

 MANNING

CSS

IN DEPTH

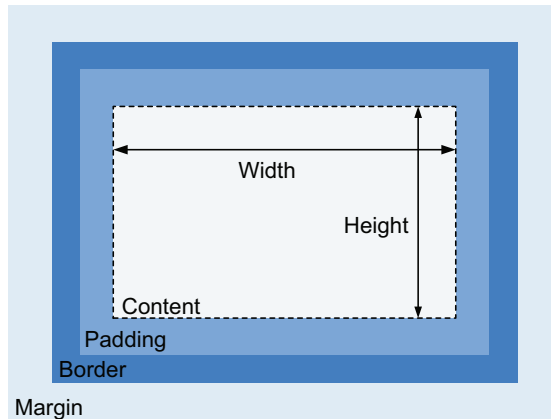
SECOND EDITION

Keith J. Grant
Foreword by Chris Coyier

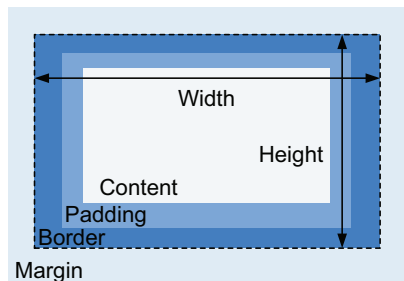


Box model and border box sizing

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 box-sizing: 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 more information on applying border box sizing to the entire page, as well as some other important concepts, including centering content and controlling spacing between elements.

Praise for the first edition

“An incredible way to learn beyond the basic aspects of CSS.”

—Andres Sacco, technical leader at TraveleX

“If I had to start over learning CSS and could choose only one book, this would be it.”

—Seth MacPherson, senior developer at Quote Factory

“One of the best, if not the best book for CSS.”

—Scott Ling, self-employed contractor

“For novices in CSS, this book provides a gradual learning path, while seasoned engineers will find comprehensive coverage of advanced concepts, thoughtfully explained. Ultimately, this book serves as a guide toward mastery.”

—Doyle Turner, software engineer at Incremental Systems

“The most comprehensive and up-to-date book on CSS. Easily modernize your CSS skills with this painless, progressive training.”

—Becky Huett, engineer at Maxar Intelligence

CSS in Depth, Second Edition

KEITH J. GRANT

FOREWORD BY CHRIS COYIER



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

©2024 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% recycled and processed without the use of elemental chlorine.

The authors and publisher have made every effort to ensure that the information in this book was correct at press time. The authors and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

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

Development editor: Sarah Harter
Technical editor: Chris Chinchilla
Review editors: Isidora Isakov and Radmila Ercegovic
Production editor: Andy Marinkovich
Copy editor: Kari Lucke
Proofreader: Katie Tennant
Technical proofreader: Doyle Turner
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781633437555
Printed in the United States of America

brief contents

PART 1	REVIEWING THE FUNDAMENTALS.....	1
1	■ Cascade, specificity, and inheritance	3
2	■ Working with relative units	35
3	■ Document flow and the box model	62
PART 2	MASTERING LAYOUT.....	91
4	■ Flexbox	93
5	■ Grid layout	119
6	■ Positioning and stacking contexts	153
7	■ Responsive design	179
PART 3	MODERN CODE ORGANIZATION.....	209
8	■ Cascade layers and nesting	211
9	■ Modular CSS and scope	238
10	■ Container queries	270

PART 4	VISUAL ENHANCEMENTS.....	291
11	■ Color and contrast	293
12	■ Typography and spacing	326
13	■ Gradients, shadows, and blend modes	357
14	■ Masks, shapes, and clipping	383
PART 5	ADDING MOTION	409
15	■ Transitions	411
16	■ Transforms	431
17	■ Animations	459

contents

foreword xv
preface xvii
acknowledgments xix
about this book xxi
about the author xxvi
about the cover illustration xxvii

PART 1 REVIEWING THE FUNDAMENTALS.....1

1 *Cascade, specificity, and inheritance* 3

1.1 The cascade 5

Stylesheet origin 8 ■ *Inline styles* 11 ■ *Selector specificity* 11
Source order 16

1.2 Inheritance 19

1.3 Special values 21

The inherit keyword 21 ■ *The initial keyword* 23 ■ *The unset keyword* 23 ■ *The revert keyword* 24

1.4 Shorthand properties 25

Beware shorthands silently overriding other styles 25 ■ *Remember the order of shorthand values* 26

- 1.5 Progressive enhancement 29
 - Using the cascade for progressive enhancement* 29 ▪ *Progressively enhancing selectors* 30 ▪ *Feature queries using @supports()* 31

2 *Working with relative units* 35

- 2.1 The power of relative units 36
 - The rise of responsive design* 36
- 2.2 Ems and rems 37
 - Using ems to define font-size* 39 ▪ *Using rems for font-size* 43
- 2.3 Stop thinking in pixels 44
 - Setting a sane default font size* 45 ▪ *Making the panel responsive* 47 ▪ *Resizing a single component* 48
- 2.4 Viewport-relative units 50
 - Selecting from the newer viewport units* 52 ▪ *Using viewport units for font size* 54
- 2.5 Unitless numbers and line-height 55
- 2.6 Custom properties (aka CSS variables) 57
 - Changing custom properties dynamically* 59

3 *Document flow and the box model* 62

- 3.1 Normal document flow 63
 - Centering content horizontally* 65 ▪ *Using logical properties* 67
 - Adopting useful shorthand logical properties* 69
- 3.2 The box model 70
 - Avoiding magic numbers* 72 ▪ *Adjusting the box model* 73
 - Using universal border box sizing* 74
- 3.3 Element height 75
 - Controlling overflow behavior* 75 ▪ *Using alternatives to percentage-based heights* 76 ▪ *Using min-height and max-height* 77
- 3.4 Negative margins 78
- 3.5 Collapsed margins 79
 - Collapsing between text* 80 ▪ *Collapsing multiple margins* 80
 - Collapsing outside a container* 81
- 3.6 Spacing elements within a container 83
 - Considering changing content* 85 ▪ *Creating a more general solution* 86

PART 2 MASTERING LAYOUT 91

4 Flexbox 93

- 4.1 Flexbox principles 94
 - Building a basic flexbox menu* 97 ▪ *Adding padding and spacing* 98
- 4.2 Flex item sizes 100
 - Flex basis* 102 ▪ *Flex grow* 102 ▪ *Flex shrink* 104
 - Some practical examples* 105
- 4.3 Flex direction 106
 - Changing the flex direction* 108 ▪ *Styling the login form* 109
- 4.4 Alignment, spacing, and other details 110
 - Understanding flex container properties* 114 ▪ *Understanding flex item properties* 116 ▪ *Using alignment properties* 117

5 Grid layout 119

- 5.1 Building a basic grid 120
- 5.2 Anatomy of a grid 122
 - Numbering grid lines* 127 ▪ *Working together with flexbox* 128
- 5.3 Alternate syntaxes 132
 - Naming grid lines* 132 ▪ *Naming grid areas* 134
- 5.4 Explicit and implicit grid 136
 - Adding variety* 140 ▪ *Adjusting grid items to fill the grid track* 142
- 5.5 Subgrid 146
 - Additional options* 150
- 5.6 Alignment properties 150

6 Positioning and stacking contexts 153

- 6.1 Fixed positioning 154
 - Creating a modal dialog with fixed positioning* 154 ▪ *Preventing the screen from scrolling while the modal dialog is open* 158
 - Controlling the size of positioned elements* 159
- 6.2 Absolute positioning 160
 - Absolutely positioning the Close button* 160 ▪ *Positioning a pseudo-element* 161

- 6.3 Relative positioning 162
 - Creating a dropdown menu* 163 ■ *Creating a CSS triangle* 166
- 6.4 Stacking contexts and z-index 169
 - Understanding the rendering process and stacking order* 169
 - Manipulating stacking order with z-index* 171 ■ *Understanding stacking contexts* 172
- 6.5 Sticky positioning 175

7 *Responsive design* 179

- 7.1 Mobile first 180
 - Creating a mobile menu* 188 ■ *Adding the viewport meta tag* 191
- 7.2 Media queries 192
 - Understanding types of media queries* 194 ■ *Adding breakpoints to the page* 197 ■ *Adding responsive columns* 200
- 7.3 Fluid layouts 202
 - Adding styles for a large viewport* 203 ■ *Dealing with tables* 204
- 7.4 Responsive images 206
 - Using multiple images for different viewport sizes* 206 ■ *Using srcset to serve the correct image* 207

PART 3 MODERN CODE ORGANIZATION.....209

8 *Cascade layers and nesting* 211

- 8.1 Manipulating the cascade with layers 212
 - Defining layers* 212 ■ *Layer order and priority* 216
 - The revert-layer keyword* 219
- 8.2 A recommended organization for cascade layers 221
 - Reset layer* 221 ■ *Theme layer* 223 ■ *Global layer* 224
 - Layout layer* 225 ■ *Modules layer* 226 ■ *Utilities layer* 227
 - Additional layers* 228
- 8.3 The :is() and :where() pseudo-classes 228
 - More forgiving selectors* 229 ■ *Manipulating specificity* 229
- 8.4 Nesting 230
 - Using the nesting selector* 231 ■ *Understanding the nuances of the nesting selector* 233 ■ *Nesting media queries and other at-rules* 235

9 Modular CSS and scope 238

9.1 Defining modules 239

Modules and global styles 240 ▪ *A simple module* 241
Variations of a module 242 ▪ *Modules with multiple elements* 246

9.2 Modules composed into larger structures 249

Dividing multiple responsibilities among modules 250
Naming modules 254

9.3 CSS scope 256

Scope proximity 258 ▪ *Scoping limit* 261 ▪ *Implicit scope* 264
Scope and layers 265

9.4 Pattern libraries 265

Using a CSS-first workflow 266 ▪ *Refactoring and breaking changes* 267

10 Container queries 270

10.1 A basic example of a container query 271

Using container size queries 274

10.2 A closer look at containers 276

Container types 276 ▪ *Container names* 278 ▪ *Containers and modular CSS* 279

10.3 Container-relative units 280

10.4 Container style queries 283

Decoupling a module from its container 284 ▪ *Reducing code duplication* 285

PART 4 VISUAL ENHANCEMENTS.....291

11 Color and contrast 293

11.1 Communicating with contrast 295

Establishing patterns 296 ▪ *Implementing the design* 297

11.2 Defining color 299

Gamuts and color spaces 306 ▪ *CSS color notations* 308

11.3 Using OKLCH to work with color 313

Switching the stylesheet to OKLCH 315 ▪ *Naming color variables* 316 ▪ *Selecting new colors for the palette* 318
Deriving colors from others on the page 318

11.4 Considering contrast for font colors 323

12 *Typography and spacing* 326

12.1 Spacing 327

Using ems vs. px 327 ▪ *Factoring in line height* 329
Spacing inline elements 333

12.2 Web fonts 335

12.3 Google fonts 337

12.4 How @font-face works 341

Font formats and fallbacks 343 ▪ *Multiple variants of the same typeface* 343

12.5 Performance considerations 344

The font-display property 346 ▪ *Variable fonts* 348

12.6 Adjusting space for readability 350

Body copy spacing 351 ▪ *Headings, small elements, and spacing* 353

13 *Gradients, shadows, and blend modes* 357

13.1 Gradients 358

Multiple color stops 360 ▪ *Color interpolation* 363
Radial gradients 366 ▪ *Conic gradients* 368

13.2 Shadows 370

Adding depth with gradients and shadows 370 ▪ *Creating elements with a flat design* 372 ▪ *Creating buttons with a hybrid look* 373

13.3 Blend modes 374

Tinting an image 376 ▪ *Types of blend modes* 377
Adding texture to an image 378 ▪ *Mix blend modes* 380

14 *Masks, shapes, and clipping* 383

14.1 Filters 384

Types of filters 386 ▪ *Backdrop filter* 387

14.2 Masks 389

Masking with a gradient 391 ▪ *Masking using luminance* 392
Other mask properties 393

14.3 Clipping paths 394

Polygon clipping paths 396 ▪ *Firefox clip-path tools* 397
Other clip-path types 399

- 14.4 Floats and shapes 399
 - Floating* 400 ■ *Defining a shape* 404

PART 5 ADDING MOTION 409

15 Transitions 411

- 15.1 From here to there 412
- 15.2 Timing functions 415
 - Custom Bézier curves* 416 ■ *Steps* 419
- 15.3 Non-animatable properties 420
 - Properties that cannot be animated* 423 ■ *Fading in and out* 424
- 15.4 Transitioning to auto height 426
- 15.5 Transitioning custom properties 428

16 Transforms 431

- 16.1 Rotate, translate, scale, and skew 432
 - Changing the transform origin* 435 ■ *Applying multiple transforms* 436 ■ *Individual transform properties* 436
- 16.2 Transforms in motion 437
 - Scaling up the icon* 443 ■ *Creating “fly in” labels* 446
 - Staggering the transitions* 448
- 16.3 Animation performance 449
 - Looking at the rendering pipeline* 449
- 16.4 3D transforms 451
 - Controlling perspective* 452 ■ *Implementing advanced 3D transforms* 455

17 Animations 459

- 17.1 Keyframes 460
- 17.2 Animating 3D transforms 463
 - Building the layout without animations* 463 ■ *Adding animation to the layout* 466
- 17.3 Animation delay and fill mode 468
- 17.4 Conveying meaning through animation 471
 - Responding to user interaction* 471 ■ *Drawing the user’s attention* 475

17.5	Scroll-based timelines	477
17.6	One final piece of advice	479
<i>appendix A</i>	<i>Selectors reference</i>	<i>481</i>
<i>appendix B</i>	<i>Preprocessors</i>	<i>488</i>
	<i>index</i>	<i>501</i>

foreword

“A minute to learn, 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 just a few functions. 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 few 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 dive deep 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 "Oh!" bits that surprise you and extend that base.

