hamburger menu

Hamburger menus have become popular in recent years. They can solve the problem of fitting more on a small screen, but they come with a cost. Hiding important elements (such as your main navigational menu) has been shown to reduce user interaction with those elements.

These are considerations you'll need to evaluate with your team or designer. Sometimes they're the right choice; other times, they may not be. Regardless, it's important to know the techniques involved in building a hamburger menu.

In listing 8.1, notice that the `<nav>` appears after the `<header>` as a sibling element. This means it'll flow to the space beneath the header. You'll have to do one unusual thing here to match the design: you'll use absolute positioning to pull the menu-toggle button up so it appears inside the header element. Add this listing to your stylesheet to style the menu.

Listing 8.3 Mobile menu styles

```
.menu {
  position: relative;
}

.menu-toggle {
  position: absolute;
  top: -1.2em;
  right: 0.1em;

  border: 0;
  background-color: transparent;

  font-size: 3em;
  width: 1em;
  height: 1em;
  line-height: 0.4;
  text-indent: 5em;
  white-space: nowrap;
  overflow: hidden;
}
```

Establishes containing block for both absolutely positioned children

A negative top pulls the button up outside its containing block.

Overrides user agent button styles

Hides the text content of the button and fixes its size at 1 em

```

.menu-toggle::after {
  position: absolute;
  top: 0.2em;
  left: 0.2em;
  display: block;
  content: "\2261";
  text-indent: 0;
}

.menu-dropdown {
  display: none;
  position: absolute;
  right: 0;
  left: 0;
  margin: 0;
}

.menu.is-open .menu-dropdown {
  display: block;
}

```

Overlays the button with a unicode symbol, the hamburger icon

Displays the dropdown menu when the class is-open is added to the menu

A lot is going on here, but it's mostly a series of techniques you've already seen. The menu is relatively positioned to establish a containing block for both its child elements: the toggle button and the dropdown. The toggle button is pulled upward with a negative top, and the right property positions it on the right side of the screen. This makes it appear in the header to the right of the page title.

You then use a replacement trick on the button: A constrained width, a large text-indent, and hidden overflow all work together to hide the text of the button (toggle menu). Then you give the button's `::after` pseudo-element a unicode character (`\2261`) for its content. This character is a mathematical symbol with three horizontal lines: a hamburger menu. If you want to tailor the icon further, you could instead use a background image on the pseudo-element.

If you're unsure why any of these particular styles are used, comment them out and look at the effect they have on the page. The page will look a little funny on a large viewport; resize your browser window to a narrower size for a better approximation of the mobile look.

The `is-open` class is a new "trick." When this class is present, the final selector (`.menu.is-open .menu-dropdown`) targets the dropdown. When this class is absent, the selector will not. This enables the dropdown's functionality. Figure 8.7 shows the dropdown menu before the rest of the styling is applied (note the four links in front of the hero image on the left).

The JavaScript in listing 8.4 adds and removes the `is-open` class when the toggle button is pressed. Add it to your page before the closing `</body>` tag.



SINGLE-ORIGIN

We have built partnerships with small farms around the world to hand-select beans at the peak of season. We then carefully roast in **small batches** to

Figure 8.7 Hamburger button working

Listing 8.4 JavaScript for dropdown functionality

```
<script type="text/javascript">
(function () {
  var button = document.getElementById('toggle-menu');
  button.addEventListener('click', function(event) {
    event.preventDefault();
    var menu = document.getElementById('main-menu');
    menu.classList.toggle('is-open');
  });
})();
</script>
```

Click-event listener
(also fires on a
touchscreen tap
event)

Toggles is-open
class on the menu

Now when you click the hamburger icon, it should open the dropdown. You can see the text of the menu in front of the content behind it. Click the hamburger again to close it. This way, the CSS will do the work of showing and hiding the correct elements; the JavaScript only needs to change one class name.

Now that the dropdown works, the nav-menu needs some styling. Add this listing to your stylesheet.

Listing 8.5 Styling the navigational menu

```
.nav-menu {
  margin: 0;
  padding-left: 0;
  border: 1px solid #ccc;
  list-style: none;
  background-color: #000;
  color: #fff;
}

.nav-menu > li + li {
  border-top: 1px solid #ccc;
}
```

Applies a border
between each
menu item

```
.nav-menu > li > a {
  display: block;
  padding: 0.8em 1em;
  color: #fff;
  font-weight: normal;
}
```

← Uses a healthy amount of padding to ensure a large clickable area

Again, this is nothing new. Because the menu is a list (), you override the user agent left padding and remove the list bullets. The adjacent sibling combinator targets every menu item but the first, adding a border between each item.

An important thing to note here is the padding on the menu item links. You're designing for mobile devices, which are typically touchscreen. Key clickable areas should be large and easy to tap with a finger.

TIP When designing for mobile touchscreen devices, be sure to make all the key action items large enough to easily tap with a finger. Don't make your users zoom in in order to tap precisely on a tiny button or link.

8.1.2 Adding the viewport meta tag

At this stage, your mobile design is complete, but there's one important detail missing: the viewport *meta tag*. This is an HTML tag that tells mobile devices you've intentionally designed for small screens. Without it, a mobile browser assumes your page is not responsive, and it will attempt to emulate a desktop browser. All your hard work on a mobile design will be for naught. We don't want that. Update the <head> of your HTML to include the meta tag as shown in the next listing.

Listing 8.6 Adding the viewport meta tag for mobile responsiveness

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
    content="width=device-width, initial-scale=1">
  <title>Wombat Coffee Roasters</title>
  <link href="styles.css" />
</head>
```

Viewpoint meta tag

The meta tag's content attribute indicates two things. First, it tells the browser to use the device width as the assumed width when interpreting the CSS, instead of pretending to be a full size desktop browser. Second, it uses *initial-scale* to set the zoom level at 100% when the page loads.

TIP The DevTools in modern browsers provide the ability to emulate a mobile browser, including a smaller viewport size and the behavior of the viewport meta tag. These are helpful tools for testing out your responsive design. For more information on these modes, see <https://developers.google.com/web/tools/chrome-devtools/device-mode/> (Chrome) or https://developer.mozilla.org/en-US/docs/Tools/Responsive_Design_Mode (Firefox).

Other options are available, but the settings here are most likely the ones you'll want. For instance, you could explicitly set `width=320` to make the browser assume a viewport width of 320 pixels. This is generally not preferable, though, as mobile devices come in a wide array of sizes. Using `device-width` allows content to render at the most appropriate size.

A third common option that many developers add to the `content` attribute is `user-scalable=no`, which prohibits the user from using two fingers to zoom in and out on their mobile device. Including this is generally a bad practice, and I discourage its use. If a link is too small to tap, or the user wants to take a closer look at an image, this setting would prevent them from using zoom to assist.

For more information on the meta viewport tag, see the MDN documentation at https://developer.mozilla.org/en-US/docs/Mozilla/Mobile/Viewport_meta_tag.

8.2 *Media queries*

The second component of responsive design is the use of media queries. *Media queries* allow you to write a set of styles that only apply to the page under certain conditions. This lets you tailor your styles differently, based on the screen size. You can define a set of styles that apply to small devices, another set for medium-sized devices, and yet a third set for large screens to allow for laying out parts of the page differently.

Media queries use the `@media` at-rule to target devices that match a specified feature. A basic media query looks like this:

```
@media (min-width: 560px) {  
  .title > h1 {  
    font-size: 2.25rem;  
  }  
}
```

Any rulesets can be defined within the outer set of braces. The `@media` rule is a conditional check that must be true for any of these styles to be applied to the page. In this case, the browser checks for a `min-width: 560px`. The padding will only be applied to a page-header element if the user's device has a viewport width of 560 px or greater. If the viewport is less than this, the rules inside are ignored.

The rules inside a media query still follow the normal rules of the cascade. They can override rules outside of the media query (based on selector specificity or source order), or they can be overridden by those rules. The media query itself does not affect the specificity of the selectors within.

WARNING You should use ems for media query breakpoints. It's the only unit that performs consistently in all major browsers should the user zoom the page or change the default font size. Pixel- and rem-based breakpoints are less reliable in Safari. Em's also have the benefit of scaling up or down with the user's default font size, which is generally preferable.

I used px in the example, but it's a better idea to use ems in your media queries, based on the browser's default font size (usually 16 px). Instead of 560 px, you should use 35 em (560 / 16).

Find the `.title` styles in your stylesheet and insert the media query in this listing to add some responsive behavior to the page header.

Listing 8.7 Adding a breakpoint to the page title styles

```
.title > h1 {
  color: #333;
  text-transform: uppercase;
  font-size: 1.5rem;
  margin: .2em 0;
}

@media (min-width: 35em) {
  .title > h1 {
    font-size: 2.25rem;
  }
}
```

Now the title has two different font sizes, depending on the viewport size. It'll be 1.5 rem for viewports below 35 em and 2.25 rem for those above.

You can test these styles by resizing the width of your browser window. Make it narrow and you'll see the smaller mobile title. Then, slowly expand your browser window. You'll see the font size change fluidly because of the responsive (`calc()`) font size applied to the page (listing 8.2). Once it reaches a width of 35 em (or 560 px), the title's font size will snap to a larger 2.25 rem.

This point, where the window is 560 px wide, is known as a *breakpoint*. Most often, you'll re-use the same few breakpoints in multiple media queries throughout your stylesheet. We'll discuss how to choose these breakpoints later in the chapter.

8.2.1 Understanding types of media query

You can further refine a media query by joining the two clauses with the keyword `and`:

```
@media (min-width: 20em) and (max-width: 35em) { ... }
```

This combined media query only targets devices that meet both criteria. If you want a media query that targets one of multiple criteria, use a comma:

```
@media (max-width: 20em), (min-width: 35em) { ... }
```

This example targets both viewports 20 em and narrower and those 35 em and wider.

MIN-WIDTH, MAX-WIDTH, AND BEYOND

In the listings, you've used `min-width`, which targets devices with a viewport above a certain width, and `max-width`, which targets devices below a certain width. These are each called a *media feature*.

`min-width` and `max-width` are the most common ones you'll use by far. But you can also use a number of other types of media features. Here's some examples:

- `(min-height: 20em)`—Targets viewports 20 em and taller
- `(max-height: 20em)`—Targets viewports 20 em and shorter
- `(orientation: landscape)`—Targets viewports that are wider than they are tall
- `(orientation: portrait)`—Targets viewports that are taller than they are wide
- `(min-resolution: 2dppx)`—Targets devices with a screen resolution of 2 dots per pixel or higher; targets retina displays
- `(max-resolution: 2dppx)`—Targets devices with a screen resolution of up to 2 dots per pixel

See the MDN documentation at <https://developer.mozilla.org/en-US/docs/Web/CSS/@media> for a complete list of media features.

Resolution-based media queries can be a little tricky, as browser support for this is newer. Some browsers have limited support and/or require a proprietary syntax. IE9–11 and Opera Mini, for example, don't support the `dppx` unit, so you'll need to use the `dpi` (dots per inch) unit instead (for example, 192 dpi instead of 2 `dppx`). Safari and iOS Safari support the proprietary `-webkit-min-device-pixel-ratio` media feature. In short, the best way to target a high resolution (retina) display is to combine the two:

```
@media (-webkit-min-device-pixel-ratio: 2),  
       (min-resolution: 192dpi) { ... }
```

This approach will work on all modern browsers. Use it when you want to serve higher-resolution imagery or icons to screens that can benefit from them. This way, users with lower-resolution screens won't waste bandwidth loading larger images when they won't be able to see the difference. We'll look more at responsive imagery later in the chapter.

TIP You can also place a media query in the `<link>` tag. Adding `<link rel="stylesheet" media="(min-width: 45em)" href="large-screen.css" />` to your page will apply the contents of the `large-screen.css` file to the page only if the `min-width` media query is true. Note that the stylesheet will always download, regardless of the width of the viewport, so this is merely a tactic for code organization, not network traffic reduction.

MEDIA TYPES

One last option available in your media queries is the ability to target the *media type*. The two media types you'll generally need to think about are screen and print. Using a print media query lets you control how your page will lay out if the user prints the page, so you can do things like removing background images (to save on ink) and hiding unneeded navigation. When a user prints the page, they typically only want the main page text.

To write print styles that apply only when printing, use the query `@media print`. No parentheses are necessary as with `min-width` and other media features. To target screen only, use `@media screen`.

Considerations for print styles

When it comes to CSS development, print styles are often an afterthought, if they're considered at all. But, it's good to consider whether your users might want to print any of your pages. To help with printing, there are some common steps you should take. In most cases, it'll be helpful to apply basic print styles inside of a `@media print {...}` media query.

Use `display: none` to hide non-essential parts of the page, such as navigational menus and footers. If a user is printing a page, they'll almost certainly care only about the main content on the page.

You can also globally change font colors to black and remove all background images and colors behind blocks of text. In many cases, a universal selector does the job for this. I use `!important` here so I don't need to worry about the cascade overriding it:

```
@media print {
  * {
    color: black !important;
    background: none !important;
  }
}
```

Spending even a brief amount of time on print styles can be a great service to your users. If you're working on a site where you expect a lot of printing (for example, a recipe site), you'll want to spend more time making sure everything prints correctly.

8.2.2 Adding breakpoints to the page

Practically speaking, a mobile-first approach means the type of media query you'll use the most should be `min-width`. You'll write your mobile styles first, outside of any media queries. Then you'll work your way up to larger breakpoints. This follows the general structure shown in listing 8.8. (You don't need to add this to your page yet.)

Listing 8.8 General structure of responsive CSS

```

.title {
  ...
}

@media (min-width: 35em) {
  .title {
    ...
  }
}

@media (min-width: 50em) {
  .title {
    ...
  }
}

```

← **Mobile styles; applied to all breakpoints**

← **Medium breakpoint; overrides select mobile styles**

← **Large breakpoint; overrides select small and medium breakpoint styles**

The mobile styles are first. Because they're outside of any media queries, these rules apply to all breakpoints. This is followed by a media query to target medium screens with rules that override and build upon the mobile styles. Last is a media query to target large screens, where you add the final layer.

Depending on your design, you may have only one breakpoint, or you could have several. For many elements on your page, you may not need to add styles for every breakpoint—the rules at small or medium breakpoints may be complete to also account for larger breakpoints.

On occasion, your mobile-only styles might be complex. It can be tedious overriding these rules at a larger breakpoint. In this case, it might make sense to contain these styles in a `max-width` media query so they only apply at the smaller breakpoint. Too many `max-width` media queries, however, could be a sign you haven't followed the mobile first approach. They should be an exception and not the rule.

Let's add the rest of the styles for the medium breakpoint. On a larger screen, there's more space to work with, so you can loosen up the spacing. In listing 8.9, you'll add a little more padding to the header and main elements. Then you'll add a lot more padding to the hero image so it "pops" and adds more visual interest to the page. You no longer need to hide the navigational menu, so you'll also hide the hamburger button and reveal the menu at all times (listing 8.10). Finally, you can shift the main content to a three-column layout (listing 8.11). Afterward, the page will look like figure 8.8.

Several of these changes are straightforward, such as slight increases to padding or font size. It's generally best to add each change immediately following the rules for the associated selectors. For simplicity, I've consolidated them into listing 8.9. Add these to your stylesheet.

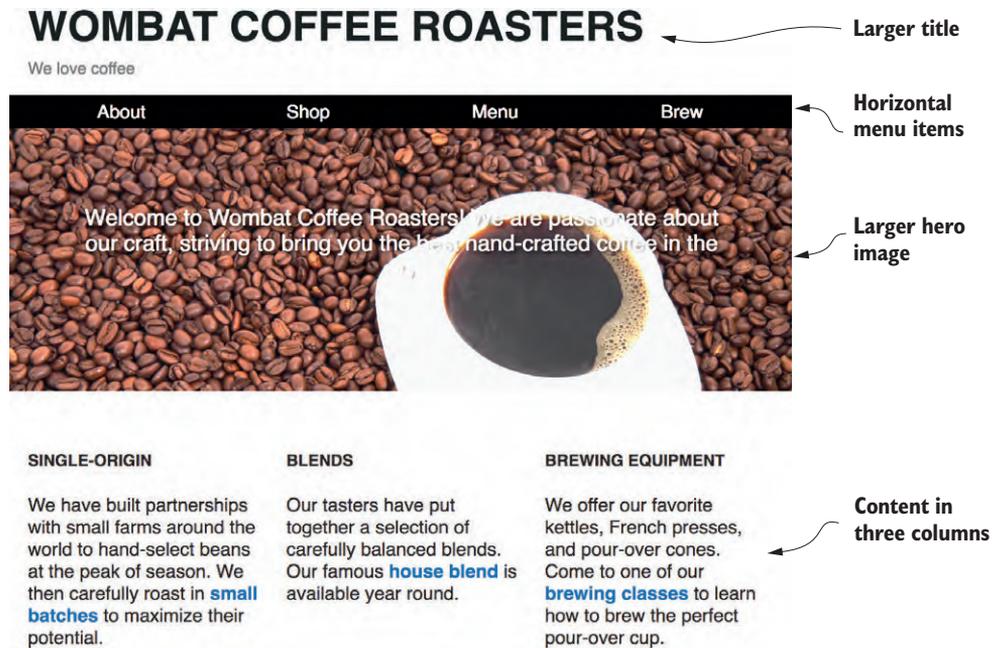


Figure 8.8 Page at medium breakpoint

Listing 8.9 Padding and font adjustments for the medium breakpoint

```
.page-header {
  padding: 0.4em 1em;
  background-color: #fff;
}

@media (min-width: 35em) {
  .page-header {
    padding: 1em;
  }
}

.hero {
  padding: 2em 1em;
  text-align: center;
  background-image: url(coffee-beans.jpg);
  background-size: 100%;
  color: #fff;
  text-shadow: 0.1em 0.1em 0.3em #000;
}

@media (min-width: 35em) {
  .hero {
    padding: 5em 3em;
    font-size: 1.2rem;
  }
}
```

← Increases header padding

← Increases hero image padding and font size

```

    }
  }

  main {
    padding: 1em;
  }

  @media (min-width: 35em) {
    main {
      padding: 2em 1em;
    }
  }

```

← Increases padding on main

Always be sure each media query comes after the styles it overrides, so the styles within the media query take precedence. Resize your browser from narrow to wide to see these changes snap into place above 35 em.

Next, let's work on the menu. This will involve two changes: First, you'll remove the open and close behavior of the dropdown so it's always visible. Second, you'll change the menu from vertically stacked links into horizontal ones. Both are done next. Add these media query blocks to your stylesheet following the styles you already have for `.menu` and `.nav-menu`.

Listing 8.10 Restructuring the nav menu for a medium breakpoint

```

@media (min-width: 35em) {
  .menu-toggle {
    display: none;
  }
  .menu-dropdown {
    display: block;
    position: static;
  }
}

@media (min-width: 35em) {
  .nav-menu {
    display: flex;
    border: 0;
    padding: 0 1em;
  }
  .nav-menu > li {
    flex: 1;
  }
  .nav-menu > li + li {
    border: 0;
  }
  .nav-menu > li > a {
    padding: 0.3em;
    text-align: center;
  }
}

```

← Hides toggle button and reveals the dropdown contents

← Overrides absolute positioning

← Changes the menu to a flex container and allows items to grow to fill the screen width

Even though the menu had a lot of complicated styles to make the mobile layout work, it doesn't take much to override those and revert the layout to a static, block-display element. You won't need to override the `top`, `left`, and `right` properties from the mobile styles, as those now have no effect with static positioning.

Flexbox makes a great approach here for the list items; they'll grow to fill the available width. You've also adjusted the padding of the menu items as you did with other elements; but this time, you've reduced the padding. With a medium breakpoint, you can assume the user isn't on a small phone, so you don't need the clickable areas quite so large.

8.2.3 Adding responsive columns

The final change to make for the medium breakpoint is the introduction of multiple columns. This is done exactly like multi-column layouts you've built in previous chapters. You simply need to wrap them in a media query, so they don't apply below a certain breakpoint.

When you wrote your markup, you added a `row` and `column` class where you anticipated a three-column layout. Let's define the styles for those now. Add this listing to your stylesheet.

Listing 8.11 Three-column layout inside a media query

```
@media (min-width: 35em) {
  .row {
    display: flex;
    margin-left: -.75em;
    margin-right: -.75em;
  }
  .column {
    flex: 1;
    margin-right: 0.75em;
    margin-left: 0.75em;
  }
}
```

Uses flexbox for equal-width columns

Uses negative margins to widen the row container to compensate for column margins (chapter 4, section 4.5.2)

Adds column gutters

Now, resize your browser to see the columns snap into place at the breakpoint. No specific styles apply to these elements below the breakpoint, so they stack atop one another according to normal document flow. Above the breakpoint, they become a flex container with flex items.

A lot of responsive design will come down to this sort of approach: when your design calls for items side-by-side, only place them beside each other on larger screens. On smaller screens, allow your elements to sit on their own line and fill the width of the screen. This technique can be applied to columns, media objects, or any other items that feel cramped on a narrow screen. You might be wondering how I arrived at a breakpoint of 35 em in listing 8.7. I chose it because this is the point where

the three columns started to feel overcrowded. In this case, below 35 ems, the columns were too narrow.

Web designer Brad Frost has compiled a list of responsive patterns that you can browse at <https://bradfrost.github.io/this-is-responsive/patterns.html>. You can arrange your columns for responsive design as a wide column and a narrow column, equal-width columns, or a two-column or three-column layout, to name a few examples. Ultimately, these arrangements come down to a variation of our approach here, with a combination of columns or column widths.

Sometimes, you won't even need the media queries, as natural line wrapping will take care of that for you. Flexbox layouts using `flex-wrap: wrap` and a reasonable `flex-basis` is an excellent way to do this. Similarly, a grid layout with `auto-fit` or `auto-fill` grid columns will determine how many items will fit in a row before wrapping to a new one. You could also use inline-block elements, though in that case, they won't grow to fill the width of the container.

BREAKPOINT SELECTION

I began this chapter by taking you through the simpler responsive elements of the page first, to get you accustomed to using media queries. Most of the time, you'll want to start setting breakpoints with the parts of your design that have multiple columns. Try a number of breakpoints until you find one that feels right. Ensure your columns aren't too narrow above that breakpoint.

It's easy to get sucked into thinking about specific devices. An iPhone 7 is this many pixels wide; a certain tablet is that many. Try not to worry about that. You'll find hundreds of devices with hundreds of different screen resolutions; you'll never test them all. Choose the breakpoints that make sense for your design, and it'll play out well, regardless of the device a user has.

Wanted: Container queries

Media queries build responsive designs based on the viewport size, but for several years, developers and browser vendors have been trying to find a better way. The feature many developers would like to see are *container queries* (initially called *element queries*).

Instead of responding to the viewport, this type of query would enable styles to respond to the size of an element's container. Consider the media object you built in chapter 4.



Change it up

Don't run the same every time you hit the road. Vary your pace, and vary the distance of your runs.

A media object with image and text side by side

This works on a larger screen when there is room for a side-by-side layout of the image and the content. But in a mobile design, this sort of pattern is often split so the image stacks above the text.



Change it up

Don't run the same every time you hit the road. Vary your pace, and vary the distance of your runs.

A stacked version of the media object might be more appropriate when horizontal space is limited.

Occasionally, you might want this mobile layout at a large breakpoint. Consider what would happen if this media object were placed inside of a narrow column (as those in listing 8.11): the container would be too narrow to allow for the side-by-side layout, even above the breakpoint. Instead, it would be more appropriate if you could define the responsive behavior of the media object based not on the viewport width, but rather the width of its container. Unfortunately, this isn't directly possible. The only way to achieve this layout is through the use of carefully constructed descendant selectors that would account for this scenario (for example, `.column .media > .media-image`), but this approach can be fragile.

Keep an eye out for container queries to address this need. Container queries are difficult to implement in the browser, which is why this feature hasn't emerged yet, but they're in high demand. Hopefully container queries, or something that can achieve the same result, will make their way into browsers in coming years.

8.3 Fluid layouts

The third and final principle of responsive design is *fluid layout*. Fluid layout (sometimes called *liquid layout*) refers to the use of containers that grow and shrink according to the width of the viewport. This is in contrast to a fixed layout, where columns are defined using pixels or ems. A fixed container (for example, one with `width: 800px`) will overflow the viewport on smaller devices, forcing the need for horizontal scrolling. A fluid container automatically shrinks to fit.

In a fluid layout, the main page container typically doesn't have an explicit width, or it has one defined using a percentage. It may, however, have left and right padding, or auto left and right margins to add breathing room between its edges and the edges of the viewport. This means it can be slightly narrower than the viewport, but never wider.

Inside the main container(s), any columns are defined using a percentage (for instance, a main column of 70% width and a sidebar of 30% width). This way, no matter

what the screen width is, the containers fit within. A flexbox layout works as well, assuming the flex items have `flex-grow` and, more importantly, `flex-shrink` values that allow the items to fit regardless of the screen width. Do make it a habit to think of container widths in percentages rather than in any fixed size.

NOTE The web page in this chapter isn't the only example of fluid containers you've seen. In fact, I've been using fluid layouts almost exclusively throughout this book. By now, you should already be somewhat familiar with this approach.

A web page is responsive by default. Before you apply any CSS, block-level elements are no wider than the viewport and inline elements line wrap to avoid horizontal overflow. As you add styles, it's your job to maintain the responsive nature of the page. That's sometimes easier said than done, but I find it helpful to know I always start out in a good state.

8.3.1 Adding styles for a large viewport

Let's add more media queries for your next breakpoint. As you do, observe how at every breakpoint you never fix the width of the containers. You allow them to grow naturally to 100% (minus some padding and/or margin). And, in the case of the three-column portion of the page, you use flexbox to allow the columns to achieve columns one-third the viewport width.

The final, large viewport layout of your page is shown in figure 8.9. It's similar to the medium viewport, but there is a lot more space to work with here! You can be liberal with padding, so that's exactly what you'll do.

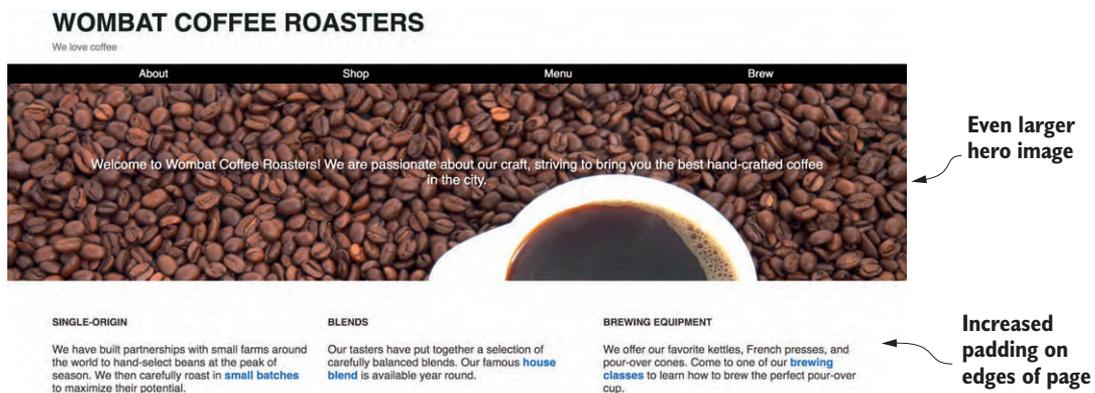


Figure 8.9 Page layout on a large viewport

The padding along the left and right edges has been increased from 1 em to 4 em. The padding on all sides of the hero image text has also been increased, allowing for a larger graphic. The additional styles are shown in listing 8.12.

Add all the (`min-width: 50em`) media query blocks to your stylesheet. Again, be sure they appear below the equivalent rules at smaller breakpoints (for `.page-header`, `.hero`, and `main`) so the styles in these media queries can override them.

Listing 8.12 Increasing paddings for large breakpoints

```
@media (min-width: 50em) {
  .page-header {
    padding: 1em 4em;
  }
}

@media (min-width: 50em) {
  .hero {
    padding: 7em 6em;
  }
}

@media (min-width: 50em) {
  main {
    padding: 2em 4em;
  }
}

@media (min-width: 50em) {
  .nav-menu {
    padding: 0 4em;
  }
}
```

Increases
left and right
padding on
edges of page
to 4 em

Increases hero padding on all
sides for a larger hero image

One last adjustment is needed. You've defined a responsive font size on the root element—`font-size: calc(1vw + 0.6em)`. On large screens, this grows too large for my taste. To fix that, you can apply an upper bound to this font size at your highest breakpoint. Update your stylesheet to match this listing.

Listing 8.13 Adding an upper limit to the responsive font size

```
:root {
  box-sizing: border-box;
  font-size: calc(1vw + 0.6em);
}

@media (min-width: 50em) {
  :root {
    font-size: 1.125em;
  }
}
```

Applies a non-scaling
font size above the
highest breakpoint

You now have a responsive page with three breakpoints. Go ahead and experiment with it. Change the breakpoint widths and see how it affects your browsing experience.

8.3.2 Dealing with tables

Tables are particularly problematic for fluid layout on mobile devices. If a table has more than a handful of columns, it can easily overflow the screen width (figure 8.10).

Country	Region/Farm	Tasting notes	Pri
Nicaragua	Matagulpa	Dark chocolate, almond	\$13.
Ethiopia	Yirgacheffe	Sweet tea, blueberry	\$15.
Ethiopia	Nano Challa	Tangerine, jasmine	\$14.

Figure 8.10 Right edge of a table clipped on a mobile device

If at all possible, I recommend you find another way to organize the data for mobile users. For instance, you could move the data from each row into its own tile, then allow the tiles to stack atop one another. Or, you could create a visual graph or chart that fits well on smaller viewports. Sometimes, however, you just need a table.

One approach you can take is to force the table to display as normal block elements. Figure 8.11 shows an example of this.

Nicaragua
Matagulpa
Dark chocolate, almond
\$13.95
Ethiopia
Yirgacheffe
Sweet tea, blueberry
\$15.95
Ethiopia
Nano Challa
Tangerine, jasmine
\$14.95

Figure 8.11 Tabular data with `display: block` applied to each row and cell

This layout is made up of `<table>`, `<tr>`, and `<td>` elements, but the declaration `display: block` has been applied, overriding their normal table, table-row, and table-cell display values. You can use a `max-width` media query to limit these changes to

small viewports. The CSS for this table is shown in the next listing. (You can apply this to any `<table>` markup to see the results.)

Listing 8.14 Forcing a responsive table layout on a mobile device

```
table {
  width: 100%;
}

@media (max-width: 30em) {
  table, thead, tbody, tr, th, td {
    display: block;
  }

  thead tr {
    position: absolute;
    top: -9999px;
    left: -9999px;
  }

  tr {
    margin-bottom: 1em;
  }
}
```

Makes all table elements block display

Hides the headings row by moving it off the screen

Adds a little space between each set of table data

This causes each cell to stack atop one another, then adds a margin between each `<tr>`. This approach makes the `<thead>` row no longer line up with columns beneath it, so I use some absolute positioning to remove the header row from view. I avoid `display: none` for accessibility: I want the headings to remain present to a screen reader. This is not a perfect solution by any means, but when all else fails, it could be the best approach.

8.4 Responsive images

In responsive design, images need special attention. Not only do you need to fit them on the screen, you must also consider the bandwidth limitations of mobile users. Images tend to be among the largest resources used on a page. The first thing you should do is always make sure your images are well compressed. Use the Save for web option in your image editor, which will greatly reduce the image’s file size, or use another image compression tool such as <https://tinypng.com/>.

You should also ensure they’re not any higher resolution than necessary. Determining what “necessary” means, however, depends on the viewport size. You don’t need to serve as large a file to smaller screens because they’ll be scaled down anyway.

8.4.1 Using multiple images for different viewport sizes

The best practice is to create a few copies of an image, each at a different resolution. If you know, based on media queries, that the screen is a certain size, there’s no sense sending an extremely large image; the browser will have to downscale it to make it fit.

Use responsive techniques to serve each to users with the appropriate screen size. For the hero image on your page, this looks like the CSS shown here. Add this to your stylesheet.

Listing 8.15 Adding a responsive background image

```
.hero {
  padding: 2em 1em;
  text-align: center;
  background-image: url(coffee-beans-small.jpg);
  background-size: 100%;
  color: #fff;
  text-shadow: 0.1em 0.1em 0.3em #000;
}

@media (min-width: 35em) {
  .hero {
    padding: 5em 3em;
    font-size: 1.2rem;
    background-image: url(coffee-beans-medium.jpg);
  }
}

@media (min-width: 50em) {
  .hero {
    padding: 7em 6em;
    background-image: url(coffee-beans.jpg);
  }
}
```

← Uses the smallest image on mobile devices

← Uses a larger image on medium-size screens

← Uses the full resolution image on large screens

If you load this in your browser, you won't notice a difference at all. And that's exactly the point. If you're on a small breakpoint, your screen isn't wide enough to show the full resolution image anyway. But you did download tens or even hundreds of kilobytes fewer than the full resolution image. On an image-heavy page, this can add up and cause a noticeable difference in the page's loading time.

8.4.2 Using *srcset* to serve the correct image

Media queries solve the problem when the image is included via the CSS, but what about images added via the HTML `` tag? For inlined images, a different approach is necessary: the `srcset` attribute (short for "source set").

This attribute is a newer addition to HTML. It allows you to specify multiple image URLs for one `` tag, specifying the resolution of each. The browser will then figure out which image it needs and download that one.

Listing 8.16 Responsive `srcset` image

```

```

← Supplies a normal `src` for browsers that don't support `srcset` (for example, IE and Opera Mini)

```
srcset="coffee-beans-small.jpg 560w,  
       coffee-beans-medium.jpg 800w,  
       coffee-beans.jpg 1280w"  
/>
```

URL of each image
and its width

Most browsers now support `srcset`, but those that don't will fall back to the specified `src`, loading whichever URL is specified there. This allows you to optimize for multiple screen sizes. Even better, the browser will make adjustments for higher resolution screens. If the device's screen has a 2x pixel density, it can download a higher resolution image accordingly.

For a closer look at responsive images, visit <https://jakearchibald.com/2015/anatomy-of-responsive-images/>. This article covers a few other useful options, such as adjusting the display size based on which image is loaded.

TIP As part of a fluid layout, you should always ensure images don't overflow their container's width. Do yourself a favor and always add this rule to your stylesheet to prevent that from happening: `img { max-width: 100%; }`.

Structuring regions of the page in responsive design can be done in countless ways. Building any of them comes down to the application of the three principles—mobile first, media queries, and fluid layout.

Summary

- Always build your designs mobile first.
- Use media queries to progressively enhance the page at larger and larger viewports.
- Use fluid layouts that fit the screen at any browser size.
- Use responsive images to fit the bandwidth limitation of mobile devices.
- Don't forget to include your meta viewport tag.

Part 3

CSS at scale

Code is communication, not just with the computer, but also with other developers who'll work with the code. How you write and organize CSS is just as important as how it renders in the browser. In Part 3, chapters 9 and 10, I'll show you how to structure your CSS so that it can be understood and easily maintained in the future.

Modular CSS

This chapter covers

- Emerging problems as a project grows
- Organizing CSS into modules
- Preventing escalating selector specificity
- Surveying popular CSS methodologies

In parts 1 and 2, we looked at the intricacies of CSS and the tools it provides for laying out elements on the page. We've made sense of the box model, margin collapsing, stacking contexts, floats, and flexbox. You'll need these skills, particularly when you first set out on a new project. In the world of software development, however, you'll spend a great deal of time not only writing new code, but also updating and adding to existing code. In CSS, this brings with it an entirely new set of difficulties.

When you make changes to an existing stylesheet, those changes can affect any number of elements on any number of pages across your site. There's an old joke: two CSS properties walk into a bar; a barstool in a different bar falls over. So, how do you ensure your change applies to all the places you want updated? And, how do you know your change won't affect elements you don't want changed?

In part 3, we'll discuss these problems. We'll look at the architecture of CSS, focusing less on the declarations in your stylesheet and more on the selectors you

choose and the HTML you pair those with. How you structure your code determines whether you can safely make changes in the future without unwanted side effects. This begins with an understanding of modular CSS, which will be the focus of this chapter.

Modular CSS means breaking the page up into its component parts. These parts should be reusable in multiple contexts, and they shouldn't directly depend upon one another. The end goal is that changes to one part of your CSS will not produce unexpected effects in another.

This is akin to the use of modular furniture, such as an IKEA kitchen. Instead of building one giant kitchen cabinet unit, you can select multiple individual pieces that are designed to look similar so they fit together visually. But, you're free to put each piece wherever you want amid the full arrangement. With modular CSS, instead of building one giant web page, you build each part of the page in a way that stands alone, then you put them together in the arrangement you want.

The idea of writing modular code is not new in computer science, but developers have only begun to apply it to CSS in the past several years. As modern websites and web applications have gotten larger and more complicated, we've had to find ways to manage the growing complexity in our stylesheets.

Instead of a stylesheet where any selector can do anything anywhere on the page, modular styles allow you to impose order. Each part of your stylesheet—which we'll call a *module*—will be responsible for its own styles, and no module should interfere with the styles of another. This is the software principle of encapsulation applied to CSS.



encapsulation—The grouping together of related functions and data to comprise an object. It's used to hide the state or values of a structured object so that outside parties cannot operate on them.

CSS doesn't have the concepts of data or traditional functions, but it does have selectors and the elements those selectors target. For the purposes of encapsulation, these will be the parts that make up our module, and each module will be responsible for styling a small number of DOM elements.

With encapsulation in mind, you'll define a module for each discrete component on the page: your navigational menus, your dialog boxes, your progress bars, and your thumbnail images. Each module will be identified by a unique class name applied to a DOM element. And, each module will have its own particular pattern of child elements to construct the module on the page. You'll nest modules inside other modules, altogether constructing a complete page.

9.1 *Base styles: laying the groundwork*

Before you dive into writing modular styles, you'll need to set up the environment. Every stylesheet begins with a set of generic rules that apply to the whole page; this is still necessary with modular CSS. These rules are often called *base rules* because they

lay the foundation upon which the rest of your styles will be built. This portion of the stylesheet won't be modular, per se, but it'll lay the groundwork for the modular styles that follow.

Create a new page and stylesheet, and apply the base styles in the following listing to the CSS. The styles here are merely an example of some base styles you could use.

Listing 9.1 Adding base styles

```

:root {
  box-sizing: border-box;
}

*,
*::before,
*::after {
  box-sizing: inherit;
}

body {
  font-family: Helvetica, Arial, sans-serif;
}

```

Box sizing reset (chapter 3)

Default font size for the page

Other base styles typically include link colors, heading styles, and margins. By default, the `<body>` has a small margin, which you may want to “zero” out. Depending on the project, you'll potentially also want to apply styles for form fields, tables, and lists.

TIP I recommend a library called `normalize.css`. This is a small stylesheet that helps even out discrepancies among the user agent stylesheets of various browsers. You can download it from <https://necolas.github.io/normalize.css/>. Add this before your stylesheet as part of your base styles.

Your base styles should be fairly generic. Only add styles here that you want applied to most or all of the page. There should be no class names or IDs in your selectors, so you only target elements by their tag type and the occasional pseudo-class. The idea is that these styles provide the global look you want, but are easy to override later when you need to.

Once your project's base styles are set, they'll rarely change. They provide a stable foundation upon which you can build your modular CSS. After your base styles, the rest of your stylesheet will consist mainly of modules.

9.2 A simple module

Let's create a simple module for brief notification messages. Because each module needs a unique name, you'll call this one “message.” You'll give it a bit of color and a border to capture the user's attention (figure 9.1).

Save successful

Figure 9.1 The Message module

The markup for this module is a single `div` with a `message` class. Add it to your page.

Listing 9.2 Markup for a Message module

```
<div class="message">
  Save successful
</div>
```

The CSS for this is one ruleset. It targets the module by its class name. Then it defines padding, border, border-radius, and the colors. To apply these styles to the `Message` module, add this listing to your stylesheet, following the base styles.

Listing 9.3 Implementing the Message module

```
.message {
  padding: 0.8em 1.2em;
  border-radius: 0.2em;
  border: 1px solid #265559;
  color: #265559;
  background-color: #e0f0f2;
}
```

← Targets the Message module by its class name

You should be familiar with all of these properties, so this may seem like nothing special at this point. It looks much like other CSS you've seen throughout the book. In fact, much of the CSS you've written follows principles of modular CSS, I just haven't drawn attention to it before now. Let's evaluate what makes this CSS modular.

It's important that the selector for this module consists only of the single class name. Nothing else in the selector restricts these styles to a certain place on the page. In contrast, using a selector like `#sidebar .message` would mean this module could only be used inside a `#sidebar` element. Without this restriction, the module is reusable in any context.

By adding this class to an element, you can now reuse these styles to give user feedback on form input, provide noticeable help text, or draw attention to a legal disclaimer to name just a few examples. By reusing the same module, you produce a consistent UI. Everywhere you use it will look the same. You won't have a slightly different teal color in some places or a slightly larger padding in others.

I've worked on projects where the CSS wasn't modular. In one, a series of similar buttons appeared throughout the application. There was a `.save-form` button and a `.login-form` button and a `.toolbar .options` button. The same code was repeated multiple times in the stylesheet, though these were imperfect copies of one another. The intent was to have a consistent experience, but slow evolution over time introduced changes that never propagated to every instance of a particular button. So, some buttons had a slightly different padding or a brighter color of red.

The solution was to refactor it into a single reusable module that worked regardless of its location on the page. By creating a module, it ensured not only simpler code (less

repetition), but also visual consistency. It looked more professional and less quickly thrown together. At a subconscious level, this helps users trust your application.

9.2.1 Variations of a module

Consistency is good, but sometimes you'll want to intentionally deviate from it. Our Message module is nice, but we might need it to look different under certain circumstances. For instance, if you need to display an error message, perhaps it should be colored red rather than teal. Or, perhaps you want to distinguish between messages that are purely informational and those that indicate a successful action, such as saving. You do this by defining *modifiers*.

You create a modifier by defining a new class name that begins with the module's name. For instance, an error modifier for the Message module might be `message--error`. By including the module name, you clearly indicate that this class belongs with the Message module.

NOTE A popular convention is to use two hyphens to indicate a modifier: `message--error`, for example.

Let's create three modifiers for your module—success, warning, and error. Add this to your stylesheet.

Listing 9.4 Message module with modifier classes

```
.message {
  padding: 0.8em 1.2em;
  border-radius: 0.2em;
  border: 1px solid #265559;
  color: #265559;
  background-color: #e0f0f2;
}

```

← Base Message module

```
.message--success {
  color: #2f5926;
  border-color: #2f5926;
  background-color: #cfe8c9;
}

```

← Success modifier changes message to green.

```
.message--warning {
  color: #594826;
  border-color: #594826;
  background-color: #e8dec9;
}

```

← Warning modifier changes message to yellow.

```
.message--error {
  color: #59262f;
  border-color: #59262f;
  background-color: #e8c9cf;
}

```

← Error modifier changes message to red.

The modifier styles don't need to redefine the entire module. They only need to override the parts they change. In this case, this means changing the color of the text, border, and background.

To use a modifier, add both the main module class and the modifier class to an element as shown next. This applies the module's default styles and then allows the modifier to selectively override them where needed.

Listing 9.5 An instance of the Message module with the error modifier applied

```
<div class="message message--error">
  Invalid password
</div>
```

← Adds both classes to the element

Likewise, add the success or warning modifier when you need to use those versions. These modifiers change the colors of the module, but others could change the size or even the layout of a module.

BUTTON MODULE VARIANTS

Let's create another module with some variants. You'll make a Button module with variants for large and small sizes, as well as color options (figure 9.2). This way, you can use color to add visual meaning to the buttons. Green will indicate a positive action, such as saving or submitting a form. Red will indicate a warning to help prevent the user from accidentally clicking a cancel button.



The styles for these buttons are given in the next listing. It includes the main Button module, as well as four modifier classes: two size modifiers and two color modifiers. Add these to your stylesheet.

Listing 9.6 Button module and modifiers

```
.button {
  padding: 0.5em 0.8em;
  border: 1px solid #265559;
  border-radius: 0.2em;
  background-color: transparent;
  font-size: 1rem;
}

.button--success {
  border-color: #cfe8c9;
  color: #fff;
  background-color: #2f5926;
}
```

← Basic button styles

← Green success color variant

```

.button--danger {
  border-color: #e8c9c9;
  color: #fff;
  background-color: #a92323;
}

.button--small {
  font-size: 0.8rem;
}

.button--large {
  font-size: 1.2rem;
}

```

Red danger color variant

Small variant

Large variant

The size modifiers work by setting a smaller or larger font size. You used this technique in chapter 2. Changing the font size adjusts the element's em size, which, in turn, changes the padding and border radius without having to override their declared values.

TIP Always keep all the code for a module together in the same place. Then your stylesheet will consist of a series of modules, one after another.

With these modifiers in place, the author of the HTML has options to choose from. They can add modifier classes to change the size of the button based on its importance. They can also choose a color to suggest contextual meaning to the user.

This listing shows some HTML, mixing and matching the modifiers to create various buttons. Add this anywhere in your page to see the results at work.

Listing 9.7 Using modifiers to create various types of buttons

```

<button class="button button--large">Read more</button>
<button class="button button--success">Save</button>
<button class="button button--danger button--small">Cancel</button>

```

Button module with large modifier

Button module with success modifier

Button module with danger and small modifier

The first button here is large. The second has the green success coloring. The third has two modifiers: one for color (danger) and one for size (small), like the buttons in figure 9.2.

The double-hyphen syntax may seem a little odd. The benefit becomes more apparent when you start creating modules with longer names, such as `nav-menu` or `pull-quote`. Adding a modifier to these produces class names like `nav-menu--horizontal` or `pull-quote--dark`.

The double-hyphen syntax shows at a glance which part of the class is the module name and which part is the modifier; `nav-menu--horizontal` indicates something

different than `nav--menu-horizontal`. This adds clarity in projects with a lot of modules that have similar names.

NOTE This double-hyphen notation has been popularized by a methodology called *BEM*. I'll introduce you to BEM and some other similar methodologies near the end of the chapter.

DON'T WRITE CONTEXT-DEPENDENT SELECTORS

Imagine you're maintaining a website that has light-colored dropdown menus. One day your boss tells you that the dropdown menu in the page header needs to be inverted so it's dark with white text.

Without modular CSS, your first inclination is going to be to target that particular dropdown with a selector that looks something like this: `.page-header .dropdown`. Using that selector, you'd then override the default colors applied by the dropdown class. With modular CSS, this selector is strictly forbidden. Although using a descendant selector may work now, this approach leads to many problems down the road. Let's consider the ramifications.

First, you must decide where this code belongs. Should it go with the styles for the page header or those for the dropdown? After adding enough single-purpose rules like this, the stylesheet will turn into a haphazard list of unrelated styles. And, if you need to modify these styles later, will you remember where you placed them?

Second, this approach has incrementally increased the selector specificity. When situations arise where you want to make further changes, you'll need to meet or exceed this specificity.

Third, you might later find you need this dark dropdown in another context. The one you created is bound to the page-header. If you want another dark dropdown in a sidebar, you'll need to add new selectors to the ruleset to make it match both scenarios or duplicate the styles entirely.

Finally, continued use of this practice produces longer and longer selectors that bind the CSS to a particular HTML structure. For example, if you've a selector like `#products-page .sidebar .social-media div:first-child h3`, that ruleset is tightly coupled to a specific place on a specific page.

These issues are at the root of many frustrations developers have concerning CSS. The longer a stylesheet is used and maintained, the worse they become. Selector specificity continually ratchets higher as new styles are needed to override old ones. Before you know it, you'll find yourself writing a selector with two IDs and five classes just to target a checkbox.

Rules become hard to find as individual elements are targeted by pieces of code in multiple disparate parts of the stylesheet. It becomes more and more difficult to understand the organization of the stylesheet and how it's doing what it does to the page. Hard-to-understand code means bugs become more common. Even the tiniest changes to the page can break a huge portion of its styling. Deleting old code becomes unsafe because nobody knows what it does and whether it's still important.

The stylesheet grows in length, and the problems only compound. These are the problems that modular CSS seeks to prevent.

When you need a module to look or behave differently, create a modifier class that can be applied directly to the specific element. Instead of writing `.page-header.dropdown`, for example, write `.dropdown--dark`. This way, the module, and only the module, is in charge of its own appearance. Other modules can't reach into it to make changes. The dark dropdown is not bound to any deeply nested structure in the HTML, so you can place it anywhere in the page you need it.

Never use descendant selectors to alter a module based on its location in the page. Following this one rule is the best thing you can do to prevent your stylesheet from ever descending into the madness of unmaintainable code.

9.2.2 Modules with multiple elements

The two modules you've built so far—message and button—are nice and simple; they consist of only one element. But many modules you'll build will need more elements. You can't build a dropdown menu or a modal with only one element.

Let's build a more complex module. This will be a media object (figure 9.3), like the one you made in chapter 4 (section 4.5.1).

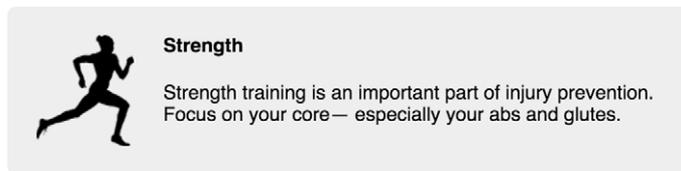


Figure 9.3 Media module made with four elements

This module consists of four elements: it has a `div` container that includes an image and a body, and inside the body is the title. As with other modules, you'll give the main container the class name `media` to match the name of the module. For the image and the body, you'll use the class names `media__image` and `media__body`. These begin with the module name, followed by a double-underscore, then the name of the sub-element. (This is another convention from BEM methodology.) As with double-hyphen modifiers, the class name tells you at a glance what role the element plays and what module it belongs to.

The rules for the Media module are shown here. Add these to your stylesheet.

Listing 9.8 Media module with sub-elements

```
.media {
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}
```

← Main container

```

.media::after {
  content: "";
  display: block;
  clear: both;
}

.media__image {
  float: left;
  margin-right: 1.5em;
}

.media__body {
  overflow: auto;
  margin-top: 0;
}

.media__body > h4 {
  margin-top: 0;
}

```

← **Clearfix**

← **The image and body sub-elements**

← **The title within the body**

You'll notice you didn't have to use many descendant selectors. The image is a child element of the Media module, so you could use the selector `.media > .media__image`, but that's not necessary. Because the name of the module is already in the `media__image` class; it's guaranteed to be unique.

I did use a direct descendant combinator for the title. I could use the class `media__title` (or even `media__body__title` to fully indicate its position in the hierarchy), but I find this isn't always necessary. In this case, I decided the `<h4>` tag is descriptive enough to indicate the title of the Media module. This precludes us from using different headings (`<h3>` or `<h5>`) for a media title. If, in your modules, you don't like a restriction like that, use a class name to target the element instead.

Add the markup for this module to your page.

Listing 9.9 Markup for the Media module

```

<div class="media">
  
  <div class="media__body">
    <h4>Strength</h4>
    <p>
      Strength training is an important part of
      injury prevention. Focus on your core&mdash;
      especially your abs and glutes.
    </p>
  </div>
</div>

```

← **Image sub-element**

← **Body sub-element**

← **Title sub-element**

This is a versatile module. It works inside containers of multiple sizes, growing naturally to fill their width. You can put multiple paragraphs inside the body. You can use it with larger or smaller images (though you might want to consider adding a `max-width` to the image so it doesn't crowd out the body).

USE VARIANTS AND SUB-ELEMENTS TOGETHER

You can also create variations of the module. Now, it's trivial to make a version where the image floats right instead of left (figure 9.4).

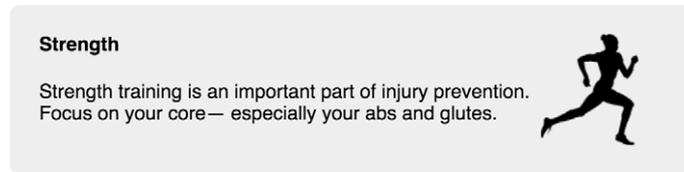


Figure 9.4 Media module with right variant

A `media--right` variant will do this work. You can add the variant class to the module's main `div` (making it `<div class="media media--right">`). Then you can use that class to target the image and float it to the right.

Add the modifier class to the element in your HTML. Append the following listing to your stylesheet to see this version.

Listing 9.10 Defining a right variant for the Media module

```
.media--right > .media__image {
  float: right;
}
```

← Targets image sub-element, but only when right modifier is present

This rule overrides the media image's original `float: left`. Because of the way floats work, you don't need to re-order the elements in the HTML.

AVOID GENERIC TAG NAMES IN MODULE SELECTORS

In the Media module, I used the selector `.media__body > h4` to target the title element. I was able to do this because `<h4>` is an element that should only represent a minor heading. I also use this technique for modules with lists; I find it simpler to target menu items with `.menu > li` than it is to add a `menu__item` class to each and every item in a list, though opinions vary on this issue.

You should avoid targeting based on generic tag types, such as `div` and `span`. A selector like `.page-header > span` is too broad. When you initially build the module, you might only be using a `span` in that module for one thing, but nothing says you won't come back later and need to add a second `span` for a different purpose. Adding a class to the spans later is difficult because it'll involve hunting down every use of that module in your markup and making the change there.

9.3 Modules composed into larger structures

In his book, *Clean Code*, Robert C. Martin says, "The first rule of classes is that they should be small. The second rule of classes is that they should be smaller than that."

He was speaking of classes in object-oriented programming at the time, but the same principle applies just as well to modules in CSS.

Your modules should each be responsible for one thing. Our Message module is responsible for making a message noticeable. Our Media module is responsible for positioning an image beside some text. You should be able to concisely summarize its purpose. Some modules will be for layout, others will be for stylistic purposes. When a module tries to do more than one thing, you should consider breaking it into smaller modules.

To demonstrate, let's build a dropdown menu (figure 9.5). This will appear like the one you built in chapter 7 (section 7.3.1).

When you start a module, ask yourself, “At a high level, what’s the module’s responsibility?” In this case, your first answer might be, “A button that visually toggles a dropdown menu and presents menu items stacked atop one another.”

It’s an apt description of what you need for this scenario, but I’ve a rule of thumb: “If I have to use the word *and* in describing the module’s responsibility, then I might be describing multiple responsibilities.” Does it toggle a menu, or does it present stacked menu items?

When you need to use the word *and* to describe a module’s responsibility, consider whether you’re potentially describing two (or more) responsibilities. You might not be—this is not a hard-and-fast rule. But, if you are, you should define separate modules for each responsibility. This is an important principle of encapsulation, which is called the *Single Responsibility Principle*. When possible, distribute multiple responsibilities to multiple modules. This will keep each module small, focused, and easier to understand.

9.3.1 *Dividing multiple responsibilities among modules*

Let’s build the dropdown menu with two different modules. The first, which I’ll call *dropdown*, will have a button responsible for controlling the visibility of a container. You could further break this down and say it’s responsible for revealing the container and hiding the container. You could also describe the button’s appearance and the small triangle that indicates the action. Describing the module in this detail requires the use of *and*, but these points are all subordinate to the primary responsibility, so I think we’re in good shape.

The second module, which I’ll call *menu*, will be the list of stacked links. You’ll then compose the full interface by placing an instance of the menu module inside the container of a Dropdown module.

To get started, place the markup from the next listing into your page. This consists of a Dropdown module, which contains a menu module. It also contains a minimal bit of JavaScript to add open and close functionality when the toggle is clicked.

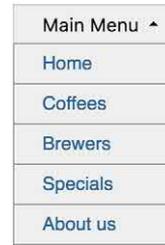


Figure 9.5
Dropdown menu

Listing 9.11 Constructing a dropdown menu from two modules

```

<div class="dropdown">
  <button class="dropdown_toggle">Main Menu</button>
  <div class="dropdown_drawer">
    <ul class="menu">
      <li><a href="/">Home</a></li>
      <li><a href="/coffees">Coffees</a></li>
      <li><a href="/brewers">Brewers</a></li>
      <li><a href="/specials">Specials</a></li>
      <li><a href="/about">About us</a></li>
    </ul>
  </div>
</div>

<script type="text/javascript">
(function () {
  var toggle =
    document.querySelector('.dropdown_toggle');
  toggle.addEventListener('click', function (event) {
    event.preventDefault();
    var dropdown = event.target.parentNode;
    dropdown.classList.toggle('is-open');
  });
})();
</script>

```

Toggle button for the dropdown

A drawer sub-element serves as the menu container

A Menu module placed inside the drawer

Toggles the is-open class when the toggle button is clicked

I've used the double-underscore notation here to indicate that toggle and drawer are sub-elements of the Dropdown module. Clicking the toggle will reveal or hide the drawer. The JavaScript does this by adding or removing the `is-open` class on the dropdown's main element.

The styles for the dropdown are shown in the next listing. Add this to your stylesheet. These styles are similar to those shown in chapter 7. I've updated the class names to match the double-underscore notation. This gives you a functioning dropdown, though the menu inside it remains unstyled.

Listing 9.12 Defining a Dropdown module

```

.dropdown {
  display: inline-block;
  position: relative;
}

.dropdown_toggle {
  padding: 0.5em 2em 0.5em 1.5em;
  border: 1px solid #ccc;
  font-size: 1rem;
  background-color: #eee;
}

.dropdown_toggle::after {
  content: "";
}

```

Establishes containing block for the absolutely positioned drawer

```

position: absolute;
right: 1em;
top: 1em;
border: 0.3em solid;
border-color: black transparent transparent; ←
}

.dropdown_drawer {
display: none;
position: absolute;
left: 0;
top: 2.1em;
min-width: 100%;
background-color: #eee;
}

.dropdown.is-open .dropdown_toggle::after {
top: 0.7em;
border-color: transparent transparent black;
}
.dropdown.is-open .dropdown_drawer {
display: block;
}

```

Draws the triangle using borders (chapter 7)

Hides the drawer initially, then displays it when the is-open class is present

Inverts the triangle while the dropdown is open

This listing uses relative positioning on the main element to establish the containing block for the drawer, which is, in turn, absolutely positioned within. It provides some styles for the toggle button, including the triangle in the `::after` pseudo-element. Lastly, it reveals the drawer and inverts the triangle when the `is-open` class is present.

This is about 35 lines of code. A handful of things is going on here, but it's not so much you can't keep it all in your head while you work on this module. When, down the road, you have to come back and make changes to a module, you'll want it to be small enough so that you can figure it out reasonably quickly.

POSITIONING IN A MODULE

This module is the first one you've built that uses positioning. Notice that it establishes its own containing block (`position: relative` on the main element). The absolutely positioned elements (the drawer and the `::after` pseudo-element) are based on positions that are defined within the same module.

When possible, I try to keep positioned elements that are related to one another within the same module. This way, the module won't behave strangely if I place it inside another positioned container.

STATE CLASSES

The `is-open` class has a special purpose in the Dropdown module. It's intended to be added to or removed from the module dynamically using JavaScript. It's also an example of a *state class* because it indicates something about the current state of the module.

It's a common convention to design all state classes so they begin with *is-* or *has-*. That way, their purpose is readily apparent; they indicate something about the module's current state and are expected to change. Other examples of state classes could be *is-expanded*, *is-loading*, or *has-error*. The precise nature of what these do would depend on the module where they're used.

IMPORTANT Group the code for state classes along with the rest of the code for the module. Then, any time you need to use JavaScript to dynamically change the appearance of a module, use the state class to trigger the change.

Preprocessors and modular CSS

A feature that all preprocessors like Sass or LESS provide is the ability to merge separate CSS files into one. This lets you organize your styles into multiple files and directories, but serve them all to the browser as one file. It reduces the number of network requests necessary from the browser, allowing you to break up your code into manageable sizes. In my opinion, this is one of the most valuable features preprocessors provide.

If you happen to use a processor, I strongly encourage you to place each module of your CSS in its own appropriately named file. Organize these into directories as needed. Then create a master stylesheet that imports all your modules. This way, you won't have to hunt through one long stylesheet for each module when you want to make a change. You'll know exactly where you need to look.

You could create a `main.scss` that contains only `@import` statements. It would look something like this:

```
@import 'base';
@import 'message';
@import 'button';
@import 'media';
@import 'dropdown';
```

The preprocessor would then bring in your base styles from `base.scss` and your module styles each from its own file, outputting a single stylesheet that contains all these styles. That way, each module has its own file, which makes for easier editing.

See appendix B for more information on preprocessors, or consult your preprocessor's documentation for more on using its import directive.

THE MENU MODULE

With the Dropdown module working, you can now focus your attention on the Menu module. You don't need to worry any longer about the opening and closing of the dropdown, as that's already taken care of by the Dropdown module. The Menu module will apply the look and feel you need to the list of links.

The styles for this are shown here. Add these to your stylesheet.

Listing 9.13 Menu module styles

```

.menu {
  margin: 0;
  padding-left: 0;
  list-style-type: none;
  border: 1px solid #999;
}

.menu > li + li {
  border-top: 1px solid #999;
}

.menu > li > a {
  display: block;
  padding: 0.5em 1.5em;
  background-color: #eee;
  color: #369;
  text-decoration: none;
}

.menu > li > a:hover {
  background-color: #fff;
}

```

← Overrides user agent styles to remove list bullets

← Adds a border between each link

← Styles large clickable links

← Adds highlight on hover

These are the same declarations you used in your dropdown in chapter 7. Each `` is a sub-element of the module, so I didn't feel the need to add a double-underscore to each and every one. The direct descendant selector `.menu > li` is specific enough.

This module is completely standalone. It has no dependency upon the Dropdown module. This keeps the code simpler because you don't have to understand one to make sense of the other. It also enables a more flexible reuse of these modules.

You could create a different style of menu—whether a variant or a completely separate module—and use it inside of a dropdown should the need arise. You can also use this menu elsewhere, outside of the dropdown. You can't often predict what your pages might need in the future, but with reusable modules, you leave the door open to mix and match new possibilities with a familiar, consistent look and feel.

9.3.2 Naming modules

Choosing appropriate module names takes thought. You can throw in a temporary name as you develop the module, but before you consider it complete, make sure you have given some attention to its name. Doing this well is possibly the most difficult part of writing modular CSS.

Consider the Media module from earlier in the chapter. You used it to display an image of a runner and running tip as shown in figure 9.6.

Imagine you haven't yet named the module, but you know a page needs it. You could name it `running-tip`. This is accurate and, initially, it seems fitting. But consider other things you might want to do with the styles in this module. What if you wanted to use the same UI element for something else? Following the theme of a running

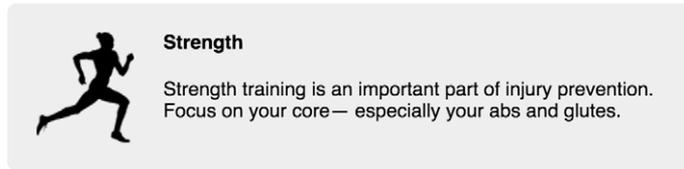


Figure 9.6 Media module with an image and a running tip

website, you might want to use a series of these to list upcoming races. If you do that, the name `running-tip` is suddenly out of place.

You need to give the module a name that's meaningful no matter what context you might want to use it. You should also avoid names that simply describe the visual appearance. Calling this a "gray-box-with-image" seems more versatile, but what if you decide later the background color should be light blue? Or, you perform a complete redesign of the site? This name would no longer be applicable, and you would have to rename it and update the name everywhere it appears in the HTML.

Instead, you need to ask yourself what this module represents conceptually. This isn't always an easy task. The name `media` works well on this level; it supports some sort of image and text. It gives a strong impression, without tying the module to any one particular use or visual implementation.

A module should be versatile. Its name will ideally be simple and memorable. As you design more pages for your site, you can reuse a module for any number of things. You and other members of your team can adopt it into the language you use when speaking about the site: "Let's use a *media* here," or, "These *tiles* are too close together."

So far, you've created a `Message`, a `Media`, a `Dropdown`, and a `Menu` module. Some other examples of good module names include `panel`, `alert`, `collapsible-section`, and `form-control`. It helps if you've a general sense of the overall design of the site from the beginning. You might know, for example, that there'll be two unrelated UI elements that could be rightly called `tiles`; in which case, you should take care to name them more precisely (perhaps `media-tile` and `headline-tile`).

Some people enforce the use of two words in the name of every module so that nothing is too generic; you never know when the need for a completely new `tile` module might arise. If your existing `tile` module is more precisely named, it can leave you a little more freedom in how you can name the new one without the two becoming conflated.

When naming variant classes for your module, you should apply similar principles. For instance, if you've a `Button` module, don't subclass red and blue variants with `button--red` and `button--blue`. The design of your site is likely to change in the future, and you don't know whether these colors will change as well. Try to find more meaningful names like `button--danger` and `button--success` instead.

Modifiers with a more relative meaning like `large` or `small` are not perfect, but they can be acceptable. Nothing says you can't change the size of a `button--large` during

a redesign, as long as it remains larger than a regular button. But certainly avoid overly precise modifiers like `button--20px`.

9.4 *Utility classes*

Sometimes you'll need a class to do one simple, very specific thing to an element. This could mean centering text, floating it left, or adding a clearfix, for example. These classes are called *utility classes*.

In some ways, utility classes are like small modules. A utility class, however, should be laser-focused. Rarely will it be more than one declaration. I like to keep these classes all near the end of my stylesheet, below all my modules.

The next listing shows four utility classes. Each perform a specific action: center text, float left, clearfix (contain floats), and hide an element.

Listing 9.14 Examples of utility classes

```
.text-center {
  text-align: center !important;
}

.float-left {
  float: left;
}

.clearfix::before,
.clearfix::after {
  content: " ";
  display: table;
}

.clearfix::after {
  clear: both;
}

.hidden {
  display: none !important;
}
```

Centers text within a container

Floats an element left

Clearfix

Hides an element on the page

Yes, I used `!important`. Twice. Utility classes are the only place you should use the important annotation. In fact, it might even be preferred. No matter where you apply the utility class, it will work. I guarantee you, any time you add the class `text-center` to an element, it's because you want its text centered and you don't want any other styles to override this. Using the important annotation ensures this.

To see these classes in action, add them to an element on your page. A `<div class="text-center">` centers any text within. Add `float-right` to an `` to float it, and `clearfix` to its container to make it contain the float.

Utility classes are meant to be quick helpers. You don't need a full module when you need to do one simple thing on the page. When this is the case, use a utility class. Don't get carried away, though. On most sites, you probably won't need more than a dozen or so of these classes.

9.5 CSS methodologies

The concept of modular CSS began to emerge several years ago. Developers who had experienced the problems of scaling up CSS for large projects began to create prescribed methodologies to enable code reuse and reduce bugs. In the years since, new methodologies have built upon these ideas. These don't come with any sort of library or technology, but they do provide a set of guidelines to help organize your CSS.

I owe a lot of the wisdom in this chapter to the people that have gone before me. If you follow the advice laid out in this chapter, you'll be a long way toward following most of these methodologies.

These practices have become transformative in the CSS world. It's worth knowing about a few of the big ones. Some of these are simple, offering only a few guidelines. Others are more rigid, providing a strict organization for your styles. Each has its own terminology and naming conventions, but fundamentally they all come back to a modular approach to CSS:

- OOCSS—Object-oriented CSS, created by Nicole Sullivan:
<https://github.com/stubbornella/oocss/wiki>
- SMACSS—Scalable and Modular Architecture for CSS, created by Jonathan Snook:
<https://smacss.com/>
- BEM—Block, Element, Modifier, developed at Yandex:
<https://en.bem.info/methodology/>
- ITCSS—Inverted Triangle CSS, created by Harry Roberts:
www.creativebloq.com/web-design/manage-large-css-projects-itcss-101517528

The order of this list is more or less chronological. It also corresponds with an increasing amount of structure. OOCSS is based on a few guiding principles, whereas ITCSS has specific rules of naming classes and categorizing styles; SMACSS and BEM fall in between.

In this chapter, I've taught you the three main sections of your stylesheet: base rules, module rules, and utility classes. SMACSS adds a section for layout rules, which deals with laying out major regions of the page (sidebar, footer, and maybe a grid system). ITCSS divides the categories further into seven sections.

If these methodologies interest you, I encourage you to read up on them further. They have differing terminology, but in many ways, they complement one another. Choose one that's your favorite or tailor your approach to modular CSS to your liking. If you work with a team, find something that all of you can agree on. If you don't like the BEM naming syntax I've shown you, find another or come up with a new one that meets your needs.

Alternate approaches in JavaScript

With large teams, writing modular styles requires a certain amount of rigor to ensure everyone is following the same conventions. It also requires taking steps to ensure nobody creates module names that conflict with one another. To solve these problems, some web development communities began experimenting with alternative approaches to modular CSS. In search of a solution, they turned to JavaScript. They've devised an approach known as *inline styles* or *CSS in JS*.

Instead of relying on class-naming conventions, this approach requires using JavaScript to either manufacture class names that are guaranteed unique or to apply all styles to the page using the HTML `style` attribute. A number of JavaScript libraries have emerged to do this work, the most popular of which are Aphrodite, Styled Components, and one called (confusingly) CSS Modules. Most of these are tied to a particular JavaScript framework or tool set, such as WebPack.

This approach is still experimental (and a bit controversial), but it's worth being aware of, especially if you develop single page applications (SPAs). It only works for applications that are rendered completely by a JavaScript framework like ReactJS. Going down this road involves some trade-offs, and it locks you in to a particular tool set. It's not a perfect solution, but has proven successful for some.

Summary

- Break your CSS up into small, reusable modules.
- Never write styles that reach into another module and change its appearance.
- Use variant classes to provide multiple versions of the same module.
- Divide large constructs into smaller modules; build your pages by piecing together a number of modules.
- Group all rules for a module together in your stylesheet.
- Use a naming convention such as double-hyphens and double-underscores to make your modules' structure easier to understand at a glance.

10

Pattern libraries

This chapter covers

- Building a pattern library to document your modules
- Incorporating a pattern library into your development process
- Applying a CSS First approach to writing styles
- Safely editing and deleting CSS
- Utilizing CSS frameworks such as Bootstrap

After you start writing your CSS in a modular way, it'll begin to shift the way you approach the task of authoring web pages and web applications. At first, the pages you build may not seem different. But at some point, you'll go to put together a particular page, and you'll find that you have already created many of the modules it requires. For example, if you need a media object, or a dropdown or a navigation menu, and you've already created one, you already have the styles ready for it. You'll need only to add the correct class names to a few elements structured in the correct manner.

Because the modules are reusable, you'll be able to build those portions of the page without adding any new CSS to your stylesheet. Instead of writing an HTML

page then applying styles, you'll find yourself taking modules that already exist, and using them to piece together a new page. The further you progress into the project, the less you'll need to write new CSS. Instead of new styles, what you'll need is an inventory of all the modules already available in your stylesheet.

It's becoming standard practice on large projects to put together a set of documentation that provides this inventory. This set of documentation is called a *pattern library* or a *style guide*. It's not part of your website or application; instead, it's a separate set of HTML pages, showcasing each CSS module. This is a development tool that you and your team will use when building the site.

In this chapter, I'll show you how to build a pattern library. Countless tools are available to help with this (though it can be done entirely without tools, if you're enterprising enough). I'll show you one such tool, called KSS, though my focus will extend beyond this to include principles that apply regardless of the tool you use.

After you get your pattern library started, I'll highlight the key benefits that it provides and how it can improve your development process, especially for large projects. This chapter is a continuation of chapter 9, so if you skipped ahead to this point, I encourage you to go back and read that chapter first.

Pattern library vs. style guide

Some pattern libraries are often called a style guide (or “living style guide”). In fact, style guide is probably the more common—there's a distinction, however.

The name *style guide* implies not only technical instruction on how to use the modules, but also opinionated direction about when and why you should or should not use them. This direction is typically for guiding a developer through requirements of the product's branding.

If branding instruction is relevant on your project, feel free to add it to your pattern library. But that gets into the realm of marketing rather than development. Because this chapter focuses on the technical documentation aspect, I'll use the name *pattern library* instead.

10.1 Introduction to KSS

I've made a point throughout the book not to talk much about tooling. The most important principles of CSS apply regardless of your toolset, and I've wanted the focus to be on those principles, not on which preprocessor or build tool you use.

Building a pattern library, although possible without any particular tooling, is much easier with some help from tools. A number of such libraries are available to help with this—run a web search for “style guide generator” and you'll find plenty. No one clear industry leader exists, but one that remains consistently near the top of the list is KSS. This stands for Knyle Style Sheets (“Knyle” being a reference to Kyle Neath, the author).

I'll show you how to get KSS set up and running. Once it's configured, it scans your stylesheet for comment blocks that have a certain Styleguide annotation. You'll use each of these comment blocks to describe the purpose and use of each module; KSS uses this to build the HTML documentation. The comment can also include a snippet of HTML, illustrating the markup required to render the module. KSS uses this to render a live demo of the module in the documentation, similar to the screenshot in figure 10.1.

Figure 10.1 Rendered KSS documentation of a Dropdown module

In this screenshot, you can see a menu on the left, listing the sections of the pattern library. On the right is the documentation for the Dropdown module (like the one you built in chapter 9). This includes a rendered version of a dropdown menu, plus the HTML used to build it. With this as a guide, anyone versed in HTML can then replicate this markup on a page, and your stylesheet would apply this appearance.

10.1.1 Setting up KSS

KSS was originally written as a Ruby application. But, because you're in the realm of front-end development, chances are you're more familiar with JavaScript, so I'll walk you through installing a Node.js implementation of KSS.

If you don't have Node.js installed, you can find it for free at <https://nodejs.org>. Download and install it according to the directions given there. Node comes with a package manager (called npm), which you'll use to install KSS. I'll show you the commands needed for this, but if you want to learn more about npm or need help troubleshooting anything, visit <https://docs.npmjs.com/getting-started/>.

INITIALIZE YOUR PROJECT

Once Node and npm are installed, create a directory for your project wherever you prefer on your file system. Navigate to it in the terminal. Run `npm init -y` to initialize a new project. The `-y` flag automatically sets the defaults for the project name, license, and other values. (If you omit the `-y` flag, npm prompts you to input these values.)

Upon initializing your project, npm creates a file called `package.json`. This file holds the npm metadata for your project in JSON format.

Listing 10.1 Generated `package.json` file

```

{
  "name": "pattern-library",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "Keith J. Grant",
  "license": "ISC"
}

```

Annotations for Listing 10.1:

- Version number**: Points to the `"version": "1.0.0"` field.
- Brief name of your npm project**: Points to the `"name": "pattern-library"` field.
- Longer description of the project can be filled in here.**: Points to the `"description": ""` field.

With your package initialized, you can install KSS as a *dependency*. Enter `npm install --save-dev kss` in the terminal. This does two things: It creates a `node_modules` directory in your project, where KSS and its dependencies are installed, and it adds “kss” into a list of development dependencies (`devDependencies`) in the `package.json` file.

ADD THE KSS CONFIGURATION

KSS will need a configuration file. This file gives KSS the paths to some directories and files that it’ll use to build the pattern library. Create a file in your project directory called `kss-config.json`. Copy the following listing into this file.

Listing 10.2 KSS configuration file (`kss-config.json`)

```

{
  "title": "My pattern library",
  "source": [
    "./css"
  ],
  "destination": "docs/",
  "css": [
    "../css/styles.css"
  ],
  "js": [
    "../js/docs.js"
  ]
}

```

Annotations for Listing 10.2:

- Path to CSS source files directory (which KSS will scan)**: Points to the `"source": ["css"]` field.
- Path to where the generated pattern library will be written**: Points to the `"destination": "docs/"` field.
- Path to stylesheet (relative to destination directory)**: Points to the `"css": ["css/styles.css"]` field.
- Path to any javascript (relative to destination directory)**: Points to the `"js": ["js/docs.js"]` field.

The source path tells KSS where to find your CSS source files, which it scans for documentation comments. It then uses the comments to produce pages of the pattern library in the destination directory.

The files listed in the `css` and `js` keys will each be added to the pages of the pattern library. I've configured these each for a `css` and `js` directory, respectively. Go ahead and create these directories and source files therein (`css/styles.css` and `js/docs.js`). Leave the files empty for now; you'll add to them shortly.

NOTE In our case, the stylesheet listed in the `css` key is in the same directory as the source directory. When you use a preprocessor, such as SASS or Less, the source directory should point to your SASS or Less files, but the `css` key should reference the compiled CSS stylesheet.

As a last bit of configuration, you'll add a command to the `package.json` file that tells KSS to build the pattern library. Add a new item to the `scripts` section of your `package.json` file so it matches the following listing.

Listing 10.3 Adding a build script to `package.json`

```
"scripts": {
  "build": "kss --config kss-config.json",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

← Defines a build command

This adds a build command to your package. Now, running `npm run build` in the terminal will tell NPM to run KSS (from the `node_modules` directory), passing it the path to the KSS configuration file you created. Run `npm run build` now and you'll see an error: "Error: No KSS documentation discovered in source files." KSS is looking for some documentation. Let's give it some.

10.1.2 Writing KSS documentation

You'll add some modules from chapter 9 to your pattern library. The first of these will be the `media` object, as shown in figure 10.2. When KSS builds this page, it'll add `Media` to the table of contents on the left and will render the documentation on the right.

KSS looks for comments in your stylesheet that follow a particular pattern. This includes a title (usually the name of the module), some descriptive text, some example HTML, and a `styleguide` annotation indicating where the module belongs in the table of contents. A blank line must separate each of these items for KSS to distinguish them. Strictly speaking, the final `styleguide` annotation is the only piece KSS requires, but you should typically include the rest as well.

My pattern library

0 Overview

1 Media

Section 1

Media

Displays an image on the left and body content on the right.

Example



Strength

Strength training is an important part of injury prevention. Focus on your core— especially your abs and glutes.

```

<div class="media">
  
  <div class="media_body">
    <h4>Strength</h4>
    <p>
      Strength training is an important part of
      injury prevention. Focus on your core&mdash;
      especially your abs and glutes.
    </p>
  </div>
</div>

```

Source: styles.css, line 15

Figure 10.2 Documentation for the Media module

Add the code shown in the next listing to your stylesheet at `css/styles.css`. This includes some base styles and the Media module. Above the module styles is the CSS comment block for KSS.

Listing 10.4 Media object with KSS documentation comment

```

:root {
  box-sizing: border-box;
}

*,
*::before,
*::after {
  box-sizing: inherit;
}

body {
  font-family: Helvetica, Arial, sans-serif;
}

```

```

/*
Media
  ← | Title (name of the module)

Displays an image on the left and body content
on the right.
  ← | Description of the module and its use

Markup:
<div class="media">
  
  <div class="media__body">
    <h4>Strength</h4>
    <p>
      Strength training is an important part of
      injury prevention. Focus on your core&mdash;
      especially your abs and glutes.
    </p>
  </div>
</div>
  ← | HTML example illustrating use of the module

Styleguide Media
*/
<div class="media">
  .media {
    padding: 1.5em;
    background-color: #eee;
    border-radius: 0.5em;
  }
  .media::after {
    content: "";
    display: block;
    clear: both;
  }

  .media__image {
    float: left;
    margin-right: 1.5em;
  }

  .media__body {
    overflow: auto;
    margin-top: 0;
  }

  .media__body > h4 {
    margin-top: 0;
  }
  ← | Styleguide annotation, adding this to the table of contents as Media

```

Now, run `npm run build` in your terminal. KSS generates a docs directory that includes a `section-media.html` file. Open this page in your browser to see the pattern library. KSS also logs a warning: “No homepage content found in homepage.md.” I’ll show

you how to fix this in a bit. For now, let's take a closer look at the parts of the documentation comment. The first few lines look like this:

```
/*
Media

Displays an image on the left and body content
on the right.
```

The first line of the comment defines the title (Media) for this section of the documentation, and then some text describing the purpose of the module. This description may be written in *markdown* format, so you can add formatting to it as you wish. The description can be multiple paragraphs.



markdown—A common text format that supports annotations for basic formatting. Surround text in asterisks to make it italic; surround text with back-ticks (`) to format it as code. See <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet> for a complete reference.

When you create a module, use the description to convey to other developers anything they'll need to know about using it. Sometimes, a simple sentence is enough. Sometimes, you'll need to indicate that the module requires JavaScript or is intended to be used in conjunction with another module. This is your documentation on the use of your stylesheet.

After the description is a Markup: annotation. This is followed by a block of HTML code that illustrates the use of the module. KSS renders this HTML into the pattern library so the reader can preview it. Then, it displays the HTML in readable format so the reader can copy it:

```
Markup:
<div class="media">
  
  <div class="media__body">
    <h4>Strength</h4>
    <p>
      Strength training is an important part of
      injury prevention. Focus on your core&mdash;
      especially your abs and glutes.
    </p>
  </div>
</div>
```

The exact text and images used in the example are not important, as long as they illustrate to the developer how the module works. In this case, I've used a generic

placeholder image from the website <http://placeholder.it>. When the developer uses this module, they can add the content they need.

WARNING It's important, however, that there are no empty lines in the middle of the HTML as this indicates to KSS that the markup section is complete.

The final line of the KSS comment must include the Styleguide annotation, which is followed by the label for the table of contents (in this case, Media):

```
Styleguide Media
*/
```

This must be the last line of the comment. Without it, KSS will ignore the entire comment block.

When you update a stylesheet, update the documentation to match. Having the documentation right there in the source code makes this easy to do. When you add a new module, add a documentation block with it. After you have made changes, run `npm run build` again to generate a fresh copy of the pattern library.

WARNING KSS doesn't delete old pages when generating new ones. If you rename or move a part of the documentation in your source code, the corresponding file in the docs directory will remain in place, alongside the new one. When you refresh your browser, be sure you're not reloading the old file.

Because the pattern library “lives” with the styles it documents, any developer with access to the stylesheet will have access to its documentation. You may also want to host the pattern library somewhere online, where your development team can access it.

10.1.3 Documenting module variants

Let's document another module (listing 10.5). You'll bring in the Button module from the last chapter. This module offered several variants: two alternate colors and two alternate sizes. KSS provides a way for you to indicate multiple variants, rendering each one in the pattern library. This will look like figure 10.3.

The documentation comment for this module will be similar to the last, but you'll add a new section after the markup to indicate each of the modifiers (listing 10.5). This will be a list of the modifier classes, each followed by a hyphen and their description. You'll also add the annotation `{{modifier_class}}` to the markup example, indicating where the modifier classes belong.

Section 1

Buttons

Buttons are available in a number of sizes and colors. You may mix and match any size with any color.

Examples

Default styling

`click here`

.button--success A green success button

`click here`

.button--danger A red danger button

`click here`

.button--small A small button

`click here`

.button--large A large button

`click here`

`<button class="button [modifier class]">click here</button>`

Figure 10.3 Button module with variants

Listing 10.5 Button module and documentation

```

/*
Buttons

Buttons are available in a number of sizes and
colors. You may mix and match any size with any
color.

Markup:
<button class="button {{modifier_class}}">
  click here
</button>

.button--success - A green success button
.button--danger  - A red danger button
.button--small   - A small button
.button--large   - A large button

```

← Indicates where the modifier classes are used

Lists available modifier classes

```
Styleguide Buttons
*/
.button {
  padding: 1em 1.25em;
  border: 1px solid #265559;
  border-radius: 0.2em;
  background-color: transparent;
  font-size: 0.8rem;
  color: #333;
  font-weight: bold;
}

.button--success {
  border-color: #cfe8c9;
  color: #fff;
  background-color: #2f5926;
}

.button--danger {
  border-color: #e8c9c9;
  color: #fff;
  background-color: #a92323;
}

.button--small {
  font-size: 0.8rem;
}

.button--large {
  font-size: 1.2rem;
}
```

KSS scans the list of modifier classes you’ve defined, rendering each one into the pattern library. The `{{modifier_class}}` tells it where to place the classes. (If you’re familiar with handlebars templates, this syntax probably looks familiar. This is what KSS uses behind the scenes to render the module.) Run `npm run build` to re-build your pattern library and view the documentation in your browser.

TIP Re-running KSS every time you make a change can get tedious. If you’re using a task runner such as Gulp for your projects, I suggest configuring a task that watches for changes and automatically re-runs KSS for you. Most task runners have a plugin or other mechanism to do this.

You should now have three items in your pattern library’s table of contents (`docs/index.html`): Overview, Buttons, and Media. The latter two each link to the parts of documentation you’ve written. The Overview link is broken as you haven’t created a home page yet. This is the cause of the “No homepage content” warning.

10.1.4 *Creating an overview page*

Let's add a home page to the pattern library. Inside the `css` directory, create a new file at `css/homepage.md`. This will be a file in markdown that serves as an introduction to the pattern library. Copy this listing into the file.

Listing 10.6 Home page markdown

```
# Pattern library 
This is a collection of all the modules in our
stylesheet. You may use any of these modules when
constructing a page.
```

Now run `npm run build` and the warning about home page content should be gone. If you open `docs/index.html` in your browser, you'll see this content rendered.

In your projects, use this page to serve as an introduction to your pattern library. You can provide instruction on how to include the stylesheet or stylesheets on the page, how to include the correct web fonts (see chapter 13), or anything else to help developers get familiar with using your stylesheets.

Because you're opening the pattern library files directly from disc, you may notice the Overview link in the table of contents still doesn't work. This is because KSS links it to the url `./` rather than to `index.html`. To make this work, you'll need to serve the pattern library via an HTTP server so the `./` url will resolve to `index.html` in the browser. I'll leave this for you to do, depending on the toolset you're most familiar with. If you're unsure where to start, try the npm package `http-server` (<https://www.npmjs.com/package/http-server>).

10.1.5 *Documenting modules that require JavaScript*

Some modules are designed to work with the help of JavaScript. In these cases, it's often helpful to add a bit of bare bones JavaScript to the page to demonstrate the module's behavior. You don't necessarily need to add a full functioning JavaScript library to the pattern library in order to do this. Most of the time, you'll only need enough to toggle the various state classes. You've already added the configuration to your `kss-config.json` file that adds a JavaScript file to the page:

```
"js": [
  "../js/docs.js"
]
```

KSS will add the scripts listed in this `js` array to the page for you. You can add code to these scripts that provides minimal functionality to the modules. To demonstrate this, you'll add the Dropdown module (chapter 9) to your stylesheet, along with some documentation (listing 10.7). You'll also add some JavaScript so that clicking the Toggle button opens and closes the dropdown. Then the module will work inside the pattern library to demonstrate the module's intended functionality (figure 10.4).

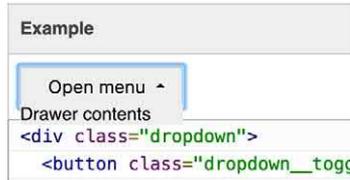


Figure 10.4 Working dropdown menu inside the pattern library. (Note the Drawer contents are unstyled because we expect a separate module to style the menu.)

Begin by adding the styles and documentation in listing 10.7 to your stylesheet. It's also important to give some direction about how the JavaScript will need to work. Developers will be using this to build the website or web application. They'll need enough information to be able to do it correctly. Add this code to your CSS.

Listing 10.7 Dropdown module and documentation

```

/*
Dropdown

A dropdown menu. Clicking the toggle button opens
and closes the drawer.

Use JavaScript to toggle the `is-open` class in
order to open and close the dropdown.

Markup:
<div class="dropdown">
  <button class="dropdown__toggle">Open menu</button>
  <div class="dropdown__drawer">
    Drawer contents
  </div>
</div>

Styleguide Dropdown
*/
.dropdown {
  display: inline-block;
  position: relative;
}

.dropdown__toggle {
  padding: 0.5em 2em 0.5em 1.5em;
  border: 1px solid #ccc;
  font-size: 1rem;
  background-color: #eee;
}

.dropdown__toggle::after {
  content: "";
  position: absolute;
  right: 1em;
  top: 1em;
  border: 0.3em solid;

```

Provides instructions indicating how the developer will need to use JavaScript for this module

Markup example

Dropdown module rules (copied from chapter 9)

```

border-color: black transparent transparent;
}

.dropdown__drawer {
display: none;
position: absolute;
left: 0;
top: 2.1em;
min-width: 100%;
background-color: #eee;
}

.dropdown.is-open .dropdown__toggle::after {
top: 0.7em;
border-color: transparent transparent black;
}

.dropdown.is-open .dropdown__drawer {
display: block;
}

```

Running `npm run build` builds this documentation, but at this point, it's static. Let's add JavaScript to `js/docs.js` to bring it to life. Add this listing to that file.

Listing 10.8 Minimal JavaScript to demonstrate the module

```

(function () {
  var dropdowns = document.querySelectorAll('.dropdown__toggle');
  Array.prototype.forEach.call(dropdowns, function(dropdown) {
    dropdown.addEventListener('click', function (event) {
      event.target.parentNode.classList.toggle('is-open');
    });
  });
})();

```

Gets all instances of the `dropdown__toggle` button

Adds a click event listener to each instance

Toggles the `is-open` class on the dropdown element

This script toggles the `is-open` class on the dropdown any time the Toggle button is clicked. A full implementation on your website will need more code to deal with any timing delays or for closing the menu if one clicks elsewhere on the page. Again, in the pattern library, the code can be minimal; but, you'll need to get the open and closed states styled correctly. Once that's done, you (or another developer) can focus on the problem of getting the finer points of the JavaScript exactly right, outside the pattern library.

10.1.6 Organizing the pattern library into sections

You can continue to add the modules from chapter 9 into your stylesheet, entering documentation as necessary. I won't walk you through each and every one as you now have a basic understanding of the process.

The one final thing you'll need to be able to do is to organize your pattern library. The menu in figure 10.4 is fine for now, with only a few items in it. But as your projects begin to grow in size, it'll make sense to categorize your modules so they are easier to navigate.

Let's add documentation for the utility classes. Each one will need to be explained and demonstrated individually, so it makes sense to group them together. In listing 10.9, you'll create a new section called Utilities, adding the utility classes each into a subsection within that to render the sections shown in figure 10.5.

To create subsections, use a period in the styleguide annotation. You'll use annotations such as this: `Styleguide Utilities.clearfix`. This puts the block of documentation into a `clearfix` subsection within a `Utilities` section.

NOTE KSS supports sections up to three levels deep (for example, `Utilities.alignment.text-center`).

Add the following listing to your stylesheet. This includes three utility classes (`text-center`, `float-left`, and `clearfix`) and their documentation comments. I've also included a `Weight` annotation, which controls the order of the sections.

Listing 10.9 Grouping documentation in the same category

```

/*
Text center

Center text within a block by applying `text-center`

Markup:
<p class="text-center">Centered text</p>

Weight: 1

Styleguide Utilities.text-center
*/
.text-center {
  text-align: center !important;
}

/*
Float left

Float an element to the left with `float-left`

Weight: 3

```

← Uses a dot notation to place each documentation block into the same group

← Uses the `Weight` annotation to control the order of sections

My pattern library

- 0 Overview
- 1 Buttons
- 2 Dropdown
- 3 Media
- 4 Menu
- 5 Message
- 6 Utilities

- 6.1 Text center
- 6.2 Clearfix
- 6.3 Float left

Figure 10.5 Three subsections within the Utilities section

```
Styleguide Utilities.float-left
*/
.float-left {
  float: left;
}

/*
Clearfix
```

Add the class `clearfix` to an element to force it to contain its floated contents

```
Markup:
<div class="clearfix">
  <span class="float-left">floated</span>
</div>
```

```
Weight: 2
```

```
Styleguide Utilities.clearfix
*/
.clearfix::before,
.clearfix::after {
  content: " ";
  display: table;
}

.clearfix::after {
  clear: both;
}
```

Uses a dot notation to place each documentation block into the same group

Uses the Weight annotation to control the order of sections

By categorizing each of these utility classes into the same main category, they'll all be grouped together. Now when you rebuild the pattern library, there will be an item in the table of contents called Utilities. Click it to view a page with all the subsections listed.

WARNING The Styleguide annotation is case-sensitive. When placing multiple items within the same section, be sure to capitalize those consistently or KSS will create separate sections (for example, one called “Utilities” and another “utilities”).

By default, sections of a KSS pattern library are ordered alphabetically, as are subsections within a section. You can change this by using the `Weight` annotation. KSS orders sections according to their weight, with higher weights closer to the bottom. You can indicate weights on a top-level section to control its position among other top-level sections, or (as in the example) to control the order of subsections within their section.

You're now familiar with all the essential features of KSS. If you want to dive deeper on your own, you can learn to take a little more control over the look and feel of the pattern library itself. You can customize its internal stylesheet or the template it uses to

build the pattern library pages. For more information, see the documentation at <https://github.com/kss-node/kss-node>.

10.2 Shifting the way you build CSS

Pattern libraries aren't necessary for small projects, but with large projects, they'll prove invaluable. If you develop for a website with hundreds or thousands of pages, you cannot possibly style them all one at a time. But by building reusable modules and documenting them in a single place, you can provide a toolkit with which thousands of pages can be constructed.

If you work on a large web application with a dozen other developers, you cannot possibly each style your own components without running into class name conflicts and a lot of duplicated implementations of the same UIs. But with a pattern library, developers can find each other's styles, reuse them, and systematically name the modules so the class names don't conflict.

Content editors and developers who use your pattern library don't even need to know CSS—they only need to have a basic understanding of HTML. They can copy the patterns you document and place them in their page wherever they want. Modular CSS is the key to scaling your CSS, and a pattern library is a means of keeping those modules organized.

10.2.1 Using a CSS First workflow

Using a pattern library is a paradigm shift from the typical approach to CSS. Instead of taking an HTML page and then styling it, you build modular styles and then piece together a web page using those modules. This is an approach I call *CSS First* development. Instead of writing your HTML first, you begin with the CSS. You can, and should, develop your CSS within the confines of the pattern library before putting those styles to use in your project, so your development process will look something like this:

- 1 When building a page, have a sketch or mockup or some general idea what that page should look like.
- 2 Go to the pattern library. Look for existing modules that provide what you need for your page, and use them. Start from the outside (main page layout and containers) and work your way in. If you can construct your entire page using existing modules, do it. You won't need to write any new CSS.
- 3 Occasionally, you'll find you need something the pattern library doesn't provide. This will happen a lot early in the life of the project, but much less later on. You'll need to build a new module or modules, or a new variant for an existing module. Set aside the page you're working on, and build it within the pattern library. Document it and make sure it looks and behaves like you expect.
- 4 Go back to your page and, using the new stylesheet, add the new module(s) to your page.

This approach has several benefits. First, it helps provide a more consistent interface for your site. It encourages you to reuse existing styles rather than rolling out new ones. For example, instead of ten different pages on your site with ten different list styles, you'll tend toward reusing the same few types of lists. It forces you to stop and think each time whether you need a new style or whether one you already have is sufficient.

Second, when you develop a module within the confines of the pattern library, you'll be able to focus on that problem in isolation. You can remove yourself from the bigger picture of the particular web page and focus on the singular task of styling a module. Instead of solving one single problem on one single page, it'll be easier to think about where else the new module might be used. You'll create a more general, more reusable solution.

Third, this approach will also allow a few members of your team to specialize in CSS. A developer who is less adept at it can hand off a piece of work to one who is more experienced. Once that CSS-minded developer finishes the module, they can send a link to the other developer, pointing to the module in the pattern library.

Finally, this approach will ensure your documentation is up-to-date. The pages of your pattern library are where you test changes to the CSS, which means they always demonstrate the current, correct behavior. When you edit the CSS, the documentation is right there in a comment block. This makes it trivial to keep the documentation current as you make changes. (I'll talk more about editing existing modules in a bit.)

Developers often ask how they can write HTML that is easy to style. I believe this is the wrong question. Instead, we should ask how we can write styles that can be reused in any number of pages. We should write CSS first; well-structured HTML will follow.

10.2.2 *Using a pattern library as an API*

When you use a pattern library, you're documenting an *API* for interfacing with your CSS. Each module comes with some class names and a small bit of DOM structure. As long as the relevant portion of HTML follows this structure, the stylesheet will style it correctly (figure 10.6).



Figure 10.6 The class names and HTML structure are an API



API—Application Programming Interface. A set of subroutine definitions that describe how to use or interact with a system. Traditionally, this includes method names and parameters (in the case of a programming language) or URLs and query parameters (in the case of an HTTP API). I use the phrase in regards to modular CSS to illustrate that the class names and HTML elements are the way the HTML interfaces with the styles.

The markup example in each module illustrates a sort of contract your CSS makes with the HTML. It shows how the HTML should interface with the CSS.

When you build your modules, this API is the most important part because it's the hardest thing to change later. The HTML is free to change: it can change the contents within each element. In some cases, it can add, remove, or even re-arrange the order of the DOM elements within the module (be sure to indicate clearly in your documentation if elements are optional or if things can be re-arranged). And, the HTML can stop using a module entirely, changing instead to a different module.

Likewise, the CSS can change as long as it still honors this API. You can make small edits, such as increasing padding or adjusting a color or fixing any bugs that arise. Or, you can make large edits, such as reworking a media object to use flexbox instead of floats, or redesigning a module to stack vertically instead of horizontally. As long as the key pieces of the API (class names and DOM structure) remain unaltered, you are free to edit the CSS however you want.

Be aware that making these edits can affect many parts of your website. But as long as the HTML follows the API instructions, these changes will be according to plan. If you want to change the look of all dropdown menus throughout your site, you can. Because all dropdown menus on your site use the same module (and the same API), the changes will be consistent.

EDITING AN EXISTING MODULE

To illustrate, let's assume a hypothetical scenario where you want to make a change to the way the Media module works. Instead of one image, you find you need it to support two images, one on either side of the content, as in figure 10.7.

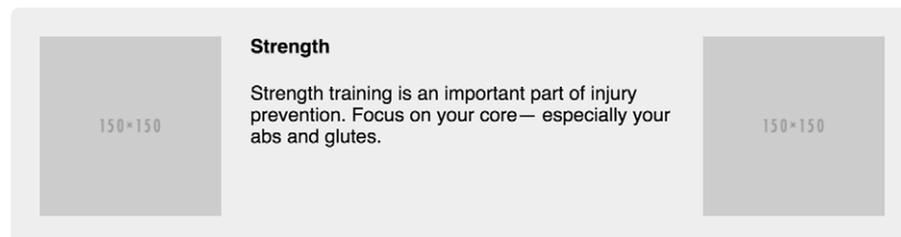


Figure 10.7 A hypothetical media object with two images

This requires some changes to the CSS. As long as you make sure your changes still honor the API (that is, existing media objects on your site continue to work as expected with only one image), you're free to change the styles. You'll do this by reworking the module to use flexbox. Let's make these changes.

First, you'll need to add to the example markup in the comment block. Keep the old example there, so you can test that it remains unchanged after you make your edits. But you'll add a second example to the markup to test the new behavior as well. Update the documentation comment to match the next listing.

Listing 10.10 Adding a new media example to the documentation

```

/*
Media
    Displays images and/or body content beside one
    another.

Markup:
<div class="media">
  
  <div class="media__body">
    <h4>Strength</h4>
    <p>
      Strength training is an important part of
      injury prevention. Focus on your core&mdash;
      especially your abs and glutes.
    </p>
  </div>
</div>
<div class="media">
  
  <div class="media__body">
    <h4>Strength</h4>
    <p>
      Strength training is an important part of
      injury prevention. Focus on your core&mdash;
      especially your abs and glutes.
    </p>
  </div>
  
</div>

Styleguide Media
*/

```

Updates description to allow for multiple images

Keeps original markup example in place

Adds new example with two images

This listing gives you two instances of the module in your pattern library. Rebuild the pattern library to see it render. Before you make changes to the CSS, you can see that one works and the other doesn't. You can then make changes (listing 10.11) until they both work. Afterward, you'll have the result shown in figure 10.8.



Figure 10.8 Both types of media objects illustrated in the pattern library.

The pattern library now serves as a backstop. It tells you if your changes would break the existing media objects on your site and acts as a test for the validity of the code.

You can now refactor the CSS to account for the new scenario. Make these changes to your stylesheet so the second example works, ensuring the first example doesn't break in the process.

Listing 10.11 Media module reworked to use flexbox

```
.media {
  display: flex;
  align-items: flex-start;
  padding: 1.5em;
  background-color: #eee;
  border-radius: 0.5em;
}

.media > * + * {
  margin-left: 1.5em;
}

.media__body {
  margin-top: 0;
}

.media__body > h4 {
  margin-top: 0;
}
```

← Changes the container to a flex container; `media__image` and `media__body` will become flex items

← Aligns items on the top rather than stretching to fill and prevents image distortion

← Removes the image's right margin and replaces it with a general margin between all flex items

Run `npm run build` and open the page in the pattern library. You'll see that your changes were successful. The media object is now a bit more versatile. And, because you still honor the original API of the module, you can be confident your changes didn't break things on the website.

Without modular styles and a pattern library, editing CSS can wreak havoc on a website; you have no idea exactly how the HTML might be structured everywhere and whether the selectors will still target the correct elements. But with a stable, documented API, edits can be painless, even satisfying.

USING SEMVER AND REFACTORING CODE

Sometimes, you won't be able to make the changes you want without modifying the API. This is okay. It'll mean a little more work, but it's doable. You could make the changes you want, then go through your entire site or application and update every instance of the HTML to match the new API. But often I find the best course of action is to deprecate the module (indicating so in the documentation), and create an entirely new module for the new functionality I need. This way, the old module continues working where it's used, but I can start migrating to the new module while both are supported.

To help facilitate this, I find it's highly beneficial to version my CSS using a three-numbered *semver*. When the version number changes, it communicates to developers the nature of those changes.



semver—Short for Semantic Versioning, a system for versioning software packages using three numbers, each separated by a period (for example, 1.4.2). The three numbers stand for the major, minor, and patch versions, respectively. See <http://semver.org/> for more information.

When I make small adjustments, such as bug fixes, I increment the patch version number (from 1.4.2 to 1.4.3, for example). When I add a new module or new functionality that does not break the API, or when I mark a module as deprecated, I increment the minor version number, resetting the patch version to 0 (for example, 1.4.2 to 1.5.0). Then, on rare occasions, I go through the stylesheet and delete deprecated modules, bumping to the next major version (for example, 1.4.2 to 2.0.0). I also create a major version release when I make substantial design changes (such as a site redesign), even if the API remains intact.

Practically speaking, there are a number of ways you could go about this versioning. This depends on the nature of the project where you use the styles. If you package the CSS in a NodeJS module or Ruby Gem, for instance, use the versioning built into these systems. Or, if you're hosting your CSS statically on a server, include the version number in the URL (`http://example.com/css/1.4.2/styles.css`) and host multiple versions simultaneously.

This way, the project can be configured to use whichever version of the CSS it needs. You can release a version 3.0.0 with breaking changes, but the web application can continue using the old version until developers are able to go through and update the HTML wherever it uses deprecated modules. Changes you make to your CSS won't break the application until the application deliberately upgrades to a new version of the stylesheet.

Your pattern library documents the use of the stylesheet, but the authors of the HTML are in control of whether they use the styles and which version they use. The HTML and CSS are decoupled. The CSS must be developed first before it can be used by the HTML, but the HTML is in control when it comes to upgrading to a new stylesheet. This is the benefit of CSS First development.

These decisions can't happen in a vacuum. You'll need to communicate with other developers on your team when you want to deprecate or delete modules. You'll need their input in regards to which modules are still valuable and which are no longer needed.

Bootstrap, Foundation, and other frameworks

You may be familiar with one or more CSS frameworks that provide a pre-packaged set of styles. These usually include styles for buttons, forms, menus, and a grid system of some sort. Popular frameworks include Bootstrap (<http://getbootstrap.com/>), Foundation (<http://foundation.zurb.com/>), and Pure (<http://purecss.io/>). There are countless others as well. Some of these are robust libraries with dozens of modules; others are minimal and provide only bare essentials.

As you go about building your pattern library, you may start to feel like you're building your own framework along the same lines. That's exactly what you're doing! This is why these frameworks are successful—each one is a pattern library. They consist of CSS built with attention given to making styles reusable in many contexts. Some follow the principles of modular CSS better than others, but they all follow them to some degree. And, they're always versioned.

The difference between these frameworks and your own pattern library is that frameworks are general-purpose. In your pattern library, you're able to make modules tailored specifically to your project, and you're able to precisely match a brand-specific look and feel. You can create two different types of a Tile module if you need to, and you can adapt more quickly when you need to.

Developers often ask me whether I think they should use a framework like Bootstrap. My answer is both yes and no.

Frameworks are helpful for getting a project off the ground quickly. With almost no work, you can have styled buttons, tiles, and dropdown menus. But, in my experience, they never provide all the modules you'll need. Except for small projects, you'll always have to add more modules of your own. They also provide a lot of modules you probably won't need.

(continued)

If you want to use a framework you're familiar with, my suggestion is to take only the pieces of it you need and leave the rest. Don't just stick a `bootstrap.css` file onto your page. Instead, copy only the modules you want into your own stylesheet (assuming the framework's license allows this). Take those pieces of CSS and make them your own.

When you add a framework to the page before your own stylesheet, you'll find yourself writing a bunch of styles to override and augment the framework. If you instead bring the framework's styles into your stylesheet, you'll be able to modify them directly. This will keep the page's CSS leaner and easier to keep track of.

Instead of blindly using a framework, take on the mindset of a framework. Imagine your pattern library is a general-purpose library for use by unknown third parties. This will help you keep your styles reusable and provide a means for making changes in the future with fewer breakages on the page.

Often, CSS is an “additive-only” language. Developers are afraid to edit or delete any existing styles because they have no way of knowing all the ramifications of those changes. They only modify CSS, adding more to the end of the stylesheet, overriding earlier rules and ever increasing selector specificity until the stylesheet is an unmaintainable tangle of code.

By taking care to organize your CSS in a way that is modular and maintaining a pattern library for it, you don't have to fall into this trap. You always know where the styles for a module reside. Each module is responsible for one thing. And the pattern library helps developers keep tabs on everything going on within the stylesheet.

Summary

- Use a tool such as KSS to document and inventory your modules.
- Use a pattern library to document markup examples, module variants, and JavaScript for your modules.
- Develop your modules “CSS First.”
- Consider the API your CSS defines, taking care to never break it unpredictably.
- Version your CSS using semver.
- Don't blindly add a CSS framework to your page; selectively take only the pieces you need.

Part 4

Advanced topics

A polished UI is important. Users tend to trust a professional-looking application, and they may be inclined to spend more time using a site if it's aesthetically pleasing. In the final six chapters, we'll look at important considerations for design. These are small details that can have a big impact on the look and feel of your site.

11

Backgrounds, shadows, and blend modes

This chapter covers

- Linear and radial gradients
- Box shadows and text shadows
- Sizing and positioning background images
- Using blend modes to combine backgrounds and content

We've covered a lot of ground by this point. You've deepened your understanding of the fundamental workings of CSS. You've learned multiple aspects of layout. And, you've taken time to make sure your code is organized and maintainable. We've covered the essentials needed to build a site from the ground up. You could take this knowledge, apply it to your projects, and be in fairly good shape. But don't stop there.