—CHRIS COYIER, cofounder of 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 backend and the frontend, 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. 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 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. We’ve seen the rise of newer, modern layout methods and effects like blend modes, filters, transformations,

transitions, and animations. Even in the last year or two, logical properties, cascade layers, and container queries have dramatically changed the way we write the language. 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:

- *It's essential.* There are many fundamentals of the language that, sadly, many developers don't fully understand. These include the cascade, page layout, and positioning. I'll take a deep look at these, explaining how they work.
- *It's new.* A lot of new features have emerged in the last few years or are just now emerging. 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 future of CSS development, and my goal is to provide you with an understanding of the language that is ready for emerging changes.
- *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. It is a comprehensive guide intended to teach practical proficiency in the language.

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 few 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.

The excellent staff at Manning worked hard with me to make this the best book it can be. Thanks go to Manning’s publisher, Marjan Bace, who saw the potential in this idea and the need for an updated edition. I thank my editors Toni Arritola and Sarah Harter for their commitment to quality. This is a much better book because of their input. Thanks go to my technical editor, Chris Chinchilla, for his insight throughout the writing process, and to Doyle Turner for a meticulous technical proofread of the final manuscript and all the code listings.

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: Al Pezewski, Andres Sacco, Becky Huett, Craig Sharkie, Darek Kazik, Esref Durna, Harrison Maseko, Holger Voges, Joe Cuevas, Marlen Waaijer, Mikael Byström, Patrick Regan, Paulo Vieira, Rick Bunnell, Rodney Weis, Santosh Joseph, Sara Wallén, Scott Ling, Seth MacPherson, Simon Verhoeven, Stephan Max, Sudheer Kumar Reddy Gowrigari, and Yves Dorfsman. Your feedback offered valuable early insight into how the book would be received by developers of all skill levels.

Finally, I feel 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. Keep up the good work!

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—I said this when I wrote the first edition, and it’s true all over again now, in the second. This book aims to meet both of 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, in strategic places, 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 some 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 knowledge gaps and help you understand the world of CSS as a whole.

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 imbue you with 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: A roadmap

The book is 17 chapters long, divided into five 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 document flow and the box model. This involves understanding how the HTML elements are laid out in a page and how you can control the size of those elements 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 teaches flexbox, which is used for laying out a series of elements on the page. It begins with the fundamental concepts and moves on to practical layout examples.
- Chapter 5 introduces grid layout. This allows you to structure the contents of the page into rows and columns—an indispensable part of modern page layout.
- Chapter 6 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 7 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, “Modern code organization,” you’ll look at some new and upcoming features of the language, as well as some coding 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 8 covers cascade layers and nesting, which are essential tools for organizing your code.
- Chapter 9 teaches how to organize your CSS in a modular way, so that your code is reusable and maintainable. I also introduce scope, which is a new feature of CSS that helps with this.
- Chapter 10 walks you through container queries. This builds on the responsive design principles from chapter 7.

In part 4, “Visual enhancements,” 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 shows how to work with color and visual contrast. Attention to these details goes a long way toward making a good design a great one.
- Chapter 12 is about web typography: using online font files to bring a unique personality to your site or app.
- Chapter 13 discusses shadows, gradients, and blend modes. These work together to build an elegant user interface.
- Chapter 14 looks at more advanced visual effects using filters, masks, clipping, and shapes.

Finally, in part 5, “Adding motion,” you’ll learn how to bring your page to life with movement:

- Chapter 15 brings motion to the page with transitions, changing the shape, color, or size of an element on the page.
- Chapter 16 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 17 discusses keyframe animations. You’ll learn how to use complex motion to communicate meaning to the user.

There are also two appendixes:

- Appendix A is a reference of all the types of CSS selectors.
- Appendix B is an introduction to preprocessors.

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 (➞). 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.

You can get executable snippets of code from the liveBook (online) version of this book at <https://livebook.manning.com/book/css-in-depth-second-edition>. The complete code for the examples in the book is available for download from the Manning website at www.manning.com and from GitHub at <https://github.com/cssinddepth/css-in-depth-2>. 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.

liveBook discussion forum

Purchase of *CSS in Depth, Second Edition*, includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to <https://livebook.manning.com/book/css-in-depth-second-edition/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/discussion>.

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 some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website for as long as the book is in print.

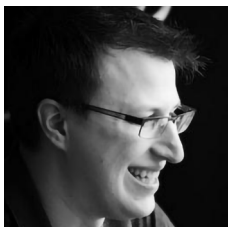
Browser versions

Cross-browser testing is an important part of web development. Most of the code in this book is supported in recent versions of Chrome, Firefox, Safari, Microsoft Edge, 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 older browsers as an acceptable compromise. I show examples of this in several places.

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

about the author



KEITH J. GRANT is currently a principal software engineer at Red Hat, where he works on frontend code for the Ansible Automation Platform using JavaScript and CSS.

He is self-taught in HTML and CSS, and he had several years of informal experience working with the technology before graduating with a degree in computer science from Gonzaga University. He has 18 years of professional experience building and maintaining web applications and websites using HTML, CSS, and JavaScript.

Though Keith has primarily been a JavaScript developer, he has become an important CSS instructor at every company he's worked for. He writes about CSS and other web development topics at <https://keithjgrant.com>.

ABOUT THE TECHNICAL EDITOR

Chris Chinchilla is a technical writer, podcaster, and video maker. Chris has worked for small and large technical projects and companies and creates fiction, interactive fiction, and games in his spare time.

about the cover illustration

The figure on the cover of *CSS in Depth, Second Edition*, is taken from a nineteenth-century collection of works by many artists, edited by Louis Curmer and published in Paris in 1841. Each illustration is finely drawn and colored by hand.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture 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, document flow, and box model. These fundamentals, covered in these first three chapters, control what styles apply 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 six criteria 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
- Working with new and upcoming CSS features

CSS is unlike a lot of things in the world of software development. It's not a programming language in the conventional sense, 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 the language 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.

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*. The following declaration comprises 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 `<a href="/">`, `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. The selector indicates the element or elements on the page where the declarations will be applied. See appendix A for a comprehensive reference of selector types. Here is an example with a `body` selector, targeting the `<body>` element:

```
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.

Part 1 begins with the most fundamental principles of CSS: 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, I'll begin with the *C* in *CSS*—the cascade. I'll articulate how it works and then show you how to work with it practically. I'll 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 are 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, you 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.

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 you will build in this chapter

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-2>. 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>
<html lang="en-US">
<head>
  <meta charset="utf-8" />
  <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>
</html>
```

Diagram annotations for Listing 1.1:

- 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 `<li><a href="/specials" class="featured">Specials</a></li>` 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 the following code 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;
}
```

Diagram annotations for Listing 1.2:

- Tag (or type) selector**: Points to the `h1 {` line.
- ID selector**: Points to the `#page-title {` line.
- Class selector**: Points to the `.title {` line.

Rulesets with conflicting declarations can appear one after the other, or they can be scattered throughout your stylesheet. Either way, given your HTML, these 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 wins due to the ID in its selector; the title will have a sans serif font (figure 1.2).

Wombat Coffee Roasters

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

Sans serif title font

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

This set of rules is called the *cascade*. 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.

Let's unpack the cascade. When declarations conflict, the cascade considers six criteria in the following order to resolve the difference. We will look at each of these in depth:

- 1 *Stylesheet origin*—Where the styles come from. Your styles are applied in conjunction with the browser's default styles.
- 2 *Inline styles*—Whether a declaration is applied to an element via the HTML style attribute or a CSS selector.
- 3 *Layer*—Styles can be defined in layers, each with a different priority.
- 4 *Selector specificity*—Which selectors take precedence over which.
- 5 *Scope proximity*—Whether the styles are scoped to a portion of the DOM.
- 6 *Source order*—Order in which styles are declared in the stylesheet.

Some of these criteria are influenced by the use of the `!important` annotation, which we will look at later in the chapter. Figure 1.3 shows how the rules are applied at a high level.

These rules allow browsers to behave predictably when resolving any ambiguity in the CSS. In the previous example, the `#page-title` selector received precedence over the other selectors due to these rules—in particular, the criteria of selector specificity, which I'll explain in detail shortly.

Layers and *scope* are new additions to CSS that provide more explicit control over the cascade. We will look at these in greater depth in chapters 8 and 9.

Without any layers or scoped styles in your stylesheets, the remaining four aspects of the cascade continue to behave as they have for years, and that is where I'll focus first. I'll step you through them one at a time: origin, inline styles, specificity, and source order.

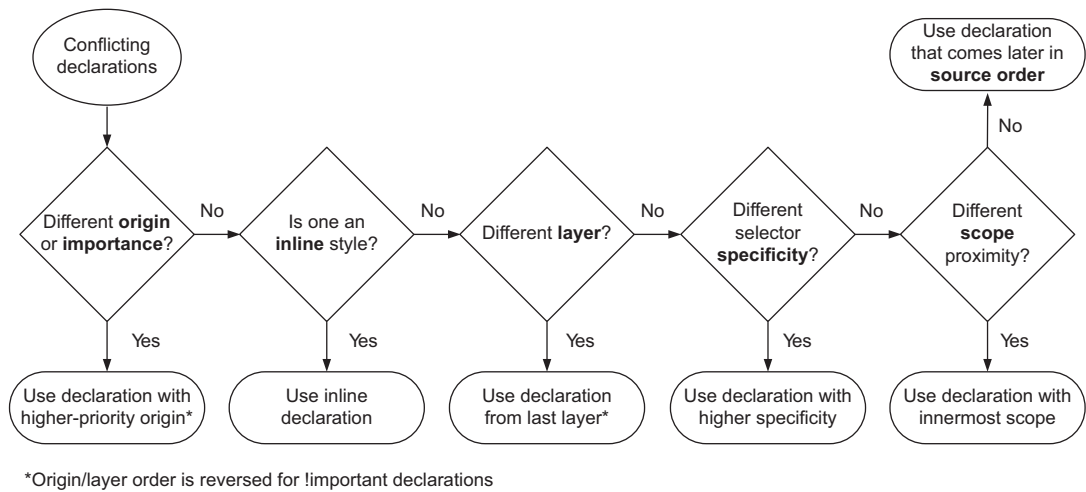


Figure 1.3 High-level flowchart of the cascade showing declaration precedence

1.1.1 Stylesheet origin

The stylesheets you add to your web page aren't the only ones the browser applies. There are three different types, or origins, of stylesheets. The styles you add to your page are called *author* styles. There are also *user* styles, which are customizations added by the end user, and *user-agent* styles, which are the browser's default styles. User-agent styles have lower priority, so your styles override them. User styles are rare, but if present, they have a priority between user-agent styles and author styles.

NOTE User stylesheets can only be defined in certain browsers or with the use of browser extensions. User styles are not commonly used and beyond your control, so you generally don't need to worry about them when styling a page.

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 (`<ol>` and `<ul>`) 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, and the heading has a large font size.

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

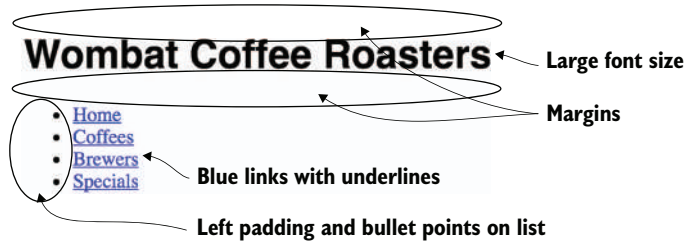


Figure 1.4 User-agent styles set defaults for your web page header.

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. They also tend to be very consistent across recent versions of all modern browsers. 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.

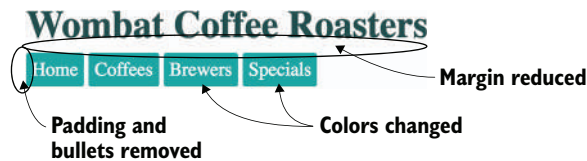


Figure 1.5 Author styles override user-agent styles because they have higher priority.

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;
}
```

← Reduces the margins

← Removes user-agent list styles

← Makes list items appear side by side rather than stacked

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

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 they will always override normal (nonimportant) styles from any origin. All taken together, the overall order of preference, in decreasing order, is as follows:

- Important user-agent
- Important user
- Important author
- Normal author
- Normal user
- Normal user-agent

Declarations from origins earlier in this list will always win over declarations from later origins. Also, notice that the order of important origins is inverted from that of normal origins—we’ll see this concept again when we get to cascade layers in chapter 8.

NOTE Transitions and keyframe animations (covered in chapters 15 and 17) add two additional origins. These effects create “virtual” rules for dynamically changing values that receive special consideration in the cascade so they behave predictably.

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 margins from the user-agent stylesheet can still apply. The `!important` annotation is an interesting quirk of CSS, which we’ll come back to shortly.

1.1.2 Inline styles

If conflicting declarations can't be resolved based on their origin, the browser next considers whether they are added to an element via inline styles. When you use an HTML style attribute to apply styles, the declarations are applied only to that element. These will 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.

On your page, let's make the **Specials** link in the navigation menu orange, as shown in figure 1.6. I'll evaluate several ways you can accomplish this, beginning with inline styles in listing 1.4.