The difference between a site that looks good and one that looks great is attention to detail. After you lay out and style a component of your page, train yourself to slow down and look at it with a critical eye. Does it look better if you increase, or decrease, paddings? Adjust the colors a bit—do they look better a little darker or little lighter, or a little less vivid? If you're working from a detailed mockup by the designer, does your implementation match everything as closely

as possible? Your designer spent a lot of time on those details. Make sure you're doing the design justice.

These details are where the artistic portion of CSS comes into play. If you're like many developers, you may not consider yourself a designer or an artist. But if you're working with CSS, you'll need to play the role of one from time to time. These concerns will be the main focus in part 4 of this book.

These final chapters are about the details—things you can do to add a special something to the page. I'll teach you some design tips, but don't worry; it won't be all subjective art. I'll focus on concrete rules. I'll show you how to make use of color, space, typography, and animation. If you've wondered how to make your page look not only functional but also visually appealing, these chapters will give you the tools to do so.

In this chapter, I'll show you techniques for adding visual interest to the page. Consider the button shown in figure 11.1 for example. This uses two effects that give it the illusion of depth: a background gradient and a drop shadow. The background color transitions from a medium blue at the top (color #57b) to a darker one at the bottom (#148). You might not even consciously notice it, but this, in combination with the shadow along the bottom and right edges, adds an illusion of depth to the button.



Figure 11.1 A button with a gradient background and a shadow effect

This chapter covers how gradients and drop shadows work and looks at some practical uses for them. Then we'll take a look at a fun effect called *blend modes* that you can use to blend multiple background images or colors together in various ways.

You'll not often need to add all these things to your page at the same time, so rather than build one large page, we'll be constructing multiple smaller examples. This will give you a number of tools you can choose from in your various projects.

11.1 Gradients

As you've been following along, we've used solid color backgrounds and a few background images in previous chapters. But there's a lot more to explore when it comes to the background property; it's, in fact, a shorthand for eight properties:

- `background-image`—Specifies an image from a file or a generated color gradient.
- `background-position`—Sets the initial position of the background image.
- `background-size`—Specifies how large to render the background image within the element.
- `background-repeat`—Determines whether to tile the image if necessary to fill the entire element.

- `background-origin`—Determines whether background positioning is relative to the element’s border-box, padding-box (initial value), or content-box.
- `background-clip`—Specifies whether the background should fill the element’s border-box (initial value), padding-box, or content-box.
- `background-attachment`—Specifies whether the background image will scroll up and down along with the element (the initial value), or if the image will be fixed in place in the viewport. Note that using the value `fixed` can have negative performance implications on the page.
- `background-color`—Specifies a solid background color. This will render behind any background image.

We’ll explore many of these properties throughout the chapter. For now, keep in mind that using the shorthand property (`background`) will set the values you specify while also resetting all the others to their initial value. For this reason, I tend to prefer using individual properties when I’m doing anything that requires more than a few of them.

The `background-image` property is particularly interesting. You’ve seen that this accepts a path to an image URL (`background-image: url(coffee-beans.jpg)` in chapter 8), but it can also accept a gradient function. For example, you can define a gradient that blends from white to blue, as in figure 11.2.



Figure 11.2 A white-to-blue linear gradient

Gradients are a useful effect. Let’s examine how they work before we get to some practical examples. To try out gradients, create a new page and stylesheet. Add the CSS shown in the following listing, which uses the `linear-gradient()` function to define the gradient.

Listing 11.1 A basic linear gradient

```
.fade {  
  height: 200px;
```

```
width: 400px;
background-image: linear-gradient(to right, white, blue);
```

← | Fades to the
right from
white to blue

The gradient is a background image, which by itself won't do anything to affect the size of the element. For the purposes of this example, I've set an explicit height and width on the element. The element is empty, so you'll need to force it to have some height in order to see the gradient.

The `linear-gradient` function has three parameters defining its behavior: angle, starting color, and ending color. The angle here is `to right`, meaning the gradient starts on the left edge of the element (where it's white) and blends evenly to the right edge (where it's blue). You can also use other color syntaxes, such as hex (`#0000ff`), RGB (`rgb(0, 0, 255)`), or the transparent keyword. Add the element from this listing to see the gradient on the page.

Listing 11.2 Element with a background gradient

```
<div class="fade"></div>
```

You can specify the angle of the gradient in several ways. In this example, you used `to right`, but you can also use `to top` or `to bottom`. You can even specify a corner such as `to bottom right`; in which case, the gradient will begin at the top-left corner of the element and blend to the bottom-right corner.

For more precise control of the angle, you can use more specific units, such as degrees. The value `0deg` points straight up (equivalent to `to top`); higher values move clockwise around the circle, so `90deg` points to the right, `180deg` points down, `360deg` points up again. Thus, this is equivalent to the previous example.

Listing 11.3 Gradient using degree (deg) units

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(90deg, white, blue);
}
```

← | 90deg value is
equivalent to the
value to right

Degrees are the most common unit, but there are a few others that you can use to indicate angle:

- `rad`—Indicates radians. One full circle is 2π , or roughly 6.2832 radians.
- `turn`—Indicates the number of full turns around the circle. One turn equals 360 degrees (`360deg`). Use decimal values to represent less than one full turn: `0.25turn`, for example, is equal to `90deg`.
- `grad`—Indicates gradians. One full circle is 400 gradians (`400grad`) and `100grad` equals `90deg`.

Go ahead and experiment with various values in the gradient to see how they affect it.

11.1.1 Using multiple color stops

Most of the time, your gradients will have two colors, transitioning from one color to the other. But, you can define a gradient with multiple colors, which are each called a *color stop*. Figure 11.3 shows a gradient with three color stops (red, white, then blue).

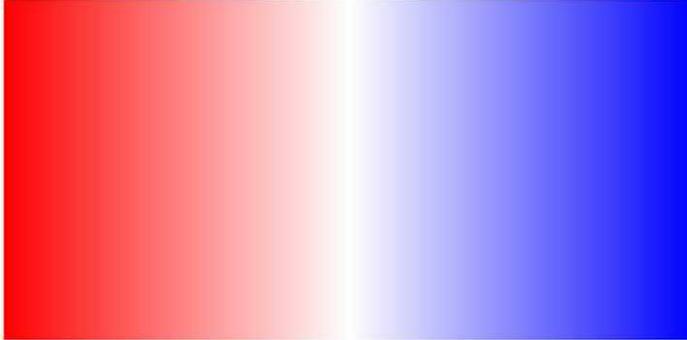


Figure 11.3 Gradient with three color stops (red to white to blue)

You can insert more color stops by adding more colors to the `linear-gradient()` function. To see this gradient in your page, update your stylesheet to match the next listing.

Listing 11.4 Linear gradient with multiple color stops

```
.fade {  
  height: 200px;  
  width: 400px;  
  background-image: linear-gradient(90deg, red, white, blue);  
}
```

← Specifies multiple color stops

A gradient can accept any number of color stops, each separated by a comma. It will automatically spread them out evenly. In this example, the gradient fades from red on the left edge (0%) to white in the center (50%) to blue on the right edge (100%). You can also explicitly set the position of these color stops in the gradient function. The gradient in listing 11.4 is equivalent to this one:

```
linear-gradient(90deg, red 0%, white 50%, blue 100%)
```

As you might imagine from this example, you can adjust the position of the color stops however you want; they don't need to be evenly spaced. They also don't need to be measured in percentages. Pixels, ems, and other length units are perfectly valid.

STRIPES

If you place two color stops at the same position, the gradient's color will instantly switch from one to the other, rather than a smooth transition. Figure 11.4 shows a

gradient that begins red, switches immediately to white, then switches immediately to blue. This creates the appearance of stripes.

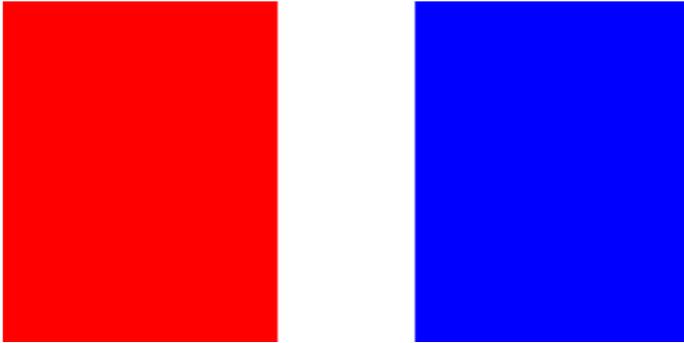


Figure 11.4 Gradient used to create stripes by placing two color stops at the same points

The code for the gradient is shown here. Notice that this gradient has four color stops, two of which are white.

Listing 11.5 Placing two color stops on the same points to create stripes

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(90deg,
    red 40%, white 40%,
    white 60%, blue 60%);
}
```

| **Color stops on the same points**

The first color stop is red at 40%, so the gradient is solid red from the left edge all the way to 40%. The next color stop is white, also at 40%, so the gradient makes a hard switch to white. This is followed by another white color stop at 60%, so the gradient is pure white from 40% to 60%. Then the final color stop, also at 60%, is blue. This makes a hard switch to blue, then remains blue all the way to the right edge.

REPEATING GRADIENTS

Even though the previous example is a bit contrived, the technique can be used for some interesting effects. In particular, it can be used with a slightly different gradient function, `repeating-linear-gradient()`. This works like the regular `linear-gradient()` function, except the pattern repeats. This can be used to produce stripes similar to a barber pole, which looks nice on progress bars (figure 11.5).



Figure 11.5 A repeating linear gradient for a striped bar

With repeating gradients, it's better to use a specific length rather than a percentage, because specified values determine the size of the pattern to repeat. The code for the striped bar is shown next. Update your stylesheet to match.

Listing 11.6 Creating a diagonally striped bar

```
.fade {
  height: 1em;
  width: 400px;
  background-image: repeating-linear-gradient(-45deg,
    #57b, #57b 10px, #148 10px, #148 20px);
  border-radius: 0.3em;
}
```

Stripes alternating
between light and
dark blue

Rather than coding up a whole gradient from scratch, I sometimes find it easier to start with a working example and modify it to meet my needs. You can find more examples at <https://css-tricks.com/stripes-css/>.

11.1.2 Using radial gradients

Another type of gradient is a radial gradient. Instead of starting at one end of the element and proceeding to the other end in a linear direction, radial gradients start at a single point and proceed outward in all directions. A basic example is shown in figure 11.6.



Figure 11.6 A radial gradient from white to blue

Edit your stylesheet to match the radial gradient code in this listing.

Listing 11.7 A basic radial gradient

```
.fade {
  height: 200px;
  width: 400px;
  background-image: radial-gradient(white, blue);
}
```

Blends from a
white center
to blue edges

By default, the gradient is centered in the element, transitioning evenly to its corners. It's elliptical in shape, matching the proportions of the element (that is, wider for wide elements, taller for tall elements).

Radial gradients support color stops, the same as linear gradients. You can provide multiple stops or explicitly define their position within the gradient using percentages or length units. You can also make the radial gradient a circle rather than an ellipse, or you can specify where the gradient should be centered. A `repeating-radial-gradient()` function repeats the pattern in concentric rings.

Most of these features are best explained by example, so I've included several with the corresponding code in figure 11.7. I encourage you to try these in your page, or to experiment with making your own.

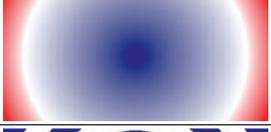
Value	Result
<code>radial-gradient(white, midnightblue)</code> Basic gradient (ellipse)	
<code>radial-gradient(circle, white, midnightblue)</code> Circle gradient	
<code>radial-gradient(3em at 25% 25%, white, midnightblue)</code> Sized 3 em, centered 25% from the left and top edges	
<code>radial-gradient(circle, midnightblue 0%, white 75%, red 100%)</code> Radial gradient with explicit color stop positions	
<code>repeating-radial-gradient(circle, midnightblue 0, midnightblue 1em, white 1em, white 2em)</code> Repeating gradient with stripes	

Figure 11.7 Examples of radial gradients

In the real world, I find I rarely need to do anything complex with a radial gradient as the most basic form tends to meet my needs. If you want to dive deeper into how this works, visit the MDN documentation at <https://developer.mozilla.org/en-US/docs/Web/CSS/radial-gradient>.

11.2.1 Defining depth with gradients and shadows

Let's use gradients and shadows to build a button as shown in figure 11.10. A top-to-bottom gradient gives it a curved, 3-D appearance. The shadow enhances this effect. In this example, you'll also use the `:active` pseudo-class to create an alternate type of shading when the button is depressed.



Figure 11.10 A button with gradient and shadow. Active (clicked) styling is shown on the right.

The gradient here is subtle. You might not notice it immediately, but it gives the button a slight rounded appearance. The shadow has some blur, making it appear more natural. When the button is clicked, the shadow is removed and, instead, an inset shadow appears inside the borders of the button. This gives it the illusion of being depressed, as if the user physically pressed it on the page. Upon releasing the mouse button, the button returns to its original state. This is done using the button's `:active` state.

Start a new page and a new stylesheet for this button. Add the button markup.

Listing 11.8 Button markup

```
<button class="button">Sign up now</button>
```

Next, add the styles from the following listing. These override user agent styles for font size and border, as well as apply some basic sizing and a gradient background with a box shadow.

Listing 11.9 Button styles with a gradient and shadow

```
.button {
  padding: 1em;
  border: 0;
  font-size: 0.8rem;
  color: white;
  border-radius: 0.5em;
  background-image: linear-gradient(to bottom, #57b, #148);
  box-shadow: 0.1em 0.1em 0.5em #124;
}

.button:active {
  box-shadow: inset 0 0 0.5em #124,
             inset 0 0.5em 1em rgba(0,0,0,0.4);
}
```

Gradient from light blue to medium blue
Dark blue shadow with a 0.5 em blur
Two inset box shadows

The `background-image` is a gradient between two similar colors of blue. The box shadow is not offset far, just 0.1 em right and down, with a moderate blur of 0.5 em. The larger a shadow's offset, the further it "lifts" the image from the page, making the image seem deeper. In the active state, the box shadow changes.

I've done two new things here. Instead of a normal box shadow, I've added the `inset` keyword. This makes the shadow appear inside the border of the element, rather than outside. I've also added more than one shadow definition, separating them with a comma. Multiple shadows can be added in this way.

The first inset shadow (`inset 0 0 0.5em #124`) has offsets of zero and a slight blur. This adds a ring of shade inside the edges of the element. The second (`inset 0 0.5em 1em rgba(0,0,0,0.4)`) has a bit of vertical offset, making the shadow more prevalent along the top of the button. The RGBA color definition defines a semi-transparent black. I encourage you to experiment with these values to see how they affect the final rendering.

NOTE In Chrome, you'll notice a light blue glow around the button when you click it. This is an outline applied by the user agent in the `:focus` state. You can remove this with `.button:focus { outline: none; }`. I suggest any time you remove it that you replace it with something else, so the focused state is still visible to a user navigating via a keyboard.

This sort of design was used extensively for several years: onscreen elements made to resemble real-world counterparts (an approach known as *skeuomorphism*). In the real world, objects don't have perfectly flat colors. Even on smooth surfaces, light reflects off the object in various ways, producing highlights and shadows.

By giving a button a rounded appearance and a box shadow, it looks more like a physical object. Other skeuomorphic elements include stitched borders and leather-like textured images. Between 2010 and 2013, this sort of design gave way to a new trend called *flat design*.

11.2.2 Creating elements with a flat design

Where skeuomorphic design seeks to emulate the physical world, flat design embraces the modern world's digital nature. It emphasizes vivid, uniform colors and a simpler appearance. This means fewer gradients, shadows, and rounded corners. Ironically, this trend emerged only after these long-awaited effects arrived in CSS. (Before then, shadows and gradients had to be created using images.)

Flat design doesn't necessarily mean none of these effects are used, however. Instead, they're used subtly. Instead of a gradient from light blue to medium blue, for instance, there may be a gradient from one blue to another, which is nearly indistinguishable. Or, perhaps an element has only a faint shadow. It might not have both at the same time.

Let's convert our button to one with a flat design. This new button is shown in figure 11.11. It doesn't look like something from the physical world, though it does have a slight shadow beneath it.



Figure 11.11 Button with a flat appearance

The styles for this are shown next, along with hover and active states. Change your CSS to match.

Listing 11.10 A flat button with hover and active states

```
.button {
  padding: 1em;
  border: 0;
  color: white;
  background-color: #57b;
  font-size: 1rem;
  padding: 0.8em;
  box-shadow: 0 0.2em 0.2em rgba(0, 0, 0, 0.15);
}
.button:hover {
  background-color: #456ab6;
}
.button:active {
  background-color: #148;
}
```

← Solid background color (no gradient)

← Very subtle shadow

← Slightly darker hover and active states

The box shadow here has a few changes. It only has a vertical offset, so the shadow is straight down rather than a more natural-looking angle. I also used an RGBA color with red, green, and blue values of zero (producing black) and an alpha value of 0.15 (almost completely transparent). The hover and active states are also flat in appearance. These states change the background color to slightly darker shades of blue. I also increased the font size, which is another trend that emerged along with flat design.

11.2.3 Creating buttons with a more modern look

Flat design is still popular, but it's evolving. One common approach is to come back toward the middle between flat design and skeuomorphism. Let's remake our button one last time to match figure 11.12. This design will use elements of both design styles.



Figure 11.12 A different type of flat button (active state shown on the right)

This is still a minimal design, but it has a thick border along the bottom, giving it an appearance much like the front side of a 3-D cube-like object. This dark line is in fact not a border at all, but rather a box-shadow with no blur, which allows for the curved edges mirroring the border radius.

In the active state, the button is shifted down by a few pixels. This produces the illusion of it moving into the page as it's pressed. Update your stylesheet to match the styles for this version.

Listing 11.11 Modern button styles

```

.button {
  padding: 0.8em;
  border: 0;
  font-size: 1rem;
  color: white;
  border-radius: 0.5em;
  background-color: #57b;
  box-shadow: 0 0.4em #148;
  text-shadow: 1px 1px #148;
}
.button:active {
  background-color: #456ab5;
  transform: translateY(0.1em);
  box-shadow: 0 0.3em #148;
}

```

Adds back the rounded corners
 Puts a solid shadow beneath the button (no blur)
 Adds a subtle text shadow
 Shifts the button down when clicked
 Reduces the size of the shadow to offset the transform

This button uses `box-shadow` in a different way. Rather than adding a blur and replicating a shadow, I've kept the edges of the shadow crisp. This produces an effect similar to a thick bottom border. It's slightly different from a border, however, because it has curved corners matching the `border-radius` of the element.

I've also added a text shadow. This behaves much like a box shadow, except it casts a shadow of the rendered text rather than the element box. Its syntax is nearly identical: `x-offset`, `y-offset`, blur radius (optional), and a color. But it doesn't support the `inset` keyword or a `spread-radius` value. Here I've cast a dark blue shadow of the text, offset by only 1 px in each direction.

In the active state, I've done something else that's new. I used the `transform` property with a `translateY()` function. This shifts the element down 0.1 em on the screen. (I'll unpack this a bit further when we take an in-depth look at transforms in chapter 15.) I then reduced the vertical offset of the box shadow by the same amount (0.3 em instead of 0.4 em). When you press the button, the button moves, but the box shadow doesn't. Click the button to watch this effect.

Gradients and shadows can be used in all sorts of ways. As time goes on, new design trends will arise. Down the road, when you see a new look on a website, take a few minutes to inspect it in your browser and learn how it's implemented. And, don't be afraid to experiment.

11.3 Blend modes

The majority of the time, an element will have only one background image, whether an actual image or a gradient. But there are instances where you might want two or more backgrounds. CSS allows for this.

The `background-image` property accepts any number of values, each separated by a comma:

```
background-image: url(bear.jpg), linear-gradient(to bottom, #57b, #148);
```

When you apply multiple background images, those listed first render in front of those listed afterward. In this example, bear.jpg will cover the linear gradient. The gradient won't be visible. But if you've added two images, you'll likely want the second image to show through. You can do this by using a *blend mode*.

If you're familiar with photo-editing software, you may have seen blend modes. They control the way stacked images blend together. They have enigmatic names like screen, color-burn, and hard-light. As an example, figure 11.13 shows two backgrounds combined with the multiply blend mode. Both backgrounds use the same image, positioned differently.



Figure 11.13 Two backgrounds combined with the multiply blend mode

This produces an interesting effect, where both copies of the image are still clearly visible, even if they overlap. Yet it doesn't wash out or fade the color like simple transparency does.

NOTE If a background image has some transparency, other backgrounds behind it will show through the transparent areas, even without the use of blend modes. You can achieve this with a transparent png or gif, or with a gradient that uses the transparent keyword as one of its colors.

Let's create an element with two backgrounds and blend them to match figure 11.3. On a new page, add the element shown next. You'll re-use this markup for the next several examples.

Listing 11.12 A div for blending backgrounds

```
<div class="blend"></div>
```

You'll use an empty element for now to illustrate the effect. Next, add the code from the next listing to a stylesheet and link it to the page.

Listing 11.13 Blending two background images

```
.blend {
  min-height: 400px;
  background-image: url(images/bear.jpg), url(images/bear.jpg);
  background-size: cover;
  background-repeat: no-repeat;
  background-position: -30vw, 30vw;
  background-blend-mode: multiply;
}
```

A comma separates two background images.

Specifies one value to apply to both background images

Applies the blend mode

Applies different background positions to each image

Most of these background properties can accept multiple values, separated by a comma. The background position has two such values. The first will apply to the first image, the second to the second image. The `background-size` and `background-repeat` properties also accept multiple values, but by specifying only one, that value is applied to both background images. The `min-height` property is included to ensure that the element doesn't collapse to a height of zero (because it's empty).

The `background-size` property accepts two special keyword values, `cover` and `contain`. Using `cover` resizes the background image so it fills the entire element; this can result in edges of the image being clipped. Using `contain` ensures the entire image is visible, though this may result in some of the element not being covered by the background (a "letterboxing" effect). This property also accepts length values to explicitly set the height and width of the element.

Try changing the blend mode to other values like `color-burn` or `difference` to see the types of effect they can have. This can be amusing to play with, but you might be wondering what the practical applications are. Here are a few uses:

- Tint an image with a single color or gradient
- Apply texture to an image, such as a scratched or grainy film reel look
- Lighten, darken, or reduce the contrast of an image so the text in front of it is more readable
- Overlay a text banner while still allowing the image to show through

Let's look at some examples of these. Afterward, I'll give you a brief breakdown of all the blend modes available.

11.3.1 *Tinting an image*

You can use a blend mode to take a full-color image and tint it with a single hue. To illustrate, you'll take the bear image and color it blue, as in figure 11.14. (Note that if you're reading this in the print edition, the figure isn't printed in color. See the ebook or follow the example in your browser to see the full effect.)



Figure 11.14 Image colored with a uniform blue hue

A background-blend-mode merges not only multiple background images, but also the background-color. These stacked layers are then combined by the blend mode, so you can set a background color to the desired hue, and blend it into the image. To do this, update your CSS to match this listing.

Listing 11.14 Blending a background image with the background color's hue

```
.blend {  
  min-height: 400px;  
  background-image: url("images/bear.jpg");  
  background-color: #148; ← Blue background color  
  background-size: cover;  
  background-repeat: no-repeat;  
  background-position: center;  
  background-blend-mode: luminosity; ← Uses luminosity blend mode  
}
```

The luminosity blend mode takes the luminosity from the front layer (the bear image) and blends it with the hue and saturation from the back layer (the blue background color). In other words, the blended result has all the color of the background color, but the brightness and contrast of the image.

It's important to know that this blend mode (and several others) behave differently, depending on which image is in front of the other. In these cases, the background color is the furthest layer back with the image(s) stacked in front of it.

If, for instance, you were to place a blue layer in front of the bear image rather than behind (using a gradient instead of a background color), the result would be different. In that case, the color blend mode would be needed to achieve the same result—the color blend mode is effectively an inverse of luminosity, taking the hue and saturation from the front layer, while luminosity is taken from the back.

11.3.2 Understanding types of blend mode

CSS supports 15 blend modes. Each uses a different mathematical formula to control how images are blended. For every pixel, the color of one layer is combined with the corresponding color in the other, resulting in a new pixel color for the composite image.

The blend modes are shown in table 11.1. They each fall into one of five types: darkening, lightening, contrasting, compositing, or comparing. Some are more practically useful than others. Choosing the right one often involves a bit of trial and error.

Table 11.1 Blend modes in five basic categories

Type of effect	Blend modes	Description
Darken	multiply	The lighter the front color, the more the base color will show through.
	darken	Selects darker of the two colors.
	color-burn	Darkens the base color, increasing contrast.
Lighten	screen	The darker the front color, the more the base color will show through.
	lighten	Selects the lighter of the colors.
	color-dodge	Lightens base color, decreasing contrast.
Contrast	overlay	Increases contrast by applying <code>multiply</code> to dark colors and <code>screen</code> to light colors, at half strength.
	hard-light	Greatly increases contrast. Like <code>overlay</code> , but applies <code>multiply</code> or <code>screen</code> at full strength.
	soft-light	Similar to <code>hard-light</code> , but uses <code>burn/dodge</code> instead of <code>multiply/screen</code> .

Table 11.1 Blend modes in five basic categories (*continued*)

Type of effect	Blend modes	Description
Composite	hue	Applies hue from the top color onto the bottom color.
	saturation	Applies saturation from the top color onto the bottom color.
	luminosity	Applies luminosity from the top color onto the bottom color.
	color	Applies hue and saturation from the top color onto the bottom color.
Comparative	difference	Subtracts the darker color from the lighter one.
	exclusion	Similar to difference, with less contrast.

At the time of writing, most blend modes are supported in most major browsers, with the exception of IE and Edge. The composite blend modes are also not supported in Safari. Use feature queries and provide fallback behaviors when necessary (see chapter 6, section 6.5). Check <http://caniuse.com/#feat=css-backgroundblendmode> for up-to-date support details.

11.3.3 Adding texture to an image

Another application of blend modes is to add texture to an image. You may have a clear, modern image, but sometimes you want it to appear differently for stylistic reasons. You can use a grayscale image to artificially add film grain or some other texture to the image.

Consider the image shown in figure 11.15. This is the same bear image from earlier, but it has been blended with a textured image to give it the appearance of a rough-hewn canvas. This sort of effect can be achieved with one of the contrast blend modes: `overlay`, `hard-light`, or `soft-light`. In this case, I didn't want to alter the hue of the image, so I used a grayscale image to supply the texture. This way the coloring of the original still shows through.

The code for this textured overlay is shown in the next listing. The texture image is tiled for a repeating pattern and layered over the top of the bear image. Change your stylesheet to match this listing to see this effect in your browser.

Listing 11.15 Using `hard-light` to add texture to the image

```

.blend {
  min-height: 400px;
  background-image: url("images/scratches.png"), url("images/bear.jpg");
  background-size: 200px, cover;
  background-repeat: repeat, no-repeat;
  background-position: center center;
  background-blend-mode: soft-light;
}

```

Layers the texture in front of the primary image

Tiles the texture image every 200 px

Blends with `soft-light`



Figure 11.15 Image with texture blended in

The background size of the texture image (figure 11.16) is set to 200 px, with background repeat enabled. This tiles the image repeatedly to fill the entire element. Meanwhile, the second image has a background size of cover and repeat is disabled, so it's not tiled.

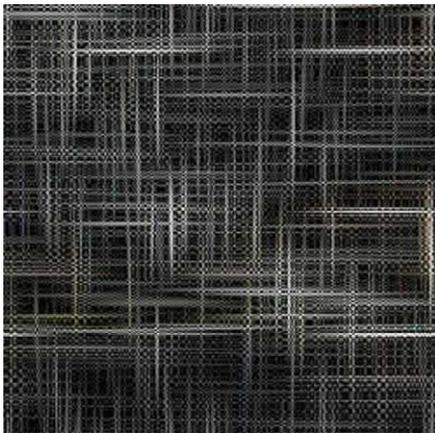


Figure 11.16 Grayscale canvas texture image

I've found that `soft-light` tends to work better with darker texture images, while `hard-light` and `overlay` are better suited for lighter texture images. (This tendency

is reversed when the texture is behind the primary image.) Your results may vary, though, depending on your design's need and darkness of the base image.

11.3.4 Using mix blend modes

Although the background-blend-mode property lets you blend multiple images, it's limited to the background colors or images of one element. Another property, mix-blend-mode, lets you blend multiple elements. This allows you to do more than blend images: text and borders of one element can be blended with the background image of its container. Using a mix blend mode, you can display a heading in front of an image and still allow part of the image to show through, as in figure 11.17.



Figure 11.17 A heading blended with the image behind it

This produces an interesting effect where the text looks transparent, as if it's cut out of the red banner. This works because I used the `hard-light` blend mode and a medium gray text color. The contrast blend modes have more effect with either very light or very dark colors, but a medium gray (`#808080`) allows the background layer to show through unchanged.

To see this in your browser, first you'll need to add the heading to the markup as a child of the container. Update your HTML to match this.

Listing 11.16 Adding a heading inside the container

```
<div class="blend">
  <h1>Ursa Major</h1>
</div>
```

You'll style this `<h1>` into a red banner with a solid background color, thick light gray borders along the top and bottom, and gray text. Then, you'll apply a mix

blend mode. This will treat the entire element as a layer, blended with the background image of the container that's behind it. Change your stylesheet to match this code.

Listing 11.17 Using mix blend mode to blend multiple elements

```
.blend {
  background-image: url("images/bear.jpg");
  background-size: cover;
  background-position: center;
  padding: 5em 0 10em;
}

.blend > h1 {
  margin: 0;
  font-family: Helvetica, Arial, sans-serif;
  font-size: 6rem;
  text-align: center;
  mix-blend-mode: hard-light;
  background-color: #c33;
  color: #808080;
  border: 0.1em solid #ccc;
  border-width: 0.1em 0;
}
```

Uses hard-light
blend mode

Sets text and background
color for the foreground
element

The heading text doesn't have a high contrast here, so you'll have to be careful. I've made it large and bold to help with legibility. It'll also work better over a low contrast background image. In this example, I've placed the heading over the darker part of the image, which is of a lower contrast.

Blend modes can be a lot of fun in a design. Use these in conjunction with gradients and shadows to add a lot of visual interest to the page. But be judicious—these effects are usually best when used in moderation.

Summary

- Use gradients and shadows to add the appearance of depth to the page.
- Even basic flat designs can benefit from some subtle shadows or gradients.
- Use gradients with explicit color stops to add stripes to an element.
- A subtle background gradient rather than a flat color provides a little more complexity to the design.
- Use blend modes to colorize or texture an image.

12

Contrast, color, and spacing

This chapter covers

- Converting a designer mockup into HTML and CSS
- Using contrast to draw attention to the right parts of a page
- Selecting colors
- Leveraging white space
- Working with line height

An important part of web development involves taking a mockup from a designer and bringing it to life with CSS. When you do this, you're effectively translating art into code. Sometimes that translation is straightforward. Other times, you may have to work with the designer to make compromises. Where the designer has fine-tuned every piece of the mockup, you'll need to consider how to organize the CSS for easy re-use. Your CSS will be more general-purpose than their single-page mockup.

After the translational work is done, it'll fall to you, the developer, to continue adding to the site, building upon the vision laid out by the designer. It's important that you at least have a general sense of the designer's concerns about things like spacing, colors, and typography. You'll need to know how to ensure your

implementation is accurate. If you respect the designer's goals, it'll make this process less frustrating.

I also recognize you'll not always work with a designer. If you're at a small startup or working on a personal project, you may be on your own. In which case, it's beneficial to have a basic understanding of design principles so you can implement them yourself.

In this chapter, we'll consider a page mockup as a designer might provide, and convert it into code. I'll focus primarily on the aspects of spacing and color. I'll also highlight some considerations a designer may have made in regard to those. My goal is that, in understanding these concerns, you'll be able to apply them to some degree as you maintain the design or even as you work on projects that have no designer. For that purpose, you'll build the page shown in figure 12.1.

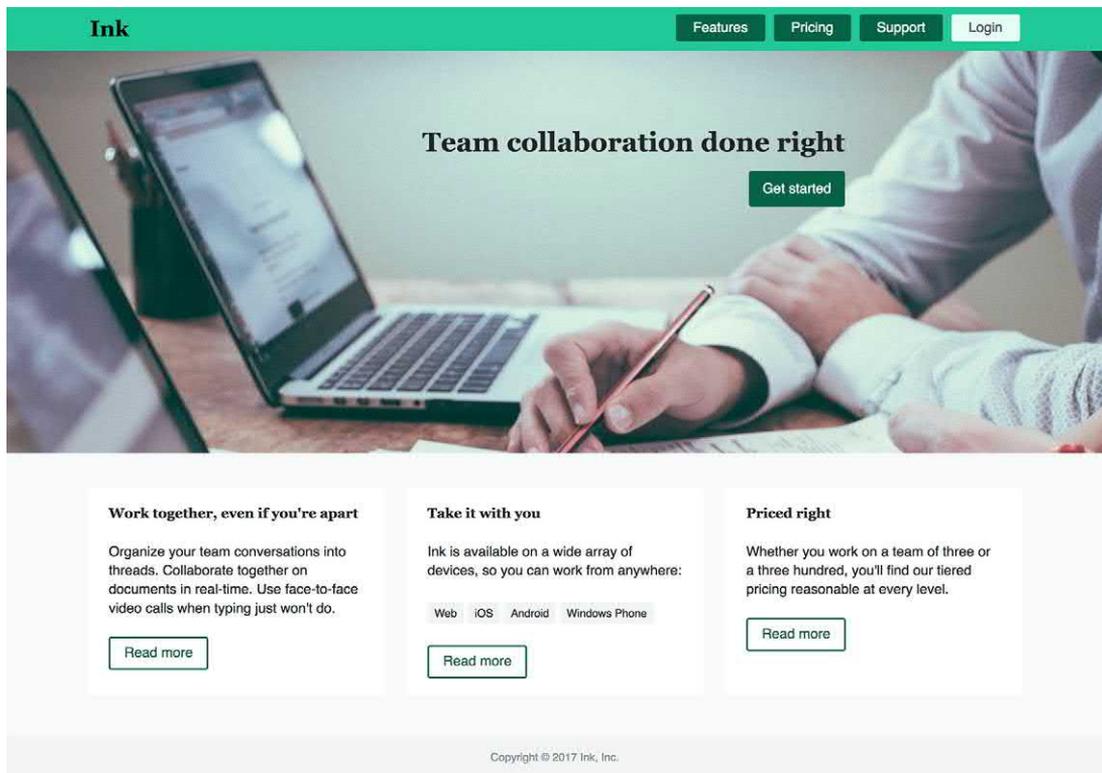


Figure 12.1 The page design for Ink collaboration software

This screenshot shows the finished page. If you receive a design from a designer, it'll probably have a lot more information along with it. You'll see that in a bit, but first I want to point out a few things about the design.

12.1 Contrast is king

When you look at the screenshot in figure 12.1, notice where your eyes go. They should be drawn, primarily, to the slogan “Team collaboration done right” and to the Get started button below it. You’ll also see a number of other things on the page—company name on the top left, navigation on the top right, the three columns below—but the content in the middle of the image has the strongest pull. The reason for this is *contrast*.

Contrast in design is a means of drawing attention to something by making it stand out. Our eyes and our minds naturally find patterns. When something breaks a pattern, our attention goes right to it.

For contrast to work, the page must establish patterns; you cannot have an exception to the rule unless you first have a rule. In figure 12.1, the spacing between the Read more navigational buttons is consistent, as is the size and spacing of the three columns. Additionally, all three Read more buttons are identical. You can also see a few different splashes of color on the page, but they’re all the same hue of green, with only varying darkness. This is one reason why modular CSS and pattern libraries are so important (chapters 9 and 10)—instead of using nested selectors to build a “button-in-a-tile,” build a button that can be used anywhere.

When you promote reuse of styles, you ensure identical patterns will appear on your site. One of a professional designer’s key concerns is to establish patterns and then to break those patterns to highlight the most important parts of the page.

Some key ways you can establish contrast is by using color, spacing, or size. If several items are light, but one is dark, you’ll notice the dark item first. When one item is surrounded by a lot of unused space (called *white space*), that item stands out. And, large elements stand out amid a series of smaller ones. For stronger contrast you can combine one or more of these effects, as I have with the Team collaboration done right slogan on this page: it has larger text, loads of white space, and a striking, dark button.

The slogan isn’t the only contrast evident on the page, though. You’ll find a hierarchy of information importance, communicated via contrast. Apart from the slogan and the Get started button, there’s also contrast evident in the navigation menu (figure 12.2) and within each of the text columns at the bottom of the page (figure 12.3). These elements are not as strong as the slogan, but they attract attention within their respective regions. The footer, being the least important content on the page, is smaller and of lesser contrast.

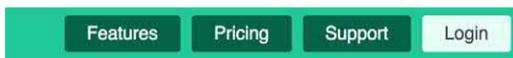


Figure 12.2 The lighter login button draws more attention than the three darker green buttons

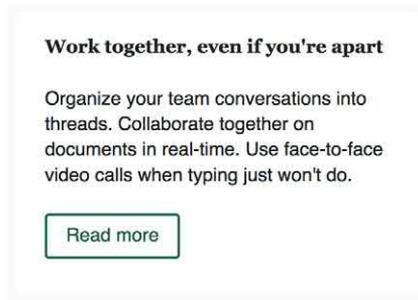


Figure 12.3 The button with colored text and border stands out amid plain black-and-white text

Every web page should have a purpose. It may be to convey a story, collect information, or allow the user to complete some sort of task. In addition to a main purpose, there may be navigation elements, ads, paragraphs of text, and a footer full of copyright information and links to various pages. The designer's job is to make the most important thing stand out. Your job is to not mess that up.

12.1.1 *Establishing patterns*

To establish patterns, your designer may be meticulous about things that seem unimportant to you: precise spacing between certain elements, using the same border radius or same box shadow throughout multiple different components, and even a lot of care for the spacing between letters and lines of text.

An example mockup is shown in figure 12.4, highlighting the precise spacing, in pixels, of various items. It can be tedious (and sometimes even difficult) to keep this precision intact when you convert a design to code.

This mockup uses pink boxes to indicate which spaces are being measured. For example, 10 px between buttons in the navigational menu; 40 px between the bottom of the hero image and the top of the three white columns; 30 px between each column's heading and the following paragraph of text; and so on. Certain measurements on the page will appear commonly throughout, helping establish a visual pattern. For instance, 10 px and 25 px spaces are common on this page.

Let's take a closer look at two aspects of a cohesive design: color choice and controlling space. (Typography is also an important part of this, which we'll focus on in the next chapter.) I'll show you how to accurately implement the design shown in figure 12.4. It's also important to realize that a website evolves over time. Implementing a mockup is part of the work, but you'll also need to be able to add new features or content down the road, while still remaining true to the designer's vision. For this reason, we'll look at some considerations for this work as well.

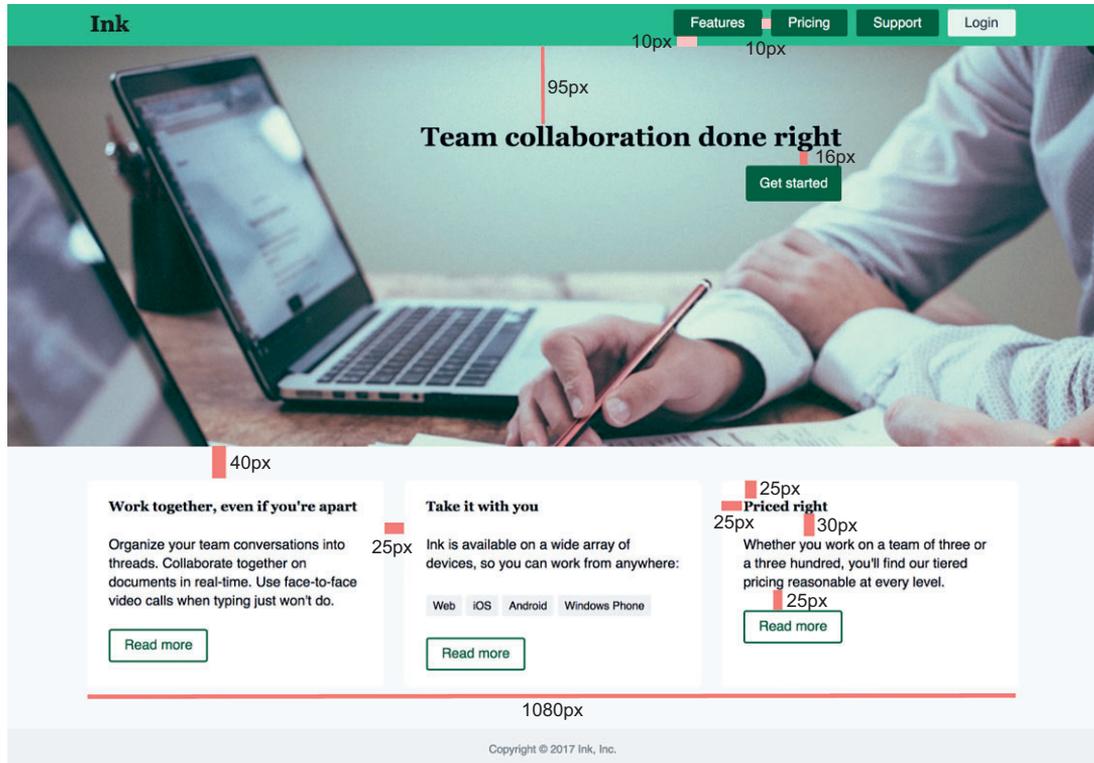


Figure 12.4 A mockup of the page design with annotated measurements

12.1.2 Implementing the design

Create a new page and link it to a new stylesheet. Then copy in the markup shown in the following listing. I've broken up the page design into various modules, which you'll style throughout the rest of the chapter.

Listing 12.1 Page markup

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <nav class="nav-container">
    <div class="nav-container__inner">
      <a class="home-link" href="/">Ink</a>
      <ul class="top-nav">
        <li><a href="/features">Features</a></li>
        <li><a href="/pricing">Pricing</a></li>
        <li><a href="/support">Support</a></li>
        <li class="top-nav__featured"><a href="/login">Login</a></li>
      </ul>
    </div>
  </nav>
```

Top navbar
container

```

        </ul>
    </div>
</nav>

<div class="hero">
    <div class="hero__inner">
        <h2>Team collaboration done right</h2>
        <a href="/sign-up" class="button button--cta">Get started</a>
    </div>
</div>

<div class="container">
    <div class="tile-row">
        <div class="tile">
            <h4>Work together, even if you're apart</h4>
            <p>Organize your team conversations into threads. Collaborate
            together on documents in real-time. Use face-to-face <a
            href="/features/video-calling">video calls</a> when typing just won't
            do.</p>
            <a href="/collaboration" class="button">Read more</a>
        </div>

        <div class="tile">
            <h4>Take it with you</h4>
            <p>Ink is available on a wide array of devices, so you can work from
            anywhere:</p>
            <ul class="tag-list">
                <li>Web</li>
                <li>iOS</li>
                <li>Android</li>
                <li>Windows Phone</li>
            </ul>
            <a href="/supported-devices" class="button">Read more</a>
        </div>

        <div class="tile">
            <h4>Priced right</h4>
            <p>Whether you work on a team of three or a three hundred, you'll
            find our tiered pricing reasonable at every level.</p>
            <a href="/pricing" class="button">Read more</a>
        </div>
    </div>
</div>

<footer class="page-footer">
    <div class="page-footer__inner">
        Copyright &copy; 2017 Ink, Inc.
    </div>
</footer>
</body>

```

Large hero
image

Three-column
tile row

I've used BEM-style notation for some of these class names to make it clear which elements belong to which modules: double-underscores indicate sub-elements of a module, such as `hero__inner`, and double-hyphens indicate variants of a module, such as

button--cta (chapter 9). We'll work our way through these modules. Our first stop is to take a closer look at the colors they use.

12.2 Color

When a designer delivers a design, typically you'll get a large PDF document with several sections. A large portion of the PDF will likely consist of full-page mockups similar to that shown in figure 12.4. But before that, a designer will establish some basics. The PDF may include a page or two of typography examples for various headings and body copy. The document will probably have a detailed breakdown of a few common UI elements, like links and buttons, including their various states, like hover and active. And it'll include a color palette for the site.

The color palette will typically look something like that shown in figure 12.5. It shows color swatches for all the colors used on the site and the associated hex color values. The designer will often give a name to each color, which will be used throughout the rest of the specifications.

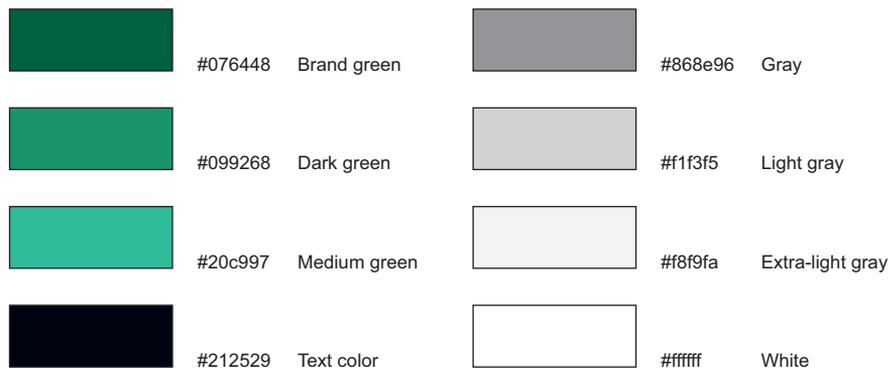


Figure 12.5 The color palette for the site

A palette will typically have one primary color that everything else is based on. It's often derived from the corporate branding or logo. In our page, this is the brand green color (top, left in the figure). Other colors in the palette will often be varying shades of the same hue, or complementary colors to it. Most palettes also have a black, a white (though they may not be pure #000000 or #ffffff), and a handful of grays.

Because these colors will appear repeatedly throughout the CSS, you can save yourself a lot of time by assigning them to variables. Otherwise, you'll be typing the hex values countless times and will, almost certainly, make mistakes.

Let's put together some base styles for the page. This includes assigning each of the palette colors to a variable so you can reuse them. The page as shown in figure 12.6 won't look like much yet, but the colors will start to look closer to the mockup.

Add these base styles to your stylesheet.

- [Ink](#)
- [Features](#)
- [Pricing](#)
- [Support](#)
- [Login](#)

Team collaboration done right

[Get started](#)

Work together, even if you're apart

Organize your team conversations into threads. Collaborate together on documents in real-time. Use face-to-face video calls when typing just won't do.

[Read more](#)

Figure 12.6 Page with base styles and some colors in place

Listing 12.2 Base styles including color variables

```
html {
  --brand-green: #076448;
  --dark-green: #099268;
  --medium-green: #20c997;
  --text-color: #212529;
  --gray: #868e96;
  --light-gray: #f1f3f5;
  --extra-light-gray: #f8f9fa;
  --white: #fff;

  box-sizing: border-box;
  color: var(--text-color);
}
*,
*::before,
*::after {
  box-sizing: inherit;
}

body {
  margin: 0;
  font-family: Helvetica, Arial, sans-serif;
  line-height: 1.4;
  background-color: var(--extra-light-gray);
}

h1, h2, h3, h4 {
  font-family: Georgia, serif;
}

a {
  color: var(--medium-green);
}
a:visited {
  color: var(--brand-green);
}
```

Assigns each color to a variable

Sets heading fonts

Uses variables to assign color where needed

```

a:hover {
  color: var(--brand-green);
}
a:active {
}

```

← Uses variables to assign color where needed

← Placeholder for active links. You'll need a red color for this later on.

I've used custom properties for these colors (see chapter 2, section 2.6 if you need a refresher). By using variables you might save yourself some work down the road should any of these values change. I once worked on a project where the designer decided to make an adjustment to the brand color late in the process. It was trivial to update the variable in one place, but it would have meant updating a hundred or more spots in the code without it.

NOTE I've used CSS custom properties in this example for simplicity; you won't need any special tooling to follow along. In your projects, if you need to support IE or other older browsers, you should favor the use of preprocessor variables instead. See appendix B for an introduction to preprocessors.

I've also left a placeholder here for active links. We'll come back and fill in a color there shortly. Before we do that, let's get the page roughly laid out. You'll get all the main sections of the page in the correct position in relation to each other and then apply some colors and font settings (figure 12.7). We won't fuss much with the spacing yet.

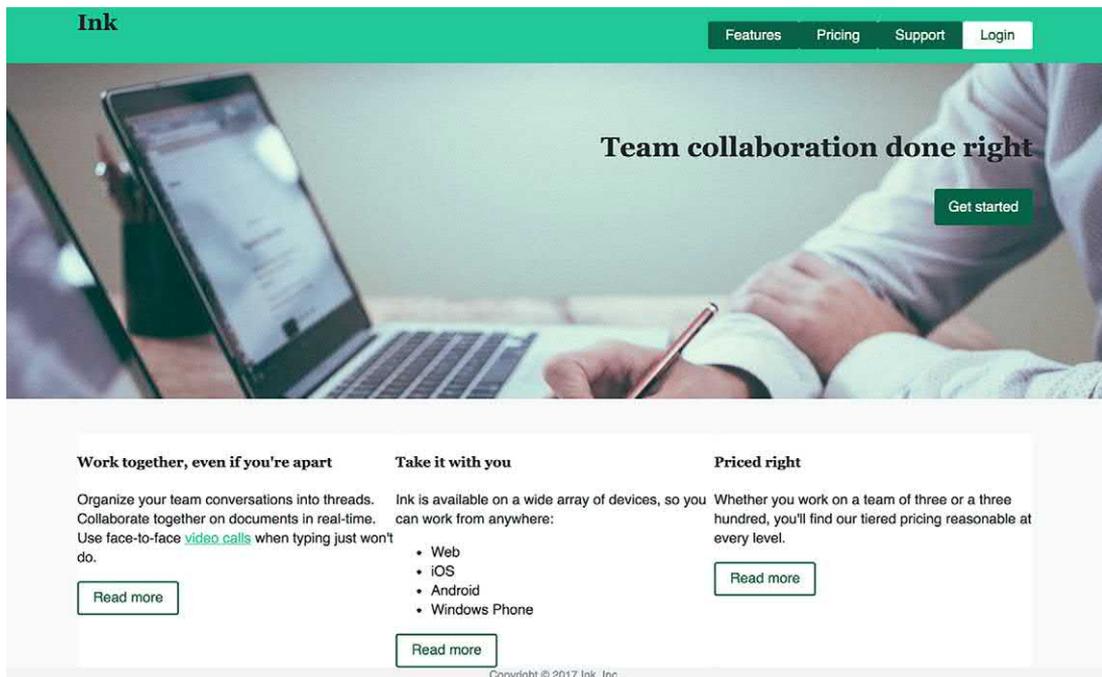


Figure 12.7 Page elements roughly positioned and the initial styles applied.

You'll start at the top and work your way down in three steps: the header, the hero image, and the main section with three columns. For the most part, I'll be reusing techniques covered in earlier chapters. Then we'll circle back and take a closer look at fine-tuning the page.

First up is the header and the navigational bar within. This section consists of three modules: `nav-container`, `home-link`, and `top-nav`, shown in the following listing. Add these to your stylesheet.

Listing 12.3 Header styles

```
.nav-container {
  background-color: var(--medium-green);
}
.nav-container__inner {
  display: flex;
  justify-content: space-between;
  max-width: 1080px;
  margin: 0 auto;
}

.home-link {
  color: var(--text-color);
  font-size: 1.6rem;
  font-family: Georgia, serif;
  font-weight: bold;
  text-decoration: none;
}

.top-nav {
  display: flex;
  list-style-type: none;
}
.top-nav a {
  display: block;
  padding: 0.3em 1.25em;
  color: var(--white);
  background: var(--brand-green);
  text-decoration: none;
  border-radius: 3px;
}
.top-nav a:hover {
  background-color: var(--dark-green);
}
.top-nav__featured > a {
  color: var(--brand-green);
  background-color: var(--white);
}
.top-nav__featured > a:hover {
  color: var(--medium-green);
  background-color: var(--white);
}
```

Centers the contents and restricts their width to 1,080 px

Uses flexbox to display nav items in a row

Adds color and padding to each nav item link

The whole header is wrapped in a `nav-container`. I've used the double-container pattern here to center the inner element (see chapter 4 for a review of this technique). This allows the background color to bleed to the edges of the page, while the width of the main contents is constrained. This element is a flexbox container with `justify-content: space-between` pushing its content to the two edges: `home-link` on the left and `top-nav` on the right. The `top-nav` module is another flexbox, so the links all appear in a row, and all colors are assigned using custom properties.

Next, let's add styles for the hero image area. This will involve two more modules, one for the hero image and another for the button. Add the styles to your stylesheet.

Listing 12.4 Hero image and button styles

```
.hero {
  background: url(collaboration.jpg) no-repeat;
  background-size: cover;
  margin-bottom: 2.5rem;
}
.hero__inner {
  max-width: 1080px;
  margin: 0 auto;
  padding: 50px 0 200px;
  text-align: right;
}
.hero h2 {
  font-size: 1.95rem;
}

.button {
  display: inline-block;
  padding: 0.4em 1em;
  color: var(--brand-green);
  border: 2px solid var(--brand-green);
  border-radius: 0.2em;
  text-decoration: none;
  font-size: 1rem;
}
.button:hover {
  background-color: var(--dark-green);
  color: var(--white);
}
.button--cta {
  padding: 0.6em 1em;
  background-color: var(--brand-green);
  color: var(--white);
  border: none;
}
```

Double-container pattern
 Uses padding to roughly position the slogan and button

Standard button styles

CTA button variant

The Hero module uses the double-container pattern, much like the heading. It also has some padding added to the inner element. The padding values are rough estimates at this point. Once everything is in about the right place, I'll circle back and point out key concerns as we more precisely conform to the designer's mockup.

I've also defined a Button module. The default appearance is a white button with a green border and green text. This is the style of the buttons at the bottom of the main section of the page. I then defined a CTA variant for this button, with a solid green background and white text. (As explained earlier, CTA, or Call to Action, is a marketing term for the key element you want the user to interact with; in this case, the prominent Get started button.) Finally, we'll add the styles for the main portion of the page with the three columns as shown in figure 12.8.

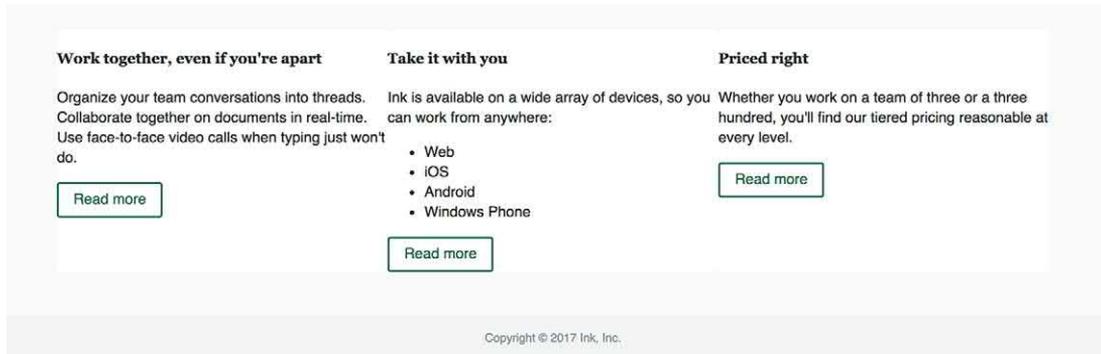


Figure 12.8 Rough styles for the main portion of the page

The main portion of the page consists of a container to constrain the width, a tile-row to shape the three columns and a tile for the white box in each column. Add the next listing to your stylesheet, which also adds styles for the footer. You've already added styles for the button, so you don't need to add anything for those.

Listing 12.5 Three columns and tiles

```
.container {
  margin: 0 auto;
  max-width: 1080px;
}

.tile-row {
  display: flex;
}

.tile-row > * {
  flex: 1;
}

.tile {
  background-color: var(--white);
  border-radius: 0.3em;
}
```

← Same 1,080 px constrained width as with other page sections

Makes each column equal width

```

.page-footer {
  margin-top: 3em;
  padding: 1em 0;
  background-color: var(--light-gray);
  color: var(--gray);
}
.page-footer__inner {
  margin: 0 auto;
  max-width: 1080px;
  text-align: center;
  font-size: 0.8rem;
}

```

← Same 1,080 px constrained width as with other page sections

Again, this listing uses the double-container pattern, restricting the width to 1,080 px. You also set a white background and a border radius for the tiles.

The footer is an example of contrast being used to remove focus, rather than to draw attention. It has a smaller font and gray text on a light gray background. This is the least important part of the page, so it doesn't need to stand out. Instead, it blends in and subconsciously tells the user, "This section of the page is probably not what you're looking for."

12.2.1 Understanding color notations

The colors in our palette are specified using hex notation. This is a concise notation, and one that web developers have been using since the early days of the web. But it's not always the easiest to work with. CSS now has support for other notations as well, using the `rgb()` and `hsl()` functions.

The `rgb()` function is a way to represent the red, green, and blue values of a color using decimal rather than hexadecimal notation. Instead of 00 through FF, it uses 0 through 255: `rgb(0, 0, 0)` is pure black (equal to #000) and `rgb(136, 0, 0)` is brick red (equal to #800).

Both RGB and hex colors are a bit cryptic. It's hard to see a color like #2097c9, or its RGB equivalent, and know how it will render on the page. To break the color value down, its red value (20) is fairly low, its green value (97) is middle of the road, and its blue value (c9) is higher. It's predominantly blue and green, but how dark or how vivid is it? The truth is, RGB colors aren't intuitive. They were designed to be read by a computer, not by a person.

HSL is a type of notation intended to be more human-readable. It stands for Hue, Saturation, and Lightness (or Luminosity). The syntax looks like `hsl(198, 73%, 46%)`, which is equivalent to the hex color #2097c9.

The `hsl()` function takes three values. The first value, representing hue, is an integer between 0 and 359. This indicates the 360 degrees around the color circle, transitioning evenly through red (0), yellow (60), green (120), cyan (180), blue (240), magenta (300), and back to red. The second value, representing saturation, is a percentage defining the intensity of the color: 100% makes the color vivid, and 0% means no color is present, resulting in a shade of gray. The third value, representing lightness,

is a percentage defining how light (or dark) the color is. A lightness of 50% provides for the most vivid colors—setting it higher makes the color lighter, with 100% resulting in pure white; setting it lower makes the color darker, with 0% resulting in black. For example, the value `hsl(198, 73%, 46%)` has a cyan-blue hue, reasonably high saturation (73%) and a lightness near 50%, so it'll produce a rich blue, a little darker than sky blue.

Table 12.1 shows a side-by-side comparison of several colors and their various representations in hex, RGB, HSL, and named colors. (There are about 150 such named colors that are valid CSS values.)

Table 12.1 Comparison of color systems and values

Name	Hex	RGB	HSL
blue	#0000ff	rgb(0, 0, 255)	hsl(240, 100%, 50%)
lavender	#e6e6fa	rgb(230, 230, 250)	hsl(240, 67%, 94%)
coral	#ff7f50	rgb(255, 127, 80)	hsl(16, 100%, 66%)
gold	#ffd700	rgb(255, 215, 0)	hsl(51, 100%, 50%)
green	#008000	rgb(0, 128, 0)	hsl(120, 100%, 25%)
tan	#d2b48c	rgb(210, 180, 140)	hsl(34, 44%, 69%)

The best way to get familiar with HSL is to play around with it. Again, I encourage you to visit <http://hslpicker.com/>. It provides an interactive color picker with three sliders for the three values, as well as a fourth for transparency. Watch how each slider affects the rendered color as you move it.

NOTE RGB and HSL notations each have a corresponding notation with an added alpha channel: `rgba()` and `hsla()`. These accept a fourth value, a number between 0 and 1, representing transparency. Additionally, some browsers are also beginning to support an eight-character hex notation where the last two characters specify an alpha channel.

CONVERTING COLORS IN THE BROWSER

Let's convert the hex colors to HSL. Many online resources such as hslpicker.com will give you a color in all three notations. But the easiest place to do conversion is often in your Chrome or Firefox DevTools, because you always have them on hand. I'll show you how to do this in Chrome.

With the page loaded, open the browser's DevTools (Cmd+Option+I on Mac; Ctrl+Shift+I on Windows). In the Elements panel, click the `<html>` tag to select it. Its associated styles are shown in the Styles panel, including your custom properties (figure 12.9).

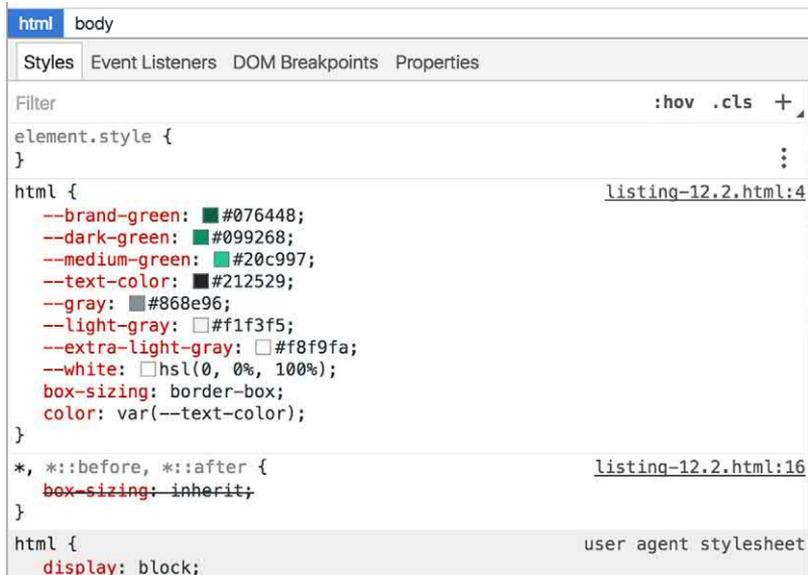


Figure 12.9 The applied colors are shown in the DevTools Styles panel. Shift-click the small square beside a color to cycle between hex, RGB, and HSL color notations.

Beside each color is a small square displaying an example of the color. If you press and hold Shift and then click your mouse on this square, it changes the hex value to an RGB value. Clicking again changes the RGB to an HSL value. Clicking a third time returns to the hex notation.

NOTE This method for cycling between color notations also works in the Firefox DevTools. Unfortunately, it only works for regular properties and not for colors assigned to custom properties as in our example.

If you want to dig deeper, clicking the square opens a full color picker dialog (figure 12.10). This allows you to fine tune colors, select from a palette, or cycle between hex, RGB, and HSL notations. It also includes an eye-dropper to extract colors from the page. Firefox offers a similar color picker, although it isn't as full-featured.

SWITCHING THE STYLESHEET TO HSL

Often, a hex color is all you need. But converting to HSL can help with fine-tuning colors or finding new colors to add to your site. Let's convert our colors to HSL and then make some observations about the site's colors.

Copy the HSL values from your DevTools into your stylesheet. This portion of your CSS should match this listing.

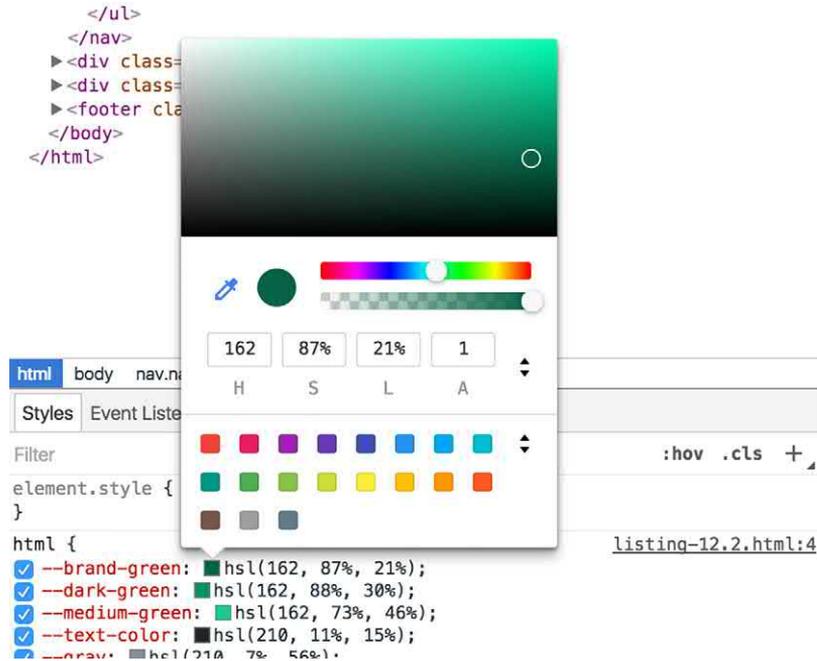


Figure 12.10 Use the color picker dialog to fine tune colors.

Listing 12.6 Converting hex colors to HSL

```
html {
  --brand-green: hsl(162, 87%, 21%);
  --dark-green: hsl(162, 88%, 30%);
  --medium-green: hsl(162, 73%, 46%);
  --text-color: hsl(210, 11%, 15%);
  --gray: hsl(210, 7%, 56%);
  --light-gray: hsl(210, 17%, 95%);
  --extra-light-gray: hsl(210, 17%, 98%);
  --white: hsl(0, 0%, 100%);

  box-sizing: border-box;
  color: var(--text-color);
}
```

Green colors all have the same hue.

The text and gray colors aren't pure gray.

A couple things become apparent when the colors are in HSL notation. First, you can now see that all three green colors have the exact same hue. You may not know off-hand that 162 is a teal green until you look in the browser, but you can see there's a symmetry between the three colors. This is impossible to discern from the hex values, but in HSL it's obvious. With this knowledge, it's easy to add another green color to the palette. If the need arises for a lighter shade of the same color, I'd try something

like `hsl(162, 50%, 80%)`, then fine-tune the saturation and luminescence in the browser's DevTools until it looked appropriate.

You can also observe that the gray colors aren't a pure gray: they've a small bit of saturation, each with the same hue. It's not likely you could tell this by looking at the colors, but it's a small detail that can help the page look richer. True colorless grays almost never happen in the real world, and our eyes expect to see some color, even if it's subtle.

NOTE A designer will typically include several shades of gray in the palette. It's my experience, however, that you'll always need another gray. Whether it's an even lighter color than the extra light gray or something between gray and light gray, the need will eventually arise. This makes naming the variables problematic. For this reason, consider using names with numeric values like `--gray-50` or `--gray-80`, where the number roughly corresponds to luminescence. This way you can insert another value between two existing ones when needed.

Again, you don't necessarily need to convert every color to HSL in a real-world project. But when the need arises, go for it. Often this makes working off the color easier.

12.2.2 *Adding new colors to a palette*

Occasionally, you'll find you'll need a color that your designer didn't plan for. Maybe you need a red error message or a blue informational box. Experienced designers usually account for common instances like this, but you may still find yourself in a situation where you need to add to the palette.

Our stylesheet has a placeholder for an active link color. Traditionally, active links are red, which is in the provided by the user agent stylesheet. But, it's a bright, cartoony red that looks out of place on this page. Let's find a less vivid color that works with the green.

The simplest way to find to a color that works well with another is to find its *complement*. This is the color on the opposite side of the color wheel: the complement of blue is yellow; the complement of green is magenta (or purple); and the complement of red is cyan.

With HSL color values, finding the complementary color is easy: add or subtract 180 to the hue value. Our primary brand green has a hue of 162. Adding 180 to this gets us a hue of 342, which is red with a touch of magenta. You can also subtract 180 to find its complement, which results in a hue of -18. This is equivalent to hue 342, so `hsl(-18, 87%, 21%)` will render the same as `hsl(342, 87%, 21%)`. But I prefer to keep my values between 0 and 360 so they are in more familiar territory.

Now that we have a hue, we need a saturation and lightness. The page's regular link color is medium green—`hsl(162, 73%, 46%)`—so let's start there. Because green is our primary brand color, we don't want secondary colors to steal the show too

much, so we'll drop the saturation down a little—let's say 10%. This gives us a color of `hsl(342, 63%, 46%)`. Figure 12.11 shows an active link with this red color.

Organize your team conversations into threads.
Collaborate together on documents in real-time.
Use face-to-face [video calls](#) when typing just won't do.

Figure 12.11 A red active link

Let's add this color to the stylesheet. Edit your code to match this listing. This includes assigning the color to a custom property `--red`, and then using it for active links.

Listing 12.7 Adding the red color to the palette

```
html {
  --brand-green: hsl(162, 87%, 21%);
  --dark-green: hsl(162, 88%, 30%);
  --medium-green: hsl(162, 73%, 46%);
  --text-color: hsl(210, 11%, 15%);
  --gray: hsl(210, 7%, 56%);
  --light-gray: hsl(210, 17%, 95%);
  --extra-light-gray: hsl(210, 17%, 98%);
  --white: hsl(0, 0%, 100%);
  --red: hsl(342, 63%, 46%);
}

box-sizing: border-box;
color: var(--text-color);
}
...
a:active {
  color: var(--red);
}
```

← Assigns a red variable

← Uses the variable for active links

With this in place, load the page and see it in action. Unfortunately, this is a little tricky because a link's active state isn't shown by default. You can trigger it by clicking and holding on a link, but it'll return to green as soon as you let go. To make this easier, you can use the DevTools to force an active state.

Right-click a link and choose *Inspect* or *Inspect element* from the context menu. This opens DevTools. In the *Elements* pane, right-click the `<a>` tag and select `:active` (or `active` in Firefox) from the context menu (figure 12.12). This will force the browser to display the element's active styles.

With the element forced to an active state, you can see how the red color looks. When needed, you can edit styles live in the DevTools and see how it affects this active element.

Choosing colors that look good is never an exact science, but working in HSL can help make it easier. Try colors that are complementary to colors already on your

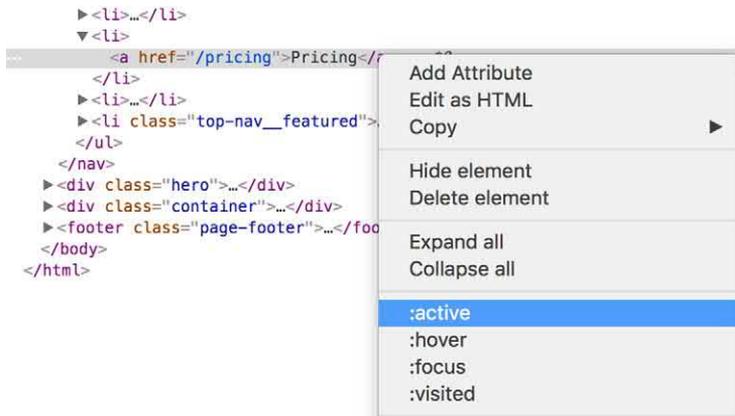


Figure 12.12 DevTools lets you force an element into an active, hover, focus, or visited state so you can preview how the styles will appear.

page. Play with the saturation and luminescence in DevTools to find something that looks nice.

If you want to dive deeper into color selection, look online for articles on Color Theory. One great article, created by Natalya Shelburne, to get you started is available at <http://tallys.github.io/color-theory/>.

12.2.3 Considering contrast for font colors

You probably noticed our font color is a dark gray and not a true black (#000). In HSL notation, it has a luminescence value of 15%, not 0%. The use of gray rather than true black is a common practice. On a backlit computer screen, true black text on a pure white background (#fff) produces too much contrast. This can produce eye fatigue when reading, especially for larger blocks of text. The same is true for white text on a black background. In these instances, you should either use a dark gray in place of black or a light gray instead of white, or both. To your eye, it still appears as black and white, but it'll be more comfortable to read.

While you don't want too much contrast for your text, you also don't want too little. Gray text on a light gray background can be hard to read for users with impaired vision. It can also be hard to read on a smartphone in bright sunlight. So, how do you know when you have enough contrast?

To help guide this decision, the W3C's Web Content Accessibility Guidelines (WCAG) provide a minimum recommended contrast ratio (called level AA), as well as a stricter, enhanced contrast ratio (called level AAA). And because larger text is easier to read, both levels include a less strict contrast ratio for large text. The recommended contrast ratios are shown in table 12.2.

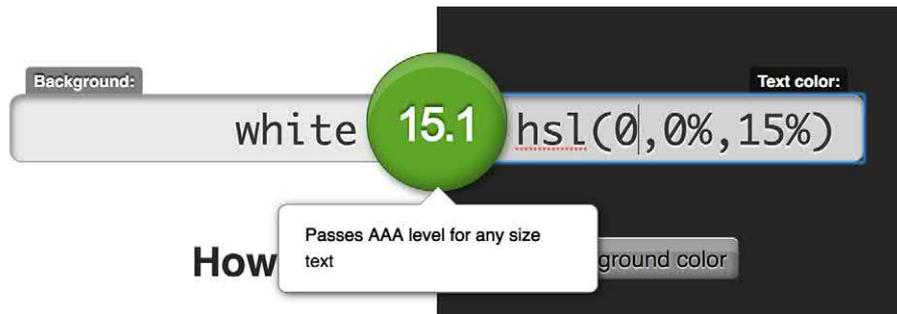
Table 12.2 WCAG text contrast recommendations

	Level AA	Level AAA
Regular text	4.5:1	7:1
Large text	3:1	4.5:1

The WCAG defines large text as 18 pt (24 px) or larger for a regular font weight, or 14 pt (18.667 px) for bold fonts. In short, this generally means your body fonts should meet or exceed the regular text recommendation, and your headings should meet or exceed the large text recommendation.

The WCAG provides a formula for computing this contrast ratio, but I never bother with the math. It's much easier to use a tool. Several tools are available online: search for CSS color contrast checker.

Each has its own strengths and weaknesses. One of my favorites is available at <http://leaverou.github.io/contrast-ratio>. This checker supports any valid CSS color format. Paste in your background color and your text color, and it shows you the computed contrast ratio (figure 12.13). Hover over this number to see whether this passes WCAG level AA or AAA, and for which font sizes.

**Figure 12.13** The background and text colors have a contrast ratio of 15.1:1.

With many designs, it's not practical for every bit of text to meet level AAA contrast levels. The WCAG recommendations acknowledge this. It's a good idea for your main body text to meet level AAA, but you can be a little more relaxed and aim for level AA with colored labels or other decorative text.

Also, keep in mind that the mathematical analysis isn't the full story. Some typefaces are easier to read than others. This is especially true if your design uses thin fonts. Figure 12.14 shows two copies of the same paragraph. Even though the colors are the same in each, the perceived contrast is different.

The paragraphs in figure 12.14 are both set using the Helvetica Neue typeface. The one on the left has a font weight of 300 (often called light or book); the one on

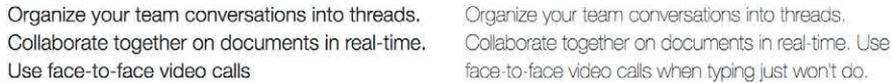


Figure 12.14 A thin font face results in less visual contrast, even with the same color.

the right has a font weight of 100 (thin). A contrast ratio of 7:1 may be excellent with the font on the left, but the font on the right might need one that's a bit stronger.

TIP Only some fonts provide thin weights, but when you use them, make sure you have a strong color contrast so they're readable.

12.3 Spacing

With colors squared away, we can turn our attention to the precise spacing the designer provided in the mockup. This can be a tedious part of the development process, and it may involve some back and forth with your designer when reviewing your work, pointing out inconsistencies that you'll need to fix.

A lot of this work simply comes down to setting the correct margins on elements. Doing this is generally the easiest place to start, though you may need to make a few adjustments from there. Let's look at two things you'll need to consider: whether or not to work with relative units and how line heights can impact your vertical spacing.

12.3.1 Using ems vs. px

One important decision you'll have to make is whether you want to use ems or pixels. Designers typically provide measurements in pixels, so these are the easiest to use. But there are benefits to converting to a relative unit, whether ems or rems.

Consider the measurements specified around the navigational menu (figure 12.15). The design calls for 10 px between each item, as well as 10 px between their bottom edges and the bottom edge of the navbar.



Figure 12.15 The specification calls for 10 px around and between each nav item.

Throughout chapter 2, I discussed the benefits of using relative units. In particular, they let you define a responsive font size (`font-size: calc(0.5em + 1vw)`), then allow your design to scale proportionally with the font. On a larger screen, the font size will be larger, as well as the em- and rem-based margins. This benefit also applies if the user customizes their browser's default font size.

This technique of using responsive font sizes is, however, relatively new, so most designers aren't used to working with relative units. If you want to use this technique, you should probably discuss it with the designer. You'll also have to do the unit conversions yourself.

If you decide to use pixels, you make the work easier on yourself in the short term, but it means a less flexible design going forward. This could potentially mean more work later, but it's impossible to know for sure. If you decide to use relative units, you'll have a little more work up front, but your design will be more robust.

Because using pixels is the more straightforward option (for example, set 10 px margins or paddings where needed), I'll walk you through the more involved process building the navigational menu with ems.

In the design specification, the spacing measurements in the navbar call for 10 px around the menu items (figure 12.15). Because the base font size is 16 px, you can do the math, dividing the desired length by the base font size: 10 divided by 16 is 0.625, so our distances here will be 0.625 em. Now you're ready to add the declarations indicated here to your stylesheet.

Listing 12.8 Using padding and margin to set nav spacing

```
.nav-container {
  background-color: var(--medium-green);
}
.nav-container__inner {
  display: flex;
  justify-content: space-between;
  max-width: 1080px;
  margin: 0 auto;
  padding: 0.625em 0;
}

/* ... */

.top-nav {
  display: flex;
  list-style-type: none;
  margin: 0;
}
.top-nav > li + li {
  margin-left: 0.625em;
}
```

← Adds 10 px top and bottom padding to the whole navbar

← Removes the list margin applied from the user agent stylesheet

← Adds a 10 px horizontal margin between each nav item

When working with space, it's important to consider when you should use padding and when you should use margins. In this case, it makes sense to use padding for the vertical spacing on the `nav-container__inner` so it'll also apply to the whole container, padding the page title on the far left as well as the `top-nav` list. Then I used margins for the horizontal spacing between nav items because I wanted each item to have the space between them.

The space below the hero image and between each of the three columns is also straightforward (figure 12.16). Because these gaps both need to be applied to the outside of elements with a background image or background color, you'll use margins to set this space.

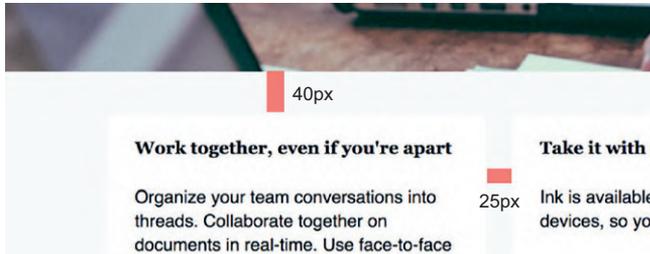


Figure 12.16 The page margins below the hero image (40 px) and between the columns (25 px)

Again, divide these pixel values by the base font size to convert the lengths to ems. The 40 px beneath the hero image is equal to 2.5 em ($40 / 16 = 2.5$ —this margin is already in place) and the 25 px between each column is equal to 1.5625 em ($25 / 16$). Add these margins as shown in the next listing.

Listing 12.9 Adding margins below the hero image and between the columns

```
.hero {
  background: url(collaboration.jpg) no-repeat;
  background-size: cover;
  margin-bottom: 2.5rem;
}
/* ... */

.tile-row {
  display: flex;
}
.tile-row > * {
  flex: 1;
}
.tile-row > * + * {
  margin-left: 1.5625em;
}
```

← Ensures 40 px space beneath the hero image

Adds 25 px between each column

Spacing between containers like this (with background images or background colors) is generally straightforward. It can be a little more finicky when you need to adjust the space between lines of text, such as with paragraphs or headings.

12.3.2 Factoring in line height

Our mockup also defines spaces around text. Figure 12.17 shows the measurements specified for this. (It may be difficult to see here, but this is a white tile on an extra light gray background. The 25 px measurements on the top and left are from the edges of this white tile.)

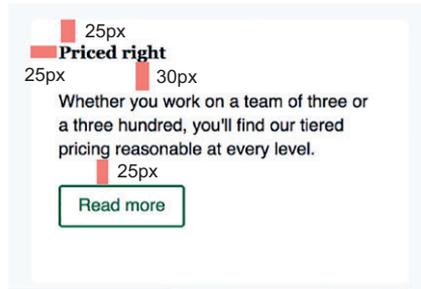


Figure 12.17 Desired spacing in the tile and around the text

Applying the 25 px space around the text edges is a matter of adding padding to the tiles: $25 / 16 = 1.5625$ em. The 30 px between the heading and the paragraph isn't quite as simple, however. If you were to apply a 30 px margin between the two elements, the space between the two lines of text would be closer to 36 px. To understand why this is, let's look at how the element's height is determined.

In the box model, the element's content box is surrounded by padding, then border, then margin. But with elements like paragraphs and headings, there's more to the content box than the printed text: the element's line height contributes to the overall height, beyond the top and bottom of the characters. This is illustrated in figure 12.18. The text is 1 em high, but the line height extends a little further above and below the edge of the text.

On your page, you have a line height of 1.4. This is applied to the `<body>` element and inherited down from there. Thus, an element with one line of text has a content



Figure 12.18 The line height defines the height of the content box.

box 1.4 em high, and the text is vertically centered within. With a font size of 16 px, this makes the total height of the content box 22.4 px. The extra 6.4 px are split evenly above and below the text.

So, if you give the heading a bottom margin equal to 30 px, there will visually be an extra 3.2 px between the text and the top of the margin. There will also be an extra 3.2 px in the content box of the paragraph beneath. (The spacing is the same because both heading and paragraph have the same line height and font size.) This produces a perceived space between the two of 36.4 px.

NOTE Designers are accustomed to working with *leading*, which is a measure of the space between lines of text. In CSS, this space is controlled by line height, which is not directly analogous to leading. We'll look closer at fine-tuning this spacing in the next chapter.

A designer won't usually fuss over a one- or two-pixel discrepancy, but an extra six-and-a-half pixels might bother them. It'll be even larger if you have a bigger line height or one of the elements has a larger font size.

The way to fix this discrepancy is to account for the extra space and subtract it from the margin. Instead of a 30 px margin, subtract the extra 6 pixels and aim for 24 px. Divide by 16, and this gets us an em value of 1.5. Add these spacings from the following listing to your stylesheet.

Listing 12.10 Setting the tile and paragraph spacing

```
p {
  margin-top: 1.5em;
  margin-bottom: 1.5em;
}

/* ... */

.tile {
  background-color: var(--white);
  border-radius: 0.3em;
  padding: 1.5625em;
}
.tile > h4 {
  margin-bottom: 1.5em;
}
```

Adds margins to paragraphs in your base styles

Adds padding inside the tiles

Adds the margin beneath the tile headings

You've applied the 1.5 em margins to the base styles, so all paragraphs will have the same spacing throughout the page. You repeated this measurement again below the tile title (`.tile > h4`), so the space beneath the heading is always the same, even if it's not followed immediately by a paragraph. Because of margin collapsing, the two margins will overlap, producing the 30 px of space between the header and the paragraph.

One final set of measurements in the mockup remains: the spacing in the hero image around the slogan. This portion of the mockup is shown in figure 12.19.



Figure 12.19 The design calls for 95 px above the slogan and 16 px beneath it (between it and the button).

The line height of the slogan will play into these spacings as well because it has such a large font size. The font size is 1.95 rem, which, multiplied by the 16 px base font size, produces a 31.2 px font. This multiplied by a line height of 1.4 produces a computed line height of 43.68 px, or about an additional 6 px above and 6 px below the text.

Because the line height contributes 6 px of space above the text, you only need to add 89 px of additional space to achieve the desired 95 px. Likewise, beneath the slogan, you only need to add an additional 10 px to achieve the 16 px gap shown in the mockup. This is a lot of arithmetic, and sometimes the easiest approach is to sit down with the designer and edit values live in the browser until the designer approves.

Now that you know you need to add 89 px above and 10 px below the slogan, you can convert these values to relative units and add them to the stylesheet: $89/16$ equals 5.5625 em and $10/16 = 0.625$ em. Update the portion of the stylesheet shown in the next listing, adding annotated declarations to position the slogan.

Listing 12.11 Positioning the slogan and button within the hero image

```
.hero {
  background: url(collaboration.jpg) no-repeat;
  background-size: cover;
  margin-bottom: 2.5rem;
}

.hero__inner {
  max-width: 1080px;
  margin: 0 auto;
  padding: 5.5625em 12.5em 12.5em 0;
  text-align: right;
}

.hero h2 {
  font-size: 1.95rem;
  margin-top: 0;
  margin-bottom: 0.625rem;
}
```

← Replaces the rough estimate with the newly calculated spacing

← Removes the top margin as the hero__inner padding provides all the space needed

← Defines the space between the slogan and the button

The top padding of the `hero__inner` provides the spacing above the slogan. I've added padding to the right and bottom edges, though specific values weren't specified in the

mockup. I then set the slogan's top margin to zero, so it won't add more space inside the `hero__inner`'s padding. I also used rems rather than ems for the slogan's bottom margin because the slogan doesn't have the default 16 px font size.

12.3.3 Spacing inline elements

One final detail to put into place in the page design remains. The center column has a list of operating systems where the Ink application is available for use (figure 12.20). I've left this as a regular unordered list until now. Let's get these laid out inline as they appear in the mockup.

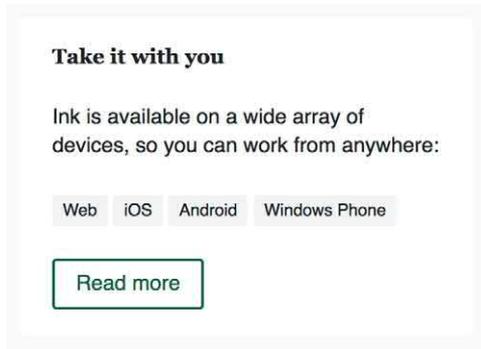


Figure 12.20 The list items need to be styled and made inline.

This sort of mini-layout is common for things like listing tags on blog posts or categories for merchandise. I have included it here because it can come with a few quirks that you should be familiar with.

A few options are available for this type of layout. Two that jump to mind are flexbox or inline elements. We've looked at a number of flexbox layouts throughout the book, so I want to take a look at the concerns when using inline elements.

A number of styles here are straightforward. Each item will need `display: inline`, as well as a small bit of padding, background color, and a border radius. At first, this will look like enough, but a problem will emerge if the content line wraps. The result is shown in figure 12.21, which could occur at certain viewport widths or if the content changes down the road.

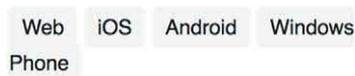


Figure 12.21 The list items overlap when they line wrap.

The gray background of items in one row will overlap with that of items in another row. The reason for this is the line height. As you saw earlier in the chapter, the height

of the line of text is determined by the font size times the line height. If you add padding to an inline element, the element itself will get taller, but it will not increase the height of the line of text. That's determined exclusively by the line height.

To fix this, you'll need to increase the line height of each item. Add the code shown in the next listing to style these tags on your page. Go ahead and edit the line height to different values to see the affect it has.

Listing 12.12 Styling the tags

```
.tag-list {  
  list-style: none;  
  padding-left: 0;  
}  
.tag-list > li {  
  display: inline;  
  padding: 0.3rem 0.5rem;  
  font-size: 0.8rem;  
  border-radius: 0.2rem;  
  background-color: var(--light-gray);  
  line-height: 2.6;  
}
```

Overrides user agent list styles

Sets a large line height to add vertical space when line wrapping

This behavior is unique to inline elements. If an element is a flex item (or inline-block), the line will grow if necessary to accommodate it. But you'll also need to add both horizontal and vertical margins to maintain spacing between items. By using inline items, you can allow the natural white space between elements to provide the spacing for you.

NOTE Notice the text Windows Phone within the inline element is allowed to line wrap. In a flexbox or inline block, this is not permitted, and the entire element will wrap to the next line. If this is a concern either way, choose whichever approach provides the solution that makes the most sense in your context.

This completes the page design. Your page should now match the full mockup as shown in figure 12.22.

We spent a lot of time focusing on very particular details. Many developers won't put this much attention to detail when implementing a design, but for those that do, it pays off. These details are the difference between a good design and a great one.

As you work in CSS, I encourage you to take the time to fine tune the design. Even if you don't have a professional designer behind your design, trust your eye. Try a little more space here or a little less there and see what feels right. Take time to tweak values. Don't over-use color, but selectively put it in the places you want to draw attention. Establish consistent patterns, then break those patterns to draw the users eye to the most important things on the page.

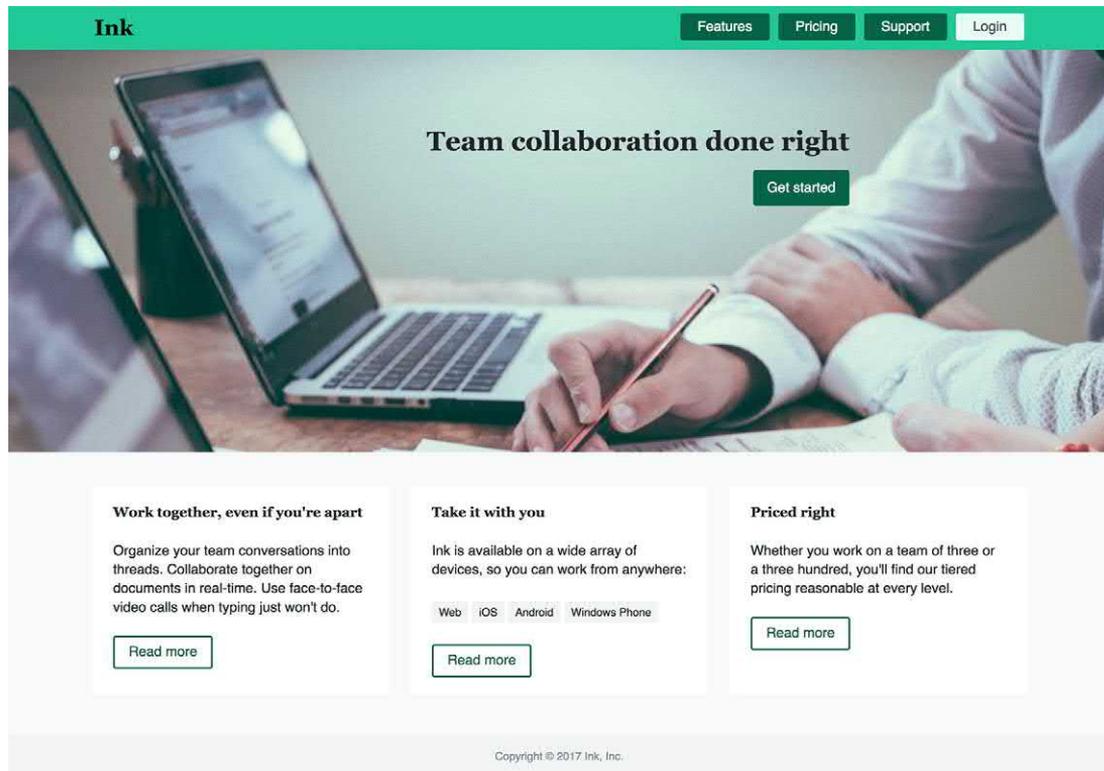


Figure 12.22 Completed page design

Summary

- Use contrast selectively to draw attention to important parts of the page.
- Use HSL color to make working with colors easier and more understandable at a glance.
- Trust your designer when they get picky about nitty gritty details.
- Take the time to fine-tune spacing.
- Remember that line height can impact your vertical spacing.

13

Typography

This chapter covers

- How web fonts can give your page a unique feel
- Using the Google Fonts API
- Tuning font spacing (tracking and leading)
- Web font performance concerns and optimizations
- Dealing with FOUT and FOIT

You can make or break a page design with its fonts. For years, web developers had to choose from a limited set of typefaces, referred to as *web safe fonts*. These are font families like Arial, Helvetica, and Georgia that are commonly installed on most users' systems. Browsers could render the page using only these system fonts, so that's what we had to work with. We could specify a more exotic font, such as Helvetica Neue, but it would only show up for those users who happened to have it installed; other users would see a more generic fallback.

This all changed with the rise of web fonts. Web fonts use the `@font-face` at-rule to tell the browser where it can find and download custom fonts for use on a page; applying a custom typeface can transform an otherwise dull page. This opens a whole new world of possibilities. It also involves a lot more decisions than we used to make.

You can use fonts that make a page feel playful or serious, trustworthy or informal. Look at the font examples in figure 13.1. The same text is set with three different pairs of fonts. The example on the top left uses News Cycle for the heading and EB Garamond for the body. It looks quite formal, like it might appear on a newspaper's website. The top right uses Forum and Open Sans and looks more informal. These might be fonts you'd use on a personal blog or for a small tech company. The bottom left column uses Anton and Pangolin. Its appearance is playful, or even cartoonish, which would be fitting on a children's site. By doing nothing more than changing the typeface, you can completely change the tone of the page.

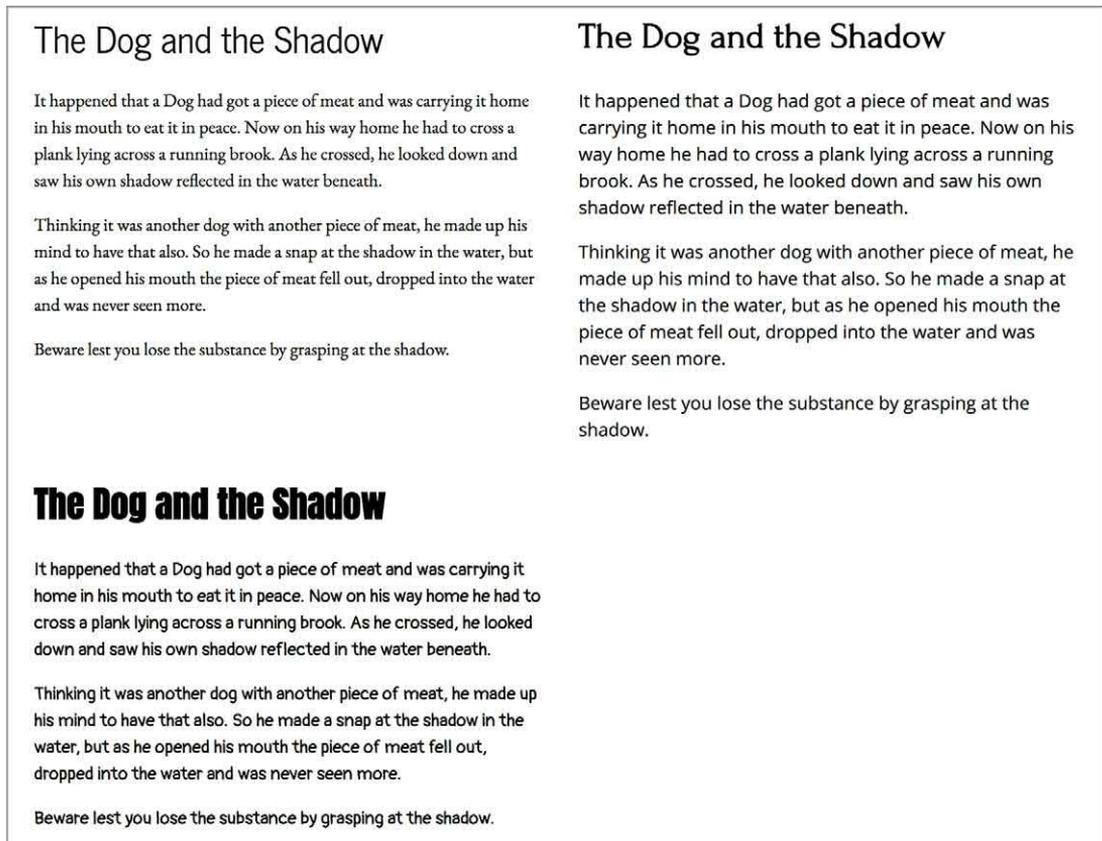


Figure 13.1 The fonts you use can have a marked impact on the feel of your site.

In this chapter, we'll take a look at web fonts. I'll show you how they work and introduce you to some online services that provide a large selection of fonts to choose from. We'll also look at the CSS properties that control the layout, spacing, and size of

fonts. Understanding these properties will allow you to improve the readability of your site or more closely match designs as provided by a designer.

Typography is an art form as old as the printing press. This makes it the only topic in this book with hundreds of years of history behind it. As such, I won't claim to exhaust the subject here, but I'll show you some of the essentials and how to bring them to bear on the modern web.

13.1 Web fonts

The easiest and most common way to use web fonts is through an online service. Common ones are:

- Typekit (www.typekit.com/)
- Webtype (www.webtype.com)
- Google Fonts (<https://fonts.google.com/>)

Whether paid or free, these services take care of many concerns for you, including both technical (hosting) and legal (licensing) issues. They each offer a large library of typefaces to choose from. Though sometimes, if you need a particular typeface, it might only be available with a particular service.

Because Google Fonts has a lot of high-quality, open source fonts—and it's free—I'll walk you through using this service to add web fonts to a page. Google does a lot of the work for you, so it's mostly a straightforward process. After that, we'll look under the hood to get a closer look at how it all works.

You'll take the page you built in the previous chapter and add web fonts to improve the design. Afterward, the page will render as shown in figure 13.2. The Roboto font is the main body font used on most of the page, and Sansita is the font used in the headings.

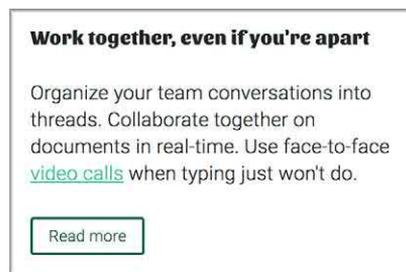


Figure 13.2 A portion of the page with Sansita and Roboto web fonts applied

It's common to use two different fonts like this: one for headings and another for body text. Often, one font will be a *serif* font and the other will be *sans-serif*, though, in this case, both are sans-serif. You may also see designs that use different weights of the same typeface for headings and body text.



serif—A small line or “foot” at the end of a stroke in a letter. A typeface with serifs is known as a serif typeface (Times New Roman, for example). One without serifs is known as a sans-serif typeface (Helvetica, for example).

If you’ve followed along in the last chapter, you should already have this page built, minus the web fonts. (The HTML for this page is shown in listing 12.1, and the CSS was built up in listings throughout the rest of chapter 12, so your page should already match these listings from the previous chapter.) Next, let’s add web fonts.

13.2 Google fonts

To see the directory of fonts available from Google Fonts, go to <https://fonts.google.com/>. The page shows typefaces in a grid of tiles (figure 13.3). You can select fonts from this screen, or you can search for a particular font by clicking the magnifying glass on the top right corner.

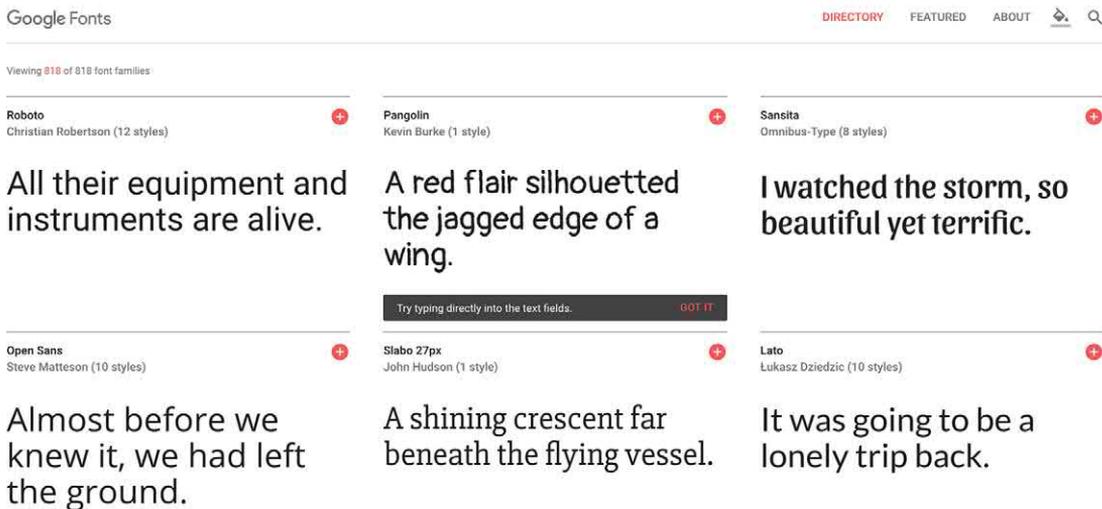


Figure 13.3 Font selection interface for Google Fonts

For any font you want to use, click the red + icon, and Google adds it to your selected fonts, which show in a drawer near the bottom right (figure 13.4). Click the red minus (–) icon to remove a font.

Because you know the fonts you want, you can search for them by name. In the search menu, type *Sansita*. All other font tiles are filtered out of the main view. Click the + icon to add it to your selected fonts. Then delete *Sansita* from the search box

2 Families Selected

Your Selection [Clear All](#)  

Sansita  Roboto 

EMBED CUSTOMIZE Load Time **Fast**

Embed Font

To embed your selected fonts into a webpage, copy this code into the <head> of your HTML document.

STANDARD @IMPORT

```
<link href="https://fonts.googleapis.com/css?family=Roboto|Sansita" rel="stylesheet">
```

Specify in CSS

Use the following CSS rules to specify these families:

```
font-family: 'Roboto', sans-serif;
font-family: 'Sansita', sans-serif;
```

For examples of how fonts can be added to webpages, see the [getting started guide](#).

Figure 13.4 Currently selected fonts appear in a drawer, along with sample code snippets.

and type Roboto. Google will pull up several related typefaces, including Roboto, Roboto Condensed, and Roboto Slab. Add Roboto to your selected fonts.

If you open the selected font families drawer, it shows you Sansita and Roboto along with code snippets in HTML (to embed the fonts in your page) and CSS (to use the fonts in your styles). Before you use these snippets, however, you'll need to make a change to the fonts to select the font weights needed for the page. Click the Customize tab to see the options (figure 13.5).

You're probably familiar with working with regular and bold font weights, but some typefaces are designed for several different weights. For example, Roboto comes in six different weights, ranging from thin to black, as well as an italic variant of each. Check the boxes beside the fonts you want to download to your page.

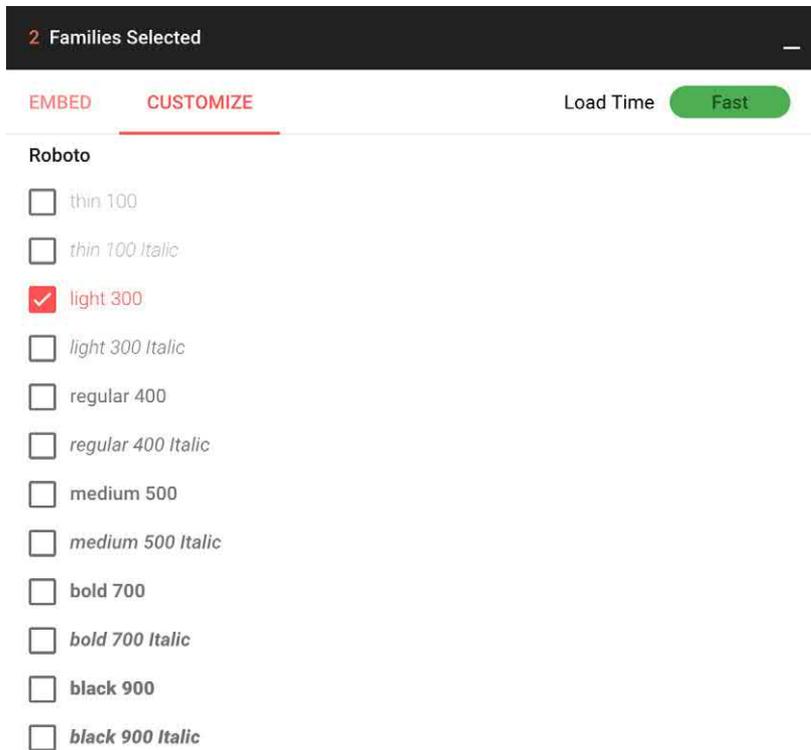


Figure 13.5 Select which font weights and styles to include on your page

NOTE The terms *typeface* and *font* are often conflated. Traditionally, *typeface* refers to an entire family of fonts (Roboto), usually created by the same designer. Within a typeface there may be multiple variants or weights (light, bold, italic, condensed, and so on). Each of these variants is a *font*.

In an ideal world, you could select all the font variants, giving you plenty of options to choose from for your page design. If you start checking boxes, however, you'll notice the Load Time indicator (upper right) change from Fast to Moderate to Slow. The more fonts you select, the more the browser will have to download. And web fonts, after images, are one of the biggest offenders in slowing down loading time. You'll need to be judicious, selecting only the fonts you need.

Under Roboto, select Light 300 and under Sansita, select Extra-bold 800. These are the weights you'll be using for this example. (You'll often need the italic version of the main body font as well, but it's not a bad idea to hold off until you know for sure it's needed on your site.) Click the Embed tab to return to the code snippets, and you'll see they've been updated to specify the font weights you selected.

Copy the `<link>` tag and add it to your page's `<head>` as shown in the following listing. This adds the stylesheet containing the font definitions to your page. Your page will now have two stylesheets: yours and the font stylesheet.

Listing 13.1 Stylesheet `<link>` tag for Google Fonts