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 as important, then nothing can override them. It's preferable to do this from within the stylesheet. Delete the `style` attribute in listing 1.4 from your page, and we'll look at better approaches.

1.1.3 Selector specificity

Specificity is a feature of CSS that is often not readily apparent when learning the language, but understanding it is essential. You can go a long way without an understanding of stylesheet origin because most of the styles will be ones you add, all belonging to the author origin. But if you don't understand specificity, it will almost certainly cause problems.

If the previous aspects of the cascade cannot resolve a conflict between declarations, the browser seeks to resolve them by looking at the specificity of their 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 if you 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 {           ← Higher-specificity selector
  color: white;
  background-color: #13a4a4;           ← Teal background color
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

.featured {              ← The orange background declaration
  background-color: orange;  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 {   ← Four tags
  color: blue;
}

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

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

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

The most specific selector here is `#page-title`, with one ID, so its color declaration of red is applied to the title. The next specific is `.page-header .title`, with two class names. This would be applied if the ID selector were absent. The third selector, `.page-header .title`, has a higher specificity than the second (`body header.page-header h1`) despite its length: two classes are more specific than one class. Finally, the first selector (`html body header h1`) 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. See appendix A for more on these types of selectors.

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 ON 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 and 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
<code>html body header h1</code>	0	0	4	0,0,4
<code>body header.page-header h1</code>	0	1	3	0,1,3
<code>.page-header .title</code>	0	2	0	0,2,0
<code>#page-title</code>	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 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 because inline styles were originally defined as a subset of specificity in earlier versions of the CSS specification. In this case, an inline style has a notation 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 the following code.

Listing 1.7 Possible fix number 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 override something already set as 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.

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 number two

```
#main-nav a {  
  color: white;  
  background-color: #13a4a4;  
  padding: 5px;  
  border-radius: 2px;
```

Specificity
remains 1,0,1


```

text-decoration: none;
}
#main-nav .featured {
  background-color: orange;
}

```

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 you 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 number 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;
}

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

```

Changes #main-nav to .nav throughout the stylesheet

Lowest the first specificity (0,1,1)

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.

1.1.4 Source order

The final step to resolving the cascade is source order, sometimes referred to as *order of appearance*. If all other criteria 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. This uses a compound selector, `a.featured`, without a space. This selector targets an element that matches both criteria: an `<a>` that has the `featured` class.

Listing 1.10 Possible fix number 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`? For example, if you added `<a class="featured">our specials</a>` elsewhere on the page, you'd 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



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

Figure 1.7 The `featured` class used outside the `nav` 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 websites, 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 number 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, and 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. Add this to the beginning of your stylesheet, before the `nav` styles.

Listing 1.12 Link styles

```

a:link {
  background-color: blue;
  color: white;
  text-decoration: none;
  padding: 2px;
}
a:visited {
  background-color: purple;
}

```

```
a:hover {  
    background-color: transparent;  
    color: blue;  
    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.

TIP You can also use the `:any-link` pseudo-class to target links that match either `:link` or `:visited`.

CASCADED VALUES

The browser follows these steps to resolve every property for every element on the page. A declaration that “wins” the cascade is called a *cascaded value*. There is 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.

DEFINITION A cascaded value is 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.

THE CASCADE IS GETTING EASIER TO WORK WITH

Historically, there are two common rules of thumb for working with the cascade. First, don’t use IDs in your selectors. Second, don’t use `!important`. These both make it difficult to override styles in the future should the need arise.

These two rules can be good advice, but things are changing. There are many situations where using them can be acceptable—using layers and scope, specifically. However, it’s important to never use them as a knee-jerk reaction to win a specificity battle. In part 3, we will examine modern tools for controlling the cascade, and I will show you examples where it is appropriate to break these rules.

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 npm 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.

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 descendant 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.

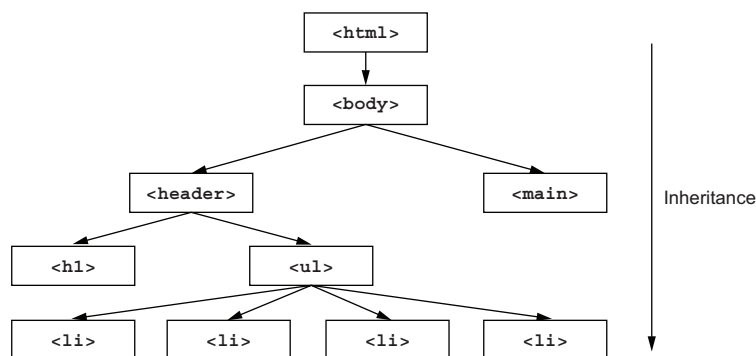


Figure 1.8 Inherited properties are passed down the DOM tree from parent nodes to their descendants.

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 the border behavior of tables, not the more commonly used properties for specifying 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 Applying a font-family to the body allows all descendant elements to 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;
}
```



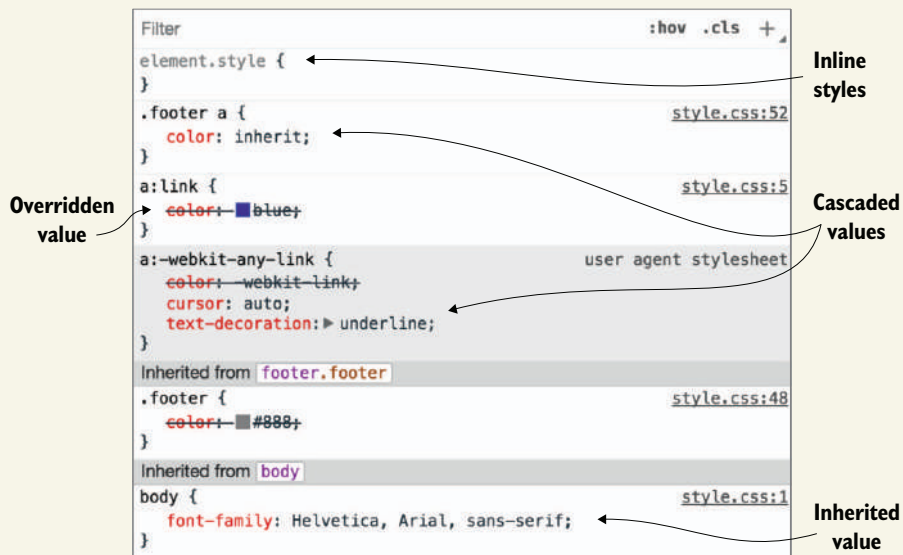
An inherited property will pass down to descendant elements.

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 the value will be inherited by 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. The following figure is an example of what you'll see.



Browser DevTools are the best way to see which styles apply to an element.

The style inspector shows every selector targeting the inspected element, ordered by specificity. Below that are all inherited properties. This shows all 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 some special values that you can apply to any property to help manipulate the cascade: `inherit`, `initial`, `unset`, and `revert`. Let's take a look at these.

You may have noticed I provided slightly unusual styles for links back in listing 1.12: rather than a more traditional blue text, I specified white text and a blue background color. I did this intentionally to help illustrate the behavior of these keywords.

1.3.1 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 isn't an important part of the page. So you'll make the links in the footer dark gray (figure 1.10).

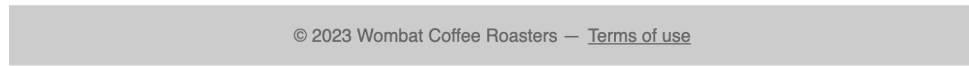


Figure 1.10 The Terms of Use link when it inherits the gray text color

Listing 1.14 shows the markup for this footer. Add this to the end of your page, just before the closing `</body>` tag. 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; 2023 Wombat Coffee Roasters &mdash;
  <a href="/terms-of-use">Terms of use</a>
</footer>
```

Earlier, you applied some styles to all links on the page, and they will be 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

```
.footer {
  color: #666;           ← Footer text color set to gray
  background-color: #ccc;
  padding: 15px 0;
  text-align: center;
  font-size: 14px;
}

.footer a {
  color: inherit;         ← Specifies that font color should inherit from the footer
  background-color: transparent;
  text-decoration: underline;
}                          ← Overrides blue background color
```

The second ruleset here overrides the 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, though you will likely not find many practical uses for this.

1.3.2 The initial keyword

Sometimes you'll find you have styles applied to an element that you want to undo, as you did with the background color in listing 1.15. 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.

Instead of setting `background-color: transparent`, you can use `initial`. This is done in the following listing. Because `transparent` is the initial value for the `background-color` property, this accomplishes the same thing as the previous listing.

Listing 1.16 The initial value

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



Resets the background
color to its initial value

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. In this case, consider using the `revert` keyword instead.

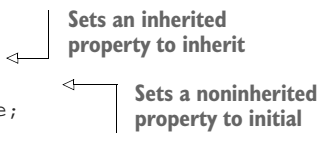
1.3.3 The unset keyword

The `inherit` and `initial` keywords are useful for clearing values you've set on properties that are either inherited or noninherited. The `unset` keyword is a combination of the two. When applied to an inherited property, it sets the value to `inherit`, and when applied to a noninherited property, it sets the value to `initial`.

You can, of course, do this with `inherit` and `initial`, respectively, but using `unset` instead makes it a little simpler and helps you avoid using the wrong keyword by mistake. In this footer, you can use `unset` to fix both the font color and background color of links.

Listing 1.17 The `unset` value

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



Sets an inherited property to inherit

Sets a noninherited property to initial

This sets the color to `inherit`, allowing it to inherit the same gray color as the rest of the footer. But the background color is set to `transparent`, the initial value for the `background-color` property.

You will notice I still had to declare `text-decoration: underline` to restore link underlines. In this case, `unset` would not work, because the initial value of the `text-decoration` property is `none`, not `underline`. Remember, this is the initial value for the CSS property itself, which is always the same. It does not consider the element type it's being applied to. To “undo” the text decoration style we've set, we need one more keyword.

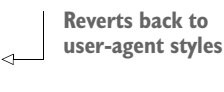
1.3.4 The revert keyword

The `initial` and `unset` keywords essentially override all styles, from both author and user-agent stylesheets. Sometimes what you want is to override your previously set author styles but leave the user-agent styles intact. This is what the `revert` keyword does.

In our footer, if you set all three properties to `revert`, it would revert all the way back to a blue link with an underline (the browser's default styles). But since we like the design we have with the gray link, you can use the keyword more selectively, applying it only to `text-decoration`. Update your page to match listing 1.18 to see this.

Listing 1.18 The `revert` value

```
.footer a {  
  color: unset;  
  background-color: unset;  
  text-decoration: revert;  
}
```



Reverts back to user-agent styles

These keywords may seem like overkill in this example, where the properties are probably familiar to you; you likely know the values you hope for here and could declare them explicitly. But when working with newer CSS features or something like flexbox properties that have multiple keyword values, you may not know off the top of your head what the default values are. When this occurs, these keywords come in handy.

There is also the added benefit of subtly indicating to future authors of the stylesheet that your goal is to effectively undo styles applied elsewhere in the stylesheet.

WARNING These keywords are normal cascaded values. That means it is still possible to override them with other values when another selector with higher specificity targets the same element.

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;
```

Here are some other common shorthand properties:

- `background` is a shorthand 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 1.11).

Wombat Coffee Roasters

Figure 1.11 Using the shorthand `font: 32px sans-serif` sets `font-weight` and other omitted values to their initial value.

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

Listing 1.19 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 the following code.

Listing 1.20 Expanded equivalent to the shorthand in listing 1.19

```
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 Remember 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 *TRouBL*e is a 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 1.12 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 1.12 Declaring padding: 10px 15px 0 5px applies different paddings to each side.

The following listing shows the CSS for these links.