```
<link href="https://fonts.googleapis.com/css?family=Roboto:300|Sansita:800"
      rel="stylesheet">
```

With this stylesheet, Google has taken care of everything needed to set up the web fonts for your page. With this in place, you can now use the fonts throughout your styles. You'll add these to the page, resulting in the page shown in figure 13.6.

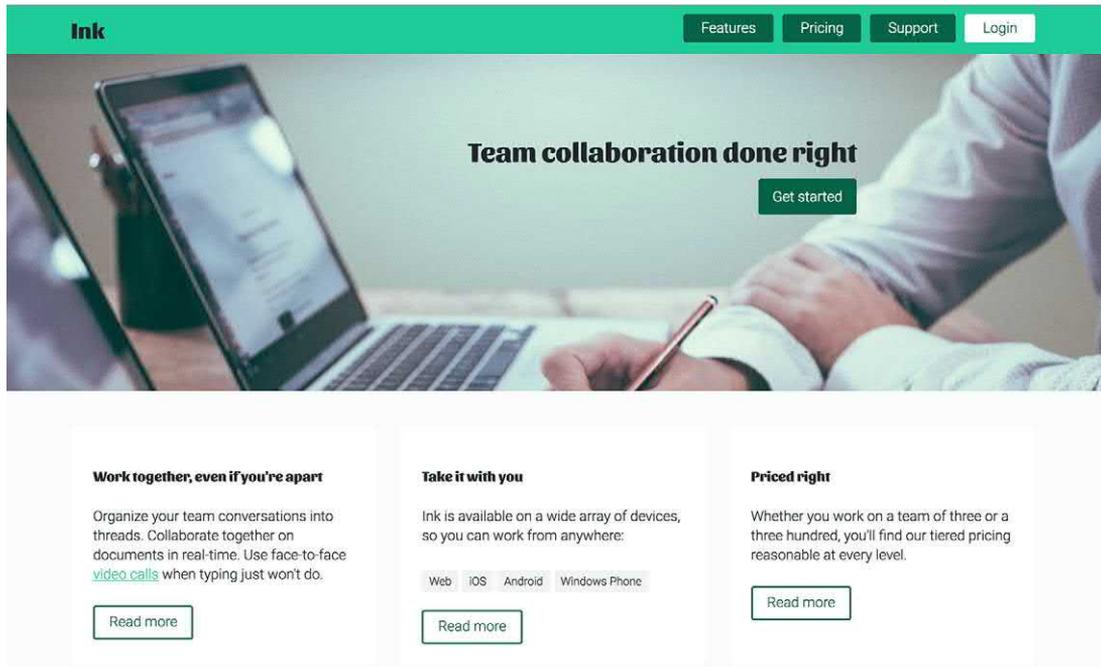


Figure 13.6 The page with Roboto and Sansita fonts applied

To use the fonts, you'll need to specify Roboto and Sansita using the `font-family` property. Let's update the CSS to do this. You'll set Roboto as the main font on the `<body>` element, where it's inherited by the entire page. Then you'll set Sansita for headings and the Ink homepage link in the top left corner. Change the corresponding portions of the code to match the next listing.

Listing 13.2 Using the web fonts

```

body {
  margin: 0;
  font-family: Roboto, sans-serif;
  line-height: 1.4;
  background-color: var(--extra-light-gray);
}

h1, h2, h3, h4 {
  font-family: Sansita, serif;
}

/* ... */

.home-link {
  color: var(--text-color);
  font-size: 1.6rem;
  font-family: Sansita, serif;
  font-weight: bold;
  text-decoration: none;
}

```

← Applies Roboto globally to the page

← Sets headings to the Sansita font

← Sets the homepage link to Sansita

With the Google Fonts stylesheet on the page, the browser now understands that these font families refer to the downloaded web fonts, and it'll apply them to the page. If you use another web font service, such as Typekit, the process will be similar. The service will either provide the URL to the CSS you need or a snippet of JavaScript that will add it to the page for you.

I'll show you how to tweak the spacing of fonts and share some considerations for loading performance. But first, let's see what Google Fonts is doing for us.

13.3 *How @font-face works*

Font providers make the process of adding fonts to your page nice and easy, but it's worth knowing how they work. Let's look at that CSS file Google provided. Open the URL <https://fonts.googleapis.com/css?family=Roboto:300|Sansita:800> in your browser to see Google's CSS. I've copied a portion of it into the following listing.

Listing 13.3 Google's font definition stylesheet

```

/* latin */
@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 300;
  src: local('Roboto Light'), local('Roboto-Light'),
       url(https://fonts.gstatic.com/s/roboto/v15/Hgo13k-
         tfSpn0qil1SFdUFZBwlxU1rKptJj_0jans920.woff2) format('woff2');
}

```

The @font-face ruleset, defining a single font for use elsewhere in your page's CSS

← Declares the name for this font

← Defines which font style and font weight this @font-face applies to

← Location(s) where the font file can be found

```

    unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02C6, U+02DA, U+02DC,
                  U+2000-206F, U+2074, U+20AC, U+2212, U+2215;
}

/* latin */
@font-face {
  font-family: 'Sansita';
  font-style: normal;
  font-weight: 800;
  src: local('Sansita ExtraBold'), local('Sansita-ExtraBold'),
       url(https://fonts.gstatic.com/s/sansita/v1/M0VOVsEPZWhxh-
       yeRPQtpQzyDMXhdD8sAj6OAJTFsBI.woff2) format('woff2');
  unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02C6, U+02DA, U+02DC,
                  U+2000-206F, U+2074, U+20AC, U+2212, U+2215;
}

```

← The unicode character ranges this @font-face applies to

The @font-face ruleset defines the fonts for the browser to use in your page’s CSS. The first ruleset here effectively says, “If the page needs to render Latin characters with a Roboto font-family using a normal font style (not italic) and a weight of 300, use this font file.” The second is similar, defining a bold version (weight 800) of the Sansita font.

The font-family defines the name you’ll use to reference this font elsewhere in your stylesheet. The src: provides a comma-separated list of locations where the browser will look, beginning with local (Roboto Light) and local (Roboto-Light): If the users operating system happens to have an installed font called either Roboto Light or Roboto-Light, then that font will be used. Otherwise, the woff2 font file will be downloaded from the given url () and used.

NOTE The file, as hosted by Google, includes similar portions of code repeated for other character sets, such as Cyrillic, Greek, and Vietnamese. These characters are kept in separate font files, so your browser doesn’t have to download them unless needed. The principles are the same, so I’ve left them out for simplicity.

13.3.1 Font formats and fallbacks

Google’s stylesheet is making the assumption that my browser supports WOFF2 font files. It’s able to do this because Google checked my browser’s user agent string and determined that my browser (Chrome) supports these font files. If I were to load this same URL in IE10, however, it’ll serve a slightly different stylesheet that references a WOFF font.

WOFF stands for Web Open Font Format. It’s a compressed format designed specifically for use over a network. All modern browsers support WOFF, but some don’t yet support WOFF2 (which has better compression and, therefore, smaller files). You probably don’t want to have to sniff user agent strings like Google is doing. Instead, a robust solution should provide URLs for both WOFF and WOFF2 font files as shown in the next listing. (I’m using shorter URLs than Google’s to make it more readable.)

Listing 13.4 A WOFF2 web font declaration with fallback to WOFF

```

@font-face {
  font-family: "Roboto";
  font-style: normal;
  font-weight: 300;
  src: local("Roboto Light"), local("Roboto-Light"),
       url(https://example.com/roboto.woff2) format('woff2'),
       url(https://example.com/roboto.woff) format('woff');
}

```

The first supported format listed will be used.

Fallback to WOFF for browsers that don't support WOFF2.

When web fonts were just getting started, developers had to include as many as four or five different font formats because browsers each supported different ones. WOFF is now almost fully supported, though WOFF2 loads faster, so provide both URLs, if possible.

13.3.2 Multiple variants of the same typeface

If you need multiple fonts from the same typeface, each needs its own `@font-face` rule. If, in the Google Fonts interface, you selected both light and bold versions of Roboto, Google will give you a stylesheet URL that looks something like this: <https://fonts.googleapis.com/css?family=Roboto:300,700>. Open this URL in your browser to see the code. I have copied a portion of it into the following listing.

Listing 13.5 Defining two different weights for the same typeface

```

/* latin */
@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 300;
  src: local('Roboto Light'), local('Roboto-Light'),
       url(https://fonts.gstatic.com/s/roboto/v15/Hgo13k-
         tfSpn0qilSFdUfZBwlxUlrKptJj_0jans920.woff2) format('woff2');
  unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02C6, U+02DA, U+02DC,
                 U+2000-206F, U+2074, U+20AC, U+2212, U+2215;
}
...
/* latin */
@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 700;
  src: local('Roboto Bold'), local('Roboto-Bold'),
       url(https://fonts.gstatic.com/s/roboto/v15/d-
         6IYp1oFocCacKzxwXSOJBwlxUlrKptJj_0jans920.woff2) format('woff2');
  unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02C6, U+02DA, U+02DC,
                 U+2000-206F, U+2074, U+20AC, U+2212, U+2215;
}

```

Roboto light

Roboto bold

This listing shows two different definitions for a Roboto font. If the page needs to render Roboto with a weight of 300, it'll use the first definition. If it needs to render Roboto with a weight of 700, it'll use the second.

If the page's styles call for some other version (for example, `font-weight: 500` or `font-style: italic`), the browser approximates as best it can from the two weights provided. Typically, this means the browser will choose whichever of the two is closer to the needed font. But, depending on the browser, it may occasionally italicize or make bold one of the provided fonts artificially to approximate the desired effect. It does this by transforming the letter shapes geometrically. This never looks as good as using a properly designed font, so I don't recommend relying on it.

When you use Google Fonts or another font provider, you can use their interface and they'll give you all the code you need. Sometimes, you may want to use a font that's not available from a provider. In this case, you'll need to host your own font, using `@font-face` rules to define them as needed for the browser.

13.4 Adjusting space for readability

Let's return to our page. Now that your web fonts are loaded, let's tune them to your design. This involves two properties: `line-height` and `letter-spacing`. These properties control the space between lines of text (vertically) and the distance between individual characters (horizontally).

These are two properties that many developers tend to overlook. If you take time to adjust them in your designs, it makes a significant improvement on the look of your site. Furthermore, it can make reading more comfortable for your user, increasing engagement on the page.

If the text spacing is too compact, it can take more effort to read anything more than a few sentences or even words. The same is true if the spacing is too great. See figure 13.7 for examples of text with various spacing.

If you try to read the compressed text in the top left, you'll find that it takes greater concentration. You might find your eye skipping a line, or reading the same line twice. You'll want to stop reading sooner. It also makes the page feel cramped and busy. The text on the bottom left is a little too spread out. It draws too much attention to each and every letter, taking a little more effort to form the words in your mind. Meanwhile, the text in the top right is comfortable. It looks "right," and it's the easiest of the three to read.

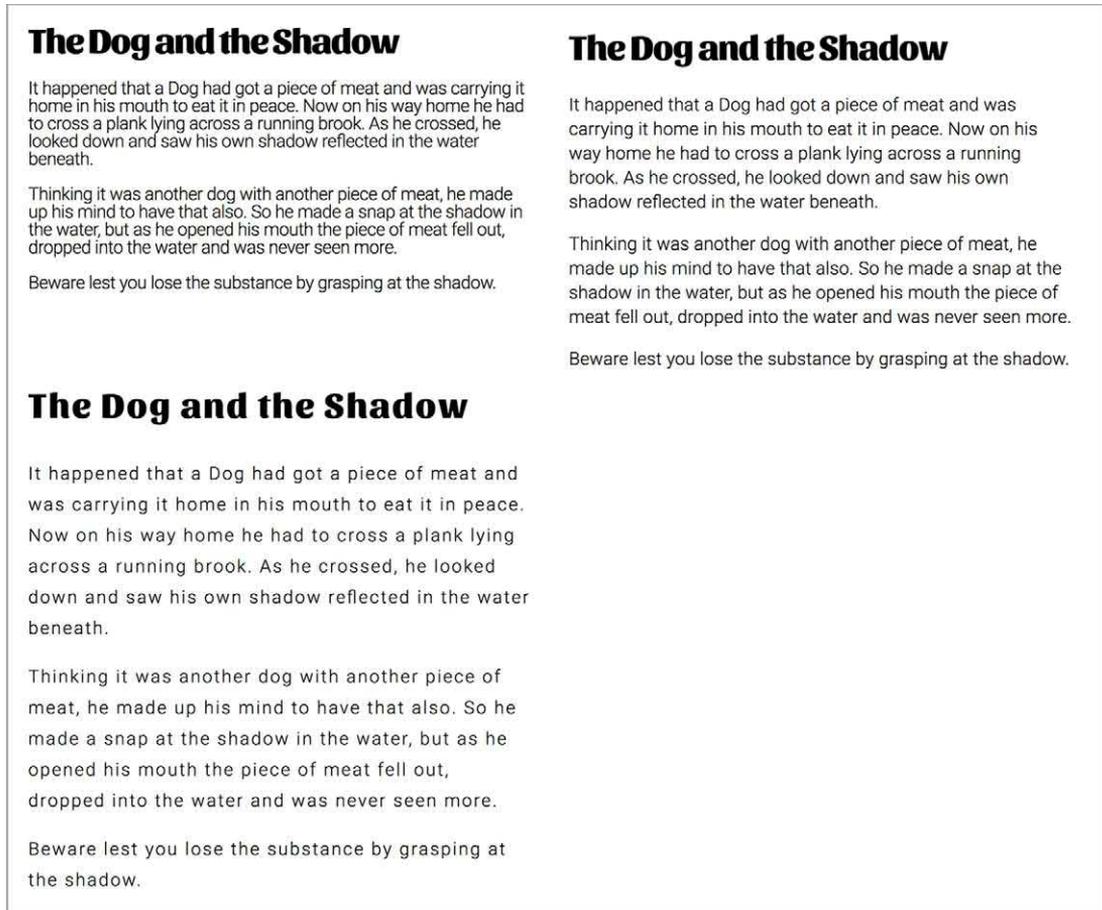


Figure 13.7 The spacing of the text can have a marked impact on readability.

13.4.1 *Body copy spacing*

Finding values for both line-height and letter-spacing is subjective. The best approach is generally to try several values; find one that's too tight and another that's too loose, then settle on a value in between. Fortunately, there are some rules of thumb to help guide you.

The initial value for the line-height property is the keyword `normal`, which is equal to about 1.2 (the exact value is encoded in the font file, as is the font's `em` value). In many cases, however, this is too small. For body copy, a value between 1.4 and 1.6 is usually closer to ideal.

On your page, you already have a line height of 1.4 applied to the `<body>`, which you added in the previous chapter. This value is inherited down to the rest of the page. But consider if that were missing. Figure 13.8 shows one of the tiles. The one on

the left has the initial values for both line-height and letter-spacing; the one on the right has been adjusted. (You'll make your page match the spacing of the tile on the right.)

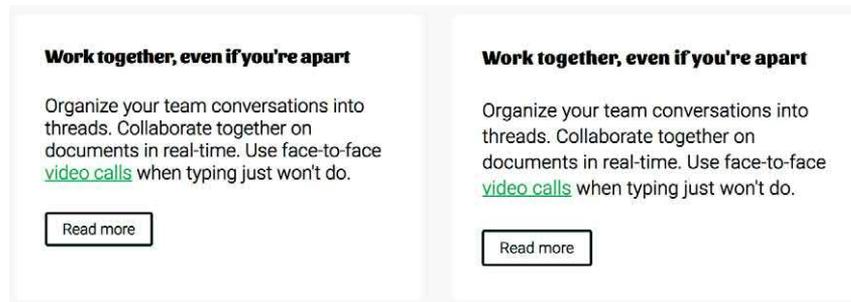


Figure 13.8 A tile from Ink's page with initial spacing values (left) and purposely chosen values (right)

Change the value for the `line-height` to 1.3 or 1.5 to see how they look. See if you like these more or less than the value of 1.4 I've provided.

TIP Longer lines of text should have a larger line height. This makes it easier for the reader's eye to scan to the next line without losing their place. Ideally, you should aim for line lengths that hold between 45 and 75 characters per line, as this is generally considered the most easily readable.

Next, we can look at letter-spacing. This property takes a length, measuring the amount of space to add between each character. Even a spacing of 1 px can be drastic, so this should typically be a small value. When I'm looking for a value, I generally change this in increments of 1/100ths of an em (for example, `letter-spacing: 0.01em`). Add letter spacing to your CSS as shown here.

Listing 13.6 Setting letter spacing on the body element

```
body {
  margin: 0;
  font-family: Roboto, sans-serif;
  line-height: 1.4;
  letter-spacing: 0.01em;
  background-color: var(--extra-light-gray);
}
```

Line height and letter spacing will be inherited by everything on the page.

Adds 0.01 em of extra space between characters

Increment the letter spacing to 0.02 em or 0.03 em to see how it looks. You may not have the eye of a designer to know which is best, but that's okay. Go with your gut. When in doubt, don't overdo it. The point isn't to draw attention to the spacing; in fact, it's quite the opposite. On Ink's page, I find both 0.01 em and 0.02 em reasonable, so let's keep the more conservative 0.01 em.

Converting leading and tracking to CSS

In the design world, the spacing between lines of text is called *leading* (rhymes with bedding). This originates from strips of lead that were added between rows in a letterpress. Spacing between characters is called *tracking*. If you work with a designer, they may specify leading and tracking for the design, but these values might look nothing like the `line-height` and `letter-spacing` CSS properties.

Leading is often expressed in points, such as 18 pt, measuring a line of text plus the space between it and the next line of text. This is effectively the same as the CSS `line-height`, but it's not expressed as a unitless number. You must first convert it to pixels so it's the same as your font, then to a unitless number.

To convert from pt to px, multiply the point value by 1.333 (that's 96 px per inch and 72 pt per inch, so $96/72 = 1.333$): $18 \text{ pt} * 1.333 = 24 \text{ px}$. Then divide this by your font size to find the unitless line height: $24 \text{ px} / 16 \text{ px} = 1.5$.

Tracking is usually given as a number, such as 100. This number represents thousands of an em, so to convert it to ems, divide by a 1000: $100 / 1000 = 0.1 \text{ em}$.

13.4.2 Headings, small elements, and spacing

The spacing for headings won't always be the same as that for the body copy. After you have spacing set for the body copy, check the headings and see if they need any adjusting.

Headings are normally short, often only a few words. But, occasionally, a long one will pop up. A common mistake during design is to test only short headings. Now that your line height is set on the page, temporarily add extra text to various headings to force them to line wrap (figure 13.9).

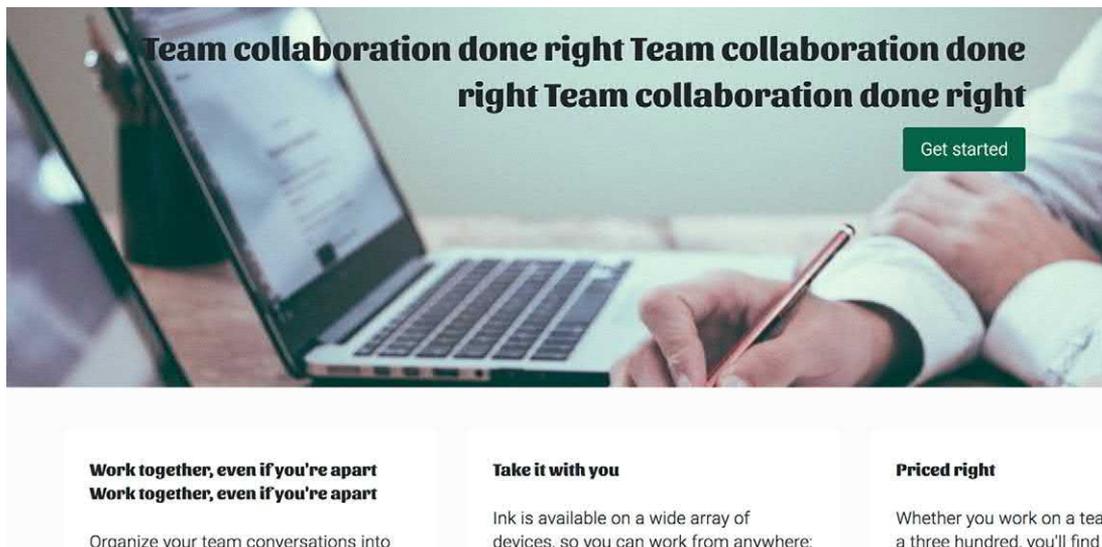


Figure 13.9 Force headings to line wrap to make sure their line height is appropriate.

In this case, I think the vertical spacing looks fine, so I won't change it. But it's always worth checking. Depending on the typeface, a line height of 1.4 can appear too far apart, especially in large font sizes. I've had to bring the line height on headings down as low as 1.0 on some sites.

The letter spacing, on the other hand, could stand to be a little further apart. Add the declaration shown in the next listing to your stylesheet. This is a subtle change in the letter spacing.

Listing 13.7 Increasing the letter spacing for headings

```
h1, h2, h3, h4 {
  font-family: Sansita, serif;
  letter-spacing: 0.03em;
}

/* ... */

.home-link {
  color: var(--text-color);
  font-size: 1.6rem;
  font-family: Sansita, serif;
  font-weight: bold;
  letter-spacing: 0.03em;
  text-decoration: none;
}
```

Increases the
letter spacing
for headings

For the body copy, spacing was focused on maximizing readability. But this matters less for headings and other elements with little content, such as buttons. In these cases, you can get more creative. You can get away with broader spacing. You can also use negative letter spacing to compress the characters. Figure 13.10 shows the slogan with a `letter-spacing: -0.02em` applied.



Figure 13.10 Tight letter spacing is one option for short, stylistic parts of the page.

This spacing is dramatic. Several paragraphs of text like this would be hard to read, but it can work for short pieces of text (only a few words). Let's apply this to the slogan text. Add the next listing to your stylesheet.

Listing 13.8 Tightening up the spacing of the slogan

```
.hero h2 {
  font-size: 1.95rem;
  letter-spacing: -0.02em;
  margin-top: 0;
  margin-bottom: 0.625rem;
}
```

← Uses a negative letter spacing to compress text

You can also evaluate the spacing and text of small elements on the page, like the buttons. I think they look a little large right now, especially the navigational buttons in the header bar. Let's adjust those. Figure 13.11 shows how they look now (top) and how they'll look after making the changes (bottom).

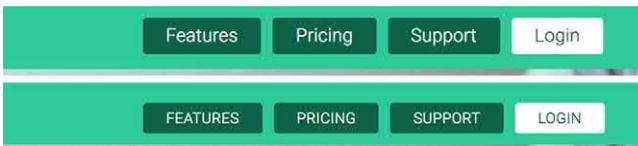


Figure 13.11 Changing the text properties can improve the look of the nav buttons (bottom).

I've made a few changes here: I've reduced the font size, capitalized the text using `text-transform`, and increased the letter spacing.

TIP Text set in all caps generally looks better with a larger letter spacing.

Add the declarations shown next to your stylesheet. This also includes a reduced font size for other buttons on the page, making them slightly smaller. But, in this listing, you'll only change the capitalization and letter spacing for the navigational links.

Listing 13.9 Adjusting size and spacing of nav menu items

```
.nav-container__inner {
  display: flex;
  justify-content: space-between;
  align-items: flex-end;
  max-width: 1080px;
  margin: 0 auto;
  padding: 0.625em 0;
}

/* ... */

.top-nav a {
  display: block;
  font-size: 0.8rem;
  padding: 0.3rem 1.25rem;
}
```

← Aligns items in the nav container to the bottom

← Decreases font size of nav links and buttons

← Changes padding values from em to rem

```

color: var(--white);
background: var(--brand-green);
text-decoration: none;
border-radius: 3px;
text-transform: uppercase;
letter-spacing: 0.03em;
}
...
.button {
display: inline-block;
padding: 0.4em 1em;
color: hsl(162, 87%, 21%);
border: 2px solid hsl(162, 87%, 21%);
border-radius: 0.2em;
text-decoration: none;
font-size: 0.8rem;
}

```

Capitalizes nav links and increases letter spacing

Decreases font size of nav links and buttons

Because you've reduced the size of the navigational links, they'll no longer fill the height of the nav-container's content box. By default, they'll align to the top, leaving more space beneath them. Aligning the nav-container's flex items to the bottom (flex-end) fixes this.

Because the font size of the navigational items has changed, their padding (previously specified in ems) would change as well. To prevent this, I've changed the units to rems. We could do the math to find the corresponding values with the new em size, but there's no compelling reason to do so.

The text-transform property might be new to you. This changes the text to all uppercase, regardless of how it is authored in the HTML. I strongly encourage you to use this rather than capitalizing text in the HTML. That way, if the design changes in the future, you can change one line of CSS without having to edit multiple places throughout all your HTML pages. But, if something should be capitalized according to the rules of grammar (such as an acronym), capitalize it in the HTML. If it's purely a design decision, do it in the CSS.

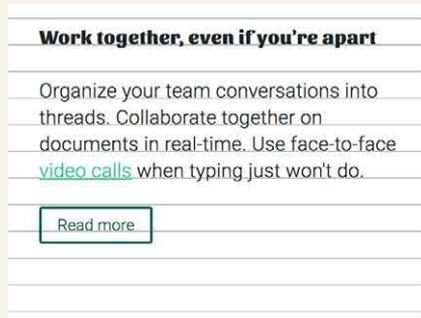
Another value for text-transform is lowercase, which makes every character lowercase. You can also set it to capitalize, which capitalizes the first letter of each word, but leaves all other characters as authored in the HTML.

Going the extra mile: vertical rhythm

In chapter 12, I discussed the importance of establishing consistent patterns in a design, including consistent spacing of elements on the screen. *Vertical rhythm* is the practice of applying this principle to lines of text throughout the page. This is done by defining a *baseline grid*, a repeating measure between lines of text. Most or all text on the page should align with this baseline grid.

(continued)

A baseline grid is illustrated in this figure with equally spaced horizontal lines. Notice how the heading, the main text, and the button text all align to this grid:



Elements with various text sizes and margins conform to a repeating vertical rhythm—the baseline grid.

Applying this principle to your site can take some work, but it can also pay off by the subtle consistency it produces. If you have an eye for detail and want to try this yourself, I recommend the article at <https://zellwk.com/blog/why-vertical-rhythms/>.

A word of warning: building a vertical rhythm typically requires using units in your `line-height` declarations. Because this changes the way the values are inherited (see chapter 2), you'll have to be sure to explicitly define an appropriate line height anywhere on the page where the font size changes.

13.5 **The dreaded FOUT and FOIT**

Before we're done with fonts, we need to consider performance. Font files are big. I've already mentioned that you should minimize the number of font files you use on the page, but even then, there can be problems. In the browser, there's usually a moment when the content and layout of the page are ready to render, but the fonts are still downloading. It's important to consider what'll happen for that brief moment.

Originally, most browser vendors decided to render the page as soon as possible, using available system fonts. Then, a moment later, the web font would finish loading, and the page would re-render with the web fonts. This is illustrated in figure 13.12.

Your web fonts will likely take up a different amount of space on screen than the system fonts. When the second render occurs, the page layout shifts and the text suddenly jumps around on the page. If it happens quickly enough after the first render, the user may not notice. But if there's a delay in font loading (or the fonts are too big), it can take as long as a few seconds to render the page. When this happens, it can be annoying to some users. The user may have already started reading content on the page, only to see it suddenly shift, making them lose their place. This became known as *FOUT*, or Flash of Unstyled Text.

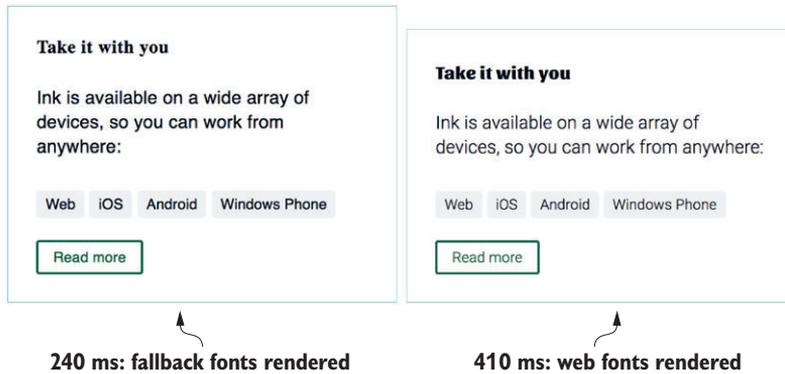


Figure 13.12 FOUT—Flash of Unstyled Text

Developers didn't like this, so most browser vendors changed the behavior of their browsers. Instead of rendering the fallback font, they rendered everything on the page except the text. More precisely, they rendered the text as invisible, so it still takes up space on the page. This way, the page's containers are put in place so the user can see the page is loading. This resulted in a new acronym: *FOIT*, for Flash of Invisible Text (figure 13.13). Background colors and borders show up, but the text only appears during the second render, when web fonts are ready.

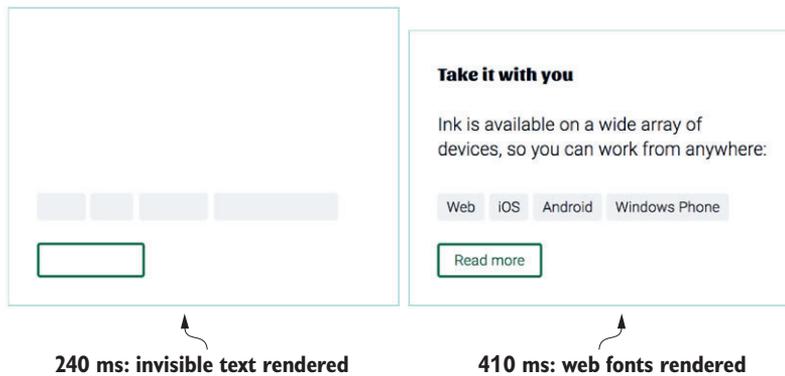


Figure 13.13 FOIT—Flash of Invisible Text

This approach solves one problem, but it creates another: What happens if the web fonts take a long time to load? Or fail to load altogether? In this case, the page remains blank, a shell of colored boxes that are entirely useless to the user. When this happens, it leaves you wanting the system font you see during FOUT.

Developers have come up with a number of approaches to address these problems. It seems every year or so, a “better” method emerges. But the simple fact of the matter is this: Both FOUT and FOIT are undesirable. And, in the world of web fonts, they’re never completely avoidable. All we can hope to do is to mitigate the problem as best we can.

Thankfully, the dust is starting to settle on this issue, so I won’t need to walk you through a half dozen different techniques. I’ll show you what I consider to be the most reasonable approach. It uses a little JavaScript to provide some control over font loading. I’ll also show you an upcoming CSS property that will eventually provide this control without the need for JavaScript. You can use either one or both of them together.

13.5.1 Using Font Face Observer

Using JavaScript, you can monitor font-loading events. This lets you take better control over the FOUT versus FOIT experience. You can use a library to take care of a lot of this for you. One I like is called Font Face Observer (<https://fontfaceobserver.com/>). This library lets you wait for the web fonts to load, then responds accordingly. What I like to do is to add a `fonts-loaded` class to the `<html>` element using JavaScript as soon as fonts are ready. You can then use this class to style the page differently, both with and without web fonts.

Download a copy of `fontfaceobserver.js` and save it into the same directory as your page. Then add the following to the end of your page, prior to the closing `</body>` tag.

Listing 13.10 Using Font Face Observer to detect font loading

```

<script type="text/javascript">
  var html = document.documentElement;
  var script = document.createElement("script");
  script.src = "fontfaceobserver.js";
  script.async = true;

  script.onload = function () {
    var roboto = new FontFaceObserver("Roboto");
    var sansita = new FontFaceObserver("Sansita");
    var timeout = 2000;

    Promise.all([
      roboto.load(null, timeout),
      sansita.load(null, timeout)
    ]).then(function () {
      html.classList.add("fonts-loaded");
    }).catch(function (e) {
      html.classList.add("fonts-failed");
    });
  };
  document.head.appendChild(script);
</script>

```

Dynamically creates a `<script>` tag to add the Font Face Observer to the page

Creates observers for both Roboto and Sansita fonts

When both fonts are loaded, adds the `fonts-loaded` class to the `<html>` element

When font loading fails, adds the `fonts-failed` class to the `<html>` element

This script creates two observers, one for each of the fonts Roboto and Sansita. The method `Promise.all()` waits for both fonts to complete loading, then the script adds the `fonts-loaded` class to the page. If loading fails, or if loading times out (after two seconds), the `catch` callback is called, which instead adds the `fonts-failed` class. Now, when the page loads, this script will add either the `fonts-loaded` or `fonts-failed` class to the page.

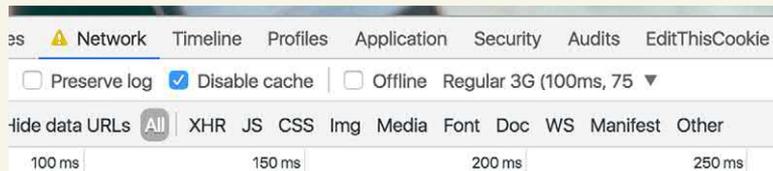
NOTE Both this code and Font Face Observer use a feature of JavaScript called promises, which aren't supported in IE. Thankfully, Font Face Observer includes a polyfill to add support. If you already use a polyfill of your own, use the standalone version of Font Face Observer available on their site.

Next, I'll show you how to use the `fonts-loaded` and `fonts-failed` classes to control how the fonts behave during loading.

Throttling your network to test font-loading behavior

If you're developing over a fast network connection, it can be difficult to test your site's font-loading behavior. One solution is to artificially slow down your download speeds in Chrome or Firefox DevTools.

In the Chrome Network tab, there's a dropdown menu in the top bar that provides several preset network speeds. You can use this to artificially slow your connection down to slower speeds by selecting the Regular 3G option in the select box as shown here:



I suggest you also check the box beside `Disable Cache`. This way, every time you load the page, all resources will be downloaded anew. This more closely mimics the initial page load of your site as a user on a slower connection would see it.

These settings only apply while you leave DevTools open. Be sure to restore these settings to normal when you're finished, so they don't catch you by surprise the next time you open DevTools.

13.5.2 Falling back to system fonts

You can take two basic approaches to font loading. First, you can apply the fallback fonts in your CSS, then using `.fonts-loaded` in a selector, change them to your desired web fonts. This'll change your browser's FOIT (invisible text) into a FOUT (unstyled text).

Second, you can apply the web fonts in your CSS, then using `.fonts-failed` in a selector, change the fonts to the fallback fonts. This'll still produce a FOIT, but it'll time out and revert to system fonts, so the page doesn't get stuck with invisible text when loading fails.

Between the two options, I generally prefer the second. But this is purely my opinion, and the “right” answer may be different depending on your preferences or even the particulars of the project you're working on. Even the exact timeout length you choose is a matter of taste.

Let's implement the second approach. The next code adds the fallback styles using the `.fonts-failed` class. Add these styles to your CSS.

Listing 13.11 Defining fallback styles so text stuck in FOIT reappears

```
body {
  margin: 0;
  font-family: Roboto, sans-serif;
  line-height: 1.4;
  letter-spacing: 0.01em;
  background-color: var(--extra-light-gray);
}
.fonts-failed body {
  font-family: Helvetica, Arial, sans-serif;
}

h1, h2, h3, h4 {
  font-family: Sansita, serif;
  letter-spacing: 0.03em;
}
.fonts-failed h1,
.fonts-failed h2,
.fonts-failed h3,
.fonts-failed h4 {
  font-family: Georgia, serif;
}
...
.home-link {
  color: var(--text-color);
  font-size: 1.6rem;
  font-family: Sansita, serif;
  font-weight: bold;
  letter-spacing: 0.03em;
  text-decoration: none;
}
.fonts-failed .home-link {
  font-family: Georgia, serif;
}
```

**If web fonts
fail to load,
falls back to
system fonts**

When the fonts fail to load (or the loading times out), the `fonts-failed` class is added to the page, and these fallback styles will be applied to the page. On a fast connection, there'll be a brief FOIT before the web fonts appear. On a slow connection, there'll be a FOIT for up to two seconds, then the fallback fonts will appear.

TIP We spent time adjusting letter spacing for our web fonts. You may want to go through the same process again with the fallback system fonts, as their spacing will likely be different. Add these spacing adjustments within the `.fonts-failed` rulesets so they only apply if the web fonts fail to load. If you want to go the extra mile, tune the fallback font so its spacing is nearly identical to the web font, so a FOUT is less noticeable. The tool at <https://meowni.ca/font-style-matcher/> can help with this.

There's no one right answer to handling font loading. If you have analytics of your site's loading times, use that to help you when deciding on an approach. A FOIT generally looks better on a fast connection, but a FOUT is preferable on a slow connection. Use your best judgment to decide between the two.

13.5.3 Getting ready for font-display

A new CSS property, `font-display`, is in the works to provide better control over font loading without the help of JavaScript. At the time of writing, it's only available in Chrome and Opera and is soon to appear in Firefox. I'll show you briefly how this works so you can be on the lookout for it in the future.

This property goes inside a `@font-face` rule. It specifies how the browser should treat web font loading. An example is shown in the next listing.

Listing 13.12 An example of the `font-display` property

```
@font-face {
  font-family: "Roboto";
  font-style: normal;
  font-weight: 300;
  src: local("Roboto Light"), local("Roboto-Light"),
       url(https://example.com/roboto.woff2) format('woff2'),
       url(https://example.com/roboto.woff) format('woff');
  font-display: swap;
}
```

← Uses the swap behavior when loading fonts: a FOUT

This tells the browser to display the fallback font immediately, then *swap* in the web font when available. In short, a FOUT.

This property also supports a few other values:

- `auto`—The default behavior (a FOIT in most browsers).
- `swap`—Displays the fallback font, then swaps in the web font when it's ready (a FOUT).
- `fallback`—A compromise between `auto` and `swap`. For a brief time (100 ms), the text will be invisible. If the web font isn't available at this point, the fallback font is displayed. Then, once the web font is loaded, it'll be displayed.
- `optional`—Similar to `fallback`, but allows the browser to decide whether to display the web font based on the connection speed. Typically, this means the web font may not appear at all on slower connections.

These options provide more control than a few lines of JavaScript can. For fast connections, `fallback` works best, providing a brief FOIT, but it'll produce a FOUT if the web font takes longer than 100 ms to load. For slow connections, `swap` is a bit better, rendering the fallback font immediately. Use `optional` in cases where the web font is a less vital part of your design.

Controlling the performance of web fonts can be complicated. For a deeper dive into the subject, check out *Web Performance in Action* by Jeremy L. Wagner (Manning, 2016). It features an entire chapter focused on web font performance, as well as chapters on other issues relevant to CSS.

Summary

- Use a font provider such as Google Fonts for easy web font integration.
- Strictly limit the number of web fonts you add to the page to keep page size under control.
- Use `@font-face` rules when hosting your own fonts.
- Take the time to adjust `line-height` and `letter-spacing` to set your page apart.
- Use Font Face Observer or other JavaScript to help control loading behavior and prevent invisible text problems.
- Keep an eye out for `font-display` support in the future.

14

Transitions

This chapter covers

- Bringing motion to the page with transitions
- Understanding timing functions and choosing the right one
- Coordinating with JavaScript

In traditional print media, things are static. Text cannot move around on paper; colors cannot shift. But the web is a living medium, where we can do so much more. Elements can fade out. Menus can slide in. Colors can shift from one value to another, and the easiest way to do any of these is with *transitions*.

With a CSS transition, you can tell the browser to “ease” one value into another when the value changes. For example, if you’ve blue links with a red hover state, a transition will cause the link to blend from blue through purple to red when the user mouses over—and back again when the user moves the mouse away. Used correctly, transitions can enhance the interactive feel of the page and, because our eyes are drawn to motion, can bring the user’s attention to a change as it occurs.

Often, transitions can be added to the page with little effort. In this chapter, we’ll look at how that’s done, along with some of the decisions you’ll have to make

in the process. Because there're some use cases where things can get a bit more complicated, we'll also take a look at how to address those issues.

14.1 From here to there

Transitions are done with the `transition-*` family of properties. If these are applied to an element while one of its property values changes, then that property will transition instead of changing immediately to the new value.

Let's build a basic example using a button, then examine how it works. It'll be a teal button with square corners that, when hovered over, transitions to a red button with rounded corners. These two states are shown in figure 14.1, along with the in-transition intermediate state.

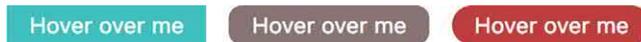


Figure 14.1 The element before, during, and after transition

Add a button to a new page and link it to a stylesheet. The markup for the button is shown here.

Listing 14.1 Adding a simple button to the page

```
<button>Hover over me</button>
```

Next, add the styles to the stylesheet. These styles define both the normal and hover states. The two transition properties instruct the browser to transition fluidly between the two.

Listing 14.2 Button styles with a transition

```
button {
  background-color: hsl(180, 50%, 50%);      ← Teal button
  border: 0;
  color: white;
  font-size: 1rem;
  padding: .3em 1em;
  transition-property: all;                  ← Transitions all
  transition-duration: 0.5s;               ← property changes
}
button:hover {
  background-color: hsl(0, 50%, 50%);      ← Hover state red button
  border-radius: 1em;                      ← with border radius
}
```

The `transition-property` property specifies which properties to transition. In this case, the special keyword `all` means to transition any properties that change. The `transition-duration` property indicates how long the transition will take before reaching the final value. In this case, `0.5s` is given, meaning 0.5 seconds.

Load the page and watch the transition take place as you mouse over the button. Notice that the `border-radius` property transitions fluidly from 0 to 1 em, even though you didn't explicitly set a border radius of zero in the non-hover state. The button has an initial value of zero automatically, and the transition works from there. Try changing other properties within the hover state, such as the `font-size` or `border`.

A transition takes place any time a property on this element is changed. This can occur on a state change like `:hover` or if JavaScript changes something, such as adding or removing a class that affects the element's styles.

Note that you didn't apply the transition properties in the `:hover` ruleset; you applied them with a selector that targets the element at all times, even though you're doing so with the hover rule in mind. You want to transition both while in the hover state (transitioning in) and after the hover state (transitioning out). While other values are changing, you typically don't want the transition properties themselves to change.

You can also use the shorthand property, `transition`. The syntax for this is shown in figure 14.2. The shorthand accepts up to four values for the four transition properties: `transition-property`, `transition-duration`, `transition-timing-function`, and `transition-delay`.

The diagram shows the CSS shorthand `transition: background-color 0.3s linear 0.5s;` with four annotations:

- Affected property:** A bracket above `background-color`.
- Duration:** A bracket below `0.3s`.
- Timing function:** A bracket above `linear`.
- Delay:** A bracket below `0.5s`.

Figure 14.2 The transition shorthand property syntax

The first value specifies which properties to transition. The initial value is the keyword `all`, which affects all properties, but if you need the transition to only apply to one property, specify that property here. For example, `transition-property: color` would apply only to the element's color, leaving other properties to change instantaneously. Or, you can apply multiple values: `transition-property: color, font-size`, for example.

The second value is the duration. This is a time value expressed in either seconds (0.3s, for example) or milliseconds (300ms).

WARNING Unlike length values, 0 isn't a valid time. You must include a unit for time values (0s or 0ms) or the declaration will be invalid and ignored by the browser.

The third value is the timing function. This controls how the intermediate values of the property are computed, effectively controlling how the rate of change accelerates or decelerates throughout the transition effect. This is either a keyword value, such as `linear` or `ease-in`, or a custom function. This is an important part of the transition that we'll look at shortly.

The final value, the delay, allows you to specify a waiting period before the transition begins to take effect after the property value changes. If you hover over a button with a 0.5 s transition delay, you'll not see the change begin until half a second after your mouse cursor enters the element.

If you need to apply two different transitions to two different properties, you can do that by adding multiple transition rules, each separated by a comma:

```
transition: border-radius 0.3s linear, background-color 0.6s ease;
```

Alternately, use the long-hand properties. The following is equivalent:

```
transition-property: border-radius, background-color;
transition-duration: 0.3s, 0.6s;
transition-timing-function: linear, ease;
```

You'll look at an example using multiple transitions later in this chapter.

14.2 Timing functions

The timing function is an important part of the transition. The transition makes a property value “move” from one value to another; the timing function says *how* it moves. Does it move at a steady speed? Does it start slowly and accelerate?

You can use several keyword values, such as `linear`, `ease-in`, and `ease-out`, to define this movement. With a `linear` transition, the value changes at a constant rate. With `ease-in`, the rate of change starts out slow, but accelerates until the end of the transition. `Ease-out` decelerates, starting with a rapid change and ending slowly. Figure 14.3 illustrates how a box would move from left to right with the various timing functions.



Figure 14.3 A linear transition moves at a steady rate, while `ease-in` accelerates and `ease-out` decelerates.

This may be a little difficult to visualize from a static image, so let's build an example to see it live in the browser. Create a new HTML page and add this code.

Listing 14.3 A simple timing function demo

```
<div class="container">
  <div class="box"></div>
</div>
```

← You'll transition this box across the screen from left to right.

Next, you'll style the box with a little color and some sizing. Then you'll absolutely position it and use a transition to move its position on hover. Add a new stylesheet to the page and copy this listing into it.

Listing 14.4 Transitioning the box from left to right

```
.container {
  position: relative;
  height: 30px;
}
.box {
  position: absolute;
  left: 0;
  height: 30px;
  width: 30px;
  background-color: hsl(130, 50%, 50%);
  transition: all 1s linear;
}
.container:hover .box {
  left: 400px;
}
```

Starts positioned to the left

Applies a transition

Moves 400 px to the right on hover

This demo should render a small green box in the top, left corner of the page. When you hover over the container, the box transitions to the right. Notice that it moves at a constant, steady speed.

WARNING This demo illustrates an element moving across the screen by absolutely positioning it and transitioning the `left` property. There are, however, performance reasons to avoid transitioning certain properties, including `left`. I'll cover these issues in the following chapter, along with a better alternative using transforms.

Now edit the transition property to see how different timing functions behave. Try ease-in (`transition: all 1s ease-in`) and ease-out (`transition: all 1s ease-out`). These keywords get the job done, but sometimes you'll want more control. You can do this by defining your own timing functions. Let's look at how to do this.

14.2.1 Understanding Bézier curves

Timing functions are based on mathematically defined Bézier curves. The browser uses these curves to calculate a property's value as a function of change over time. The Bézier curves for several timing functions are shown in figure 14.4, as well as all the keyword values that can be used as a timing function.

These curves begin at the bottom left and proceed to the top right. Time will progress to the right, and the curve represents how the value changes during that progression before arriving at the final value. The linear timing function is a steady progression throughout the duration of the transition—a straight line. The other values curve, representing acceleration and deceleration.

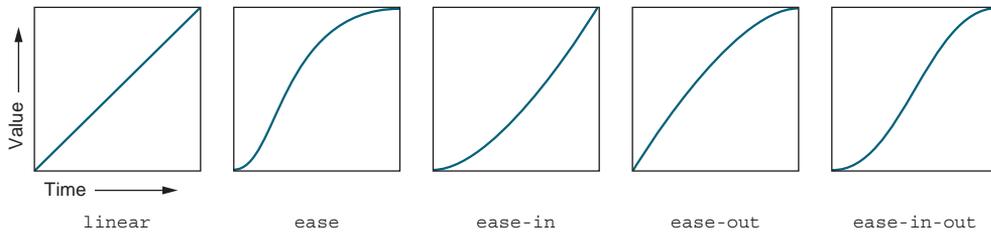


Figure 14.4 The Bézier curves of timing functions illustrate how the value changes over time.

You're not limited to these five keyword values, however. You can define your own cubic Bézier curve for more gentle or more drastic transitions. You can even add a bit of a “bounce” effect. Let's explore this.

In the page you just created, open your DevTools and inspect the green box element. You should see a small symbol beside the timing function in the Styles pane (Chrome) or Rules pane (Firefox). Click that symbol and a small popup opens, allowing you to modify the timing function's curve (figure 14.5).

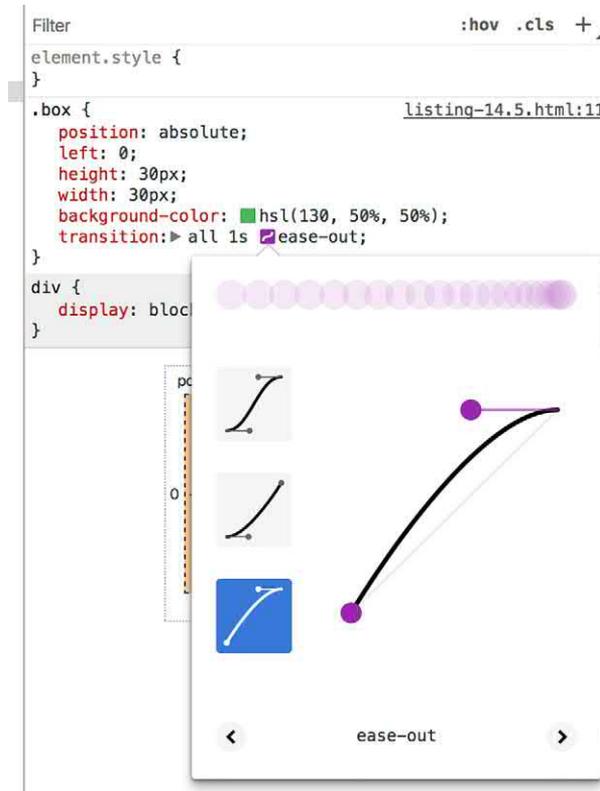


Figure 14.5 Editing a Bézier curve in the Chrome DevTools

On the left of the popup, this interface provides a series of pre-made curves to choose from. (Firefox offers far more than Chrome.) You can click a curve to select it. On the right, the selected Bézier curve is shown.

At each end of the curve is a short, straight line with circles on the end (*handles*). These are the *control points*. Click and drag one of these circles to manipulate the shape of the curve. Notice how the length and direction of the handle “pulls” the curve.

Click outside of this popup to close it, and you’ll see that the timing function has been updated. Instead of a keyword like `ease-out`, it’ll now be something like `cubic-bezier(0.45, 0.05, 0.55, 0.95)`. This `cubic-bezier()` function and the four values within define the custom timing function.

Selecting a timing function

Whether you use keyword timing functions or custom Bézier curves, it’s helpful to know when to use which. Each site or application should have a decelerating curve (`ease-out`), an accelerating curve (`ease-in`), as well as the `linear` keyword. It’s best to re-use the same few curves to provide a more consistent user experience.

You can use each of the three functions in the following scenarios:

- *Linear*—Color changes and fade in/out effects.
- *Decelerating*—User-initiated changes. When the user clicks a button or hovers over an element, use `ease-out` or something similar. This way, the user will see a fast, instant response to their input, easing out as the element comes to a stop.
- *Accelerating*—System-initiated changes. When content finishes loading or a timeout event triggers, use `ease-in` or something similar. This way, the element will ease in at first to draw the user’s attention before the element speeds up and completes its motion.

These are soft rules. They provide a good starting place, but don’t be afraid to break them if something doesn’t “feel” right. Occasionally, you’ll also want a fourth curve for larger or more playful motions: use either an `ease-in-out` (accelerate then decelerate) or a bounce effect (see chapter 15 for an example of a bounce).

Let’s take a closer look at how `cubic-bezier()` works. Another example curve is shown in figure 14.6.

This figure shows a custom Bézier curve. This curve accelerates at the beginning, proceeds the fastest in the middle (the steepest part of the curve), then decelerates at the end. The curve exists on a Cartesian grid. It begins at point (0, 0) and ends at point (1, 1).

With the end points set, the position of the two handles is all you need to define the curve. In CSS, this curve can be defined as `cubic-bezier(0.45, 0.05, 0.55, 0.95)`. The four values represent the x- and y-coordinates of the two handles’ control points.

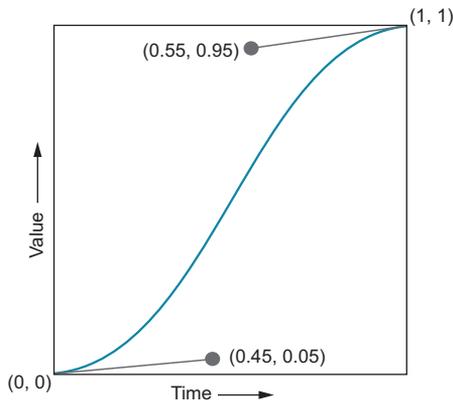


Figure 14.6 A Bézier curve representing a timing function

Curves are hard to visualize from these numbers alone. Editing them via a GUI is much more intuitive, so I like to edit and test a transition in the browser before copying the resulting cubic Bézier to my stylesheet. I prefer the DevTools for this, but you can also use an online resource such as <http://cubic-bezier.com/>.

14.2.2 Steps

One last type of timing function uses the `steps()` function. Instead of providing a fluid, Bézier-based transition from one value to the next, this moves it in a number of discrete, instant “steps.”

The function takes two parameters: the number of steps and a keyword (either `start` or `end`), indicating whether each change should take place at the start or end of each step. Some step functions are illustrated in figure 14.7.

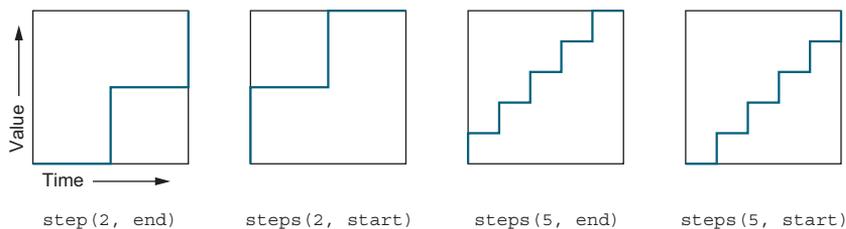


Figure 14.7 The `steps()` function changes the value incrementally.

Note that `end` is the default value for the second parameter, so `steps(3)` can be used in place of `steps(3, end)`. To see steps in action, edit your stylesheet to match this listing.

Listing 14.5 Using `steps()` to increment the value

```
.box {
  position: absolute;
  left: 0;
  height: 30px;
  width: 30px;
  background-color: hsl(130, 50%, 50%);
  transition: all 1s steps(3);
}
```

Transitions in three discrete steps

Now, instead of moving fluidly from left to right for one second (the transition duration), the time is divided into thirds, or three steps. For each step, the box appears at the starting position, the one-third position, then the two-thirds position, before moving to the final position at the 1-second mark.

NOTE By default, the property’s value changes at the end of each step, so the transition doesn’t begin immediately. You can change this behavior so the changes take place at the beginning of each step rather than at the end by adding the `start` keyword: `steps(3, start)`.

Practical uses for `step()` are uncommon, but there’s a clever list of ideas at <https://css-tricks.com/clever-uses-step-easing/>.

14.3 Non-animatable properties

Many transitions are straightforward. For example, you can apply `transition: color 200ms linear` to links so they fade from one color to another when hovered over or clicked. Or, you can transition the background color of a clickable tile or the padding of a button.

If JavaScript changes something on the page, you might want to consider whether adding a transition is appropriate. In some cases, this is as simple as adding a transition property to that element. In other cases it can require a little more orchestration. For the rest of this chapter, you’ll build a dropdown menu and apply transitions so it opens seamlessly rather than snapping into view.

First, you’ll make it fade in, transitioning its `opacity` value. After that, you’ll change the dropdown to use a different effect, transitioning its `height`. Both of these effects introduce particular problems that require a little more thought.

The menu will look like figure 14.8. You’ll start by getting the menu opening and closing. After that, you’ll add in the transition effects. I’ve included a link beneath the



Figure 14.8 The dropdown menu in its closed (left) and open (right) states

menu. Notice how the menu's drawer appears in front of this link when it's open; this'll be important.

Create a new page for the dropdown, adding the markup shown next. This is similar to the dropdown menus you've built in previous chapters and includes some JavaScript to toggle the menu's open and closed states.

Listing 14.6 Transitional dropdown menu

```
<div class="dropdown" aria-haspopup="true">
  <button class="dropdown__toggle">Menu</button>
  <div class="dropdown__drawer">
    <ul class="menu" role="menu">
      <li role="menuitem">
        <a href="/features">Features</a>
      </li>
      <li role="menuitem">
        <a href="/pricing">Pricing</a>
      </li>
      <li role="menuitem">
        <a href="/support">Support</a>
      </li>
      <li role="menuitem">
        <a href="/about">About</a>
      </li>
    </ul>
  </div>
</div>
<p><a href="/read-more">Read more</a></p>
```

The drawer that will appear and disappear to reveal the menu

A link that will appear below the dropdown

```
<script type="text/javascript">
  (function () {
    var toggle = document.getElementsByClassName('dropdown__toggle')[0];
    var dropdown = toggle.parentElement;
    toggle.addEventListener('click', function (e) {
      e.preventDefault();
      dropdown.classList.toggle('is-open');
    });
  })();
</script>
```

Toggles the is-open class on the container when the button is clicked

The styles before adding the fade-in effect are shown in the next listing. Add these to a stylesheet and link it to the page. I've added a few transitional effects so colors transition smoothly on hover. Other than that, there's not a lot new here yet, but this gets the page set up so you can focus on creating the fade-in effect.

Listing 14.7 Transitional dropdown menu styles

```
body {
  font-family: Helvetica, Arial, sans-serif;
}

.dropdown__toggle {
  display: block;
```

```

padding: 0.5em 1em;
border: 1px solid hsl(280, 10%, 80%);
color: hsl(280, 30%, 60%);
background-color: white;
font: inherit;
text-decoration: none;
transition: background-color 0.2s linear;
}
.dropdown__toggle:hover {
  background-color: background-color: hsl(280, 15%, 95%);
}
.dropdown__drawer {
  position: absolute;
  display: none;
  background-color: white;
  width: 10em;
}
.dropdown.is-open .dropdown__drawer {
  display: block;
}

.menu {
  padding-left: 0;
  margin: 0;
  list-style: none;
}
.menu > li + li > a {
  border-top: 0;
}
.menu > li > a {
  display: block;
  padding: 0.5em 1em;
  color: hsl(280, 40%, 60%);
  background-color: white;
  text-decoration: none;
  transition: all .2s linear;
  border: 1px solid hsl(280, 10%, 80%);
}
.menu > li > a:hover {
  background-color: hsl(280, 15%, 95%);
  color: hsl(280, 25%, 10%);
}

```

Transitions the background color when it changes

Changes the background color on hover

Transitions background and text colors

Changes the colors on hover

Open this page in your browser and try it out. You should be able to open and close the menu by clicking on the toggle button. Notice how both the button and the menu links transition their colors smoothly when you hover your mouse over them, and again when you move the mouse away.

I used a transition duration of 0.2 seconds for these hover effects. As a rule of thumb, most of your transitions should be somewhere between 200 and 500 ms. Any longer, and users will perceive your page as slow and feel as if they are waiting unnecessarily for the page to respond. This is doubly true for effects the user will see often and repeatedly.

TIP Use quick transition speeds for hover effects, fades, and small scaling effects. Keep these below 300 ms; you may even want to go as low as 100 ms in some instances. For transitions that involve large moves or complex timing functions, such as bounces (see chapter 15), use slightly longer transitions between 300 and 500 ms.

When I'm working on a transition, I sometimes slow it down to two or three full seconds. This way I can watch carefully what it's doing and ensure that it behaves the way I want. If you do this, be sure to set it back to a nice short speed after you're done.

14.3.1 *Properties that cannot be animated*

Not all properties can be animated. The `display` property is one of these. You can only toggle it between `display: none` and `display: block`; you can't transition between values, so any transition properties applied to `display` are ignored.

If you look up a property in a reference guide such as MDN (<https://developer.mozilla.org/en-US/>), it'll typically tell you whether you can animate a property and what type of value (for example, length, color, percent) can be interpolated. The details for the `background-color` property from <https://developer.mozilla.org/en-US/docs/Web/CSS/background-color> are shown in figure 14.9.

Initial value	<code>transparent</code>
Applies to	all elements. It also applies to <code>::first-letter</code> and <code>::first-line</code> .
Inherited	no
Media	visual
Computed value	If the value is translucent, the computed value will be the <code>rgba()</code> corresponding one. If it isn't, it will be the <code>rgb()</code> corresponding one. The <code>transparent</code> keyword maps to <code>rgba(0,0,0,0)</code> .
Animation type	a <code>color</code>
Canonical order	the unique non-ambiguous order defined by the formal grammar

Figure 14.9 MDN documentation provides a technical summary box for each property.

The `background-color` property, as shown in the figure, can be animated only as a color value, meaning from one color to another (which makes sense, as this property must be set as a color). A property's Animation Type applies both to transitions as well as animations, which you'll use in chapter 16. The documentation also lists other useful information about the property, such as its initial value, what sort of elements it

can be applied to, and whether it's inherited. If you need a good technical summary of how a property can be used, find the property in the MDN documentation and look at its properties box.

NOTE Most properties that accept a length, number, color, or the function `calc()` can be animated. Most properties that take a keyword or other discrete values, like `url()`, can't.

If you were to look up the `display` property, you'd see that its animation type is discrete, meaning it can only be changed between discrete values, and it can't be interpolated in an animation or transition. If you want to fade an element in or out, you can't transition the `display` property; but, you can use the `opacity` property.

14.3.2 Fading in and out

Next, let's use an opacity transition to give our menu a fade in and fade out effect as it opens and closes. The result will be like that shown in figure 14.10.



Figure 14.10 Menu fading in

The `opacity` property can be any value between 0 (fully invisible) and 1 (fully opaque). The next listing shows the basic idea. This alone won't work, however, for reasons you'll soon see. Go ahead and edit your stylesheet to match.

Listing 14.8 Adding opacity and transition rules

```
.dropdown_drawer {
  position: absolute;
  background-color: white;
  width: 10em;
  opacity: 0;
  transition: opacity 0.2s linear;
}
.dropdown.is-open .dropdown_drawer {
  opacity: 1;
}
```

Replaces display:
none with opacity: 0

Transitions
the opacity

Replaces display:
block with opacity 1

Now the menu fades in and out as you open and close it. Unfortunately, the menu isn't gone when it's closed—it's fully transparent, but it's still present on the page.

And, if you try to click the Read more link, it won't work. Instead, you'll click the transparent menu item in front of it, taking you to the Features page.

You need to transition the opacity, but also fully remove the menu drawer when it's not visible. You can do this with the help of another property, `visibility`.

The `visibility` property lets you remove an element from the page, similar to the `display` property. You can give it the values `visible` or `hidden`. Unlike `display`, `visibility` is animatable. Transitioning it doesn't make it fade, but it'll obey a `transition-delay`, where the `display` property won't.

NOTE Applying `visibility: hidden` to an element removes it from the visible page, but it doesn't remove it from the document flow; meaning it'll still take up space. Other elements will continue to flow around its position, leaving an empty area on the page. In our case, this doesn't affect the menu because we've also applied absolute positioning.

You'll take advantage of `visibility`'s ability to animate with a little trick. Change your CSS to match the code in this listing, then I'll walk you through how it works.

Listing 14.9 Using a transition delay to manipulate when `visibility` changes

```
.dropdown_drawer {
  position: absolute;
  background-color: white;
  width: 10em;
  opacity: 0;
  visibility: hidden;
  transition: opacity 0.2s linear,
              visibility 0s linear 0.2s;
}
.dropdown.is-open .dropdown_drawer {
  opacity: 1;
  visibility: visible;
  transition-delay: 0s;
}
```

Hidden and transparent when closed

Delays the visibility transition 0.2s

Visible and fully opaque when opened

Removes the transition delay while the is-open class is applied

Here, you've split the transition into two sets of values. This defines the fade-out behavior. The first set of values transitions the opacity for 0.2 seconds. The second set transitions the visibility for 0 seconds (an instant) after 0.2 seconds of delay. This means the opacity transitions first and when it ends, the visibility transitions. This enables the menu to fade out slowly, then, when it's fully transparent, the visibility switches to hidden. The user can then click the Read more link without the menu interfering.

When the menu fades in, you'll need the order to be different: visibility needs to toggle on immediately, followed by an opacity transition. This is why, in the second ruleset, you changed the transition delay to `0s`. This way, visibility is hidden while the menu is closed, but visible throughout both the fade-in and fade-out transitions.

TIP You can use the JavaScript `transitionend` event to perform an action after a transition completes.

This fading effect can also be achieved with some JavaScript instead of the `transition-delay`, but I find this takes more code and can be prone to error. Sometimes, however, JavaScript will be necessary to achieve a desired effect (which you'll see in a moment), but if a transition or animation can be accomplished through CSS alone, that's almost always preferable.

14.4 Transitioning to auto height

Let's repurpose the dropdown menu to use a different effect that's also common: sliding open and closed with a transitioned height. This effect is illustrated in figure 14.11.



Figure 14.11 Slide open the element by transitioning its height.

When the menu opens, it transitions from a height of zero to its natural height (auto). When it closes, it transitions back to zero. The general idea is shown in the following listing. Unfortunately, it doesn't work. Change this portion of your CSS to match these rules, then we'll look at what the problem is and how you can address it.

Listing 14.10 Transitioning the height value

```
.dropdown_drawer {
  position: absolute;
  background-color: white;
  width: 10em;
  height: 0;
  overflow: hidden;
  transition: height 0.3s ease-out;
}
.dropdown.is-open .dropdown_drawer {
  height: auto;
}
```

Closed drawer has no height and overflow is hidden.

Transitions the height

Height of the open drawer is determined by its contents.

Overflow is hidden to cut off the contents of the drawer when it's closed or transitioning. But this doesn't work because a value cannot transition from a length (0) to auto.

You can set a height explicitly, instead, to something like 120 px, but the problem is you don't know exactly what the height should be. That's only known once the contents are set and rendered in the browser, so you'll have to use JavaScript to determine the height.

After the page loads, you'll access the DOM element's `scrollHeight` property. This'll give you the appropriate value for its height. You'll then change the code slightly to set the element's height to this value. Edit the script on your page to match the next listing.

Listing 14.11 Setting the height explicitly so the transition works

```
(function () {
  var toggle = document.getElementsByClassName('dropdown__toggle')[0];
  var dropdown = toggle.parentElement;
  var drawer = document.getElementsByClassName('dropdown__drawer')[0];
  var height = drawer.scrollHeight;

  toggle.addEventListener('click', function (e) {
    e.preventDefault();
    dropdown.classList.toggle('is-open');
    if (dropdown.classList.contains('is-open')) {
      drawer.style.setProperty('height', height + 'px');
    } else {
      drawer.style.setProperty('height', '0');
    }
  });
})();
```

← Gets the computed auto height of the drawer

← Sets the height explicitly to open

← Restores the height to zero to close

Now, in addition to toggling the `is-open` class, you also explicitly specified a height in pixels so the element transitions to the correct height. You then set the height back to zero upon closing so the menu can transition back.

WARNING An element's `scrollHeight` property is equal to 0 if the element is hidden using `display: none`. If this is ever the case for you, you can set the display to `block` (`el.style.display = 'block'`), access the `scrollHeight`, then restore the display value (`el.style.display = 'none'`).

Transitions sometimes require coordination between the CSS and the JavaScript. It may be tempting in some cases to move the logic entirely into JavaScript. For instance, you could reproduce the height transition by repeatedly setting a new height in JavaScript alone. But you should generally let CSS do as much of the heavy lifting as possible. It's better optimized in the browser (and, therefore, more performant) and provides some features like easing that can take a lot of code to mimic by hand.

You aren't done with transitions just yet. They'll be useful in conjunction with transforms in the next chapter.

Summary

- You can use transitions to smooth sudden changes in the page.
- To catch the user's attention, use an accelerating motion.
- To show the user that their action has taken effect, use a decelerating motion.
- You can use JavaScript to coordinate transitions with class name changes when CSS alone cannot do what you need.

15

Transforms

This chapter covers

- Manipulating elements using transforms for performant transitions and animations
- Adding a “bounce” effect to a transition
- The browser’s rendering pipeline
- Looking at 3D transforms and perspective

In this chapter, we’ll explore the `transform` property, which you can use to change or distort the shape or position of an element on the page. This can involve rotating, scaling, or skewing the element in two or three dimensions. Transforms are most commonly used in conjunction with transitions or animations, which is why I’ve sandwiched this chapter between those two topics. In these last two chapters, you’ll build a page that makes heavy use of transitions, transforms, and animations.

First, I’ll walk you through applying transforms to a static element. This’ll give you a grasp on how they work in isolation before we add them into some transitions. Then you’ll build a small but complex menu with multiple transforms and transition effects. Finally, we’ll take a look at working in 3D and utilizing perspective. This will carry over into the next chapter, where we’ll use 3D transforms in conjunction with animation.

15.1 Rotate, translate, scale, and skew

A basic transform rule looks something like this:

```
transform: rotate(90deg);
```

This rule, when applied to an element, rotates it 90 degrees to the right (clockwise). The transform function `rotate()` specifies how the element is to be transformed. You'll find several other transform functions, but they generally all fall into one of four categories (illustrated in figure 15.1).

- *Rotate*—Spins the element a certain number of degrees around an axis
- *Translate*—Moves the element left, right, up, or down (similar to relative positioning)
- *Scale*—Shrinks or expands the element
- *Skew*—Distorts the shape of the element, sliding its top edge in one direction and its bottom edge in the opposite direction

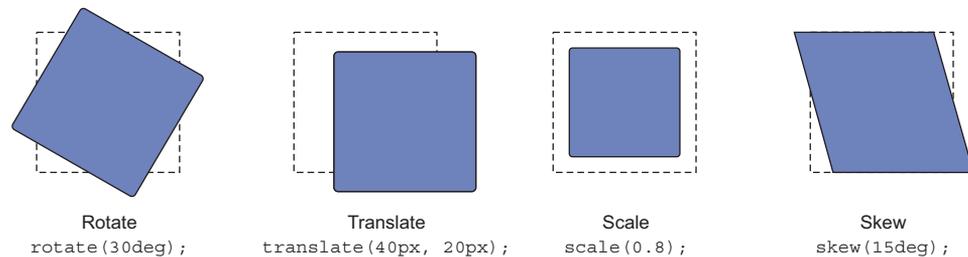


Figure 15.1 The four basic types of transform (a dashed line represents the initial elements' positions)

Each transform is applied using the corresponding function as a value of the `transform` property. Let's create a simple example to try these out in your browser. This'll be a card with an image and text (figure 15.2), which you can apply transforms to.

Create a new page and stylesheet and link them. Add the HTML shown here.

Listing 15.1 Creating a simple card

```
<div class="card">
  
  <h4>Mrs. Featherstone</h4>
  <p> She may be a bit frumpy, but Mrs Featherstone gets the job done. She
    lays her largish cream-colored eggs on a daily basis. She is gregarious
    to a fault.</p>
  <p>This Austra White is our most prolific producer.</p>
</div>
```



Figure 15.2 A basic card with a rotate transform applied

Next, in the stylesheet, add the CSS in the following listing. This includes a few base styles, colors, and a card with a rotation transform applied.

Listing 15.2 Styling a card and applying a transform

```
body {
  background-color: hsl(210, 80%, 20%);
  font-family: Helvetica, Arial, sans-serif;
}

img {
  max-width: 100%;
}

.card {
  padding: 0.5em;
  margin: 0 auto;
  background-color: white;
  max-width: 300px;
  transform: rotate(15deg);
}
```

Centers the card

Rotates the card 15 degrees to the right

Load this into your browser and you'll see the card rotated. Experiment with this a bit to get a feel for how the `rotate()` function behaves. Use a negative angle to rotate the card left (for example, try `rotate(-30deg)`).

Next, try changing the transform to some of the other functions. Use the following values and observe how they each behave:

- `skew(20deg)`—Skews the card 20 degrees. Try a negative angle to skew in the other direction.
- `scale(0.5)`—Shrinks the card to half of its initial size. The `scale()` function takes a unitless number. Values less than 1 shrink the element; values greater than 1 expand it.
- `translate(20px, 40px)`—Shifts the element 20 pixels right and 40 pixels down. Again, you can use negative values to transform in the opposite direction.

One thing to note when using transforms is that, while the element may be moving to a new position on the page, it doesn't shift the document flow. You can translate an element all the way across the screen, but its original location remains unoccupied by other elements. Also, when rotating an element, a corner of it may shift off the edge of the screen. Similarly, it could potentially cover up portions of another element beside it (figure 15.3).

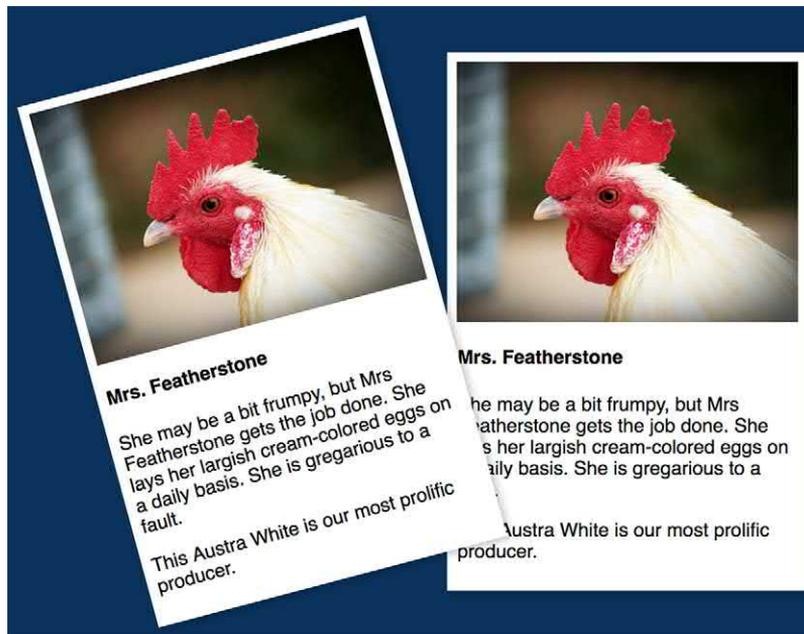


Figure 15.3 Transforming one element doesn't cause other elements to move, so they might overlap.

In some cases, I find it's helpful to set plenty of margin for one or both of the elements to prevent unwanted overlapping.

WARNING Transforms cannot be applied to inline elements like `` or `<a>`. To transform such an element, you must either change the `display` property to something other than `inline` (such as `inline-block`) or change the element to a flex or grid item (apply `display: flex` or `display: grid` to the parent element).

15.1.1 Changing the transform origin

A transform is made around a *point of origin*. This point serves as the axis of rotation, or the spot where scaling or skewing begins. This means the origin point of the element remains locked in place, while the rest of the element transforms around it (though this doesn't apply to `translate()` as the whole element moves during a translation).

By default, the point of origin is the center of the element, but you can change this with the `transform-origin` property. Figure 15.4 shows some elements transformed around different points of origin.

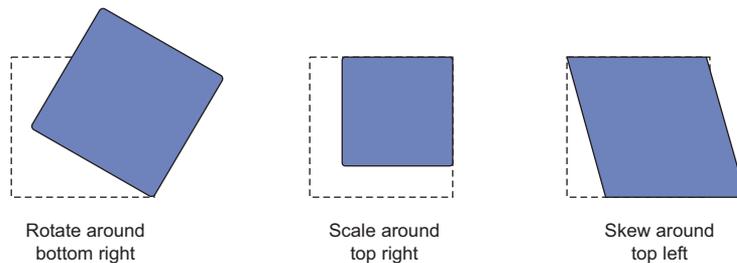


Figure 15.4 Rotate, scale, and skew made with the transform origin at various corners of the element.

For the element on the left, the rotation pivots about the origin, which is set using `transform-origin: right bottom`. The element in the middle scales toward the origin (right top). And the element on the right skews in such a way that its origin (left top) remains in place while the rest of the element stretches away.

The origin can also be specified in percentages, measured from the top left of the element. The following two declarations are equivalent:

```
transform-origin: right center;  
transform-origin: 100% 50%;
```

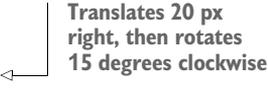
NOTE You can also use a length to specify the origin in pixels, ems, or another unit. Though, in my experience, the keywords `top`, `right`, `bottom`, `left`, and `center` are all you'll need in most instances.

15.1.2 Applying multiple transforms

You can specify multiple values for the `transform` property, each separated by a space. Each transform value is applied in succession from right to left. If you apply `transform: rotate(15deg) translate(15px, 0)`, the element is translated 15 px to the right, then rotated 15 degrees clockwise. Edit your stylesheet to work with this a bit.

Listing 15.3 Applying multiple transforms

```
.card {
  padding: 0.5em;
  margin: 0 auto;
  background-color: white;
  max-width: 300px;
  transform: rotate(15deg) translate(20px, 0);
}
```



It's probably easiest to see this effect if you open your browser's DevTools and manipulate the values live to see how they affect the element. Notice that changing the values for `translate()` seems to move the element along a diagonal axis, rather than the normal cardinal directions; this is because the rotation takes place after the translation.

This can be a little tricky to work with. I generally find it's easier to do `translate()` manipulations last chronologically (first in source order for `transform`), so I can work with the normal left/right, up/down coordinates. To see this, reverse the order to `transform: translate(20px, 0) rotate(15deg)`.

15.2 Transforms in motion

Transforms by themselves aren't all that practical. A box with a `skew()` applied may look interesting, but it's not exactly easy to read. But when used in conjunction with motion, transforms become much more useful.

Let's build a page that makes use of this concept. A screenshot of the page you'll make is shown in figure 15.5. You'll be adding a lot of motion to this page.

In this section, you'll build the navigational (`nav`) menu on the left. Initially, it appears as just four icons stacked vertically, but, upon hover, the text for the menu appears. This example will include several transitions and a couple transforms. Let's get the page set up, then we'll take a closer look at the `nav` menu. (In the next chapter, you'll build the main cards section in the center and add more transforms and some animation to it.)

Create a new page and a new stylesheet named `style.css` and add the following markup. This markup includes a link to two web fonts (Alfa Slab One and Raleway) from the Google Fonts API. It also has the markup for the page header and the `nav` menu.

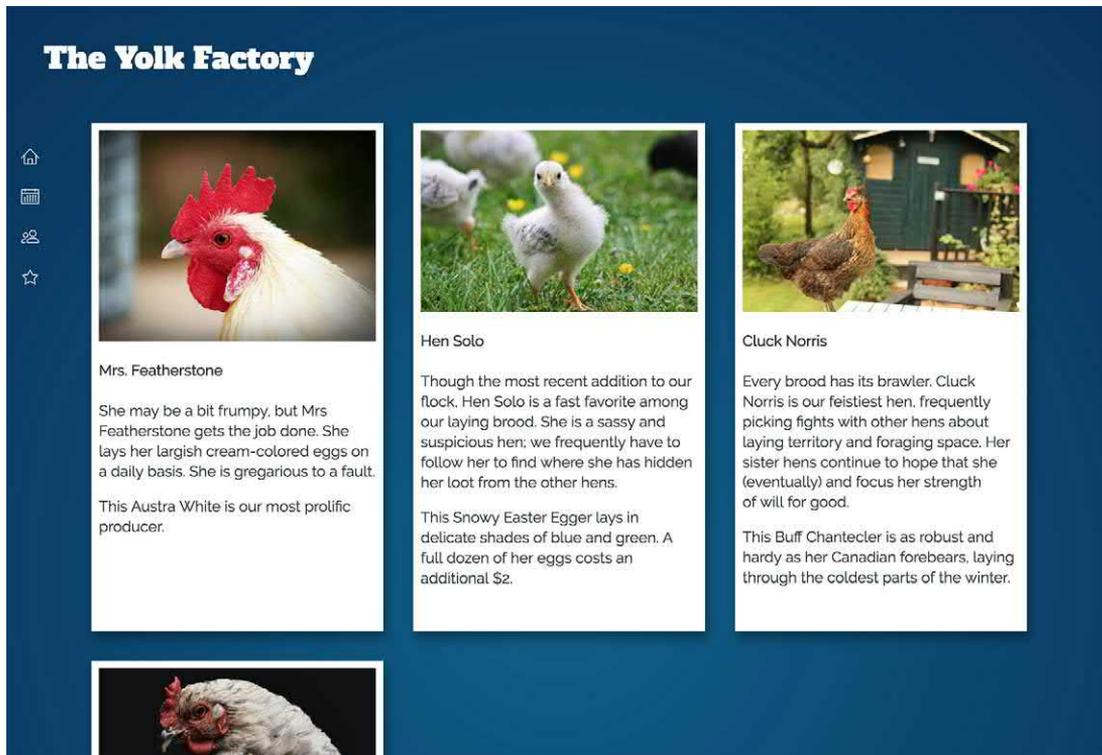


Figure 15.5 The menu icons on the left will feature several transforms and transitions.

Listing 15.4 Page markup for transforms in motion

```
<!doctype html>
<html lang="en">
  <head>
    <title>The Yolk Factory</title>
    <link
      href="https://fonts.googleapis.com/css?family=Alfa+Slab+One|Raleway"
      rel="stylesheet">
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <header>
      <h1 class="page-header">The Yolk Factory</h1>
    </header>
    <nav class="main-nav">
      <ul class="nav-links">
        <li>
          <a href="/">
```

← Adds Alfa Slab One and Raleway fonts to the page

Nav links
each contain
an image
and a label.

```

    
    <span class="nav-links__label">Home</span>
  </a>
</li>
<li>
  <a href="/events">
    
    <span class="nav-links__label">Events</span>
  </a>
</li>
<li>
  <a href="/members">
    
    <span class="nav-links__label">Members</span>
  </a>
</li>
<li>
  <a href="/about">
    
    <span class="nav-links__label">About</span>
  </a>
</li>
</ul>
</nav>
</body>
</html>

```

The nav element contains the largest part of this markup. It includes an unordered list () of links. Each link consists of an icon image and a text label. Notice that the icon images here are in *SVG* format. This'll become important later on. You'll add more content to the page when you're ready to style it in the next chapter.



SVG—Short for Scalable Vector Graphics. This is an XML-based image format that defines an image using vectors. Because the image is mathematically defined, it can scale up and down to any size. SVG is broadly supported in all browsers.

Next, you'll add some base styles, including a background gradient and padding around the main heading. You'll also apply the web fonts to the page. Copy or add the following listing into your stylesheet. These are just the base styles and page-header; you'll work on laying out the menu next.

Listing 15.5 Base styles and heading

```

html {
  box-sizing: border-box;
}
*,
*::before,

```

```

*::after {
  box-sizing: inherit;
}

body {
  background-color: background-color: hsl(200, 80%, 30%);
  background-image: radial-gradient(hsl(200, 80%, 30%),
                                   hsl(210, 80%, 20%));
  color: white;
  font-family: Raleway, Helvetica, Arial, sans-serif;
  line-height: 1.4;
  margin: 0;
  min-height: 100vh;
}

h1, h2, h3 {
  font-family: Alfa Slab One, serif;
  font-weight: 400;
}

main {
  display: block;
}

img {
  max-width: 100%;
}

.page-header {
  margin: 0;
  padding: 1rem;
}

@media (min-width: 30em) {
  .page-header {
    padding: 2rem 2rem 3rem;
  }
}

```

Deep blue background gradient

Ensures that the body fills the viewport so the gradient fills the screen

Smaller padding for the header on mobile viewports

Larger padding for the header on larger screens

This example uses a number of concepts from earlier chapters. I've used a radial gradient for the body background. This adds a nice bit of depth to the page. (The background-color provides a fallback value for Opera Mini, which doesn't support radial gradients.) The web font Alfa Slab One is applied to the headings and Raleway to the body copy. I've also provided responsive styles for the page header using a media query, adding a larger padding when the screen size can afford it.

We'll take the menu in several stages. First, let's get the menu laid out and then provide some responsive behavior. You'll do this with a mobile first approach (chapter 8), so let's start with the small viewport. The heading and menu should look like figure 15.6.

Because you want to lay out the navigational links horizontally for smaller viewports, an approach using a flexbox makes sense. You can evenly space the navigational



Figure 15.6 Mobile design for the nav menu

items across the width of the page by applying `align-content: space-between` to the flex container. Next, you'll set font colors and align the icons. Add the following listing to your stylesheet.

Listing 15.6 Mobile styles for the nav menu links

```
.nav-links {
  display: flex;
  justify-content: space-between;
  margin-top: 0;
  margin-bottom: 1rem;
  padding: 0 1rem;
  list-style: none;
}
.nav-links > li + li {
  margin-left: 0.8em;
}
.nav-links > li > a {
  display: block;
  padding: 0.8em 0;
  color: white;
  font-size: 0.8rem;
  text-decoration: none;
  text-transform: uppercase;
  letter-spacing: 0.06em;
}
.nav-links__icon {
  height: 1.5em;
  width: 1.5em;
  vertical-align: -0.2em;
}
.nav-links > li > a:hover {
  color: hsl(40, 100%, 70%);
}
```

Uses a flexbox to spread the nav items across the screen horizontally

Styles the link text

Moves icons down slightly to center them with the text labels

You'll keep the menu like this on smaller viewports. But on larger screens, you can layer on more effects. For the desktop layout, you'll dock it to the left-hand side of the screen using fixed positioning. This'll look like figure 15.7.

This menu is built from two modules: I've named the outer element `main-nav` and the inner structure `nav-links`. The `main-nav` serves as the container, which you'll position to the left. The `main-nav` also provides the dark background. Let's get this into place.

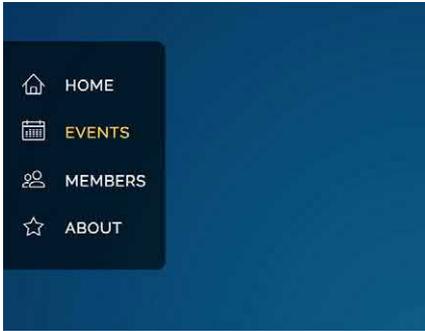


Figure 15.7 The nav menu docked on the left side of the screen for large viewports.

Add the next listing to your stylesheet; make sure the second media query and its contents are placed after the existing main-nav styles so they can override the mobile styles where necessary.

Listing 15.7 Positioning the menu for large viewports

```

@media (min-width: 30em) {
  .main-nav {
    position: fixed;
    top: 8.25rem;
    left: 0;
    z-index: 10;
    background-color: transparent;
    transition: background-color .5s linear;
    border-top-right-radius: 0.5em;
    border-bottom-right-radius: 0.5em;
  }

  .main-nav:hover {
    background-color: rgba(0, 0, 0, 0.6);
  }
}

/* ... */

@media (min-width: 30em) {
  .nav-links {
    display: block;
    padding: 1em;
    margin-bottom: 0;
  }
  .nav-links > li + li {
    margin-left: 0;
  }
  .nav-links__label {
    margin-left: 1em;
  }
}

```

← Applies styles only to medium and larger screens

← Ensures the nav shows in front of other content added to the page later

← Initially leaves the background color transparent

← Adds a transition effect to the background

← Applies a dark semi-transparent background on hover

← Overrides the flexbox from mobile styles to make links stack vertically

The `position: fixed` declaration puts the menu into place and keeps it there, even as the page scrolls. The `display: block` rule overrides the `display: flex` from the mobile styles, causing the menu items to stack atop one another.

Now let's start layering in some transition and transform effects. For that, you'll do three things:

- 1 Scale up the icon size while a link is hovered.
- 2 Hide the link labels, then make them all appear with a fade-in transition when the user hovers over the menu.
- 3 Use a translate to add a "fly in" effect to the link label in conjunction with the fade-in.

Let's take these in each in turn.

15.2.1 Scaling up the icon

Look at the structure of the navigational links. Each list item contains a link (`<a>`), which in turn contains an icon and a label:

```
<li>
  <a href="/">
    
    <span class="nav-links__label">Home</span>
  </a>
</li>
```

NOTE The list items, in conjunction with the parent ``, is a much larger, more deeply nested module than I prefer to make. I'd typically look for a way to split it up into smaller modules, but we'll need to keep it all together in order to achieve some of these effects.

First, let's scale up the icon on hover. You'll do this with a scale transform, then apply a transition to it so the change happens seamlessly. In figure 15.8, the Events menu item is moused over, and its calendar icon is scaled up slightly.



Figure 15.8 Default icon size (left). Hovering over a link causes its icon to scale up (right).

The Events image has a set height and width, so you could make it larger by increasing these properties. But, this would cause some other elements to move around as the document flow would get recalculated.

By using a transform instead, the elements around it aren't affected, and the Events label doesn't shift to the right. Update your CSS to add this effect when the element is either hovered or focused.

Listing 15.8 Scaling up the icon when its link is hovered or focused

```
@media (min-width: 30em) {

  .nav-links {
    display: block;
    padding: 1em;
    margin-bottom: 0;
  }
  .nav-links > li + li {
    margin-left: 0;
  }
  .nav-links__label {
    margin-left: 1em;
  }

  .nav-links__icon {
    transition: transform 0.2s ease-out;
  }

  .nav-links a:hover > .nav-links__icon,
  .nav-links a:focus > .nav-links__icon {
    transform: scale(1.3);
  }
}
```

← | **Transitions the transform property**

← | **Scales up the icon size**

Now, as you swipe your mouse across the menu items, you'll see the icons grow a bit to help indicate which item you're hovering over. I intentionally chose to use SVG image assets here, so there's no pixelation or other odd distortions when the image size changes. The `scale()` transform is a perfect way to do this.

SVG: a better approach to icons

Icons are an important part of some designs. The techniques used for icons have evolved and, for a long time, the best practice was to put all your icons into a single image file, called a *sprite sheet*. Then—using a CSS background image and some careful sizing and background positioning—display one icon from the sprite sheet in an element.

Next, *icon fonts* became popular. Instead of embedding sprites in an image, this approach involves embedding each icon as a character in a custom-made font file. Using web fonts, a single character would render as an icon. Services like Font-Awesome (<http://fontawesome.io/>) provide hundreds of general-use icons to make this easy.

These techniques still work, but I encourage you to make the switch to SVG icons. SVG is much more versatile and more performant. You can use an SVG as an `` source, as you've done in this chapter, but SVG provides other options as well. You can create an SVG sprite sheet, or because SVG is an XML-based file format, you can inline it directly in your HTML. For example:

```
<li>
  <a href="/">
    <svg class="nav-links__icon" width="20" height="20" viewBox="0 0 20
      20">
      <path fill="#ffffff" d="M19.871 12.1651-8.829-9.758c-0.274-0.303-
        0.644-0.47-1.042-0.47-0 0 0 0 0-0.397 0-0.767 0.167-1.042
        0.471-8.829 9.758c-0.185 0.205-0.169 0.521 0.035 0.706 0.096
        0.087 0.216 0.129 0.335 0.129 0.136 0 0.272-0.055 0.371-
        0.16512.129-2.353v8.018c0 0.827 0.673 1.5 1.5 1.5h11c0.827 0 1.5-
        0.673 1.5-1.5v-8.01812.129 2.353c0.185 0.205 0.501 0.221 0.706
        0.035s0.221-0.501 0.035-0.706zM12 19h-4v-4.5c0-0.276 0.224-0.5
        0.5-0.5h3c0.276 0 0.5 0.224 0.5 0.5v4.5zM16 18.5c0 0.276-0.224
        0.5-0.5 0.5h-2.5v-4.5c0-0.827-0.673-1.5-1.5-1.5h-3c-0.827 0-1.5
        0.673-1.5 1.5v4.5h-2.5c-0.276 0-0.5-0.224-0.5-0.5v-9.12315.7-
        6.3c0.082-0.091 0.189-0.141 0.3-0.141s0.218 0.050 0.3 0.14115.7
        6.3v9.123z"></path>
    </svg>
    <span class="nav-links__label">Home</span>
  </a>
</li>
```

This allows you to target parts of the SVG directly from CSS if you want; you can dynamically change the colors—or even the size and position—of various parts of an SVG, using regular CSS. Yet the file sizes are smaller, and the images don't pixelate like GIF, PNG, or other raster-based image formats.

If you're not familiar with SVG, see <https://css-tricks.com/using-svg/> for a good primer on the various ways you can use SVG in your web pages.

Now that the icons are looking great, let's turn our attention to the labels beside them.

15.2.2 Creating “fly in” labels

The menu labels don't necessarily need to be visible at all times. You can hide them by default, leaving the icons in place to indicate to the user that the menu is there. Then, when the user moves their mouse over the menu or tabs to a menu item, you can fade in the labels. This way, when the user mouses near the icons, the entire menu appears, using a number of effects all at once: the background and the labels will fade in, with the labels starting a little to the left of their final position (figure 15.9).

This effect requires two separate transitions on the labels at the same time: one for opacity and another for a `translate()` transform. Update this portion of your stylesheet, making the changes indicated in the following listing.



Figure 15.9 Upon hover, the menu fades in, while the labels fade in and slide from the left.

Listing 15.9 Transitioning in the `nav-item` labels

```

@media (min-width: 30em) {
  .nav-links {
    display: block;
    padding: 1em;
    margin-bottom: 0;
  }
  .nav-links > li + li {
    margin-left: 0;
  }

  .nav-links__label {
    display: inline-block;
    margin-left: 1em;
    padding-right: 1em;
    opacity: 0;
    transform: translate(-1em);
    transition: transform 0.4s cubic-bezier(0.2, 0.9, 0.3, 1.3),
      opacity 0.4s linear;
  }
  .nav-links: hover .nav-links__label,
  .nav-links a: focus > .nav-links__label {
    opacity: 1;
    transform: translate(0);
  }

  .nav-links__icon {
    transition: transform 0.2s ease-out;
  }
  .nav-links a: hover > .nav-links__icon,
  .nav-links a: focus > .nav-links__icon {
    transform: scale(1.3);
  }
}

```

Makes the label an inline-block so transforms can be applied to it

Hides the label initially

Shifts the label 1 em to the left

Adds transitions to the values that will change

On hover or focus, makes the label visible and shifts it back to its correct position

This menu occupies a small portion of the screen's real estate, but there's a lot going on. Some of these selectors are fairly long and complicated.

Notice how the `:hover` pseudo-class you just added is on the top-level `nav-links` element, while the `:focus` pseudo-class is on the `<a>` within. (Focus generally can only

apply to certain elements like links and buttons.) This way, the labels all appear as soon as the menu is moused over. Additionally, an individual label also appears if the user focuses it using the Tab key on the keyboard.

When hidden, the label is shifted 1 em to the left using `translate()`. Then, as it fades in, it transitions back to its actual position. I've omitted the second parameter to the `translate()` function here and specified only the x value, which controls horizontal translation. Because you don't need to translate the element up and down, this is fine.

The custom `cubic-bezier()` function is worth looking at as well. This produces a bounce effect: the label moves right beyond the ending location before settling back where it stops. This curve is illustrated in figure 15.10.

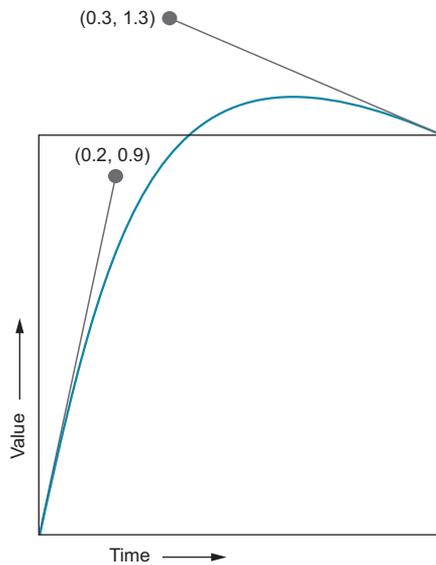


Figure 15.10 A Bézier curve with a bounce at the end

Notice that the curve extends outside the top of the box, meaning the value exceeds the value at the end of the transition. In transition from a `translate(-1em)` to `translate(0)`, the label's transform will momentarily reach a value about 0.15 em beyond the final position before easing back. You can also similarly create a bounce at the beginning of the timing function by moving the first control handle below the bottom of the box. You cannot, however, extend outside the left and right edges as this would produce an illogical transition curve.

Load the page in your browser and watch how this transition behaves. The bounce is subtle, so you may need to slow down the transition time to consciously see it, but it adds a bit of weight and momentum to the label, making the motion feel a little more natural.

15.2.3 Staggering the transitions

The menu looks pretty good at this point. Let’s make one last tweak to make it feel polished. You’ll use the `transition-delay` property to set a slightly different delay for each menu item. This’ll stagger the animations so they fly in a rolling “wave” rather than all at once (figure 15.11).

To accomplish this, you’ll use the `:nth-child()` pseudo-class selector to target each menu item based on its position in the list, and then apply a successively longer transition delay to each one. Add the next bit of code to your stylesheet after the rest of the `nav-links` styles.

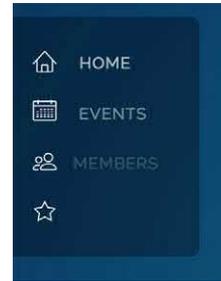


Figure 15.11 The top menu items will fly in just before the lower ones.

Listing 15.10 Adding a staggered transition delay to the menu items

```
.nav-links:hover .nav-links__label,
.nav-links a:focus > .nav-links__label {
  opacity: 1;
  transform: translate(0);
}
.nav-links > li:nth-child(2) .nav-links__label {
  transition-delay: 0.1s;
}
.nav-links > li:nth-child(3) .nav-links__label {
  transition-delay: 0.2s;
}
.nav-links > li:nth-child(4) .nav-links__label {
  transition-delay: 0.3s;
}
.nav-links > li:nth-child(5) .nav-links__label {
  transition-delay: 0.4s;
}
```

Targets the second menu item label

Delays its transition by one tenth of a second

Targets the third menu item label

Delays its transition by two tenths of a second

Repeat as many times as needed

The `:nth-child(2)` selector targets the second item in the list, to which you applied a slight delay. The third item (`:nth-child(3)`) gets a slightly longer delay. The fourth and fifth, each longer still. You don’t need to target the first item because you want its transition to begin immediately; it needs no transition delay.

Load this in your browser and hover over the menu to see the effect. It feels fluid and alive. Mouse off to see the items fade out with the same staggered timing.

You’ll find one downside to this sort of approach: the menu can only be as long as the number of these selectors you write. I added a rule to target a fifth menu item, even though our menu currently only has four items. This is a safeguard in case another menu item is added in the future. You could even add a sixth just to be safe. But be aware that as there’s a chance the menu could exceed this count at some point, you’ll then need to add more rules to the CSS.

TIP Repeating a block of code like this can be made easier with a preprocessor. See appendix B for an example.

Now that the menu is built, you can add more to this page. You'll do so in the next chapter, so keep this page handy to add to. But before that, there're a couple more things to know about transforms.

15.3 Animation performance

The existence of certain transforms might seem redundant. The result of a `translate` can often be accomplished using relative positioning, and, in the case of images or SVG, the result of a `scale` transform can be accomplished by explicitly setting a height and/or width.

Transforms are far more performant in the browser. If you animate the position of an element (transitioning the `left` property, for example) you can experience noticeably slower performance. This is particularly the case when animating a large complex element or a large number of elements on the page at once. This performance behavior applies to both transitions (covered in chapter 14) and animations (which I'll cover in the next chapter).

If you're doing any sort of transition or animation, you should always favor a transform over positioning or explicit sizing if you can. To understand why this is, we need to look closer at how the page is rendered in the browser.

15.3.1 Looking at the rendering pipeline

After the browser computes which styles apply to which elements on the page, it needs to translate those styles into pixels on the screen. This is the process of *rendering*, which can be broken down into three stages: layout, paint, and composite.

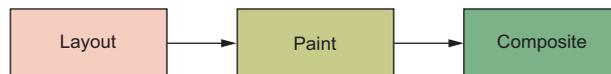


Figure 15.12 The three stages of the rendering pipeline

LAYOUT

In the first stage, *layout*, the browser calculates how much space each element is going to take on the screen. Because of the way the document flow works, the size and position of one element can influence the size and position of countless other elements on the page. This stage sorts that all out.

Any time you change the width or height of an element, or adjust its position properties (like `top` or `left`), the element's layout must be recomputed. This is also done if an element is inserted into or removed from the DOM by JavaScript. When a layout change occurs, the browser then must *reflow* the page, recomputing the layout of all other elements that are moved or resized as a result of the change.

PAINT

After layout comes *painting*. This is the process of filling in pixels: text is drawn; images and borders and shadows are all colored. This is not physically displayed on the screen, but rather drawn into memory. Portions of the page are painted into *layers*.

If you change the background color of an element, for instance, it must be re-painted. But, because the background color has no impact on the position or sizing of any elements on the page, layout doesn't need to be recalculated to account for this change. Changing a background color is less computationally intensive than changing the size of an element.

Under the right conditions, an element on the page can be promoted into its own layer. When this happens, it's painted separately from the other layer(s) on the page. Browsers can take this layer and send it to the computer's GPU (graphics processing unit) for rendering, rather than painting it on the main CPU like the main layer. This is beneficial because the GPU is highly optimized to do this sort of computation.

This is often referred to as *hardware acceleration* because it relies on a piece of the computer's hardware to give a boost to the rendering speed. Having more layers means more memory use; but, in return, it can speed up the processing time of rendering.

COMPOSITE

In the *composite* stage, the browser takes all of the layers that have been painted and draws them into the final image that'll be displayed onscreen. These are drawn in a certain order so that the correct layers appear in front of other layers, in cases where they overlap.

Two properties, *opacity* and *transform*, when changed, result in a much faster rendering time. When you change one of these on an element, the browser can promote that element to its own paint layer and use GPU acceleration. Because the element is in its own layer, the main layer won't change during the animation and won't require repeated re-painting.

When making a one-time change to the page, this optimization generally doesn't make a noticeable difference. But when the change is part of an animation, the screen needs to be updated dozens of times a second; in which case, speed matters. Most screens refresh 60 times per second. Ideally, changes during animation should be recomputed at least this fast to produce the most fluid motion possible onscreen. The more work the browser has to do for each recalculation, the harder this speed is to achieve.

Controlling paint layers with `will-change`

Browsers have come a long way with optimizing the rendering process, segmenting elements into layers as best they can. If you animate the `transform` or `opacity` property on an element, modern browsers, in order to make the animation smooth, generally make good decisions based on a number of factors, including system resources. But, occasionally, you might encounter choppy or flickering animations.

If you experience this, you can use a property called `will-change` to exert control over the render layers. This property indicates to the browser, ahead of time, that it should expect a certain property on the element to change. This usually means the element will be promoted to its own paint layer. For example, applying `will-change: transform` indicates you expect to change the `transform` property for that element.

Don't, however, apply this blindly to the page until you're seeing performance issues as it'll tend to use more system resources. Be sure to test before and after, only leaving `will-change` in the stylesheet if you experience better performance. For a deeper dive into how this property works and when you should or shouldn't use it, see the excellent article from Sara Soueidan at <https://dev.opera.com/articles/css-will-change-property/>.

I should note that one thing has changed since this article was written: it states that only 3D transforms promote an element to its own layer. This is no longer the case; the latest browsers now use GPU acceleration for 2D transforms as well.

When transitioning or animating, which we'll look at in the next chapter, try to make changes only to `transform` and `opacity`. Then, if needed, you can change properties that result in a paint but not a re-layout. Only change properties that affect layout when it's your only option and look to them first if you ever notice performance problems with your animations. For a complete breakdown of which properties result in layout, paint, and/or composite, check <https://csstriggers.com/>.

15.4 Three-dimensional (3D) transforms

So far the transforms you've used are all 2D. These are the easiest to work with (and the most common) as the page itself is 2D. But you're not confined to this limitation. Rotation and translation can be performed in all three dimensions: X, Y, and Z.

You can use the `translate()` function, as you've seen, to translate horizontally and vertically (X and Y dimensions). This can also be done with the functions `translateX()` and `translateY()`. The following two declarations produce the same result:

```
transform: translate(15px, 50px);  
transform: translateX(15px) translateY(50px);
```

You can also translate on the Z dimension using `translateZ()`, which moves an element conceptually closer to or further from the user. Similarly, you can rotate an element around axes in all three dimensions. But, unlike `translate`, `rotateZ()` is the version you're already familiar with; that is, `rotate()` is also aliased as `rotateZ()` because it rotates around the Z axis. The functions `rotateX()` and `rotateY()` rotate around the horizontal X axis (pitching an element forward and back) and around the vertical Y axis (turning—or *yawing*—the element left or right), respectively. See figure 15.13 for an illustration of these functions.

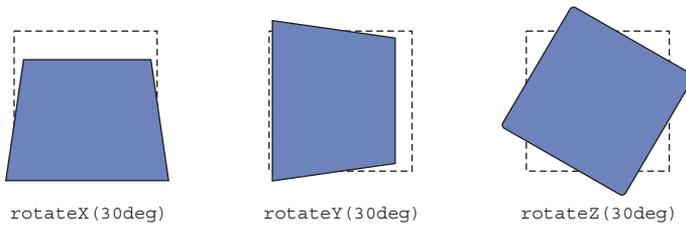


Figure 15.13 Rotation on each of the three axes with a 300 px perspective applied (a dashed line represents the initial element position)

15.4.1 Controlling perspective

Before you add 3D transforms to the page, however, you need to specify one more thing—*perspective*. The transformed elements together form a 3D scene. The browser then computes a 2D image of this 3D scene and renders it onto the screen. You can think of perspective as the distance between the “camera” and the scene. Moving the camera around changes the way the scene appears in the final image.

If the camera is close (that is, the perspective is small), then the 3D effects are much stronger. If the camera is far away (that is, the perspective is large), then the 3D effects are much more subtle. Some different perspectives are shown in figure 15.14.

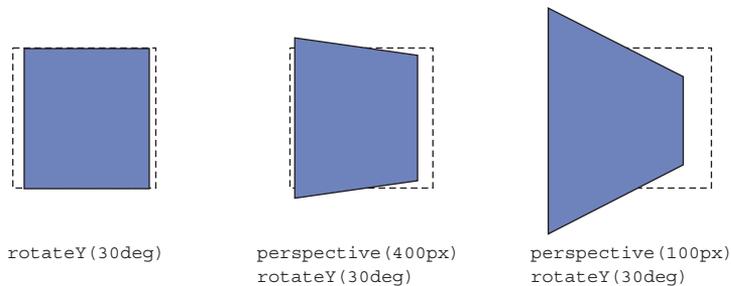


Figure 15.14 The same rotation applied at different perspectives.

The rotated element on the left, without a perspective applied, doesn’t look 3D. It appears squashed horizontally; there’s no real feel of depth. 3D transforms without perspective appear flat like this; parts of the element that are “further away” don’t appear any smaller. On the other hand, the box in the middle has a 400 px perspective applied. Its right edge—the edge that’s further from the viewer—appears a little smaller, and the edge that’s nearer appears larger. The perspective applied to the right box is much shorter, at 100 px. This exaggerates the effect so the edge of the element further away shrinks dramatically into the distance.

You can specify this perspective distance in two ways: using a `perspective()` transform or using the `perspective` property. Each behaves a little differently. Let's put together a basic example to illustrate. This example will be minimal, just to show the effects of perspective.

First, you'll rotate four elements, tilting them back using `rotateX()` (figure 15.15). Each element is rotated the same and has the same `perspective()` transform applied; thus, all four elements appear the same.



Figure 15.15 Four elements rotated about the X axis, each with a `perspective(200px)` transform applied

Create a new page for this demo and copy in the HTML shown here.

Listing 15.11 Four boxes to help illustrate 3D transforms and perspective

```
<div class="row">
  <div class="box">One</div>
  <div class="box">Two</div>
  <div class="box">Three</div>
  <div class="box">Four</div>
</div>
```

Next, you'll apply a 3D transform and a perspective transform to each of the boxes. You'll also add color and padding to fill out the size a bit and to help make the effect more apparent. Add a stylesheet to the page with the code shown in this listing.

Listing 15.12 Applying 3D transforms to the boxes