Listing 1.21 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 sides will both use the second value. Specify two values, and the top and bottom will use the first value, while the right and left will use the second value. 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:

```
margin: 1em;
margin: 1em 1em;
margin: 1em 1em 1em;
margin: 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, on our page, navigational links (figure 1.13).



Figure 1.13 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 1.22 Specifying two padding values

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

← | **Top/bottom padding, followed by left/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 shorthand 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` (which 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 value first, followed by the vertical value.

Although it seems counterintuitive 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 1.14, you'd specify the x (horizontal) value first.

Specials

Figure 1.14 Box shadow positioned at 10px 2px

The styles for this element are given here.

Listing 1.23 `box-shadow` specifies x value and 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."

1.5 Progressive enhancement

CSS is constantly evolving, and an important part of working in an evolving language is keeping tabs on which features are still relatively new to the language. In particular, it is good to know when a feature is only supported by a subset of common browsers.

The language is intentionally designed to be both forward and backward compatible, and it provides you with the tools to keep your code working in both old and new browsers at the same time. With a little forethought, you can use cutting-edge features in your CSS even when you know they won't work for all of your users.

To do this, you can add the CSS needed to provide an acceptable (but less full-featured) experience to older browsers, and then you can layer on CSS using newer features, knowing those particular features will only work in the newest browsers. The benefit of this is that it means your code is future compatible, and as more of your users receive browser upgrades, these new features will begin working for them as well. This approach is called *progressive enhancement*.

TIP To see which browser versions support a given feature, check <https://caniuse.com> or the relevant documentation page on MDN at <https://developer.mozilla.org/en-US/docs/Web>.

Some developers have an expectation that brand-new CSS features won't be available to use for years because, in the early days of the web, browser updates were slow. But now all major browsers are *evergreen*, meaning updates are frequent and mostly automatic. Some new features go from 0% support to 80% in just a few months. Progress after that is usually slower, as some users put off clicking that Install Update button (or they are prevented from doing so based on corporate policy). However, unless you have a particular need to support an older browser, you can often expect to rely on a new feature within a year or two after all major browsers add support.

With appropriate progressive enhancement, you often don't even need to wait that long to begin using it—though this will vary depending on the nature of the feature in question and how essential it is to the effect you wish to accomplish.

1.5.1 Using the cascade for progressive enhancement

The simplest way to use progressive enhancement is built into the cascade itself. If a browser encounters a declaration it does not understand, it simply ignores it. Consider the following code:

```
aside {  
  background-color: #333333;  
  background-color: #333333aa;  
}
```

Specifies a safe, broadly supported hex code

Overrides the first value with a newer hex code format

Because the second declaration appears after the first, the cascade uses it to determine the cascaded value of `background-color` for these elements. The second declaration uses a relatively new 8-digit hex color code format (the seventh and eighth

digits specify an alpha value, indicating partial transparency). This syntax is supported in most browsers but not some older browsers, including Internet Explorer (IE). So, if a user happens to be using IE, their browser simply ignores that rule, and the first declaration remains the cascaded value. That user doesn't get the full-featured experience with a bit of transparency, but they still see a perfectly usable result. The page doesn't "break" or throw an error for them; the browser simply continues parsing the CSS and discards the unrecognized declaration.

This may seem odd from a debugging perspective because the language never throws errors. But it's an essential part of how the language works, and this is intentional for this very purpose: progressive enhancement.

1.5.2 *Progressively enhancing selectors*

This approach is not limited to new properties or value syntaxes. It can also be applied with new selector syntaxes. Browsers that understand the selector will use it, and those that don't will ignore the entire ruleset.

However, there is an important nuance to be aware of: when a ruleset has multiple selectors, the browser will ignore the entire ruleset if any of the selectors are unsupported or invalid. Consider what that means for the following CSS:

```
input.invalid,  
input:user-invalid {  
    border: 1px solid red;  
}
```

At the time of writing, the `:user-invalid` pseudo-class is a new addition to CSS (see appendix A for more details on this selector). A browser that is a few versions behind may understand `input.invalid`, but because it doesn't know what `:user-invalid` means, it will never apply these styles, even when the first selector matches. When you want to use a new selector like this, the best approach is to separate the two selectors into their own rulesets:

```
input.invalid {  
    border: 1px solid red;  
}  
input:user-invalid {  
    border: 1px solid red;  
}
```

This requires unfortunate duplication, but it is the best option. This is important to keep in mind when using new pseudo-classes, pseudo-elements, or attribute selectors (see appendix A).

In simple cases like the previous example, this only requires a few lines of duplicated code, but occasionally this might require repeating a large block of CSS. Some developers who prefer to avoid the duplication will opt not to use the new feature until browsers reach a comfortable level of support for the selector in question.

1.5.3 Feature queries using @supports()

Relying on the previously mentioned techniques is sufficient when the effect of using a new feature is small and only affects one or two CSS rules, but occasionally you will need to specify multiple different declarations for browsers that support a feature compared with those that don't. In this case, you can use a *feature query* to provide a larger set of styles depending on whether or not the browser supports a given feature.

A feature query 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 will ignore 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 a grid.

Grid is now broadly supported in all modern browsers, but this was not always the case. I can use it to illustrate the behavior of `@supports`. Let's add a small grid of links to our page and lay them out in a grid, as shown in figure 1.15.

Try some of our newest coffees:

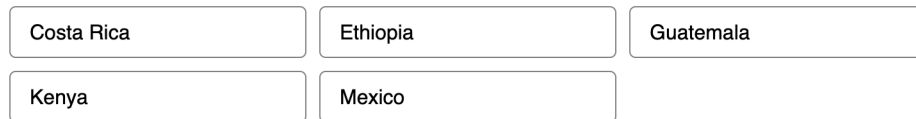


Figure 1.15 A grid of links you will define inside a `@supports` block

First, add the HTML shown in listing 1.24 to `<main>` between your page header and footer.

Listing 1.24 Adding a series of links to the page

```
<p>Try some of our newest coffees:</p>
<div class="coffees">
  <a href="/coffees/costa-rica">Costa Rica</a>
  <a href="/coffees/ethiopia">Ethiopia</a>
  <a href="/coffees/guatamala">Guatemala</a>
  <a href="/coffees/kenya">Kenya</a>
  <a href="/coffees/mexico">Mexico</a>
</div>
```

Then you can add some styles to lay these links out in a grid. First, you will provide a fallback behavior for old browsers, followed by a feature query that gives a full-featured grid layout. Add these styles to your stylesheet.

Listing 1.25 Using a feature query for progressive enhancement

```

.coffees {
  margin: 20px 0;
}

.coffees a {
  display: inline-block;
  min-width: 300px;
  padding: 10px 15px;
  margin-right: 10px;
  margin-bottom: 10px;
  color: black;
  background-color: transparent;
  border: 1px solid gray;
  border-radius: 5px;
}

@supports (display: grid) {
  .coffees {
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
    gap: 10px;
  }

  .coffees a {
    margin: unset;
    min-width: unset;
  }
}

```

Provides a fallback behavior for legacy browsers

Targets only browsers that understand grid layout

Defines the grid layout for modern browsers

Overrides the unneeded fallback styles so they don't apply in our grid

The fallback and other basic styles, such as colors, are outside of the feature query block, so they'll apply in all browsers. If you open the page in a browser that doesn't support grid, you'll see the fallback layout, which is similar to the grid layout. All styles relating to the grid-based layout are within the feature query block, so they'll only apply if the browser supports grid. We will take a closer look at grid layouts in chapter 5, so don't worry too much about these specific declarations if they're unfamiliar.

You can imagine how the styles might apply if the `@supports` block is ignored, or you can even momentarily comment it out in your stylesheet to test the fallback behavior in a modern browser and adjust it as necessary. The `@supports` rule can be used to query for all sorts of CSS features. Use `@supports (mix-blend-mode: overlay)` to query for blend mode support (see chapter 13) or `@supports (color: color-mix(in oklab, red, white))` to query for color-mix support (see chapter 11). 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.
- `@supports selector(<selector>)`—Apply rules only if the given selector is understood by the browser (for example, `@supports selector(:user-invalid)`).

Using progressive enhancement, you can define the user experience for a full spectrum of users across multiple browsers, including future browsers. A browser that understands the new code will use it, and one that doesn't will not. Web designer and educator Jen Simmons half-jokingly calls this "Quantum CSS." You can take a feature of CSS and "use it and not use it at the same time. It works and it doesn't work at the same time."

This feature of the language is known as *resilience*. CSS (and, similarly, HTML) are designed to be fault tolerant. As new features continue to roll out in CSS in the future, you can use this to your advantage.

ENABLING EXPERIMENTAL FEATURES

The W3C develops the CSS specifications in conjunction with browser development. This often means one or more browsers will begin to develop support for a feature before the specification is finalized. To discourage the use of unstable CSS in production websites, these experimental features only work for developers who intentionally turn them on in their browser settings. This allows for early experimentation and feedback while a specification is still being finalized. It's important to know how to access experimental features should you want to learn those in the future.

In Chrome and Opera, this is done by enabling a flag. 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.

Summary

- The browser follows the rules of the cascade to determine which styles are applied to which elements.
- A selector's specificity is determined by the number of IDs, classes, and tag names in the selector. Declarations with higher specificity selectors will override those with lower specificity.
- When no cascaded value is present for certain properties, elements inherit a value from their parent element. These properties include those for text, lists, and table borders.
- Providing a value of `unset` undoes other styles otherwise provided for that property, including user-agent styles. Using `revert` undoes your author styles but leaves user-agent styles intact.

- Shorthand properties provide a concise way to set values for multiple related properties at once.
- For shorthand properties such as `margin` and `padding`, specify the values in clockwise order, beginning with the top.
- Progressive enhancement allows us to use cutting-edge CSS without breaking the page in older browsers.

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 custom properties

When it comes to specifying *length* 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 a type of *absolute* unit; that is, 5 px always means the same thing. Some 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 apply it to. Naturally, this makes relative units more difficult to work with.

DEFINITION *Length* is the formal name for a CSS value that denotes a distance measurement. It's a number followed by a unit, such as 5px. Percentages are similar to lengths, but strictly speaking, they're not considered lengths.

When working with lengths, developers, even experienced CSS developers, often dislike working with relative units—the way the value of a relative unit can change makes it seem unpredictable and less clear-cut than an absolute unit, such as 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; wielded correctly, they'll make your code simpler, more versatile, and easier to work with.

2.1 *The power of relative units*

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 can, for example, 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 800 px wide by 500 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 rise of responsive design*

In the web environment, the user can set their browser window to any number of sizes, and the CSS has to apply to them. 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 on screen.

This adds a layer of abstraction to CSS. You can't style an element according to an ideal context; you 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.

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), Q (quarter-millimeter), in. (inch), pt (point—a typographic term for 1/72nd of an inch), and pc (pica—a 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 = 101.6 Q = 2.54 cm = 6 pc = 72 pt = 96 px. Therefore, 16 px is the same as 12 pt (16 / 96 × 72). Print designers are often more familiar with the use of points,

whereas 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 in the case of 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 on screen. However, this can vary on certain devices or with a user’s resolution settings.

With such a varied array of user devices and screen sizes, we have to constantly be aware of *responsive design*. If you 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 have multiple ways to solve a particular problem, you’ll need to favor the solution that works more generally under multiple and different circumstances.

DEFINITION *Responsive design* is a term coined by Ethan Marcotte, which 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 7, but in this chapter, I’ll lay some important groundwork before we get there.

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.

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.

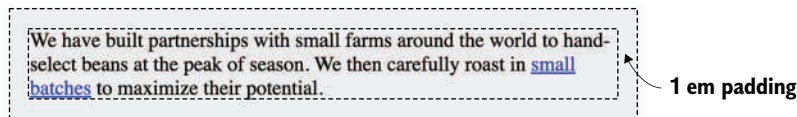


Figure 2.1 One em of padding is equal to the font size (dashed lines added to illustrate padding).

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 `1.5em` would produce a computed value of 24 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 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 are not the same.



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 `1em`, 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:

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

Now add the styles in listing 2.2 to your stylesheet. This defines a box using ems. It also defines small and large modifiers, each scaling the elements with a different font size.

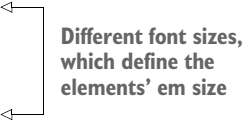
Listing 2.2 Applying ems to different elements

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



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

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



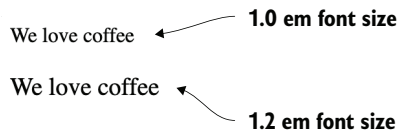
Different font sizes,
which define the
elements' em size

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



We love coffee ← 1.0 em font size

We love coffee ← 1.2 em font size

Figure 2.3 Two different font sizes using ems

Change your page to match listing 2.3. 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:

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



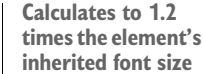
The slogan inherits its
font size from `<body>`.

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 2.3 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: $16 \times 1.2 = 19.2$, 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 multiplied by 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 A font size in ems differs from padding in ems.

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 this element's 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 on your test page.

Listing 2.4 Applying ems to font-size and padding

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

.slogan {
  font-size: 1.2em;      ← Evaluates to 19.2 px
  padding: 1.2em;        ← Evaluates to 23.04 px
  background-color: #ccc;
}
```

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 Font size of 0.8 em causing shrinking text when applied to nested lists

Shrinking text occurs when you nest lists several levels deep and apply an em-based font size to each level. Listing 2.5 provides an example of this by setting the font size of unordered lists to 0.8 em. The selector targets every on the page; so when these lists inherit their font size from other lists, the ems compound.

Listing 2.5 Applying ems to a list

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

ul {
  font-size: 0.8em;
}
```

If you were to apply these styles to a page with multiple nested lists, you would see the problem. When applied to a page with markup such as listing 2.6, each inherits its font size from the parent list, and the em value reduces it.

Listing 2.6 Nested lists

```
<ul>
  <li>Top level
    <ul>
      <li>Second level
        <ul>
          <li>Third level
```

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

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

```

        <ul>
          <li>Fourth level      | And so on
            <ul>
              <li>Fifth level</li>
            </ul>
          </li>
        </ul>
      </li>
    </ul>
  </li>
</ul>

```

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), 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 you want is to specify the font at the top level and 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 Using a font size of 1 em for nested lists keeps the text size consistent.

One way you can accomplish this is with the code in listing 2.7. This sets the font size of the first list to 0.8 em as before (listing 2.5). 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 2.7 Correcting the shrinking text

```

ul {
  font-size: 0.8em;
}

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

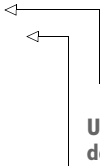
When the browser parses an HTML document, it creates a representation in memory of all the elements on the page. This representation is called the Document Object Model (DOM). It's a tree structure, where each element is represented by a node. The `<html>` element is the top-level (or root) node. Beneath 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 2.8 Specifying font size using rems

```
:root {  
  font-size: 1em;  
}  
  
ul {  
  font-size: 0.8rem;  
}
```



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 0.8 rem, which calculates to 12.8 px. Because this is relative to the root, the font size remains constant, even if you nest lists.

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-set. 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 or rems for most other measures, especially paddings, margins, and border radius.

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—though my preferences change over time and between projects. 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 either ems or rems for most other properties.

Accessibility: Using 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 (—)—or Cmd-plus/minus on a Mac—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. This is usually set in the browser's settings page, but changes at this level persist until the user 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.

Working with pixels is initially easier to understand, but if you take the time to become familiar with rems, you will have a broader range of tools at your disposal, and you can produce more accessible pages at the same time.