```
.row {
  display: flex;
  justify-content: center;
}

.box {
  box-sizing: border-box;
  width: 150px;
  margin: 0 2em;
  padding: 60px 0;
  text-align: center;
  background-color: hsl(150, 50%, 40%);
  transform: perspective(200px) rotateX(30deg);
}
```

Rotates the box back 30 degrees and applies a perspective

In this example, each box looks the same. Each has its own perspective, applied using the `perspective()` function. This method applies a perspective to a single element; in this example, you’ve applied it directly to each box. It’s as if four separate pictures were taken of each element, each from the same position.

Sometimes you’ll want multiple elements to share a common perspective, as if they all exist within the same 3D space. Figure 15.16 shows an illustration of this. These are the same four elements, but they all reach into the distance toward a common vanishing point. It’s as if one picture was taken of all four elements together. To achieve this effect, you’ll use the `perspective` property on their parent element.

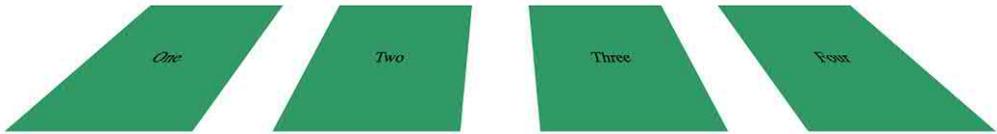


Figure 15.16 Make the elements share a common perspective by using the `perspective` property on a common ancestor element.

To see this effect, remove the `perspective()` function from the boxes and instead add it to the container using the `perspective` property. These changes are shown here.

Listing 15.13 Establishing a common perspective

```
.row {
  display: flex;
  justify-content: center;
  perspective: 200px;
}

.box {
  box-sizing: border-box;
  width: 150px;
  margin: 0 2em;
  padding: 60px 0;
  text-align: center;
  background-color: hsl(150, 50%, 40%);
  transform: rotateX(30deg);
}
```

← Adds the perspective to the container

← Don't apply a perspective transform to the boxes

By applying one common perspective to the parent (or other ancestor) container, all the elements within the parent that have 3D transforms applied will share that perspective.

Adding a perspective is an important part of 3D transforms. Without it, elements further from the viewer won’t appear smaller, and those closer won’t appear larger. This example is rather minimal. In the next chapter, you’ll use these techniques in a more practical example to “fly in” some elements onto the page from the distance.

15.4.2 Implementing advanced 3D transforms

A few other properties can be useful when manipulating elements in 3D. I won't spend a lot of time on these as real-world use cases for these are few and far between. But it's good to be aware they exist in case you ever need them. I'll point you to a few examples online if you want to delve deeper.

PERSPECTIVE-ORIGIN PROPERTY

By default, the perspective is rendered as if the viewer (or camera) is positioned directly in front of the center of the element. The `perspective-origin` property shifts the camera position left or right and up or down. Figure 15.17 shows the previous example, but with the camera shifted to the bottom left.



Figure 15.17 Moving the perspective origin increases the perspective distortion of elements toward the far edges.

To see this in your sample page, add the declaration in this listing.

Listing 15.14 Using `perspective-origin` to move the camera position

```
.row {
  display: flex;
  justify-content: center;
  perspective: 200px;
  perspective-origin: left bottom;
}
```

← Moves camera position to the element's bottom left

This is the same perspective distance as before, but here the perspective is shifted so all the boxes are to the right of the viewer. You can specify the position using the keywords `top`, `left`, `bottom`, `right`, and `center`. You can also use any percentage or length values, measured from the element's top left corner (`perspective-origin: 25% 25%`, for example).

BACKFACE-VISIBILITY PROPERTY

If you use `rotateX()` or `rotateY()` to spin an element more than 90 degrees, something interesting happens: the “face” of the element is no longer directed toward you. Instead, it is facing away, and you see the back of the element. The element in figure 15.18 has been transformed with `rotateY(180deg)`. It looks like a mirror image of the original.

This is the backface of the element. By default, the backface is visible, but you can change this by applying `backface-visibility: hidden` to the element. With this



Figure 15.18 Rotating an element to see its back side

declaration applied, the element will only be visible if it's facing toward the viewer, and hidden if it's facing away.

One possible application of this technique is to place two elements back-to-back, like two sides of a card. The front of the card will be visible, but the back of the card is hidden. Then you can rotate their container element to flip both elements around, making the front hidden and the back visible. For a demo of this card flip effect, see the article at <https://desandro.github.io/3dtransforms/docs/card-flip.html>.

TRANSFORM-STYLE (PRESERVE-3D) PROPERTY

The `transform-style` property becomes important if you go about building complex scenes with nested elements in 3D. Let's assume you've set a perspective on a container, then applied 3D transforms to elements within. That container element, when rendered, will be a 2D representation of that scene. It's like a photograph of a 3D object. This looks fine because that element must be rendered onto your 2D screen.

If you then apply a 3D rotation on the container itself, it won't look right. Instead of rotating the entire scene, it'll appear as if you're rotating a 2D photograph of a 3D scene. The perspective will be all wrong, and the illusion of depth in the scene is shattered. See figure 15.19 for an example illustrating this.

The scene rendered on the left shows a 3D cube created by transforming its six sides into place. The middle image shows what happens if you attempt to transform

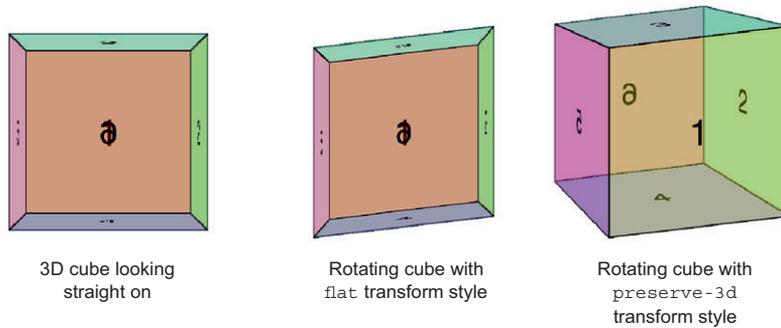


Figure 15.19 If you perform a 3D transform on the parent of other 3D-transformed elements, you'll probably want to apply `preserve-3d` (right).

the entire cube together (that is, the parent element). To correct this, you should apply `transform-style: preserve-3d` to the parent element (right).

WARNING The `preserve-3d` transform style is not supported in any version of Internet Explorer.

For a more complete explanation of this, as well as working examples, visit the tutorial from Ana Tudor at <https://davidwalsh.name/3d-transforms>. Although examples like this are fun to play with, I've never needed to use `preserve-3d` in a real-world project. But if you decide to play around with 3D transforms just to see what you can build, you may find the tutorial useful.

Summary

- Use transforms to scale, rotate, translate, and skew elements in two and three dimensions.
- Transforms are essential for performant transitions and animations.
- Understand how the rendering pipeline works and keep it in mind when building animations.
- To use a custom timing function curve to add a bounce effect to transitions.

16

Animations

This chapter covers

- Adding complex motion to the page with keyframe animations
- Playing animations when the page loads
- Using a spinner animation to provide feedback
- Drawing attention to a save button to remind the user to save

In the previous two chapters, you built several transitions that moved elements from one state to another. This brings motion to the page and visual interest to the user experience. But sometimes a transition isn't enough.

Instead of transitioning directly from one place to another, you might want an element to take a roundabout path along the way. Other times, you might want to animate an element and have it end up back where it started. These things can't be done with a transition. For more explicit control over changes on the page, CSS offers keyframe animation.

A *keyframe* refers to a specific point in an animation. You define some number of keyframes, and the browser fills in, or interpolates, all the frames in between (figure 16.1).



Figure 16.1 You define the keyframes, and the browser interpolates all the frames in between.

A transition is conceptually similar to a keyframe animation: You define the first frame (starting point) and the last (ending point), and the browser computes all the intermediate values so the element can transition smoothly between them. With keyframe animation, however, you're not limited to defining only two points. You can define as many as you like. The browser fills in the values from one to the next to the next, until it reaches the final keyframe, producing a series of seamless transitions.

In this final chapter, I'll show you how to build keyframe animations. You'll add some to the page you started in the previous chapter, then explore a few other ways they can be used. Animations aren't something you add to a page to jazz it up; they can also convey meaningful feedback to the user.

16.1 Keyframes

Animations in CSS contain two parts: the `@keyframes` at-rule, which defines an animation, and the `animation` property, which applies that animation to an element.

Let's build a basic animation to get familiar with the syntax. This animation will have three keyframes, shown in figure 16.2. In the first frame, the element is red. In the second frame, it's light blue and shifted to the right 100 px. In the final frame, it's light purple and has returned to its initial position on the left.

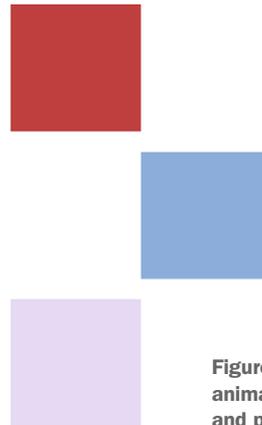


Figure 16.2 Three keyframes, animating the element's color and position

This animation applies changes to two properties: `background-color` and `transform`. The keyframe rule for this is shown in the following listing. Create a new stylesheet, `styles.css`, and add this code.

Listing 16.1 Defining a keyframe at-rule

```

@keyframes over-and-back {
  0% {
    background-color: hsl(0, 50%, 50%);
    transform: translate(0);
  }
}
  
```

← Names the animation

First keyframe declarations

```

50% {
  transform: translate(50px);
}

100% {
  background-color: hsl(270, 50%, 90%);
  transform: translate(0);
}

```

← **Second keyframe occurs halfway through the animation**

Final keyframe

A keyframe animation needs a name; this example defines an animation named `over-and-back`. It then defines three keyframes using percentages. These percentages indicate when in the animation each keyframe occurs: one at the beginning of the animation (0%), one in the middle (50%), and one at the end (100%). The declarations inside each of these blocks define how that keyframe appears.

This example animates two properties concurrently, but notice that they aren't both specified in every keyframe. The transform shifts the element from its initial position, to the right, then back again. The background color, however, isn't specified in the 50% keyframe. This means the element will animate smoothly from red (at 0%) to light purple (at 100%). At 50%, it'll be the value directly between these two colors.

Let's add this to a page to see it working. Create a new HTML document and add this markup.

Listing 16.2 Page with a single box element for animation

```

<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <div class="box"></div>
  </body>
</html>

```

← **The element you'll animate**

Next, add styles to your stylesheet to style the box and to apply the animation. Copy those in the following listing.

Listing 16.3 Applying the animation to the box

```

.box {
  width: 100px;
  height: 100px;
  background-color: green;
  animation: over-and-back 1.5s linear 3;
}

```

Gives the element a height and width for demo purposes

← **Applies the animation to the element**

Open the page in your browser. You should see the animation repeat three times, then stop. The `animation` property is a shorthand for several properties. In this demo, you've specified four of them:

- `animation-name (over-and-back)`—Indicates the name of the animation as defined by the `@keyframes` rule.
- `animation-duration (1.5s)`—Indicates how long the animation lasts; in this case, 1.5 seconds.
- `animation-timing-function (linear)`—Indicates a timing function describing how the animation accelerates and/or decelerates. This can be a Bézier curve or a keyword value, like a transition timing function (`ease-in`, `ease-out`, and so on).
- `animation-iteration-count (3)`—Indicates the number of times the animation repeats. If omitted, the initial value of 1 is used.

Reload the page to watch the animation play again. Let's observe a couple things about the way the animation behaves.

First, the color transitions smoothly from red at 0% to the light purple at 100%, but then it snaps back immediately to red as the animation repeats. If you plan to repeat an animation, you need to ensure the ending values match the beginning values if you want this change to be smooth.

Second, after the final iteration, the background color changes to green: the value specified in the regular ruleset. But notice that for the duration of the animation, this declaration is overridden by those in the `@keyframes`. In terms of the cascade, rules applied by an animation take precedence over other declarations.

If you recall from chapter 1 (section 1.1.1), the first part of the cascade is the stylesheet origin. Author styles take precedence over user agent styles because they're a higher priority origin. Declarations applied by an animation, however, are considered an even higher priority origin. While a property is being animated, it overrides those styles applied elsewhere in the stylesheet, regardless of selector specificity. This ensures all the declarations in the keyframes animate in concert with one another, regardless of what other rules might be applied to the element outside the animation.

WARNING Animations have good browser support, but a few mobile browsers require the use of the `-webkit-` prefix, both on the `animation` property (`-webkit-animation`) and the keyframes at-rule (`@-webkit-keyframes`). This requires duplicating all of this code, both with and without the prefix. Use Autoprefixer to do this (see the sidebar on “Vendor prefixes” in chapter 5).

16.2 Animating 3D transforms

Next, you'll add an animation to the page you began in the previous chapter. After listing 15.10, you should have a page with a blue background and a navigational menu on the left-hand side. You'll fill out the rest of this page with several cards of content. First, you'll get the layout built in a general shape of the overall design, then you'll add the animation.

16.2.1 Building the layout without animations

In this demo, you'll add some cards in the main area of the page (figure 16.3). Then you'll add an animation to make them fly in using 3D transforms.

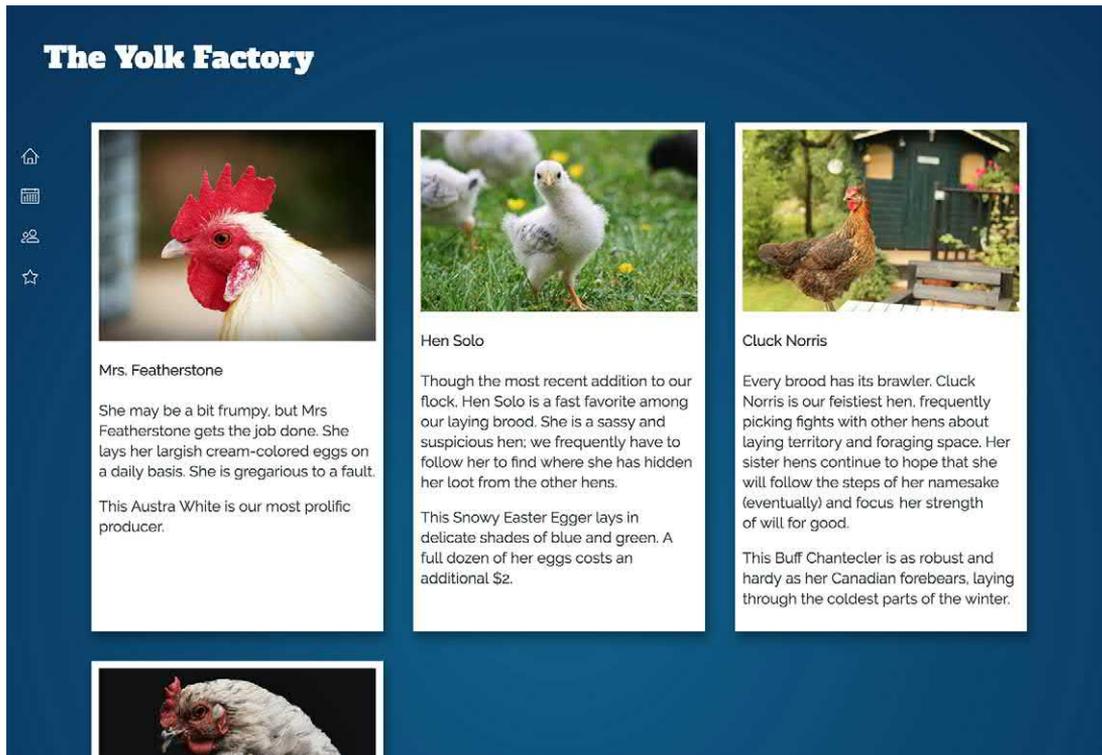


Figure 16.3 Additional cards for the main region of the page

The markup for this content is shown next. Add this to your page after the <nav> element. (I've abridged the text within the cards in this listing to save space. Feel free to add more content if you want to match the screenshot in figure 16.3 more closely.)

Listing 16.4 Building the flyin-grid and several cards

```

<main class="flyin-grid">           ← Grid container
  <div class="flyin-grid__item card"> ←
    
    <h4>Mrs. Featherstone</h4>
    <p>
      She may be a bit frumpy, but Mrs Featherstone gets
      the job done. She lays her largish cream-colored
      eggs on a daily basis. She is gregarious to a fault.
    </p>
  </div>
  <div class="flyin-grid__item card"> ←
    
    <h4>Hen Solo</h4>
    <p>
      Though the most recent addition to our flock, Hen
      Solo is a fast favorite among our laying brood.
    </p>
  </div>
  <div class="flyin-grid__item card"> ←
    
    <h4>Cluck Norris</h4>
    <p>
      Every brood has its brawler. Cluck Norris is our
      feistiest hen, frequently picking fights with other
      hens about laying territory and foraging space.
    </p>
  </div>
  <div class="flyin-grid__item card"> ←
    
    <h4>Peggy Schuyler</h4>
    <p>
      Peggy was our first and friendliest hen. She is the
      most likely to greet visitors to the yard, and
      frequently to be found nesting in the coop.
    </p>
  </div>
</main>

```

Cards are also grid items.

This portion of the page consists of two modules. The outer module, Flyin-Grid, provides the layout for the items in a grid, including a 3D fly-in effect that I'll cover in a bit. Each grid item is also an instance of the inner module, the Card. The Card module provides the styled appearance: white background, padding, and font color.

This layout is a prime example for a grid layout, so that's what you'll use. You should also consider both the mobile layout and a flexbox-based fallback for older browsers that don't support grids. You'll do the mobile first layout, then layer on flexbox styles followed by grid-based styles.

The mobile layout is shown in figure 16.4. On small screens, the cards will fill the width of the screen, with a little margin added to the left and right sides.

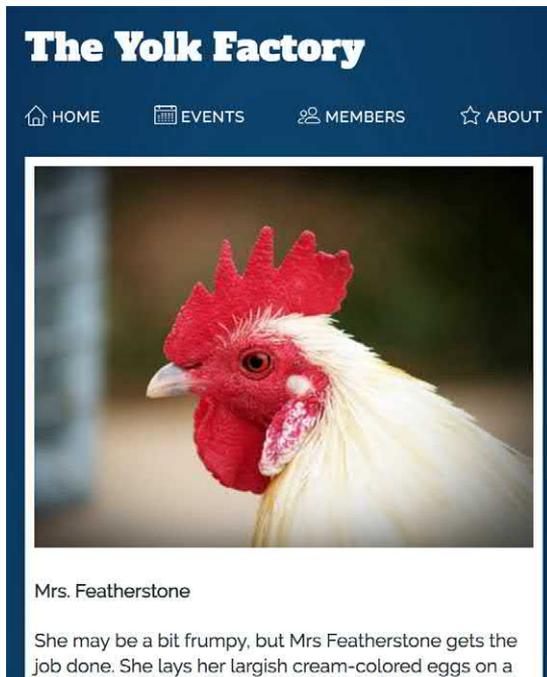


Figure 16.4 In the mobile layout, cards will fill the screen width, stacked beneath the menu.

Add these mobile styles to your stylesheet.

Listing 16.5 Mobile styles for the cards

```
.flyin-grid {
  margin: 0 1rem;
}
```

← Adds a small left and right margin around the container

```
.card {
  margin-bottom: 1em;
  padding: 0.5em;
  background-color: white;
  color: hsl(210, 15%, 20%);
  box-shadow: 0.2em 0.5em 1em rgba(0, 0, 0, 0.3);
}
```

Applies card colors and other details

```
.card > img {
  width: 100%;
}
```

← Specifies the image should fill the card width

The flyin-grid doesn't need much attention at this screen size because its items will stack correctly as normal block elements. The card styles apply the white background and the general look and feel of each card. You'll apply the more complex layouts within a media query momentarily.

Next, you'll layer in the fallback layout using flexbox, applied only to larger breakpoints. This'll get you close to the final design (figure 16.3). Add this CSS to your stylesheet.

Listing 16.6 Applying a flexbox-based fallback layout

```
.flyin-grid {
  margin: 0 1rem;
}

@media (min-width: 30em) {
  .flyin-grid {
    display: flex;
    flex-wrap: wrap;
    margin: 0 5rem;
  }

  .flyin-grid__item {
    flex: 1 1 300px;
    margin-left: 0.5em;
    margin-right: 0.5em;
    max-width: 600px;
  }
}
```

Responsive breakpoint

Establishes the flex container with wrapping

Increases padding on sides

Enables flex-grow and sets a flex-basis of 300 px

This listing establishes a responsive layout using flexbox. By applying `flex-wrap: wrap`, the flex items line wrap when they don't fit on the same line. The flex basis of 300 px establishes a minimum width, while the `max-width` establishes a maximum one; items will line wrap as needed to fit within these constraints. The flex-grow of 1 allows the cards to stretch to fill the remaining space.

The Card module doesn't need to change at all beyond the mobile styles you've already added; all colors and stylistic elements appear the same.

At certain screen sizes, the cards appear exactly like our final layout. But when the last row of cards has fewer cards than those on line(s) above, the card widths won't always be equal. This problem is shown in figure 16.5.

At this viewport width (around 1000 px), three cards fit into the top row, leaving only one for the second row. This final item grows to the `max-width` of 600 px, making it larger than the rest. When the screen size allows for two rows of two cards each, they'll all be the same size, as each row is equivalent. But other screen sizes might result in this problem. It'll vary, too, depending on how many cards there are: six cards would fit neatly into two rows of three, but large screens would see a row of four followed by a row of two.

This flexbox layout still works, as everything is usable and understandable, but it isn't ideal. You have two options for dealing with this: you could take the time to figure out several breakpoints and apply more specific control over the widths of the flex items or you could call this "good enough" as a fallback behavior, and override the flexbox with a grid layout for browsers that support grid.

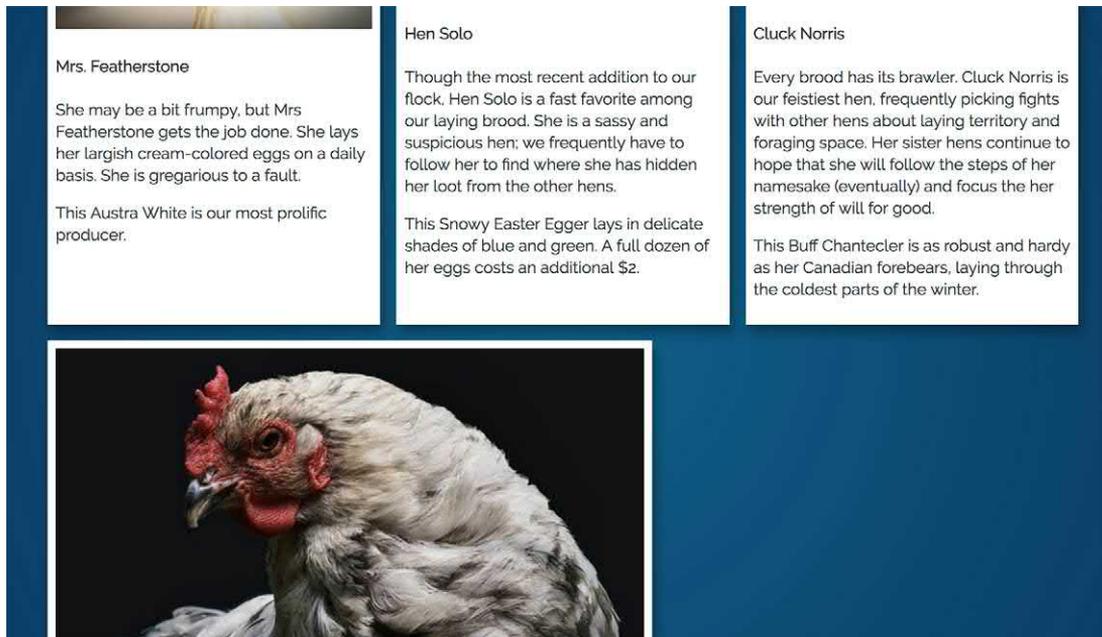


Figure 16.5 Flexbox doesn't always make the cards in the final row match the width of those above.

Let's do the second option here. After the flexbox layout, you'll use a feature query to test for grid support and add overriding styles. Update your stylesheet to match this CSS.

Listing 16.7 Applying a grid layout for browsers that support it

```

@media (min-width: 30em) {
  .flyin-grid {
    display: flex;
    flex-wrap: wrap;
    margin: 0 5rem;
  }

  .flyin-grid_item {
    flex: 1 1 300px;
    margin-left: 0.5em;
    margin-right: 0.5em;
    max-width: 600px;
  }

  @supports (display: grid) {
    .flyin-grid {
      display: grid;
      grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
      grid-gap: 2em;
    }
  }
}

```

← Fallback styles remain unchanged

← Queries for grid support within a media query block

← Defines column widths

```

.flyin-grid_item {
  max-width: initial;
  margin: 0;
}

```

← Removes margins applied by the fallback layout

Now the latest browsers will use the ideal layout. Grid columns ensure all grid items are the same width. Using `repeat()` and `auto-fit` allows the grid to determine how many columns fit at the current viewport width. This solution will gracefully degrade to the flexbox layout for older browsers, and small viewports will still display the even simpler mobile layout.

16.2.2 Adding animation to the layout

The page now has the design and layout in place, so let's work in some animation. When the page loads, you'll fly in the cards, as illustrated in figure 16.6. They'll start so they appear off in the distance, rotated 90 degrees around a vertical axis. Then the cards will fly in toward the viewer and, near the end of the animation, rotate to face the user directly. Figure 16.6 shows the three keyframes that define this animation.

This animation involves two transforms: `translateZ()` starts the cards back in the distance, and `rotateY()` rotates them. The code for this is shown in listing 16.8. This sets a perspective on the flyin-grid container, defines the keyframes, and adds the animation to each flyin-grid item. I've also added opacity, so the items fade in along with the transition effects.

Listing 16.8 Adding the fly-in animation

```

.flyin-grid {
  margin: 0 1rem;
  perspective: 500px;
}

.flyin-grid_item {
  animation: fly-in 600ms ease-in;
}

@keyframes fly-in {
  0% {
    transform: translateZ(-800px) rotateY(90deg);
    opacity: 0;
  }
  56% {
    transform: translateZ(-160px) rotateY(87deg);
    opacity: 1;
  }
  100% {
    transform: translateZ(0) rotateY(0);
  }
}

```

← Sets a shared perspective on the container

← Applies animation to each item

← Starts in the distance, rotated

← Much closer, but still mostly rotated

← Finishes in normal position

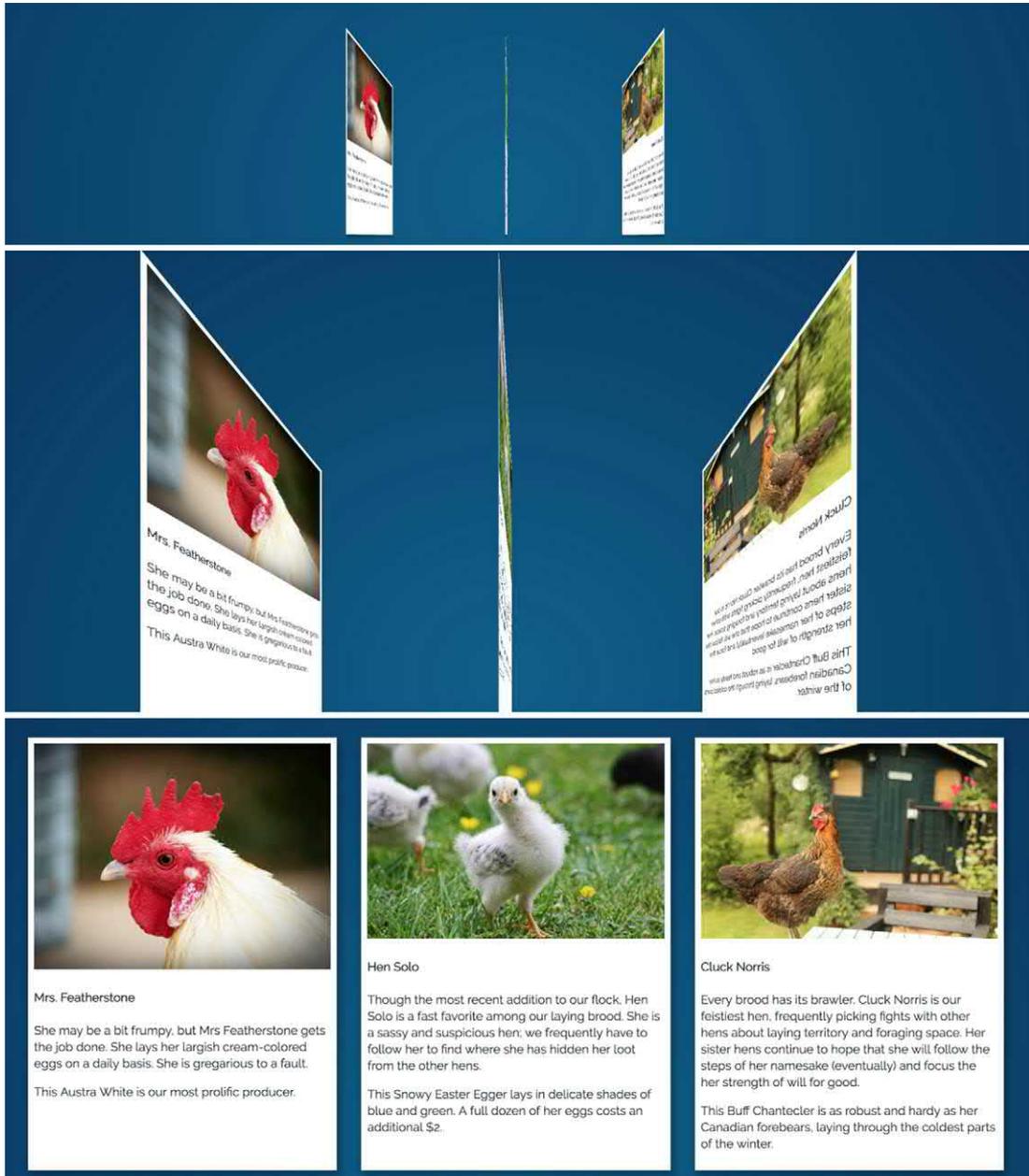


Figure 16.6 Use 3D transforms to fly in the cards from off in the distance.

This CSS sets a perspective on the container, so all the items share the same perspective. Then it applies the animation to each item. Load the page to watch the animation.

The animation begins by placing the rotated element back in the distance. Between the beginning keyframe and the middle keyframe, the element zooms forward most of the way (from 800 px to 160 px), and the opacity fades from transparent to fully opaque. From the middle keyframe to the end, the last bit of zoom finishes, while the bulk of the rotation takes place.

16.3 Animation delay and fill mode

Animations can be delayed using the `animation-delay` property, which behaves much like the `transition-delay` property. You can use this to stagger the animations, similar to the way you staggered the navigational menu transitions in the previous chapter. By staggering each item's animation for slightly different amounts of time, you can make them fly in one after the other as shown in figure 16.7.



Figure 16.7 Elements flying in with staggered animation

The next code applies these delays to the four grid items. But, it won't quite work like you want it to. Add the code to your stylesheet, then we'll take a look at the problem and how to address it.

Listing 16.9 Staggering the animation start times

```
.flyin-grid_item {
  animation: fly-in 600ms ease-in;
}
```

```

.flyin-grid_item:nth-child(2) {
  animation-delay: 0.15s;
}
.flyin-grid_item:nth-child(3) {
  animation-delay: 0.3s;
}
.flyin-grid_item:nth-child(4) {
  animation-delay: 0.45s;
}

```

← Stagger the start of each item's animation a little longer than the previous item

If you load this page in your browser, you might notice the problem. The animations are played at the expected time, but some items are visible on the page beforehand. After a moment they disappear and their animation plays (figure 16.8). This is a bit jarring, and doesn't look like the effect we're going for. Instead, we want the elements to all be invisible initially, and only appear during the course of their respective animations.

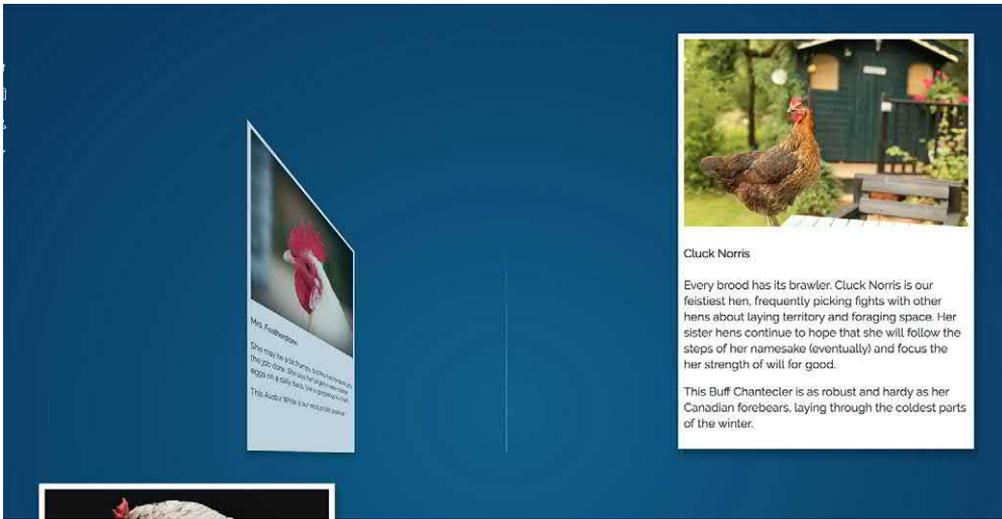


Figure 16.8 The later items appear in their final position before their animations play.

This problem occurs because the `transform` and `opacity` properties are only applied during the animation. Before the animation starts, the grid items are visible on the page, in their normal positions. Then, when the animation begins, they jump to their values applied in the keyframe at 0%. You need the animation styles to apply backward in time, as if paused on the first frame until the animation starts. This can be done with the `animation-fill-mode` property (figure 16.9).

The dark boxes here represent the duration of the animation. The initial value of `animation-fill-mode` is `none`, which means the animation styles are not applied to the element before or after the animation. By applying `animation-fill-mode: backwards`, the browser takes the values from the first frame of the animation and applies



Figure 16.9 Use `animation-fill-mode` to apply animation styles before or after the animation plays.

them to the element before the animation is played. Using `forwards` continues to apply the last frame values after the animation completes. Using `both` fills both backward and forward.

Add a backward fill mode to your page to fix the jump at the beginning of the animation. Update your stylesheet to match.

Listing 16.10 Applying a backward animation fill mode

```
.flyin-grid__item {
  animation: fly-in 600ms ease-in;
  animation-fill-mode: backwards;
}
```

← Applies first frame animation styles before animation begins

This effectively makes the animation initially pause on the first frame, waiting for the animation to play. Now, before the animation starts, the grid item is translated back 800 px, rotated 90 degrees, and set to opacity 0, ready for the animation to begin.

Because the animation ends with the element in its natural position, you don't need to fill forward; the card already seamlessly steps from the final frame of animation to the element's static position.

16.4 Conveying meaning through animation

A common misconception about animation is that it's added to the page for fun, and that it serves no practical purpose. This is sometimes the case (as with our last example), but this isn't always true. The best animations aren't added on as an afterthought. Instead, they're integrated into the experience. They convey specific meaning to the user about something on the page.

16.4.1 Responding to user interaction

Animation can indicate to the user that a button has been clicked or a message has been received. If you've ever submitted a form, only to find yourself wondering whether your mouse click registered, you know how important this can be.

On a new page, let's build a small form with a submit button. Then you'll add a spinning indicator to let the user know that the form is posting, and the browser is awaiting a response from the server. The form is pictured in figure 16.10. It consists of a label, a text area, and a button.

Tell us about your first trip to the zoo:


 A screenshot of a web form. At the top, there is a label that reads "Tell us about your first trip to the zoo:". Below the label is a large, empty rectangular text area. At the bottom left of the form, there is a blue button with the word "Save" written in white text.

Figure 16.10 A simple form with a Save button

Create a new page and a blank stylesheet for this form. Add the HTML shown here.

Listing 16.11 Form with Save button

```
<!doctype html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <form>
      <label for="trip">Tell us about your first trip to the zoo:</label>
      <textarea id="trip" name="about-my-trip" rows="5"></textarea>
      <button type="submit" id="submit-button">Save</button>
    </form>
  </body>
</html>
```

Text area

Submit button

First, you'll add some CSS to get everything laid out and styled appropriately. After that, you'll work in a few meaningful animations to enhance the user experience. Add this to your stylesheet.

Listing 16.12 Laying out and styling the form

```
body {
  font-family: Helvetica, Arial, sans-serif;
}

form {
  max-width: 500px;
}

label,
textarea {
  display: block;
```

Limits the width of the form

```

margin-bottom: 1em;
}

textarea {
width: 100%;
font-size: inherit;
}

button {
padding: 0.6em 1em;
border: 0;
background-color: hsl(220, 50%, 50%);
color: white;
font: inherit;
transition: background-color 0.3s linear;
}
button:hover {
background-color: hsl(220, 45%, 40%);
}

```

Blue button with white text

Darkens button for the hover state

Let's assume this form is part of a larger web application. When the user clicks the Save button, it'll send the data to the server and, perhaps, receive a response and then add some new content to the page. But waiting for the network takes time. While the user waits for this response, it can be reassuring to them if they see some visual indication that their content has been submitted and a response will be appearing soon. Animation is a common way to provide this indication.

You can modify the Save button, giving it an “is-loading” state. This hides the Save label, replacing it with a spinning icon (figure 16.11). When the user submits the form, you'll use JavaScript to add the `is-loading` class to the button, enacting the animation.

Tell us about your first trip to the zoo:

I took my first trip to the zoo when I was five years old. My favorite animals were the pigeons at the food court.



Figure 16.11 When the user clicks Save, a loading spinner appears in the button.

You can design a spinner in a number of ways. This is one approach I like: a rotating crescent shape is minimal in appearance but effective. Adding this spinner takes two changes to the CSS: creating the crescent shape using a border and border radius, then applying an animation to make it spin. It'll also need a little JavaScript to add the `is-loading` class in order to apply the styles when the button is clicked.

The CSS for this is shown next. This markup adds the animation to an absolutely positioned pseudo-element on the button. Add it to your stylesheet.

Listing 16.13 Defining the spin animation and *is-loading* state

```

button.is-loading {
  position: relative;
  color: transparent;
}
button.is-loading::after {
  position: absolute;
  content: "";
  display: block;
  width: 1.4em;
  height: 1.4em;
  top: 50%;
  left: 50%;
  margin-left: -0.7em;
  margin-top: -0.7em;
  border-top: 2px solid white;
  border-radius: 50%;
  animation: spin 0.5s linear infinite;
}
@keyframes spin {
  0% {
    transform: rotate(0deg);
  }
  100% {
    transform: rotate(360deg);
  }
}

```

Hides the button text

Positions pseudo-element in the center of the button

Loops spin animation repeatedly

Defines one full rotation per iteration

This defines an *is-loading* state for the button. When applied, the button's text is made invisible with `color: transparent`, and its pseudo-element is placed in the center of the button using absolute positioning.

The positioning here is a little tricky: the `top` and `left` properties move the pseudo-element down by half the button's height and right by half the button's width. This positions the pseudo-element so its *top-left corner* is at the center of the button. Then, the negative margins pull the pseudo-element back up and left by 0.7 em, which is exactly half of its height and width. Together, all four of these properties center the pseudo element vertically and horizontally within the button. Add the *is-loading* class temporarily and play with these values in your browser's DevTools to get a feel for how they work together to center the pseudo-element.

After positioning the pseudo-element, you then apply the animation. This uses a new keyword for the animation iteration count: `infinite`. This means the animation repeats endlessly as long as the *is-loading* class is applied to the button. The animation applies a rotation transform, from 0 deg to 360 deg. This rotates the pseudo-element a full rotation. The end of this animation leaves the element exactly where it began, visually, so the transition is seamless as the animation repeats.

Add the script tag from the next listing to your page. This provides JavaScript functionality to add the `is-loading` class when the button is clicked. Place this before the closing `</body>` tag.

Listing 16.14 Adding an `is-loading` class to the button when clicked

```

<script type="text/javascript">
  var input = document.getElementById('trip');
  var button = document.getElementById('submit-button');

  button.addEventListener('click', function(event) {
    event.preventDefault();
    button.classList.add('is-loading');
    button.disabled = true;
    input.disabled = true;
  });
</script>

```

← Prevents form submission
 ← Displays loading spinner
 ← Code here would submit form data using JavaScript

When the Save button is clicked, it stops the normal form submission using `preventDefault()`. This allows the user to stay on the same page without navigating away while the application submits the form data using JavaScript. In the meantime, the inputs are disabled, and the `is-loading` class is added to the button, displaying the spinning indicator. Load the page and click the button to make the spinner appear.

You aren't submitting form data here because there's no server to submit to in this demo. But, in a real application, once the server responds, you could then re-enable the form inputs and remove the `is-loading` class. For purposes of this demo, you can refresh the page to reset the form and remove the `is-loading` class.

16.4.2 Drawing the user's attention

Animation can also be used to bring the user's attention to something. If you expected the user to write more than a couple of sentences in the text area, you could encourage them to save their work frequently as they compose their response. If you use an animation to shake the button for a moment, that can serve as a reminder to the user to save their work (figure 16.12).



Figure 16.12 Move the button left and right rapidly to shake it.

This shake can be accomplished by rapidly transforming the element left and right several times. You'll define a keyframe animation that does that and apply the animation to the button element using a shake class. Add this to your stylesheet.

Listing 16.15 Defining the shake animation

```

.shake {
  animation: shake 0.7s linear;
}

```

```

@keyframes shake {
  0%,
  100% {
    transform: translateX(0);
  }
  10%,
  30%,
  50%,
  70% {
    transform: translateX(-0.4em);
  }
  20%,
  40%,
  60% {
    transform: translateX(0.4em);
  }
  80% {
    transform: translateX(0.3em);
  }
  90% {
    transform: translateX(-0.3em);
  }
}

```

Uses the same keyframe definition at multiple points during the animation

Shifts the element left

Shifts the element right

Reduces the movement for the final shake

I've done something new in this animation: I've applied the same keyframe definitions multiple times throughout the animation.

In the beginning (0%) and ending (100%) keyframes, the element is in its default position. Because these keyframes both use the same value, you can separate them with a comma and define their property values once. I've done the same with the 10%, 30%, 50%, and 70% keyframes, which all translate the element left. The 20%, 40%, and 60% each translate the element right. The 80% and 90% keyframes translate the element right and left, respectively, but to a lesser degree.

Together this animation shakes the element side-to-side four times, with the fourth shake being a little less pronounced to simulate slowing down at the end of the motion. You can temporarily add the `shake` class to the button to see the animation play when the page loads.

NOTE An animation can be used several times throughout the stylesheet, so its definition doesn't necessarily need to be located with the code for the module that uses it. I like to keep all my `@keyframe` definitions together in one place, near the end of my stylesheet.

Finally, you can use JavaScript to play the animation when you think the user might need to save their work. You can use a `keyup` event listener and a `timeout` function to do this: When the user types a character into the text area, you'll set a one second `timeout` function, which will add the `shake` class to the button. If the user enters another character before the second elapses, you'll clear the `timeout` and set a new one. Update the `script` tag in your page to match this listing.

Listing 16.16 Adding the shake class after a one-second delay

```

<script type="text/javascript">
  var input = document.getElementById('trip');
  var button = document.getElementById('submit-button');

  var timeout = null;
  button.addEventListener('click', function(event) {
    event.preventDefault();
    clearTimeout(timeout);
    button.classList.add('is-loading');
    button.disabled = true;
    input.disabled = true;
  });

  input.addEventListener('keyup', function() {
    clearTimeout(timeout);
    timeout = setTimeout(function() {
      button.classList.add('shake');
    }, 1000);
  });

  button.addEventListener('animationend', function() {
    button.classList.remove('shake');
  });
</script>

```

← Defines a variable to refer to your timeout

← Cancels the pending timeout (if present)

← Adds the shake class after a 1-second wait

← Removes the shake class after the animation ends

Now load the page and type something into the text area. Wait one second, and the Save button will shake. As long as you continue typing, the timer will continually reset, and the shake animation won't play until the next time you stop for more than 1 second. This way the shaking doesn't constantly distract the user, but only occurs when they pause.

You also made use of a JavaScript event, `animationend`. This event is fired when the shake animation finishes playing. When this happens, the shake class is removed from the button so it can be added again the next time the user types then pauses, replaying the animation a second time.

Adding and removing classes like this is perhaps the simplest way to interact with animations using JavaScript. But, if you're familiar enough with the language, there's a complete API to interact with CSS animations, including the ability to pause, cancel, and reverse them. For more information on this API, check the MDN documentation at <https://developer.mozilla.org/en-US/docs/Web/API/Animation>.

These animations—the loading indicator and the shaking Save button—communicate a lot to the user. They do so without making the user read any explanation. They indicate their meaning instantly, making for a less obtrusive UI.

As you continue to build web applications, always consider if an animation, even a subtle animation, can provide useful feedback to the user. Perhaps when sending an email, the text area can fly off the side of the screen. Or, when deleting a draft, animation can cause the draft to shrink and disappear. Animations don't need to be

obvious or flamboyant to provide reassurance to users that their actions did what they intended.

For a fantastic set of pre-defined keyframes you can use, visit <http://animista.net/>. This has a large library of animations to choose from, including bouncing in, rolling out, and wobbling like jelly.

16.5 *One final piece of advice*

To many web developers, CSS is an intimidating language. It has one foot planted in the world of design and another in the world of code. Some parts of the language aren't intuitive, especially if you're self-taught in the subject. I hope that this book has helped you find your way.

We've taken a deep look at the most fundamental parts of the language, and some of the more confusing parts of page layout. We've covered a lot of ground, from organizing CSS for easier code maintenance to the newest layout methods. We've ventured into the world of the design and built an interface that's not only utilitarian, but also intuitive and enjoyable.

My last piece of advice for you is to stay curious. I've shown you a wide array of tools in the CSS toolset. But the ways these tools can be mixed and matched are endless. When you encounter a web page that wows you, open your browser's DevTools and try to figure out how it works. Follow developers and designers online that provide creative demos or offer interesting tutorials. Try new things. And, keep learning.

Summary

- You can use keyframe animations to define key points in an animation.
- Use backward and forward fill modes to making an animation begin or end seamlessly.
- Using JavaScript, you can trigger animations at the appropriate time.
- Using animations adds meaning, not just flourish, to user interaction on the web page.

appendix A

Selectors reference

Selectors target specific elements on the page for styling. CSS provides a wide array of selector types.

A.1 Basic selectors

- `tagname`—Type selector or tag selector. This selector matches the tag name of the elements to target. Has a specificity of 0,0,1. Examples: `p`; `h1`; `strong`.
- `.class`—Class selector. Targets elements that have the specified class name as part of their class attribute. Has a specificity of 0,1,0. Examples: `.media`; `.nav-menu`.
- `#id`—ID selector. Targets the element with the specified ID attribute. Has a specificity of 1,0,0. Example: `#sidebar`.
- `*`—Universal selector. Targets all elements. Has a specificity of 0,0,0.

A.2 Combinators

Combinators join multiple simple selectors into one complex selector. In the selector `.nav-menu li`, for example, the space between the two simple selectors is known as a *descendant combinator*. It indicates the targeted `` is a descendant of an element that has the `nav-menu` class. This is the most common combinator, but there are a few others, each of which indicates a particular relationship between the elements indicated:

- *Child combinator* (`>`)—Targets elements that are a direct descendant of another element. Example: `.parent > .child`.
- *Adjacent sibling combinator* (`+`)—Targets elements that immediately follow another. Example: `p + h2`.
- *General sibling combinator* (`~`)—Targets all sibling elements following a specified element. Note this doesn't target siblings that appear before the indicated element. Example: `li.active ~ li`.

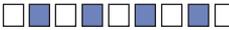
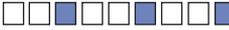
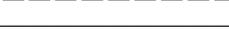
COMPOUND SELECTORS

Multiple simple selectors can be joined (without spaces or other combinators) to form a *compound selector* (for example, `h1.page-header`). A compound selector targets elements that match *all* its simple selectors. For example, `.dropdown.is-active` targets `<div class="dropdown is-active">` but not `<div class="dropdown">`.

A.3 Pseudo-class selectors

Pseudo-class selectors are used to target elements when they're in a certain state. This state can be due to user interaction or the element's position in relation to its parent or sibling elements in the document. Pseudo-class selectors always begin with a colon (:). These have the same specificity as a class selector (0,1,0).

- `:first-child`—Targets elements that are the first element within their parent element.
- `:last-child`—Targets elements that are the last element within their parent element.
- `:only-child`—Targets elements that are the only element within their parent element (no siblings).
- `:nth-child(an+b)`—Targets elements based upon their position among their siblings. The formula $an+b$, where a and b are integers, indicates which elements to target. To know exactly how a formula works, imagine solving it for all integer values of n , beginning with zero. The results of this equation indicate which children are targeted. This figure illustrates some examples:

Selector	Elements targeted	Result	Description
<code>:nth-child(n)</code>	0, 1, 2, 3, 4 ...		Every element
<code>:nth-child(2n)</code>	0, 2, 4, 6, 8 ...		Even elements
<code>:nth-child(3n)</code>	0, 3, 6, 9, 12 ...		Every third element
<code>:nth-child(3n+2)</code>	2, 5, 8, 11, 14 ...		Every third element beginning with element 2
<code>:nth-child(n+4)</code>	4, 5, 6, 7, 8 ...		All elements beginning with element 4
<code>:nth-child(-n+4)</code>	4, 3, 2, 1, 0 ...		First four elements

- `:nth-last-child(an+b)`—Similar to `:nth-child()`, but instead of counting forward from the first element, this selector counts backward from the last element. The formula in the parentheses follows the same pattern as in `:nth-child()`.
- `:first-of-type`—Similar in nature to `:first-child`, except instead of considering the position among all children, it considers an element's numerical position only among other children with the same tag name.
- `:last-of-type`—Targets the last child element of each type.
- `:only-of-type`—Targets elements that are the only child of their type.

- `:nth-of-type(an+b)`—Targets elements of their type based on their numerical order and the specified formula; similar to `:nth-child`.
- `:nth-last-of-type(an+b)`—Targets elements of their type based on a specified formula, counting from the last of those elements backward; similar to `:nth-last-child`.
- `:not(<selector>)`—Targets elements that don't match the selector within the parentheses. The selector inside the parentheses must be simple: it can only refer to the element itself; you can't use this selector to exclude ancestors. It also mustn't contain another negation selector.
- `:empty`—Targets elements that have no children. Beware that this doesn't target an element that contains whitespace as the whitespace is represented in the DOM as a text node child. At the time of writing, the W3C is considering a `:blank` pseudo-class that behaves similarly but also selects elements that contain only whitespace; `:blank` is not yet supported in any browser.
- `:focus`—Targets elements that have received focus, whether from a mouse click, screen tap, or Tab key navigation.
- `:hover`—Targets elements while the mouse cursor hovers over them.
- `:root`—Targets the root element of the document. In HTML, this is the `<html>` element. But CSS can be applied to other XML or XML-like documents, such as SVG; in which case, this selector works more generically.

Several other pseudo-classes relate to form fields. Some of these were introduced or refined with the selectors level 4 specification, so they don't work in IE10 and some other browsers. Check caniuse.com for support.

- `:disabled`—Targets disabled elements, including inputs, selects, and buttons.
- `:enabled`—Targets enabled elements, meaning they can be activated or accept focus.
- `:checked`—Targets selected checkboxes, radio buttons, or select box options.
- `:invalid`—Targets elements with invalid input values, as defined by the input type. For example, an `<input type="email">`, whose value is not a valid email address. (Level 4).
- `:valid`—Targets elements with valid values (Level 4).
- `:required`—Targets elements with a required attribute set (Level 4).
- `:optional`—Targets elements that do not have a required attribute set (Level 4).

This list of pseudo-classes isn't exhaustive. See the MDN documentation at <https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes> for a complete list.

A.4 Pseudo-element selectors

Pseudo-elements are similar to pseudo-classes, but instead of selecting elements with a special state, they target a certain part of the document that doesn't directly correspond

to a particular element in the HTML. They may target only portions of an element or even inject content into the page where the markup defines none.

These selectors begin with a double-colon (::), though most browsers also support a single-colon syntax for backward-compatibility reasons. Pseudo-elements have the same specificity as a type selector (0,0,1).

- `::before`—Creates a pseudo-element that becomes the first child of the matched element. This element is inline by default. Can be used to insert text, images, or other shapes. The `content` property must be specified to make this element appear. Example: `.menu::before`.
- `::after`—Creates a pseudo-element that becomes the last child of the matched element. This element is inline by default. Can be used to insert text, images, or other shapes. The `content` property must be specified to make this element appear. Example `.menu::after`.
- `::first-letter`—Lets you specify styles for only the first text character inside the matched element. Example: `h2::first-letter`.
- `::first-line`—Lets you specify styles for the first line of text inside the matched element.
- `::selection`—Lets you specify styles for any text the user has highlighted with their cursor. This is often used to change the background-color of selected text. Only a handful of properties can be used; those include color, background color, cursor, and text decoration.

A.5 **Attribute selectors**

Attribute selectors can be used to target elements based on their HTML attributes. These have the same specificity as a class selector (0,1,0).

- `[attr]`—Targets elements that have the specified attribute `attr`, regardless of its value. Example: `input[disabled]`.
- `[attr="value"]`—Targets elements that have the specified attribute `attr`, and its value matches the specified string value. Example: `input[type="radio"]`.
- `[attr^="value"]`—“Starts with” attribute selector. Targets by attribute and value, where the value begins with the specified string value. Example: `a[href^="https"]`.
- `[attr$="value"]`—“Ends with” attribute selector. Targets by attribute and value, where the value ends with the specified string value. Example: `a[href$=".pdf"]`.
- `[attr*="value"]`—“Contains” attribute selector. Targets by attribute and value, where the attribute value contains the specified string value. Example: `[class*="sprite-"]`.
- `[attr~="value"]`—“Space-separated list” attribute selector. Targets by attribute and value, where the attribute value is a space-separated list of values, one of which matches the specified string value. Example: `a[rel="author"]`.

- `[attr|=value]`—Targets by attribute and value, where the value either matches the specified string value, or begins with it and is immediately followed by a hyphen (–). Useful for the language attribute, which may or may not specify a language subcode (for example, Mexican Spanish, `es-MX`, or Spanish in general, `es`). Example: `[lang|=“es”]`.

CASE-INSENSITIVE ATTRIBUTE SELECTORS

All of the previous attribute selectors are case-sensitive. The selectors level 4 specification introduces a case-insensitive modifier that can be added to any attribute selector. To do this, add an `i` before the closing bracket. Example: `input[value=“search” i]`.

Many browsers don’t yet support this. Those that don’t will ignore it. For this reason, if you use case-insensitive modifiers, be sure to add a fallback to a regular case-sensitive version.

appendix B

Preprocessors

Using a preprocessor is a vital part of a modern CSS workflow. A preprocessor provides a number of conveniences to streamline your writing and to help you maintain your codebase. For instance, you can write a piece of code once and then reuse it throughout your stylesheet.

A preprocessor works by taking a source file, which you write, and translating it into an output file, which is a regular CSS stylesheet. In most cases, the source file looks much like regular CSS, but with extra features added. A simple example using a preprocessor variable looks like this