2.3 Stop thinking in pixels

One pattern, or rather, antipattern, that has been common in the past is to reset the font size at the page's root to 0.625 em or 62.5%. 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. An example of this is shown in the following CSS:

```
html {  
  font-size: 0.625em;  
}
```

I don't recommend doing this. Initially, it 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 text, so you'll have to override it throughout the page. You'll find yourself setting paragraphs to 1.6 rem and asides to 1.6 rem and nav links to 1.6 rem and so on. This introduces more places for error and more points of contact in your code when it needs to change, and it increases the size of your stylesheet.

The second problem is that when you do this, you're still thinking in pixels. You might type 1.6 rem into your code, but in your mind, you're still thinking "16 pixels." On the 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 on screen, change it. This takes some trial and error, but in reality, so does working with pixels.

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 involves arithmetic, so keep a calculator handy. 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 iterate 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.

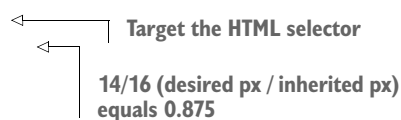
2.3.1 Setting a sane default font size

Let's say you want your default font size to be 14 px. Instead of setting a 10 px default and 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. Using ems here allows you to adjust the default font size while still respecting any user font settings.

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

Listing 2.9 Setting the true default font size

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



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.

The markup for this panel is shown in listing 2.10. Add this to your page.

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 2.7 An example panel you will create using relative units and an inherited font size

Listing 2.10 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 2.11 A 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 indicates a direct parent-child relationship between two elements in the selector. In this case, the selector targets any `h2` that's a child element of a `.panel` element. (See appendix A for a complete reference of selectors and combinators.)

In listing 2.10, 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 the way 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.

A media query uses an `@media` rule 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. This will make the panel render at different sizes based on the size of the user's screen (figure 2.8).

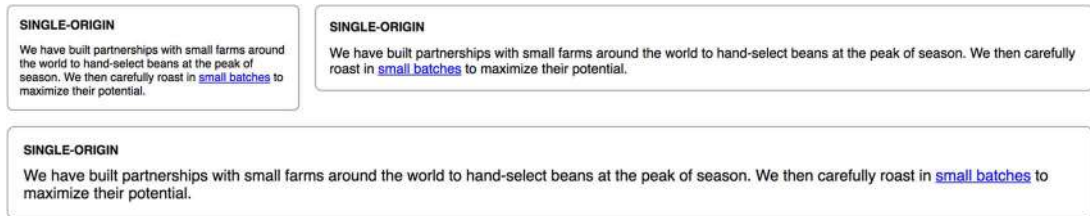


Figure 2.8 A responsive panel as seen on different screen sizes: 300 px (top left), 800 px (top right), and 1,440 px (bottom)

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

Listing 2.12 Responsive base font-size

<pre>:root { font-size: 0.85em; }</pre>	Applies to all screens but is overridden for larger screens
<pre>@media (min-width: 800px) { :root { font-size: 1em; } }</pre>	Applies only to screens 800 px and wider, overriding the original value
<pre>@media (min-width: 1200px) { :root { font-size: 1.15em; } }</pre>	Applies only to screens 1200 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 1200 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 (13.6 px); likewise, the padding and border radius will be smaller to match. And on larger screens more than 800

px and 1200 px wide, the component scales up to a 16 px and 18.4 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 size of the user's browser window. 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 you later decide the fonts on a site are too small or too large, you can change them globally by touching only 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.

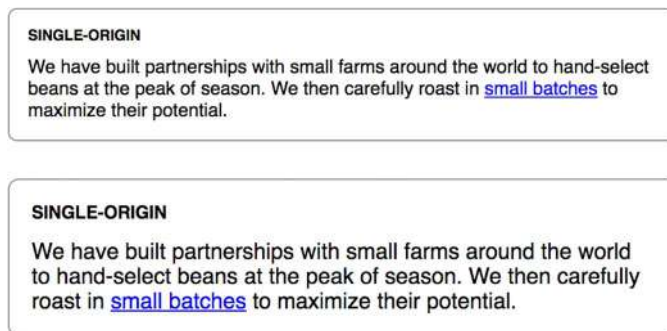


Figure 2.9 A panel defined in ems can be enlarged by increasing its font size.

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 in the following listing. Update your stylesheet to match.

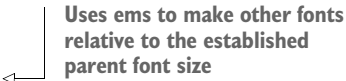
Listing 2.13 Creating a larger version of the panel

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

← Establishes a predictable font size for the component

```
border-radius: 0.5em;
}

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



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 declaration. All you have to do is override the parent element's 1 rem font size 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 following snippet to your stylesheet to define a larger version:

```
.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 would still take only this one declaration to resize it, as long as everything inside is defined using ems.

You don't need to follow this pattern exactly as I've shown. This dynamic nature of ems and rems is a powerful tool you can adjust to suit your needs. For instance, if you don't want your border radius to scale, define it using rems and use ems only for the properties you want to make scalable. Taking the time to familiarize yourself with these relative units will provide you with a lot more options than simply defining everything in pixels.

CSS is a living standard

CSS is defined by a large number of W3C specifications. It began as a single specification with a version number, but this approach changed after version 2.1.

After that point, the specification was broken up into individual modules, each independently versioned. The specification for backgrounds and borders is now separate from the one for the box model 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. Some of these specifications remain at version 3 (now called level 3), but others, such as the selectors specification, are at level 4 or higher, while others, such as a flexbox, are at level 1.

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. You may hear the term CSS3, but this does not strictly refer to version 3.0 of one specification but rather a series of specifications that were all published in a short timeframe in the early 2010s.

(continued)

The CSS Working Group has only recently resolved to support definitions of CSS4 and CSS5. These will not be specific versions of a specification but rather an assembly of features that are already stable, fundamental parts of the language (CSS4) and newer features whose adoption is steadily growing now (CSS5). I won't use these terms throughout this book as they are still not fully defined, but I will cover features from both of these categories and even a few that may eventually be considered part of CSS6.

Ems and rems are the most commonly used relative units, but they aren't the only relative unit types available. Now that you've seen the flexibility they provide, let's look at another important type of relative unit.

2.4 Viewport-relative units

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

DEFINITION The *viewport* is 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.

With some recent additions to the language, there is a grand total of 24 viewport-relative units in CSS. This probably sounds overwhelming, but these are all the permutations of a few relatively simple concepts mixed and matched in various ways. We'll start with the basics. The following are the four basic units that were first added to the language:

- *vh*—One percent of the viewport height
- *vw*—One percent of the viewport width
- *vmin*—One percent of the smaller dimension, height or width
- *vmax*—One percent of the larger dimension, height or width

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.

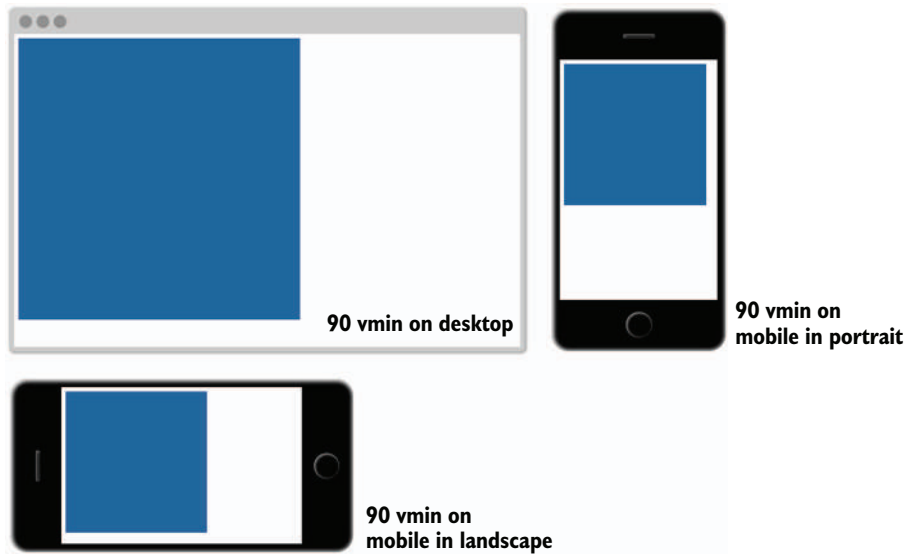


Figure 2.10 An element with a height and width of 90 vmin will always display as a square a little smaller than the viewport, regardless of its size or orientation.

Listing 2.14 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.

Listing 2.14 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—at least, that was the theory.

After developers started using these units to fill the viewport with hero images, we discovered a problem: on mobile devices, the viewport size can change dynamically. In an attempt to maximize available screen size, mobile browsers have a feature where some UX controls—the address bar at the top of the screen and the navigation buttons at the bottom—slide out of view as you scroll down the page. And they slide back into view if you scroll back up.

These dynamic changes cause a resize of the viewport, which in turn causes elements on the page using vh units to change size and content beneath them to jump around on the screen. You can imagine a scenario where the user has scrolled past

multiple 100 vh boxes but then scrolls up slightly. This can cause all the content in view to snap upward by hundreds of pixels, making for a very jarring reading experience. It also has performance implications because so much of the page layout has to be recalculated by the browser (often called *layout thrashing*).

In the end, most mobile browsers stopped this behavior by reinterpreting vh units based on the largest possible viewport size and ignoring the influence the address bar has on viewport size. But sometimes, as a developer, this is not the behavior you want, so several more units were added to the CSS specification.

2.4.1 Selecting from the newer viewport units

To address the problem of layout thrashing, CSS introduced the concept of large and small viewports. The *large viewport* is the biggest possible viewport when all the browser's UX elements are hidden. The *small viewport* is the smallest possible viewport when all the UX elements are shown (figure 2.11).

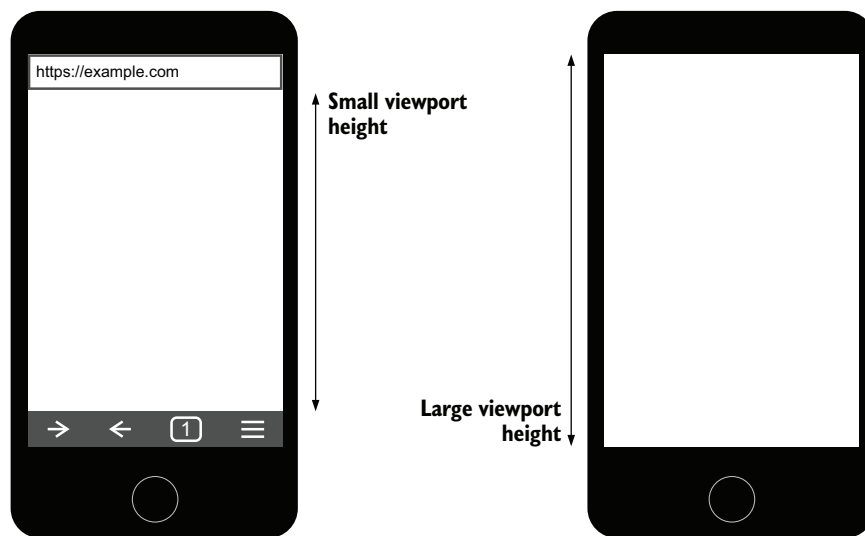


Figure 2.11 Comparison of the large and small viewports on a mobile device

Large viewport units are available for width, height, min, and max, just like the regular viewport units. Prepend the letter “l” to use large viewport units: `lvw`, `lvh`, `lvmin`, `lvmax`. Similarly, prepend the letter “s” to use small viewport units: `svw`, `svh`, `svmin`, `svmax`.

With these new units, you have the ability to choose which behavior is more important to you. Is it important for a full-height masthead to fill the entire screen, even if that means it might extend a bit past the bottom? If so, use `height: 100lvh`. Or is it more important that the entire masthead remains visible, even if there may be a little space beneath it at the bottom of the screen? Then use `height: 100svh`.

At this time, I have been unable to find examples of any browser differentiating between the widths of the small and large viewports, so for all practical purposes, vw, svw, and lw all behave the same. However, it's important to keep in mind that they are conceptually different, and it is entirely possible that browsers will make a distinction between them in the future. For example, the Edge browser on desktops has a dynamic sidebar that opens and closes; this doesn't affect svw and lw at this time, but that behavior could change down the road.

Here are a few other caveats to keep in mind:

- Viewport units do not take scrollbars into account. This unfortunately means an element of width 100 svw will introduce horizontal scrolling when a vertical scrollbar is present.
- The CSS specification does not dictate whether an onscreen keyboard should shrink the small viewport size. It is currently taken into consideration in some Android browsers but not on iOS; these could potentially change in the future.
- The original viewport units generally behave like large viewport units in most browsers, but this behavior is not guaranteed.

This leaves us with one use case that isn't addressed: there may be instances where we want the original behavior of viewport units, where they dynamically shift if/when the browser's UX elements appear or hide. For this behavior, there is a third type of viewport units: the *dynamic viewport*. For these, prepend the letter “d” to the viewport units: dvw, dvh, dvmin, dvmax. These behave like small viewport units when the viewport is small and like large viewport units when the viewport is large.

You should use the dynamic viewport units sparingly. As a user on a mobile device scrolls up and down, it can cause layout thrashing if element heights change dynamically.

Table 2.1 lists all of the available viewport units. For completeness, this includes an additional set of unit types: inline and block. These are called *logical properties* and behave like width and height, respectively, but are transposed for vertically written languages such as Japanese. I'll cover logical properties in more depth in chapter 3.

Table 2.1 All viewport unit types

	Unspecified viewport (original units)	Large viewport	Small viewport	Dynamic viewport
Width/height	vw vh	lw lh	svw svh	dvw dvh
Min/max	vmin vmax	lvmin lvmax	svmin svmax	dvmin dvmax
Inline/block	vi vb	lvi lvb	svi svb	dvi dvb

For many uses beyond mastheads or other similar full-screen or half-screen elements, there may not be a strong reason to favor units of one viewport size over another. With so many viewport units available, this can lead to a bit of decision paralysis. My go-to when this happens is to use small viewport units.

2.4.2 Using viewport units 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: 2svw` to an element. On a desktop monitor at 1200 px, this evaluates to 24 px (2% of 1200). 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 are 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 around 7.5 px on an iPhone SE. What would be nice is this scaling effect but with the extremes a little less severe. You can achieve this with either CSS's `calc()` or `clamp()` functions.

GETTING RESPONSIVE WITH THE `CALC()` FUNCTION

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 step to combine ems with `svw` units. Remove the previous base font size (and the related media queries) from your stylesheet. Add the following in its place:

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

Now open the page and slowly resize your browser to be wider or narrower. You'll see the font scale smoothly as you do. The `0.5em` here operates as a sort of minimum font size, and the `1svw` adds a responsive scalar. This gives you a base font size that scales from 11.75 px on an iPhone SE up to 20 px in a 1200 px browser window.

WARNING When using viewport units for font size, always make sure part of the calculation includes ems or rems for accessibility. This ensures the user's font settings are taken into account in the final rendered size.

You can adjust these values to your liking, but it can be a little difficult to find something that works well without being too small in a small viewport or too large in a big one.

IMPROVING THINGS WITH THE CLAMP() FUNCTION

The `calc()` function is useful for its font size, but a newer function, `clamp()`, provides a little better control. Clamp takes three arguments: a minimum value, a preferred value as an expression, and a maximum value. Update your page once more using the following instead of the previous approach:

```
:root {  
  font-size: clamp(0.9rem, 0.6rem + 0.5svw, 1.5rem);  
}
```

This specifies a font size of `0.6rem + 0.5svw`, but the clamp function ensures the final value is never smaller than `0.9rem` and never larger than `1.5rem`. This way, very large or very small viewports do not end up with text sizes outside of a reasonable range.

Now that your font size is responsive, any other sizes on your page defined using ems or rems will scale responsively as well. You’ve accomplished a large piece of your responsive strategy without a single media query. Instead of three or four hardcoded breakpoints, everything on your page will scale fluidly according to the viewport. There is more to responsive design than this—and we’ll take a much deeper look in later chapters—but this will get you off to a nice start.

TIP Two other related functions that may be useful from time to time are `min()` and `max()`. `Min` resolves to the smallest of the given values (e.g., `width: min(200px, 20svw);`). `Max` resolves to the largest of the given values (e.g., `min-height: max(200px, 20svw);`).

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 be used only 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 page.

Listing 2.15 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 on the page (figure 2.12).

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 Unitless line height is recalculated for each descendant element, generally producing well-spaced lines of text.

Add listing 2.16 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 2.16 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 were to specify the line height using a unit, you could encounter unexpected results, like that shown in figure 2.13. The lines of text overlap one another. Listing 2.17 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 2.13 Using units in a line-height can produce undesirable spacing when inherited by child elements.

Listing 2.17 Line height with units resulting in unexpected output

```

body {
  line-height: 1.2em;
}

```

Descendant elements inherit the calculated value (19.2 px).

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



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, like the overlapping text, if the child element has a different font size.

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)

Custom properties enable a higher level of dynamic, context-based styling. These function in many ways like variables: you can declare a variable and assign it a value; then you can reference this value throughout your stylesheet. You can use this technique to reduce repetition in your stylesheet, as well as some other beneficial applications, as you'll see shortly.

NOTE If you use a CSS preprocessor that supports its own variables, such as Sass or Less, you may be tempted to disregard CSS variables. Don't. 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. The next snippet is an example of a variable declaration. Start a fresh page and stylesheet, and add the following CSS:

```
: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 other 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 2.14.

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 2.14 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 2.18 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 2.19 Using custom properties for colors

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

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

Defines a blue brand-color variable

The `var()` function accepts an optional 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 2.20 Providing fallback values

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

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

Specifies a fallback value of sans-serif

```
color: var(--secondary-color, blue);
```

← The **secondary-color** variable is not defined, so the fallback value **blue** is used.

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 2.15.

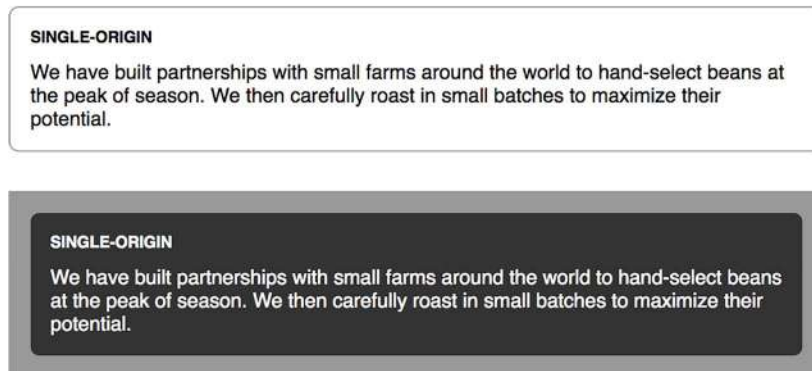


Figure 2.15 Custom properties produce different-colored panels based on local variable values.

This panel is similar to the one you created earlier but with added support for a dark version. The HTML for this is shown in listing 2.21. It has two instances of the panel: one inside the body and one inside a dark section. Update your HTML to match this.

Listing 2.21 Two panels in different contexts on the page

```

<body>
  <div class="panel">
    <h2>Single-origins</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 carefully roast in
      small batches to maximize their potential.
    </div>
  </div>

  <aside class="dark">
    <div class="panel">
      <h2>Single-origins</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 carefully 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

Next, redefine the panel to use variables for text and background color. Add the following 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 2.22 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 2.23 Styling the dark container

Puts a margin between the dark container and the preceding panel

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

Applies a dark-gray background to the dark container

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

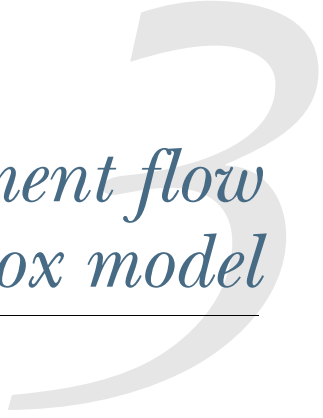
Reload the page, and the second panel has a dark background and white text. This is because when the panel uses these custom properties, they resolve to the values defined on the dark container, rather than on the root. Notice you didn't have to restyle the panel or apply any additional classes.

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.

Custom properties are an extremely versatile tool with countless applications. We will continue to use them in various ways throughout the rest of the book.

Summary

- Relative units can be used to specify sizes that adapt to the font size or viewport size.
- Ems define a length in terms of an element's font size, except when specifying font size itself, in which case ems are in terms of the element's inherited font size.
- Rems define a length in terms of the font size specified on the root `<html>` element.
- By scaling the root font size in a responsive design, elements on the page defined using ems and rems will scale at the same time.
- Viewport-relative units define a length in terms of the viewport's width or height.
- A line height defined with a unitless number will inherit more predictably to child elements.
- Custom properties work similarly to variables but can be manipulated dynamically via cascade and inheritance.



Document flow and the box model

This chapter covers

- General tips for building a page layout
- Practical advice for element sizing
- An introduction to logical properties
- 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 grids, absolutely positioned elements, and other elements of various sizes. 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 layout techniques. Before we get to those, it's important to have a solid grasp on the fundamentals of how the browser sizes and places 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 location and size of elements on the page.

In this chapter, you'll build a basic one-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.

3.1 Normal document flow

In this chapter, you'll build a simple page with a header at the top and content beneath it. The width of the content will be restricted to avoid very long lines of text. This sort of layout is often called a one-column layout because there are no blocks of text side by side anywhere.

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 the elements.

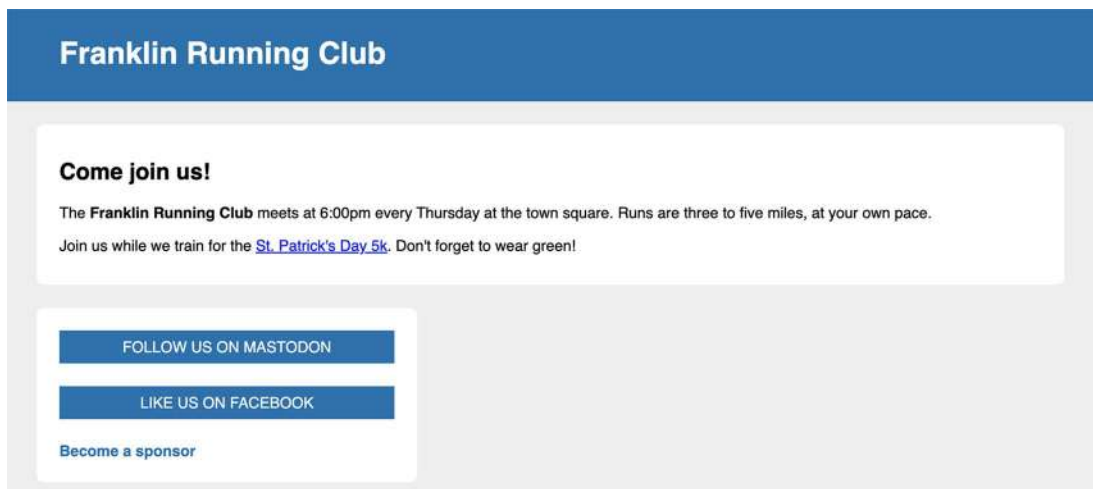


Figure 3.1 Example one-column page layout you'll build by the end of the chapter

Start a new page and an empty stylesheet. Add the markup shown in listing 3.1 to your page. Your page will have a header and a primary container that comprises the bulk of the page. Inside the container are a `<main>` with content and an `<aside>` with some social media links.

Listing 3.1 HTML for page with a one-column layout

```
<!doctype html>
<html lang="en-US">
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <header class="page-header">
    <h1>Franklin Running Club</h1>
  </header>
```

```

<div class="container">
  <main class="main">
    <h2>Come join us!</h2>
    <p>
      The <b>Franklin Running Club</b> meets at 6:00pm every Thursday at the
      town square. Runs are three to five miles, at your own pace.
    </p>
    <p>
      Join us while we train for the
      <a href="/st-patricks">St. Patrick's Day 5k</a>. Don't forget to wear
      green!
    </p>
  </main>
  <aside class="social-links">
    <a href="/mastodon" class="button-link">follow us on Mastodon</a>
    <a href="/facebook" class="button-link">like us on Facebook</a>
  </aside>
</div>
</body>
</html>

```

Let's begin with some of the obvious styles. You'll set the font for the page and 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.

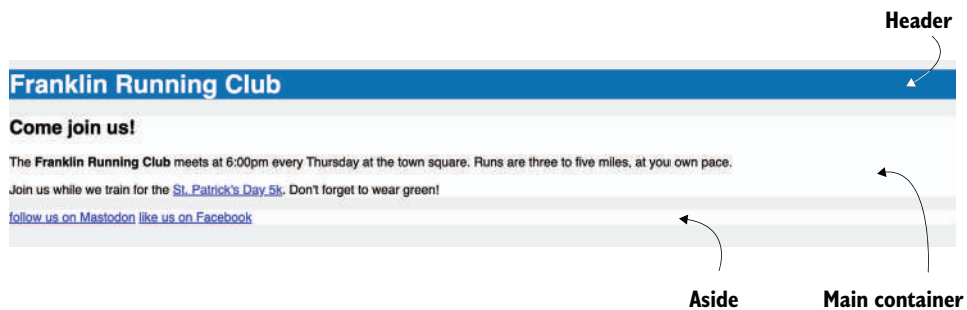


Figure 3.2 Three main containers with background colors

The styles for this are shown in the following listing. Add this code to your stylesheet.

Listing 3.2 Applying font and colors

```

:root {
  --brand-color: #0072b0;
}

```

← Sets the heading color to a custom property for reuse later

```

body {
  margin: unset;
  background-color: #eee;
}

```

← Removes a margin added by the user-agent styles so content can reach the edges of the browser window

```
font-family: Helvetica, Arial, sans-serif;
}

.page-header {
  color: #fff;
  background-color: var(--brand-color);
}

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

.social-links {
  background-color: #fff;
  border-radius: 0.5em;
}
```

Before you begin adjusting the size or layout of these elements, it's worth taking note of how the page's default layout behaves. There are two basic types of element: *inline* and *block*, each with a different behavior.

Inline elements flow along with the text of the page, from left to right. They will line wrap if they reach the edge of their container. Inline elements in our page include the `<b>` around `Franklin Running club` and the `<a>` around the links.

Block elements (usually called *block-level elements*) appear on their own individual lines. They automatically fill the width of their container. They have a line break above and below them, even if styles are added to reduce their width. Elements such as `<p>`, `<div>`, and `<header>` are set to `display: block` by the user-agent stylesheet, giving them this behavior. This behavior is called *normal document flow*.

DEFINITION *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.

The important thing to note is that height and width are fundamentally different. Normal document flow is designed to work with a constrained width and an unlimited height. Contents fill the width of their container and then line wrap as necessary.