```
$brand-blue: #0086b3;

a:link {
  color: $brand-blue;
}

.page-heading {
  font-size: 1.6rem;
  color: $brand-blue;
}
```

This code snippet defines a variable named `$brand-blue`, which is used in two separate places later in the stylesheet. When run through the Sass preprocessor, the variable is replaced throughout the stylesheet, producing the following CSS as output:

```
a:link {
  color: #0086b3;
}

.page-heading {
  font-size: 1.6rem;
  color: #0086b3;
}
```

It's important to note that, because the final output is regular CSS, a preprocessor adds no new features to the language as far as the browser is concerned. It does, however, provide useful conveniences to you as a developer.

In the example, using a variable to represent the color allows you to re-use the color countless times without having to copy and paste the exact hex code. The preprocessor does the copying for you when it generates the output file. It also means that you can edit the value in one place, and have that change propagate throughout the entire stylesheet.

The two most popular preprocessors are *Sass* (<http://sass-lang.com/>) and *Less* (<http://lesscss.org/>), though there're several others as well. Sass is the most popular, so I'll focus mostly on that in this appendix. But Sass and Less are similar, with mostly minor syntactic differences distinguishing them from one another. For instance, Sass uses a `$` to denote variables (`$brand-blue`), whereas Less uses an `@`-sign (`@brand-blue`). Every Sass feature covered in this appendix is also supported in Less; check the Less documentation for syntax differences.

B.1 Sass

When getting started with Sass, you'll need to make a few decisions. First is which implementation to use. Sass is written in Ruby, but this implementation is a little slow when compiling large stylesheets, so I recommend something called *LibSass*, which is a C/C++ port of the Sass compiler.

If you're comfortable with JavaScript and the Node environment, you can get LibSass by installing the `node-sass` package via the npm package manager. If you don't already have Node.js installed, you can find it (it's free) at <https://nodejs.org>. Download and install it according the directions given there. I'll show you the commands needed for this, but if you want to learn more about npm or need help troubleshooting anything, visit <https://docs.npmjs.com/getting-started/>.

B.1.1 Installing Sass

To install Sass, create a new project directory and navigate to it in your terminal. Then run the following two commands:

- `npm init -y`—Initializes a new npm project, creating a `package.json` file. See chapter 10 (section 10.1.1) for more on this file.
- `npm install --save-dev node-sass`—Installs the `node-sass` package and adds it to `package.json` as a development dependency.

NOTE On Windows, you will also need to install the `node-gyp` package. See <https://github.com/sass/node-sass#install> for more information.

The second decision you'll need to make is which syntax to use. Sass supports two: Sass and SCSS. They both offer the same features, but the Sass syntax omits all curly

braces and semicolons, strictly using indentation to indicate the structure of your code. For example:

```
body
  font-family: Helvetica, sans-serif
  color: black
```

This is akin to whitespace-significant programming languages, such as Ruby and Python. The SCSS syntax uses braces and semicolons, so it looks more like regular CSS. For example:

```
body {
  font-family: Helvetica, sans-serif;
  color: black;
}
```

SCSS is more commonly used. If you're unsure, I suggest using SCSS, which is what I'll use in this appendix.

NOTE SCSS files use the `.scss` file extension, whereas Sass files use `.sass`.

B.1.2 *Running Sass*

Now that Sass is installed, let's use it to build a stylesheet. In your project directory, create two subdirectories called `sass` and `build`. You'll put your source files in the `sass` directory, and Sass will use those files to produce a CSS file in the `build` directory. Next, edit the `package.json` file. Change the `scripts` entry to match this listing.

Listing B.1 Adding a sass command to package.json

```
"scripts": {
  "sass": "sass sass/index.scss build/styles.css"
},
```

This defines a `sass` command that, when run, compiles the file at `sass/index.scss` to a new file at `build/styles.css`. The file `sass/index.scss` doesn't exist yet in your project. Go ahead and create it. Your Sass code will go into this file. Running `npm run sass` executes the command, producing (or overwriting) the stylesheet at `build/styles.css`.

TIP Plugins such as `gulp-sass` are available for common task runners like Grunt, Gulp, and Webpack. If you want to use a plugin, look for one that integrates Sass or Less into the workflow you're most familiar with.

B.1.3 *Understanding important Sass features*

I've shown you one example of a Sass variable (`$brand-blue`). Add the code in the next listing to your `index.scss` file to see Sass compile it for you.

Listing B.2 A Sass variable

```

$brand-blue: #0086b3;      ← Defines a variable

a:link {
  color: $brand-blue;      ← Uses the variable
}

.page-heading {
  font-size: 1.6rem;
  color: $brand-blue;     ← Uses the variable
}

```

Run `npm run sass` to compile this into CSS. The output file (`build/styles.css`) will look like this:

```

a:link {
  color: #0086b3; }

.page-heading {
  font-size: 1.6rem;
  color: #0086b3; }

/*# sourceMappingURL=styles.css.map */

```

The variables have been replaced with the hex value, so now the browser can understand it. Sass also produced a source map file and added a comment to the end of the stylesheet, giving a path to the *source map*.



source map—A file that the computer uses to trace each generated line of code (CSS, in our case) back to the source code that produced it (Sass). This map file can be used by some debuggers, including the browser’s DevTools.

Notice that the compiled code isn’t formatted as cleanly; the closing braces are brought up onto the previous line and, in some cases, empty lines are removed. This is okay because whitespace doesn’t matter to the browser. But, for the rest of the examples in this appendix, I’ll clean up the output formatting so that its meaning is clear.

INLINE COMPUTATION

Sass also supports inline arithmetic using `+`, `-`, `*`, `/`, and `%` (for modular division). This lets you derive multiple values from one source value as shown next.

Listing B.3 Using inline computations

```

$padding-left: 3em;

.note-author {
  left-padding: $padding-left; ← Uses a variable
}

```

```

    font-weight: bold;
  }

  .note-body {
    left-padding: $padding-left * 2;
  }

```

← Multiplies variable by two

Use `npm run sass` to compile this, which produces the output:

```

.note-author {
  left-padding: 3em;
  font-weight: bold;
}

.note-body {
  left-padding: 6em;
}

```

This feature is useful when two values are related, but not the same. In this case, a `note-body` will always have twice as much left padding as a `note-author`, regardless of the value of `$padding-left`.

NESTED SELECTORS

Sass allows you to nest selectors inside other declaration blocks. You can use nesting to group related code in the same block as shown here.

Listing B.4 Nesting selectors

```

.site-nav {
  display: flex;

  > li {
    margin-top: 0;

    &.is-active {
      display: block;
    }
  }
}

```

← Nested selector

← Ampersand indicates where the outer selector will be appended.

Sass merges nested selectors with the selectors of the outer declaration block(s). This example compiles to:

```

.site-nav {
  display: flex;
}

.site-nav > li {
  margin-top: 0;
}

.site-nav > li.is-active {
  font-weight: bold;
}

```

By default, the outer `.site-nav` selector is prepended to every selector in the compiled code, and a space is added where the selectors are joined. To change this, use an ampersand (`&`) to indicate where you want the outer selector to be inserted.

WARNING Nesting increases the specificity of the resulting selectors. Be cautious about nesting and avoid nesting several levels deep.

You can also nest media queries inside a declaration block. This can be used to avoid repeating the same selector as shown in the next listing.

Listing B.5 Nesting a media query

```
html {
  font-size: 1rem;
  @media (min-width: 45em) {
    font-size: 1.25rem;
  }
}
```

← | Media query inside
a declaration block

This compiles to

```
html {
  font-size: 1rem;
}

@media (min-width: 45em) {
  html {
    font-size: 1.25rem;
  }
}
```

This way, if you change a selector, you won't have to remember to change the corresponding selector in a media query to match.

PARTIALS (@IMPORT)

Partials let you split your styles into multiple separate files, and Sass will concatenate them all together into one file. Using partials, you can organize your files however you wish, but only serve one file to the browser, thereby reducing the number of network requests.

Create a new file in your project as `sass/button.scss`. Add the styles shown here to this file.

Listing B.6 Button partial stylesheet

```
.button {
  padding: 1em 1.25em;
  background-color: #265559;
  color: #333;
}
```

Then, in `index.scss`, import the partial stylesheet using the `@import` at-rule as shown here.

Listing B.7 Importing a partial

```
@import "button";
```



When you run Sass, the partial file will be compiled and inserted where you indicated with the `@import` rule.

In my opinion, this is the most important feature of a preprocessor. As your stylesheet grows, it becomes unwieldy to scroll through thousands of lines of code to find the appropriate part of the stylesheet. This feature lets you break up the stylesheet into small, logical modules, without incurring a performance loss over the network. See the “Preprocessors and modular CSS” sidebar in chapter 9 for more on this.

MIXINS

A *mixin* is a small chunk of CSS that you can re-use throughout your stylesheet. This is useful when you have a particular font style you need to match in multiple places, or for commonly repeated rules, such as the clearfix (discussed in chapter 4, section 4.2).

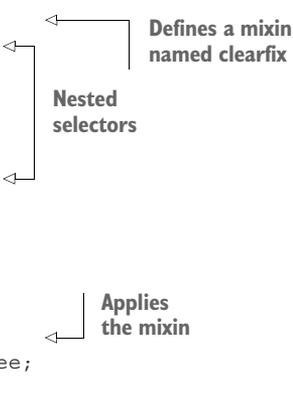
A mixin is defined using an `@mixin` at-rule and used with an `@include` at-rule. Here is an example of a clearfix mixin.

Listing B.8 Clearfix mixin

```
@mixin clearfix {
  &::before {
    display: table;
    content: " ";
  }

  &::after {
    clear: both;
  }
}

.media {
  @include clearfix;
  background-color: #eee;
}
```



The preprocessor takes the code from the mixin and copies it in place of the `@include` rule. The resulting code looks like:

```
.media {
  background-color: #eee;
}
.media::before {
  display: table;
  content: " ";
}
```

```
.media::after {
  clear: both;
}
```

Notice there's no mention of clearfix in the resulting code. The mixin's contents are only added to the stylesheet in the place(s) where it's used.

You can also define mixins that take parameters, much like a function in conventional programming. The next listing shows a mixin that defines an alert box. It takes two parameters, `$color` and `$bg-color`, which are variables defined within the scope of the mixin.

Listing B.9 Mixin with parameters

```
@mixin alert-variant($color, $bg-color) {
  padding: 0.3em 0.5em;
  border: 1px solid $color;
  color: $color;
  background-color: $bg-color;
}

.alert-info {
  @include alert-variant(blue, lightblue)
}

.alert-danger {
  @include alert-variant(red, pink)
}
```

← Defines a mixin with two parameters

Parameter variables can be used within the mixin.

← Passes values into the mixin

Each time the mixin is used, different values are passed in. These values are assigned to the two variables accordingly. This snippet produces the following CSS output:

```
.alert-info {
  padding: 0.3em 0.5em;
  border: 1px solid blue;
  color: blue;
  background-color: lightblue;
}

.alert-danger {
  padding: 0.3em 0.5em;
  border: 1px solid red;
  color: red;
  background-color: pink;
}
```

Again the mixin allows you to re-use the same chunk of code several times, but in this case, it produced two variations of the same code. These differences are based on the values you passed in.

WARNING Historically, one common use for mixins has been to mix in vendor-prefixed versions of a property. For example, a `border-radius` mixin might specify the `-webkit-border-radius`, `-moz-border-radius`, and `border-radius` properties. I encourage you to not use mixins for this; use Autoprefixer instead (for more about this, see the section “PostCSS” later in this appendix).

EXTEND

Sass also supports an `@extend` at-rule. This is similar to a mixin, but the way it compiles is different. Instead of copying the same declarations multiple times, Sass groups selectors together so they're all in the same ruleset. This is best explained by an example. In the next listing, the `.message` ruleset is extended by the two other rulesets.

Listing B.10 Extending a base class

```
.message {
  padding: 0.3em 0.5em;
  border-radius: 0.5em;
}

.message-info {
  @extend .message;
  color: blue;
  background-color: lightblue;
}

.message-danger {
  @extend .message;
  color: red;
  background-color: pink;
}
```



Shares styles with the `.message` class

This produces the following output:

```
.message,
.message-info,
.message-danger {
  padding: 0.3em 0.5em;
  border-radius: 0.5em;
}

.message-info {
  color: blue;
  background-color: lightblue;
}

.message-danger {
  color: red;
  background-color: pink;
}
```

Notice that Sass copied the `.message-info` and `.message-danger` selectors up onto the first ruleset. The benefit from this is that your markup only needs to reference one class instead of two: `<div class="message message-info">` becomes `<div class="message-info">` because the `message-info` class now also includes all styles for the `message` class as well, making the use of the `message` class redundant.

WARNING Unlike a mixin, `@extend` moves the selector to an earlier location in the stylesheet. This means the source order of your declarations might not match what you expected, which can affect the cascade.

The length of the output from `@extend` is generally shorter than that of a mixin. It's easy to see this and think that it's better because the resulting stylesheet is smaller (and, therefore, faster over the network). But, it's important to note that mixins produce a lot of repeated code—and repetitive code compresses very well when gzipped. As long as your server is gzipping all network traffic (which it should), these gains are typically much smaller than you might expect.

Don't avoid mixins and exclusively use `@extend` to provide some sort of performance optimization. Consider your code organization and which, mixins or extends, makes more sense to use on a case-by-case basis. In general, you should probably favor mixins. Only use `@extend` when you want to shorten the number of class names needed in your HTML, as in listing B.10.

COLOR MANIPULATION

Another useful feature of Sass is a series of functions that allow you to manipulate colors. If you need two related colors (for example, a lighter and darker version of the same green), you can use the functions in the next listing to produce the colors you need.

Listing B.11 Sass color functions

<code>\$green: #63a35c;</code>	
<code>\$green-dark: darken(\$green, 10%);</code>	↳ Darkens by 10%
<code>\$green-light: lighten(\$green, 10%);</code>	↳ Lightens by 10%
<code>\$green-vivid: saturate(\$green, 20%);</code>	Adjusts color saturation
<code>\$green-dull: desaturate(\$green, 20%);</code>	
<code>\$purple: adjust-hue(\$green, 180deg);</code>	Rotates hue around the color wheel
<code>\$yellow: adjust-hue(\$green, -70deg);</code>	
<code>\$green-transparent: rgba(\$green, 0.5);</code>	← Adjusts transparency

By using these functions, you can edit the value of one variable, but allow the change to affect other, related, colors. You don't have to store the value in a variable. You can edit it directly in the property where you need it:

```
.page-header {
  color: $green;
  background-color: lighten($green, 50%);
}
```

If you need to do more advanced manipulations, there're several more color functions. See <http://jackiebalzer.com/color> for a comprehensive reference.

LOOPS

Use loops to iterate over a value, producing slight variations. In chapter 15, I used several `:nth-child()` selectors to target consecutive menu items, giving each a slightly different `transition-delay` (listing 15.10). This sort of code can be done more succinctly with a Sass loop, which uses a `@for` at-rule as shown in the following listing.

Listing B.12 Iterating over a series of values

```

@for $index from 2 to 5 {
  .nav-links > li:nth-child(#{ $index }) {
    transition-delay: (0.1s * $index) - 0.1s;
  }
}

```

Iterates the \$index value from 2 to 5

Uses the variable in the selector

Multiplies the variable by a time value

This renders the same block of code several times, each time incrementing the \$index variable. Notice I used the variable in the selector, escaping it with the #{} notation. The resulting code looks like this:

```

.nav-links > li:nth-child(2) {
  transition-delay: 0.1s;
}

.nav-links > li:nth-child(3) {
  transition-delay: 0.2s;
}

.nav-links > li:nth-child(4) {
  transition-delay: 0.3s;
}

```

In plain CSS, making changes to this pattern could be tedious. If I decided to increment the transition-delay by 0.15 seconds, I'd have to manually change each of these declarations to 0.15 seconds, 0.3 seconds, and 0.45 seconds, successively. Or, if I wanted to add another iteration, I'd have to manually copy a block and change all the values. But with the Sass loop, these changes are a matter of editing the math formula or changing the iteration count.

IT'S ALL CSS

Preprocessors don't change the fundamentals of CSS. Everything I covered in this book still applies. I didn't use Sass throughout the book because I wanted to be clear that the topics covered are the essentials of the language itself, not of any one preprocessor. You still need to understand CSS in order to use Sass. But Sass (or Less) can remove much of the busy work when working with CSS. Sass is an extremely useful tool. I encourage you to become familiar with it.

B.2 PostCSS

PostCSS (<http://postcss.org/>) is a different sort of preprocessor. Like Sass or Less, it parses a source file and outputs a processed CSS file. PostCSS, however, is entirely plugin-based. Without any plugins, the output file would be an unchanged copy of the source file.

The functionality you get out of PostCSS is entirely determined by the plugins you use. You can use multiple plugins that provide all the same functionality as Sass, or you

can use one or two plugins and run your code through both Sass and PostCSS. And, if you want, you can even write your own plugins in JavaScript.

It's important to note that PostCSS runs plugins sequentially. If you configure multiple plugins, the order they execute is sometimes significant, and it might take a little trial-and-error to get PostCSS working the way you want. See the PostCSS documentation for help with this configuration.

NOTE PostCSS was initially referred to as a post-processor because it's generally run after the preprocessor. PostCSS has moved away from this particular term, however, as it implies a more limited scope of all the tool can offer.

B.2.1 Using Autoprefixer

Perhaps the most important plugin for PostCSS is Autoprefixer. This plugin adds all the appropriate vendor prefixes to your CSS. For more on vendor prefixes, see the "Vendor prefixes" sidebar in chapter 5.

If your source code looks like this:

```
.example {
  display: flex;
  transition: all 0.5s;
  background: linear-gradient(to bottom, white, black);
}
```

Autoprefixer adds additional declarations, providing vendor-prefixed fallbacks to older browsers, and then outputs:

```
.example {
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
  -webkit-transition: all .5s;
  transition: all .5s;
  background: -webkit-gradient(linear, left top, left bottom, from(white),
    to(black));
  background: linear-gradient(to bottom, white, black);
}
```

If you had to write all these vendor prefixes by hand, it would be tedious and prone to error. It also adds a lot of clutter to your source code that you probably would rather not have to think about when working.

You can configure Autoprefixer with a list of browsers you want to support, and it'll add vendor prefixes where necessary to support those browsers. For example, configuring it with the array ["ie >= 10", "last 2"] ensures that your code is compatible (when possible) with IE10 and up, and the latest two versions of all other browsers. Autoprefixer uses the latest data from the caniuse.com database to determine when prefixes are needed.

I strongly recommend you use Autoprefixer, even if you don't use any other PostCSS plugins. Throughout the book, I don't include vendor prefixes in my code examples, on the assumption that you'll use Autoprefixer to do this for you.

B.2.2 *Using cssnext*

Another popular PostCSS plugin (or rather, a bundle of plugins) is *cssnext* (<http://cssnext.io/>). These plugins attempt to emulate future CSS syntaxes that aren't supported by all browsers yet (and some of which aren't finalized in the CSS specification). This is in many ways like using a polyfill for future CSS features.

Many of the features for this plugin are similar to those provided by Sass: nesting selectors, mixin-like behavior using an `@apply` at-rule, and color-manipulation functions. Autoprefixer is also included in this bundle. See <http://cssnext.io/features/> for a complete list of the plugin features.

Be aware that a few of these features are still in the early stages of development by the W3C, and are almost certain to change before being finalized. Use *cssnext* if you want to get a feel for some up-and-coming CSS features, but I don't recommend it as your only set of preprocessor rules. As browsers add native support for some of the *cssnext* features, it may be difficult to transition from processing them using PostCSS to using them natively in the browser; it's a good idea to keep your preprocessor rules separate from your polyfilled rules.

B.2.3 *Using cssnano*

cssnano (<http://cssnano.co/>) is a PostCSS-based minifier. A *minifier* strips all extraneous whitespace from your code and makes it as small as possible, while still maintaining the same syntactic meaning.

NOTE Minification isn't a replacement for gzip compression, which should be applied by your server. It's generally a best practice to both minify and gzip CSS for the network to speed up loading time.

Several CSS minifiers are available, but it might make more sense to do this as part of your PostCSS build process, rather than in a separate step. *cssnano* allows you to do this.

B.2.4 *Using PreCSS*

PreCSS (<https://github.com/jonathantneal/precss>) is a PostCSS plugin pack that provides several Sass-like features. This includes `$` variables, inline calculation, loops, and mixins.

If it feels inefficient to run your code through both a Sass preprocessor and PostCSS, consider replacing Sass with the *PreCSS* plugin pack for PostCSS instead. It's not perfectly identical to Sass, however, so consult the *PreCSS* documentation if you choose to go this route. It's also a newer tool and may not be quite as stable as Sass.

Symbols

:: (double-colon) 420
!important annotation 9, 11, 13, 18, 217
* (universal selector) 12, 61, 81
*= comparator 109
@ symbol 7
& (ampersands) 427
- (hyphens) character 49
+ (adjacent sibling combinator) 417
> (child combinator) 417

A

<a> elements 123, 317, 374
absolute units 28, 30
absolute value 31
active pseudo-class 288
addition operators 46
adjacent sibling combinator 417
affix element 199
Alfa Slab One font 375
align-content property 137, 139, 175–176
align-items property 137, 139, 175
alignment
 with Flexbox 135–142
 of grid layouts 175–176
align-self property 138, 140, 175
all keyword 355
ampersands 427
Andrew, Rachel 176
animation property 399

animation-delay property 407
animation-duration
 property 399
animation-fill-mode
 property 408
animation-iteration-count
 property 399
animation-name property 399
animation-timing-function
 property 399
anonymous div 91
Aphrodite 252
APIs (application program
 interfaces), using pattern
 libraries as 270–276
 editing existing modules
 271–274
 refactoring code 274–276
 using semver 274–276
Archibald, Jake 143
attribute selectors 109, 420–421
attributes 7
author styles 8
auto value 64, 92, 351
auto-fill keyword 165
auto-fit keyword 166
Autoprefixer post
 processor 122, 146,
 433–434

B

background-attachment
 property 23, 281
background-chip property 23
background-clip property 281

background-color property 23,
 281, 364
background-image property 23,
 280
background-origin property 23,
 281
background-position
 property 23, 26, 280
background-repeat property 23,
 280
backgrounds 279–299
background-size property 23,
 280, 293
back-ticks 260
base rules 234
baseline grid 345
BEM (Block, Element,
 Modifier) 251
BFC (block formatting
 context) 102–105
 establishing 103–104
 overview 194
 using for media object
 layouts 104–105
blend modes 279–291
 mix blend modes 298–299
 texture 296–298
 tinting images 294–295
 types of 295–296
Block, Element, Modifier 251
block-level elements 58, 64, 99,
 123, 244, 374
blur radius 287
body copy, spacing 340–341
<body> element 18, 24, 33, 80,
 178, 196

Bootstrap 275–276
 border-box property 60
 border-box sizing, universal 61–62
 border-collapse property 18
 border-color property 23, 190
 border-radius property 33, 116, 355
 border-spacing property 18, 67
 border-style property 23
 border-width property 23
 box model 55–83

- collapsed margins 74–77
 - between text 74–75
 - multiple margins 75
 - outside containers 76–77
- element height, difficulties with 64–72
 - applying alternatives to percentage-based heights 65–70
 - controlling overflow behavior 64–65
 - max-height 70
 - min-height 70
 - vertically centering content 71–72
- element width, difficulties with 56–63
 - adjusting box models 59–61
 - gutters between columns 62–63
 - magic numbers, avoiding 59
 - universal border-box sizing 61–62
- negative margins 73
- spacing elements within containers 77
 - changing content 79–80
 - lobotomized owl selector 80
- box-shadow property 26, 287
- box-sizing property 59
- brand-color property 49
- breakpoints
 - adding to pages 217–221
 - overview 204, 215
 - selecting 222
- browsers, converting colors in 313–314
- Button modules, variants of 238–240
- button-cta module 306
- buttons, creating 290–291

C

calc() function

- for font size 45–46
- overview 59, 365

 Call to Action (CTA) 140, 142, 311
 Cartesian coordinate system 193
 Cartesian grids 26
 cascade 4–18

- animations 399
- source order 15–17
 - cascaded values 17
 - link styles and 16–17
- specificity 10–15
 - inline styles 10
 - notation for 12–13
 - selector specificity 11–12
 - troubleshooting 13–15
- stylesheet origins 8–10, 399
- important declarations 9–10
 - user agent styles 8–9
 - working with 17–18

 cascaded values 17
 cascading stylesheets. *See* CSS
 case-insensitive attribute selectors 421
 centering content, vertically 71–72, 412
 cf (contain floats) 93, 97, 99, 108, 250, 267
 child combinator 417
 class attribute 109, 417
 classes

- specificity 11–13
- state classes 246–247
- utility classes 250

 class-naming conventions 252
 clear property 95
 clearfix 93–99, 108, 250, 267
 code, refactoring 274–276
 collapsing

- containers 93–99
- margins 74–77
 - between text 74–75
 - multiple 75
 - outside containers 76–77

 colon syntax 97, 418–420
 color 320–328

- adding to palettes 316–318
- converting in browsers 313–314
- for fonts, contrast for 318–320
- manipulating 431
- notation 312–316
- overview 300–306

 color functions, Sass 431
 color picker dialog box 315
 color property 7, 18
 color stops, using multiple 283–285

- repeating gradients 284–285
- stripes 283–284

 color-burn blend mode 293
 colspan attribute 69
 columns

- of equal height 65–66, 130
- in a grid system 107–109
- gutters between 62–63
- See also* responsive columns, adding

 combinators 12, 417–418
 components, resizing 41–43
 compound selectors 418
 compression tools 227
 computed value 31
 configuration, of KSS 256–257
 contain floats (cf) 93, 97, 99, 108, 250, 267
 contain value 293
 container queries 222
 containers

- collapsing 93–99
- collapsing margins outside 76–77
- spacing elements within 77
 - changing content 79–80
 - lobotomized owl selector 80

 containing block 182
 content

- changing 79–80
- vertically centering 71–72

 content attribute 97, 184, 213
 context-dependent selectors 240–241
 contrast 300–328

- establishing patterns 303
- for font colors 318–320
- implementing designs 304–306
- overview 300–302

 contrast blend modes 295–297
 contrast ratios 318
 control points 359
 converting colors, in browsers 313–314
 cover value 293

CSS (cascading stylesheets)
 methodologies 251
 table layouts 66–69
 variables 48–54
 changing dynamically 50–53
 changing with
 JavaScript 53–54
 experimenting with 54
 CSS First workflow 269–270
 CSS3 language 44
 cssnano tool 434
 cssnext plugin 434
 CTA (Call to Action) 140, 142, 311
 cubic-bezier() function 359, 385
 cursor property 135
 custom properties. *See* CSS, variables

D

declaration block 7
 declarations, marked as
 important 9–10
 dense keyword 167
 depth, defining with gradients
 and shadows 288–289
 descendant combinator 417
 designs, implementing 304–306
 See also responsive design
 development dependencies 256
 DevTools 20, 103, 314, 358
 direct descendant
 combinators 40, 121, 417
 display property 68–69, 117, 364, 366
 div wrapper 69
 document flow 64, 87–88, 95, 103, 142, 178, 182, 185, 190, 194
 documenting
 in KSS, writing 257–261
 module variants in KSS 261–263
 modules requiring
 JavaScript 264–266
 DOM (Document Object Model) 36
 dots per inch (dpi) unit 216
 double container pattern 92, 310
 double-colon syntax 97, 420

double-hyphen notation 239–240, 305
 double-underscore notation 241–243, 305
 dpi (dots per inch) unit 216
 dropdown menu 186–188, 244–246, 264–266, 361–368

E

ease-in value 356
 ease-out value 356
 editing existing modules 271–274
 element queries 222
 elements
 creating with flat
 designs 289–290
 height of 64–72
 applying alternatives to percentage-based heights 65–70
 controlling overflow behavior 64–65
 max-height 70
 min-height 70
 vertically centering content 71–72
 modules with multiple 241–243
 generic tag names, avoiding 243
 variants with sub-elements 243
 spacing
 inline 326–327
 overview 342–345
 spacing within containers 77
 changing content 79–80
 lobotomized owl selector 80
 width of 56–63
 box models, adjusting 59–61
 gutters between columns 62–63
 magic numbers, avoiding 59
 universal border-box sizing 61–62
 Elements pane, DevTools 317
 empty string 97
 ems 31–37
 to define font-size property 32–36

pixels vs. 37–43, 320–322
 making panels responsive 40–41
 resizing single components 41–43
 setting default font sizes 39–40
 encapsulation 234
 -end suffixes 159
 equal height, columns of 65–66, 130
 error modifier 237
 evergreen browsers 66
 explicit grid 162
 @extend rule 430
 extends 430–431

F

fallbacks 337–338
 feature queries 172–175, 404
 Firefox Developer Edition 146
 Firefox Nightly 146
 first-child selector 418
 first-of-type selector 418
 Flash of Invisible Text. *See* FOIT (Flash of Invisible Text)
 flash of unstyled text. *See* FOUT (Flash of Unstyled Text)
 flat designs, creating elements with 289–290
 flex container properties 135–139
 align-content property 139
 align-items property 139
 flex-flow property 138
 flex-wrap property 138
 justify-content property 138–139
 flex direction 130–135
 changing 132–133
 styling login forms 133–135
 flex items
 properties 139–140
 align-self property 140
 order property 140
 sizes 124–130
 flex-basis property 126–127
 flex-grow property 127–128
 flex-shrink property 128–129
 practical uses of 129–130
 flex-basis property 126–128, 138

- flexbox 69–70, 116–143
 - alignment 135–142
 - cautions when using 142–143
 - Flexbugs 142
 - full-page layouts 142–143
 - flex container properties 135–139
 - align-content property 139
 - align-items property 139
 - flex-flow property 138
 - flex-wrap property 138
 - justify-content property 138–139
 - flex direction 130–135
 - changing 132–133
 - styling login forms 133–135
 - flex item properties 139–140
 - align-self property 140
 - order property 140
 - flex item sizes 124–130
 - flex-basis property 126–127
 - flex-grow property 127–128
 - flex-shrink property 128–129
 - practical uses of 129–130
 - Grid Layout and 155–158
 - menus 120–122
 - overview 117–124
 - padding 123–124
 - spacing 123–124, 135–142
 - Flexbugs 142
 - flex-direction property 136
 - flex-flow property 136, 138
 - flex-grow property 126–128, 137, 170
 - flex-shrink property 126, 128–129, 137, 224
 - flex-wrap property 136, 138
 - float-left class 267
 - floats 87–115
 - clearfix 93–99
 - container collapsing 93–99
 - grid systems 105–115
 - building 107–111
 - gutters 111–115
 - overview 106–107
 - purpose of 88–93
 - unexpected float catching 99–101
 - fluid layouts 223–227
 - overview 202
 - styles for large viewports 224–225
 - tables with 226–227
 - Flyin-Grid module 401
 - FOIT (Flash of Invisible Text) 346–352
 - Font Face Observer 348–349
 - font-display property 351–352
 - system fonts 349–351
 - Font Face Observer 348–349
 - font property 18
 - FontAwesome 382
 - font-display property 351–352
 - @font-face ruleset 336–339
 - fallbacks 337–338
 - font formats 337–338
 - font-family property 18, 23, 335
 - fonts
 - font formats 337–338
 - sizes of
 - calc() for 45–46
 - setting defaults 39–40
 - shrinking, problems with 34–36
 - vw for 45
 - system fonts 349–351
 - web fonts 331–332
 - fonts-failed class 349–350
 - font-size property
 - ems to define 32–36
 - combining with other ems 33–34
 - shrinking fonts 34–36
 - overview 18, 23
 - rems for 36–37
 - fonts-loaded class 348–349
 - font-style property 18, 23
 - font-variant property 18
 - font-weight property 18, 23, 46
 - @for at-rule 431
 - forward slash 154
 - Foundation 275–276
 - FOUT (Flash of Unstyled Text) 346–352
 - Font Face Observer 348–349
 - font-display property 351–352
 - system fonts 349–351
 - fraction unit 147
 - Frost, Brad 222
 - fuzzy values 38
- G**
-
- Google Fonts 331–336
 - GPU (graphics processing unit) 388
 - grad unit 282
 - gradients 280–287
 - defining depth with 288–289
 - multiple color stops 283–285
 - repeating gradients 284–285
 - stripes 283–284
 - radial gradients 285–287
 - graphics processing unit. *See* GPU
 - grid area 148
 - grid cell 148
 - grid container 147
 - Grid Layout module 144–176
 - alignment in 175–176
 - alternate syntaxes 158–162
 - naming grid areas 160–162
 - naming grid lines 158–160
 - anatomy of grids 148–158
 - flexbox 155–158
 - numbering grid lines 153–154
 - explicit and implicit grids 162–172
 - adjusting grid items to fill grid tracks 169–172
 - variety, adding to 166–168
 - feature queries 172–175
 - overview 145–148
 - of grid layouts 175–176
 - grid line 148
 - grid track 148
 - grid-auto-columns property 162
 - grid-auto-flow property 167
 - grid-auto-rows property 162
 - grid-gap property 147
 - grids 105–115, 144, 176
 - adjusting items to fill tracks 169–172
 - alignment in 175
 - areas, naming 160–162
 - building 107–111, 146–148
 - gutters 111–115
 - lines
 - naming 158–160
 - numbering 153–154
 - overview 106–107
 - variety, adding to 166–168
 - grid-template property 160
 - grid-template-areas property 161
 - grid-template-columns property 147, 152
 - grid-template-rows property 147, 152

gulp-sass plugin 424
 gutters, adding
 between columns 62–63
 overview 111–115
 gzip compression 434

H

handles 359
 hard-light blend mode
 296–297
 hardware acceleration 388
 has-error class 247
 <head> element 80, 335
 headings, spacing 342–345
 height 64–72
 columns of equal height
 65–66, 130
 controlling overflow
 behavior 64–65
 max-height 70
 min-height 70
 vertically centering
 content 71–72
 See also percentage-based
 heights, applying alterna-
 tives to
 height containers 72
 Helvetica Neue typeface 319
 Hero module 310
 hidden value 64, 366
 home-link module 309
 horizontal offset value 287
 horizontal overflow 65
 HSL notation, switching
 stylesheet to 314–316
 hsl() function 312–313
 <html> element 36, 313,
 348
 hyphens 49

I

icon fonts 382
 ID selector 11, 417
 images
 multiple, for different
 viewport sizes
 227–228
 srcset to serve 228–229
 texture, adding to 296–298
 tinting 294–295
 element 170
 implicit grid 162
 @import at-rule 247, 427–428

@include at-rule 428
 index.scss file 424
 inherit keyword 21–22, 62
 inheritance 18–19
 inherited font size 32
 initial containing block 183
 initial keyword 22–23
 initializing projects, in
 KSS 256
 inline computation 425–426
 inline elements, spacing
 of 326–327
 inline styles 252
 inline-block property 172
 inline-grid value 147
 inset keyword 289, 291
 installing Sass extension
 423–424
 Inverted Triangle CSS
 (ITCSS) 251
 is-expanded class 247
 is-loading class 247, 411–412
 is-open class 211, 245, 266
 ITCSS (Inverted Triangle
 CSS) 251

J

JavaScript language, changing
 CSS variables with 53–54
 js array 264
 justify-content property 124,
 136, 138–139, 175
 justify-items property 175
 justify-self property 175

K

keyframe 396
 @keyframes at-rule 397
 keywords
 inherit keyword 21–22
 initial keyword 22–23
 KSS (Kyle Style Sheets)
 254–269
 documenting module
 variants 261–263
 documenting modules
 requiring JavaScript
 264–266
 organizing pattern library into
 sections 266–269
 overview pages, creating 264
 setup 255–257
 initializing projects 256

KSS configuration 256–257
 KSS documentation,
 writing 257–261
 kss-config.json file 264

L

landmarks 210
 large class 41
 last-child selector 418
 last-of-type selector 418
 late-binding styles 29
 layouts, full-page 142–143
 See also fluid layouts
 leading 324, 342
 left property 357
 length values 46, 355
 Less (leaner style sheets) 48,
 196, 247, 423
 letter-spacing property 18, 339,
 341
 elements 123
 libraries. *See* pattern libraries
 LibSass 423
 line height, spacing 323–326
 line wrapping 155
 linear keyword 359
 linear-gradient() function
 281–283
 line-height property 18, 23,
 46–48, 339–340
 link styles, source order
 and 16–17
 link, visited, hover, active
 (LoVe/HATe) 17
 <link> tag 5, 335
 liquid layout 223
 list-style property 18
 list-style-image property 18
 list-style-position property 18
 list-style-type property 18
 lobotomized owl selector 80
 login forms, styling 133–135
 loops 431–432
 LoVe/HATe (link, visited, hover,
 active) 17

M

magic numbers, avoiding 59
 main element 160
 <main> elements 58, 125
 main-nav element 379
 Marcotte, Ethan 202
 margin property 25

margin-left property 124
 margins
 collapsing 74–77
 between text 74–75
 multiple 75
 outside containers 76–77
 negative 73
 markdown 260
 Markup annotation 260
 Martin, Robert C. 243
 max-height property 70
 max-width property 70, 92, 216
 @media at-rule 40, 202, 214
 media boxes 94
 media object 102–105, 241–243, 271–273
 media queries 214–222
 breakpoints, adding to pages 217–221
 overview 40, 202
 responsive columns 221–222
 types of 215–217
 max-width 216
 media types 217
 min-width 216
 media types 217
 Menu module 247–248
 menus
 building in flexbox 120–122
 mobile 209–213
 See also dropdown menu
 menu-toggle property 210
 meta tags, adding viewport meta tags 213–214
 min-height property 70, 293
 min-height value 181
 minmax() function 165
 min-width property 70, 216
 mix blend modes 298–299
 mixins 428–429
 mobile first approach 202–214
 mobile menus 209–213
 viewport meta tags 213–214
 mobile styles 218
 modal-body element 180, 182
 modern browsers 66
 modifier_class annotation 261, 263
 modifiers 237
 modules 233–235, 243–251
 base rules 234–235
 composed into larger structures 243–250

dividing responsibilities
 among 244–248
 Menu module 247–248
 state classes 246–247
 documenting variants in
 KSS 261–263
 editing 271–274
 methodologies 251
 naming 248–250
 positioning in 246
 requiring JavaScript, documenting in KSS 264–266
 utility classes 250
 variations of 237–241
 Button modules 238–240
 context-dependent selectors 240–241
 with multiple elements 241, 243
 generic tag names, avoiding 243
 variants with sub-elements 243
 multiply blend mode 292

N

naming
 grid areas 160–162
 grid lines 158–160
 modules 248–250
 nav spacing 321
 <nav> element 207, 400
 nav-container module 309
 navigational (nav) menu 375
 negative margins 73
 nested flexboxes 133
 nested lists 35–36
 nested selectors 426–427
 Node.js 423
 node-sass package 423
 normal document flow 64
 normal keyword 340
 normalize.css library 235
 not() selector 134, 419
 notation
 for colors 312–316
 for specificity 12–13
 npm run build command 257, 264, 274
 nth-child selector 101, 109, 418
 nth-last-child selector 418
 nth-last-of-type selector 419

nth-of-type selector 419
 numbering grid lines 153–154

O

object-fit property 170
 objects. *See* media objects
 only-child selector 418
 only-of-type selector 418
 OOCSS (Object-Oriented CSS) 105, 251
 opacity property 361, 365
 optional value 351
 order property 138, 140
 over-and-back animation 398
 overflow, behavior 64–65
 overflow-x property 65
 overflow-y property 65
 overlay blend mode 296–297
 overview pages, creating in KSS 264

P

package.json file 256
 padding property 25, 33, 59–60
 padding, adding in flexbox 123–124
 pages, adding breakpoints to 217–221
 palettes, adding colors to 316–318
 panels, responsive 40–41
 partials 427–428
 pattern libraries
 CSS First workflow 269–270
 KSS 254–269
 documenting module variants 261–263
 documenting modules requiring JavaScript 264–266
 overview pages 264
 setup 255–257
 organizing into sections 266–269
 overview 253–269, 276
 using as API 270–276
 editing existing modules 271–274
 refactoring code 274–276
 semver 274–276
 patterns, establishing 303

- percentage-based heights,
 - applying alternatives to 65–70
 - columns of equal height 65–66
 - CSS table layouts 66–69
 - flexboxes 69–70
- percentages 46
- perspective() function 391–392
- picas 30
- Pickering, Heydon 80
- pixel-perfect designs
 - end of 29–30
 - struggle for 29
- pixels. *See* px (pixels), ems vs.
- placeholders 308
- placement algorithm 154
- pointer value 135
- points (pt) 30
- polyfill 349
- position property 178
- positioning 177
 - absolute positioning 182
 - close buttons 182
 - pseudo-element 183
 - fixed 178
 - controlling size of positioned elements 182
 - modal dialogs with 178
 - relative 185
 - CSS triangles 188
 - dropdown menus 186
 - sticky 197
- PostCSS tool 432–434
 - Autoprefixer 433–434
 - cssnano 434
 - cssnext 434
 - PreCSS 434
- PreCSS tool 434
- preprocessors 422–434
 - PostCSS 432–434
 - Autoprefixer 433–434
 - cssnano 434
 - cssnext 434
 - PreCSS 434
 - Sass 423–432
 - color manipulation 431
 - with CSS 432
 - extends 430–431
 - features of 424–432
 - inline computation 425–426
 - installing 423–424
 - loops 431–432
 - mixins 428–429
 - nested selectors 426–427
 - partials (@import) 427–428
 - running 424
 - preserve-3d transform style 395
 - preventDefault() function 413
 - print media query 217
 - print styles 217
 - projects, initializing in KSS 256
 - Promise.all() method 349
 - properties. *See* shorthand properties
 - pseudo-elements
 - overview 36, 96, 98
 - positioning 183
 - selectors 419–420
 - pt (points) 30
 - Pure 275–276
 - px (pixels), ems vs. 37–43
 - making panels responsive 40–41
 - overview 320–322
 - resizing single
 - components 41–43
 - setting default font sizes 39–40

Q

 - queries
 - container 222
 - element 222
 - feature 172, 175
 - See also* media queries

R

 - rad unit 282
 - radial gradients 285–287
 - Raleway font 375
 - ReactJS 252
 - readability, adjusting spacing
 - for
 - body copy 340–341
 - headings 342–345
 - overview 339–342
 - small elements 342–345
 - refactoring code 274–276
 - relative positioning 185
 - CSS triangles 188
 - dropdown menus 186
 - relative units
 - CSS variables 48–54
 - changing dynamically 50–53
 - changing with JavaScript 53–54
 - experimenting with 54
 - ems 28–31, 37–54
 - pixels vs. 37–43
 - to define font-size property 32–36
 - line-height property 46–48
 - power of relative values 29–30
 - rems 31–37
 - unitless numbers 46–48
 - viewport-relative units 43–46
 - calc() for font size 45–46
 - vh for font size 45
 - relative values 29–30
 - rem-based breakpoints 214
 - rems
 - for font-size property 36–37
 - relative units 28, 31–37
 - rendering 191
 - repeat() function 152, 160, 405
 - repeating gradients 284–285
 - repeating-linear-gradient() function 284, 286
 - resizing components 41–43
 - responsive columns,
 - adding 221–222
 - responsive design 201–229
 - fluid layouts 223–227
 - styles for large viewports 224–225
 - tables with 226–227
 - media queries 214–222
 - breakpoints, adding to pages 217–221
 - responsive columns 221–222
 - types of 215–217
 - mobile first approach 202–214
 - mobile menus 209–213
 - viewport meta tags 213–214
 - responsive images 227–229
 - multiple images for different viewport sizes 227–228
 - srcset to serve correct images 228–229
 - rgb() function 312–313
 - Roberts, Harry 251
 - Roboto font 331
 - root node 36
 - rotate() function 371–372

rotateX() function 389, 393
 rotateY() function 389, 393, 405
 rotateZ() function 389
 row class 207
 row-reverse 130
 rowspan attribute 69
 Rules pane, Firefox 358
 rulesets 6–7

S

Safari Technology Preview 146

Sansita font 331

sans-serif font 6, 49, 331

Sass (syntactically awesome

stylesheets) 423–432

color manipulation 431

extends 430–431

features of 424–432

inline computation

425–426

installing 423–424

loops 431–432

mixins 428–429

nested selectors 426–427

overview 48, 196, 247

partials (@import) 427–428

running 424

using with CSS 432

Scalable and Modular Architecture for CSS 251

Scalable Vector Graphics 377, 382

scale() function 373, 382

screen media query 217

<script> tag 53, 178, 414

scroll value 64

scrollHeight property 368

selectors 7, 417–421

attribute selectors 420–421

basic selectors 417

combinators 417–418

context-dependent 240–241

nested 426–427

pseudo-class selectors

418–419

pseudo-element selectors

419–420

specificity 11–12

semver (Semantic Versioning) 274

specification 274–276

serif font 331

shadows 279–287, 291–299

buttons and 290–291

depth with 288–289

elements with flat designs 289–290

shake animation 413

Shelburne, Natalya 318

shorthand properties 23–27

order of values 24–27

horizontal, vertical

26–27

top, right, bottom, left

25–26

overriding other styles

with 23–24

shrinking fonts, problems

with 34–36

sibling combinator 80

single page applications. *See*

SPAs

Single Responsibility

Principle 244

single-colon syntax 97, 418–419

size attribute 135

sizes

of positioned elements,

controlling 182

of viewports, using multiple

images for 227–228

skeuomorphism 289

skew() function 375

SMACSS (Scalable and Modular Architecture for CSS) 251

Snook, Jonathan 251

soft-light blend mode 296–297

Soueidan, Sara 389

source map 425

source order 15–17

cascaded values 17

link styles and 16–17

overview 6

space-around value 139, 176

space-between value 139, 176

space-evenly value 176

spacing 300–320, 327–328

adding grid layout 175–176

adding in flexbox 123–124

adjusting for readability

339–345

for body copy 340–341

headings 342–345

small elements 342–345

elements within containers

77

changing content 79–80

lobotomized owl

selector 80

ems vs. pixels 320–322

line height 323–326

 element 374

SPAs (single page

applications) 252

special values 20–23

inherit keyword 21–22

initial keyword 22–23

specificity 10–15

inline styles 10

notation for 12–13

preventing escalation

of 240

selector specificity 11–12

troubleshooting 13–15

sprite sheet 382

srcset attribute, using to serve

images 228–229

stacking contexts 177

overview 194

z-index and 190

manipulating stacking

order with z-index

193

rendering processes 191

stacking order 191

stacking order

manipulating with z-index

193

overview 191

start keyword 361

-start suffixes 159

state classes 246–247

steps() function 360

sticky positioning 197

stripes 283–284

style attribute 10

style guide 254

style inspector 20

<style> tag 10

Styled Components 252

Styleguide annotation 255,

257, 268

Styleguide Utilities.clearfix

annotation 267

styles

for large viewports

224–225

inline 10

linking 16

shorthand properties,

overriding 23–24

Styles pane, Chrome 358

stylesheets
 origins 8–10
 important declarations 9–10
 user agent styles 8–9
 switching to HSL 314–316
 sub-elements, using module
 variants with 243
 subgrids 168
 subtraction operators 46
 Sullivan, Nicole 102, 251
 @supports rule 172, 174–175
 SVG (Scalable Vector Graphics) 377, 382
 swap value 351
 syntactically awesome
 stylesheets. *See* Sass
 syntaxes 158–162
 naming grid areas 160–162
 naming grid lines 158–160
 system fonts 349–351

T

<table> elements 69
 table-caption display type 77
 table-cell display type 77
 table-inline display type 77
 table-row display type 77
 tables, with fluid layouts 226–227
 tag names, avoiding generic tag names 243
 tag selector 11, 235, 417
 tag types 243
 <td> element 226
 text, collapsing margins
 between 74–75
 text-align property 18
 text-center class 250, 267
 text-indent property 18, 184
 text-shadow property 26, 209, 287
 text-transform property 18, 134, 342, 345
 texture, adding to images 296–298
 tinting images 294–295
 top, right, bottom, left 25
 top-nav module 309–310
 <tr> element 226
 tracking 342
 transform property 291
 transform-origin property 374

transition-delay property 355, 386, 407
 transition-duration
 property 354–355
 transition-property 354–355
 transition-timing-function
 property 355
 translate() function 374–375, 385
 translateX() function 389
 translateY() function 291, 389
 translateZ() function 389, 405
 transparent keyword 282, 292
 triangles, CSS 188
 TRouBLE (top, right, bottom, left) 25
 troubleshooting specificity 13–15, 240
 Tudor, Ana 395
 turn unit 282
 type selector 11, 235, 417
 Typekit 331
 typography 329–352
 @font-face 336–339
 fallbacks 337–338
 font formats 337–338
 FOUT and FOIT 346–352
 Font Face Observer 348–349
 font-display property 351–352
 system fonts 349–351
 Google Fonts 332–336
 spacing 339–345
 for body copy 340–341
 headings 342–345
 small elements 342–345
 variants of 338–339
 web fonts 331–332

U

unclickable elements 73
 unicode character 184, 211
 unitless numbers 46–48
 units. *See* relative units
 universal selector 12, 61, 80
 url() function 365
 user agent styles 8–9
 user stylesheets 8
 user-scalable=no option 214
 utility classes 250

V

values 20–23
 inherit keyword 21–22
 initial keyword 22–23
 See also relative values
 var() function 50
 vendor prefixes 146, 433
 See also Autoprefixer
 vertical centering 72
 vertical offset value 287
 vertical rhythm 345
 vertical-align property 71, 175
 viewport size 204
 viewport-relative units 43–46
 calc() for font size 45–46
 vh for font size 45
 viewports
 meta tags 213–214
 multiple images for different sizes 227–228
 styles for 224–225
 visibility property 366
 visible value 64
 vw unit, for font size 45

W

W3C (World Wide Web Consortium) 44
 Wagner, Jeremy L. 352
 WCAG (Web Content Accessibility Guidelines) 318
 web fonts 331–332
 Web Open Font Format 337
 web safe fonts 329
 Webkit Nightly Builds
 edition 146
 -webkit-animation property 399
 Webtype 331
 Weight annotation 267
 Westfall, Brad 92
 whitespace 148, 302
 white-space property 18
 width 56–63
 box models, adjusting 59–61
 gutters between columns 62–63
 magic numbers, avoiding 59
 universal border-box sizing 61–62
 width property 135

will-change property 389
WOFF2 (Web Open Font
Format) 337
word-spacing property 18
World Wide Web Consortium
44
writing KSS documentation
257–261

Y

-y flag 256
Yandex 251
yawing 389

Z

z-index property
manipulating stacking order
with 193
overview 46
stacking contexts and 190
rendering process 191
stacking order 191

CSS IN DEPTH

Keith J. Grant

Some websites really pop. They look great, they're visually consistent, and they feel interactive and responsive. You can bet their developers knew CSS in depth. CSS specifies everything from the structural layout of page elements to their individual look and feel. True masters know the patterns of CSS development, the techniques to implement them, and the subtle touches that result in beautiful typography, fluid transitions, and balanced graphics. Join them!

CSS in Depth exposes you to a world of CSS techniques that range from clever to mind-blowing. This instantly useful book is packed with creative examples and powerful best practices that will sharpen your technical skills and inspire your sense of design. You'll gain new insights into familiar features like floats and units, and experiment with emerging ideas like responsive design and pattern libraries. Bottom line: this book will make you a better web designer and your apps will look fantastic!

What's Inside

- Avoid common CSS pitfalls
- Master misunderstood concepts
- Use flexbox and grid layout
- Responsive designs for any device
- Code for reuse and maintainability

Written for web developers who know the basics of CSS and HTML.

Keith J. Grant is a senior web developer who builds and maintains web applications and websites, including The New York Stock Exchange site.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/css-in-depth



“Become better at writing code that lasts and is understandable and performant.”

—From the Foreword by
Chris Coyier
Cofounder of CodePen

“From zero to hero in CSS!”

—Pierfrancesco D’Orsogna
GamePix

“The bible of the most up-to-date CSS.”

—Phily Austria
Faraday Future

“A well-written, concise book. I enjoyed every minute of reading it.”

—Tanya Wilke, Sanlam

“A clear and complete guide to CSS.”

—Giancarlo Massari, Unic

ISBN-13: 978-1-61729-345-0
ISBN-10: 1-61729-345-8



9 781617 293450