This means that the width of a parent element determines the width of its children, but for height, the opposite is true: the heights of child elements determine the height of the parent. As a result, we use different approaches for manipulating each. We'll look at a number of these approaches throughout the rest of the chapter.

3.1.1 Centering content horizontally

When building the layout of a page, you should usually start with the higher-level DOM elements first. Get them sized and placed in relation to one another before focusing on their child elements. This gets a “rough cut” of the page in place and makes it easier to then focus on the smaller details.

TIP To begin laying out a page, it is best to do so from the outside in. Get the larger container elements where you want them before moving on to the smaller ones inside them.

On your page, you'll want to constrain the width of the page's main column. Because block-level elements fill the width of their container by default, you generally don't need to do anything like `width: 100%` or `width: 100svw` for full-width elements, but you will often want to reduce their width from this default.

Your page, after constraining the width of the content, is shown in figure 3.3. Notice the light gray margins on both sides and how both the header and the main container are equal widths within, aligning their content on the left side.



Figure 3.3 Page with constrained width

The layout in figure 3.3, as described in the previous two paragraphs, is common for centering content on a page. You can achieve this by placing your content inside two nested containers and then setting margins on the inner container to center it within the outer one (figure 3.4). Web developer Brad Westfall calls this the *double-container pattern*.

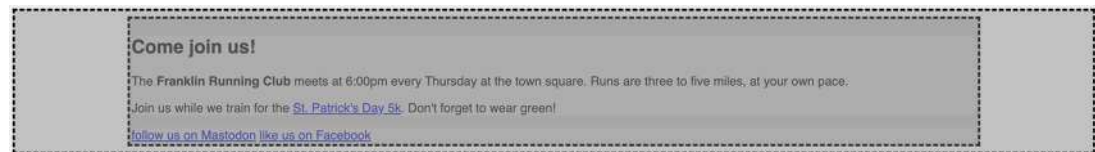


Figure 3.4 The double-container pattern horizontally centers a block of content.

In your page, `<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 main contents of the page in a `<div class="container">`, which serves as the inner container. You will restrict the width of this container and apply margins to center it.

You can also use the same pattern a second time in the header to keep it aligned with the content beneath. The `<header>` will serve as the outer container, and the `<h1>` will be the inner container. Because this needs to be the same width, you can use a custom property to control the size of both in one place. Update your stylesheet to include the code shown in the following listing.

Listing 3.3 Styles for the double container

```

:root {
  --brand-color: #0072b0;
  --column-width: 1080px;
}

.page-header h1 {
  max-width: var(--column-width);
  margin: 0 auto;
}

.container {
  max-width: var(--column-width);
  margin: 0 auto;
}

```

Sets a maximum width of 1080 px

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

By setting a left and right margin of `auto`, the margins will automatically expand as much as necessary to fill the remaining width available in the outer container. This is often the simplest way to center a block of content horizontally. It's important to note, however, that under normal document flow, this technique does not work for top and bottom margins.

Using `max-width` instead of `width` allows the element to shrink below 1080 px if the screen's viewport is narrower 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. We'll take a little closer look at `max-width` and related properties later in the chapter.

3.1.2 Using logical properties

In the past, the default behavior of document flow has made it difficult when working with websites that need to be translated into certain languages. Normal document flow goes from left to right, top to bottom. This is because most languages, including English, are written this way. But for the “worldwide” web to accommodate the whole world, it needs to also work for languages that are read in other ways. This includes right-to-left languages, such as Arabic and Hebrew, and vertically written languages, such as Japanese and traditional Chinese. For this reason, the W3C has done a lot of work to introduce the concept of *logical properties* to CSS.

DEFINITION *Logical properties* provide a way to work with elements in terms of their block and inline directions—which can change for different writing modes—rather than explicitly referring to top, right, bottom, and left or to width and height.

When using logical properties, we swap out the concepts of horizontal and vertical for *inline base direction* (the inline flow of text) and *block flow direction* (the direction in which boxes such as paragraphs stack). Instead of setting `width`, we can set `inline-size`; the two do exactly the same thing in a horizontal writing mode, but `inline-size` adapts to specify the height when used with vertical writing modes. Likewise, instead of `height`, we can use `block-size`, which adapts to specify a width in a vertical writing mode.

Logical properties also replace `top`, `right`, `bottom`, and `left` with *start* and *end*. Thus, `padding-left` and `padding-right` become `padding-inline-start` and `padding-inline-end`, respectively. Or `border-top` and `border-bottom` become `border-block-start` and `border-block-end`. These reorient their meanings according to the writing mode.

Figure 3.5 shows the block and inline directions for various writing modes. Arrows point from the “start” to the “end” in each dimension. I’ve made one border for each example a little thicker using `border-inline-start` to highlight how it behaves.

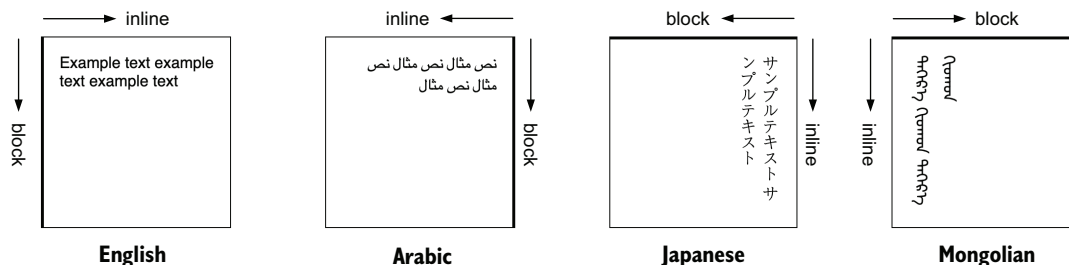


Figure 3.5 Inline and block directions in various writing modes

Adapting to use logical properties is primarily a matter of becoming familiar with these new names. You don’t need to change how you’re laying out the page—just some of the terminology that you’ve become accustomed to. You shouldn’t have to concern yourself with multiple writing modes or languages until it actually comes to enacting a translation of your page content; if your layout works in a familiar left-to-right language using logical properties, it will automatically change to work if the writing mode changes.

Do I need logical properties if I’m working in only one language?

Depending on where you live and work, you may infrequently, if ever, need to develop sites that support other writing modes. Even then, I still think it’s important to understand the principles of logical properties for three reasons.

First, they are a useful primer for concepts that are integral to flexbox and grid, which we will be diving into in the next two chapters. Second, there are a few extra logical properties that don’t have a classic counterpart, and these can be more convenient to use at times. And third, they are becoming more widely used in the industry, and it’s important to understand the code you encounter.

In all practicality, if you are working on a site that does not need alternate writing modes, it is entirely up to you whether you use logical properties, classic properties, or some combination of the two. Even when working in these languages, sometimes you will need to use some of the classic properties if the style you’re adding needs to work the same regardless of the writing direction.

If you've been inspecting elements with your browser's DevTools, you might have noticed its user-agent stylesheet is already using logical properties for default margins, the `inline-start` padding on lists, and other similar properties.

Both classic properties and their equivalent logical properties can override each other in the cascade. This way, if you set `padding-left` on a list, it will override the user-agent styles applying `padding-inline-start`. But you can also use `margin-block-start` with a higher specificity selector to override a `margin-top` applied with lower specificity. This means you can use both approaches interchangeably in your stylesheet and still have a predictable outcome.

Nearly every CSS property that deals with vertical or horizontal values has a logical counterpart. This way, you can define layouts that adapt based on the needs of the language. The following are examples of classic properties and values followed by their corresponding logical properties and values:

- `width` / `inline-size`
- `height` / `block-size`
- `margin-top` / `margin-block-start`
- `margin-bottom` / `margin-block-end`
- `margin-left` / `margin-inline-start`
- `margin-right` / `margin-inline-end`
- `text-align: left` / `text-align: start`
- `text-align: right` / `text-align: end`
- `border-top-left-radius` / `border-start-start-radius`
- `border-top-right-radius` / `border-start-end-radius`
- `border-bottom-left-radius` / `border-end-start-radius`
- `border-bottom-right-radius` / `border-end-end-radius`

This is only a partial list, but it serves to illustrate the pattern. By using block/inline and start/end, all the patterns you're already familiar with when using classic properties can be applied in the same way with logical properties. For a comprehensive list of logical properties, see <https://mng.bz/ppGK>.

3.1.3 Adopting useful shorthand logical properties

Some logical properties happen to provide shorthand approaches to common patterns. For example, `margin-inline` allows you to set the start (left) and end (right) margin at once without setting the other two margins. You can do something like `margin-inline: 2rem` to set both start and end (left and right) margins to 2 rem or `margin-inline: 2rem 4em` to set the start (left) margin to 2 rem and the end (right) margin to 4 em. The same approach is true for `margin-block` to set the block start and end (top and bottom) margins, as well as the other similar properties `padding-inline`, `padding-block`, `border-inline`, and `border-block`. There are no classic properties that behave quite like this.

You can use this on your page for a slightly cleaner approach to the double-container pattern you’ve built. Update your stylesheet to this, as shown in the next listing. This also includes changes replacing `max-width` with the logical property equivalent, `max-inline-size`.

Listing 3.4 Using logical properties for the double-container pattern

```

.page-header h1 {
  max-inline-size: var(--column-width);
  margin-inline: auto;
}

.container {
  max-inline-size: var(--column-width);
  margin-inline: auto;
}

```

Logical property equivalent for max-width

Sets left/right margins to auto without specifying top/bottom margins

These changes don’t visually change anything on your page; they are just an authoring convenience. I have found these shorthand properties useful because I frequently want to specify only top and bottom or only left and right values for margins or other properties.

3.2 The box model

The next thing to address on the page you’re building is some padding in the main container and the `social-links` box. Currently, the text in these areas is right up against the edges of the white background, so adding a little space there will make it look less crowded and more readable. Update your stylesheet to include the changes in the following listing.

Listing 3.5 Adding padding to the containers

```

.main {
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

.social-links {
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

```

Adds padding to the containers

Now the content inside two white containers has been made slightly narrower, leaving adequate space to breathe. However, by doing this, the left side of the text is no longer horizontally aligned with the text above in the page header (figure 3.6).

It might seem like we could fix this by adding a similar padding to the `<h1>` in the page header. Add this as shown in listing 3.6, but you’ll notice it doesn’t seem to make a difference.



Figure 3.6 After adding padding, the text no longer all aligns on the left side.

Listing 3.6 Adding padding to the page header

```
.page-header h1 {
  max-inline-size: var(--column-width);
  margin-inline: auto;
  padding-inline: 1.5rem;
}
```

Adds the same
padding to the
page header

If you happen to be on a smaller screen (less than about 1100 px wide), this may look like it worked, but on wider screens, it will have made no visual difference at all. Even with a padding added, the content didn't get narrower as it did with the main content box.

This is because of the default behavior of the *box model*. According to the box model, each element on the page is made up of four overlapping rectangles. The *content area* is the innermost rectangle where the contents of the element reside. The *padding area* contains the content area plus any padding. Likewise, the *border area* is the padding area plus any border, and the *margin area* is the outermost rectangle, containing the border area plus any margins.

DEFINITION The *box model* refers to the parts of an element (content, padding, border, and margin) and the size they contribute to their element. These produce rectangular boxes that are laid out by the browser on the page.

Specifying the height or width of an element sets the size of its content area; any padding, border, and margins are added outside that (figure 3.7).

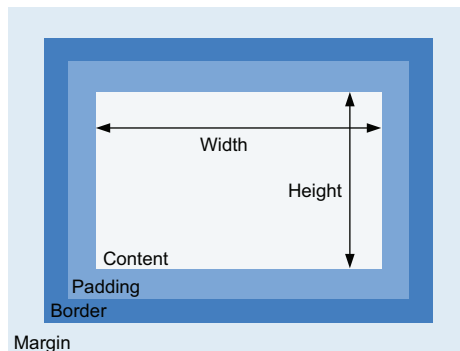


Figure 3.7 The default box model

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.

On your page, this means the `<h1>`, which had a width of 1080 px, has its padding added to its effective width. The padding is outside that 1080 px, and the main content stayed 1080 px wide.

NOTE Top and bottom margins and paddings behave a little unusually on inline elements. They will still increase the height of the element, but they will not increase the height that the inline element contributes to its container; that height is derived from the inline element's `line-height`. Using `display: inline-block` will change this behavior if necessary.

Outline: The other type of border

Similar to a border, you can also add an `outline` to an element. This behaves much like a border but does not add to the element's size and is not part of the box model. It is placed outside the border, overlapping the margin. It will not change the size or position of the element, nor will it affect the page layout in any way.

Similar to `border`, `outline` is a shorthand property for `outline-color`, `outline-style`, and `outline-width`. For example, `outline: orange solid 2px` will place a 2 px wide orange outline around an element. Unlike a border, you cannot specify a different outline for each side of the element; all four sides will always have the same outline style. Historically, an outline always had squared-off corners, but some browsers have recently changed their outline behavior to match the curve of any `border-radius` on the element.

You can manipulate the placement of an outline with the `outline-offset` property. A positive value (for example, `outline-offset: 3px`) will expand the position of the outline outward in all directions, adding space between the element's border area and the outline. A negative value will bring the outline inward, causing it to overlap with the element's border area.

Understanding the box model is an essential part of working with CSS. The way padding and border can increase an element's size can catch you off guard if you aren't ready for it. Knowing why this happens is an important first step in adapting your styles to accommodate it.

3.2.1 Avoiding magic numbers

Sometimes when you encounter problems like this, the temptation can be to fiddle with the values until it works. This is especially the case when you happen to use percent to define sizes.

Imagine if, instead of 1080 px width, your layout used 70%. A naive fix might be to reduce that percentage for the `<h1>`. Perhaps a width of 66% seems to work—but this is unreliable. The 66% is known as a *magic number*. Instead of using a desired

value, you found it by making haphazard changes to the styles until you got the result you 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 can't foresee how it will behave under different circumstances. Maybe it works with a viewport 1400 px wide, but on larger or smaller screens, the text no longer aligns. 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 your case, the `<h1>` is 3 em too wide (due to the left and right padding), so you can use the `calc()` function to reduce the width by exactly that much. A width of `calc(var(--column-width) - 3em)` gives you exactly what you need. But there's still a better way.

3.2.2 Adjusting the box model

The default box model tends to cause problems with the sizing and alignment of elements on the page. 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 sets the size of only the content box. You can assign a value of `border-box` to the box sizing instead. That way, the width, height, inline-size, and block-size properties set the combined size of the content, padding, and border, which is exactly what you want in this example.

Figure 3.8 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. These elements have the same height and width; notice that an element with border box sizing will be smaller than with content box sizing when any padding or border is present.

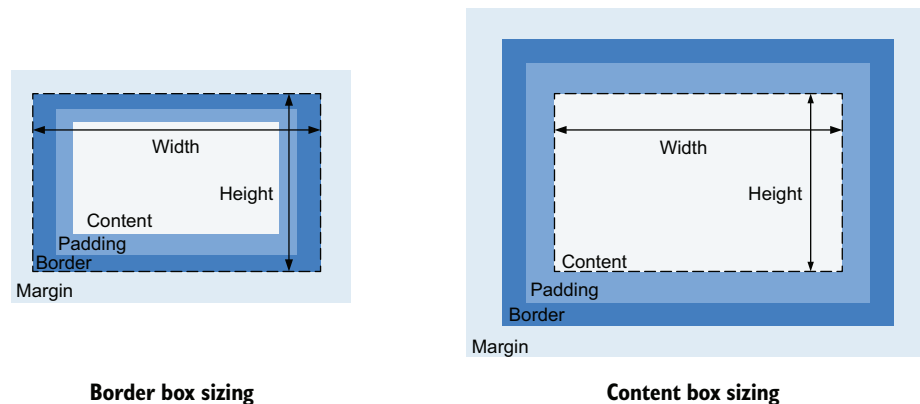


Figure 3.8 Border box sizing changes the box model so width and height are more predictable.

If you update the `<h1>` to use border box sizing, its content will align with the content in the main container, as shown in figure 3.9.

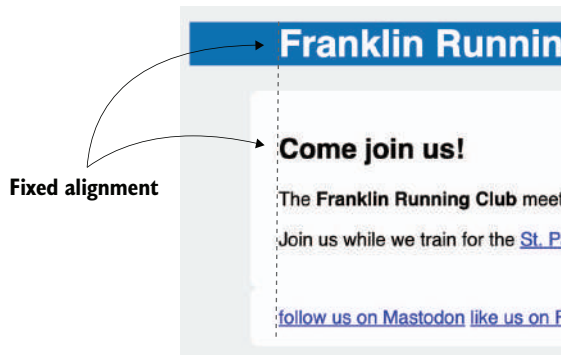


Figure 3.9 Left edge of content aligns with border box sizing.

To adjust the box model for your heading, update your stylesheet to match the following listing.

Listing 3.7 Heading with a corrected box model

```
.page-header h1 {
  box-sizing: border-box;
  max-inline-size: var(--column-width);
  margin-inline: auto;
  padding-inline: 1.5rem;
}
```

Changes the box model to border box sizing

Using `box-sizing: border-box`, the padding is now counted within the 1080 px width. The heading text now aligns with the main content beneath it.

3.2.3 Using universal border box sizing

You have made box sizing more intuitive for this one element, 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 3.8 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.

Now every element on your site will have a more predictable box model. I recommend that you add listing 3.8 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.

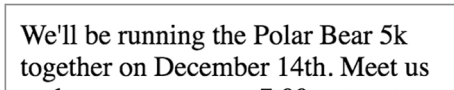
NOTE Adding this snippet near the beginning of your stylesheet has become common practice. From this point on, every example in this book will assume that this `border-box` fix is at the beginning of your stylesheet.

3.3 Element height

Working with the height (block size) of elements can be tricky. Normal document flow is designed to work with a constrained width and an unlimited height. The height of a container is organically determined by its content, not the container itself. Typically, it's best to avoid setting explicit height on elements—particularly elements with a lot of content—though there may often be a temptation to do so.

3.3.1 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 the parent element. Figure 3.10 shows this behavior. Document flow doesn't account for overflow, and any content after 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 3.10 Content overflowing its container

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

- `visible`—The default value; all content is visible, even when it overflows the edges of the content area.
- `hidden`—Content that overflows the container's padding area is clipped and won't be visible. The user will not be able to scroll to this content normally, though the element can still be scrolled programmatically via JavaScript.
- `clip`—Similar to `hidden` but programmatic scrolling is also disabled.

- `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.

Figure 3.11 illustrates four containers with various overflow settings.

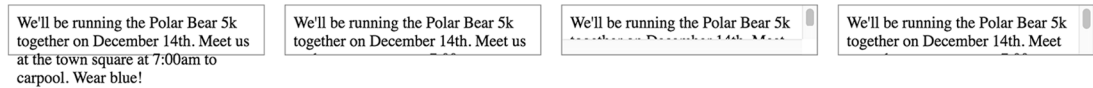


Figure 3.11 Overflow from left to right: visible, hidden, scroll, and auto

Typically, I prefer to use `auto` rather than `scroll` because, in most cases, I don't want the scrollbars to appear except when necessary.

Horizontal overflow

It's possible for content to overflow horizontally, not just vertically. This can occur when a long URL appears in a narrow container or when viewing a table with a lot of columns on a small mobile device. 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 when one of these values is `hidden`; in this case, use `clip` instead.

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.

3.3.2 Using alternatives to percentage-based heights

Specifying height using a percentage can be 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 will ignore the declaration. For percentage-based heights to work, the parent must have an explicitly defined height.

One place where percentage-based heights can be useful is when used for absolutely positioned elements, when you want to size them in relation to their container. We will take a look at positioning in a later chapter.

There are two use cases in particular where developers often try to use percentage heights that lead them into trouble. The first is when trying to make a container fill the screen. This can produce a scenario where you need to put `height: 100%` on an element, its container, and so on up the DOM tree until you've applied it to both `<body>` and `<html>`. A better approach is to use the viewport-relative units, which we reviewed in chapter 2, since a height of 100 `svh/lvh` is generally what you're looking for in this case.

The second reason is to create columns of equal height. Page designs often call for two or more side-by-side elements to have the same height. It was historically a tricky thing to size one element in CSS in relation to a sibling element. There were some very complicated tricks to make it work, but this can now be done much more easily with more modern techniques. We'll see how in the following chapters on flexbox and grid.

3.3.3 Using `min-height` and `max-height`

Two properties that can be immensely helpful are `min-height` and `max-height` (and their logical counterparts `min-block-size` and `max-block-size`). 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 3.12 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.

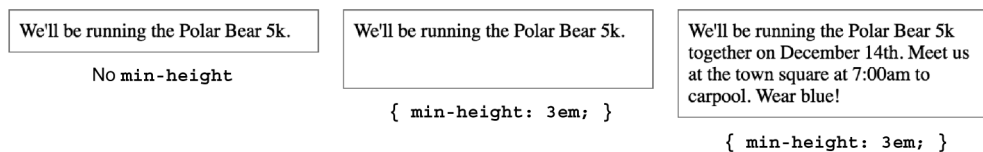


Figure 3.12 Three elements: one with no specified height and two elements with a 3 em `min-height`

Most of the time when I feel like I want to set the height on a container, I use `min-height` instead. It often saves me the headache of sorting out overflow problems later.

The corresponding `max-height` property 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` (and `min-inline-size` and `max-inline-size`) constrain an element's width.

3.4 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 3.13. 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 bottom side, a negative margin doesn't shift the element; instead, it pulls up any elements beneath it. This is not unlike giving the elements directly beneath it a negative top margin.

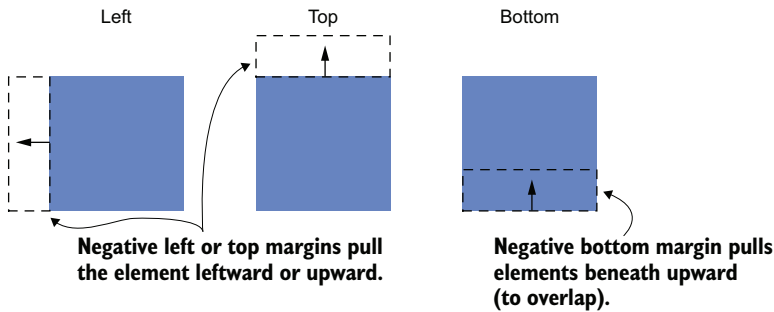


Figure 3.13 Behavior of negative margins

Although negative margins can be used to cause an overlap between multiple elements, it can be tricky to keep track of anything complicated. Often, it's a better idea to accomplish this using the `position` property instead; we'll look at positioning in detail in chapter 6.

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

Negative right margins behave differently depending on whether the element is inline or block-level. On inline elements, they behave similarly to negative bottom margins, pulling any successive content to the left so it overlaps the element. In practice, I have never had to use this.

On a block-level element, a negative right margin pulls the edge of the element right, making the element wider. This possibly brings the edge of the element outside its container.

If you join this with a negative left margin, both sides of the element will be extended outside the container, making it wider than its container. This is a sort of alternate application of the double-container pattern, illustrated in figure 3.14.

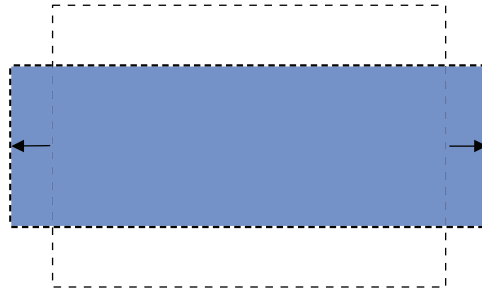


Figure 3.14 Negative left and right margins can extend a block-level element outside its container.

The code for this approach is shown in the following listing. This can be used to “bleed” an image wider than the column of text it appears with.

Listing 3.9 Using negative margins to bleed an image outside its container

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

.expanded-child {
  margin-inline: -2em;
}
```

Causes the element to widen
beyond its container in both
directions

Be aware that this can cause content to extend outside the browser viewport. Take care that things still work as expected on small screens when using this technique.

3.5 Collapsed margins

Let’s continue building 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 3.15). Why is that gap there?

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 3.15 is the result of collapsed margins. Let’s look at how this works.

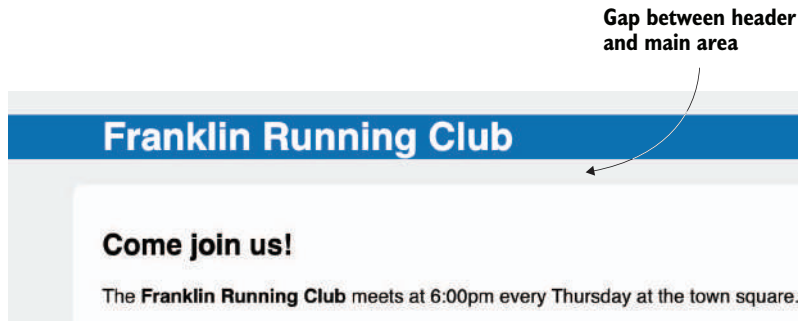


Figure 3.15 Gap caused by the margins collapsing

3.5.1 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 inside the `<main>` section on your 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 3.16. Note how the margins of each element occupy the same space on the page.



Figure 3.16 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 $\times$ 0.83 em), and the paragraph has a top margin of 16 px (16 px font size $\times$ 1 em margin). The larger of these, 19.92 px, is the amount of space rendered between the two elements.

3.5.2 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 following 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 3.10 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, three different margins are 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 occurs with only top and bottom margins. Left and right margins don't collapse. (In a vertical writing mode, this is reversed, and left/right margins will collapse instead.)

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 them, 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 appears larger only if the following element requires more space.

3.5.3 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.

This is what's causing the gap below the header on your page. The page title is an `<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 blue background of the `<header>` to be taller, so there's some space around the heading. 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 you've applied a padding to the container, and margins won't collapse if there's any padding between them.

If you add top and bottom padding to the header, the margins inside it won't collapse to the outside (see figure 3.17). This means both the margins of the `<h1>` and the padding of the `<header>` would contribute space around the text, which might be a bit much. So it's probably best to remove the margin altogether and allow the padding to define the space you need.



Figure 3.17 Adding padding to the header prevents margin collapsing.

Update your stylesheet to match listing 3.11. You'll notice this now means there's no longer any space between the header and the main content. We'll come back to address that shortly.

Listing 3.11 Applying padding to the header

```
.page-header h1 {
  max-inline-size: var(--column-width);
  margin: 0 auto;
  padding: 1em 1.5rem;
}
```

← Removes the top and bottom margins

← Uses padding instead

Any time you see unexpected spaces above or below containers, or you see text pressed up against the top or bottom of its container, margin collapsing is most likely the cause.

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 an inline block, that is floated (chapter 12), or that has an absolute or fixed position (chapter 6).
- When using a flexbox or grid layout, margins won't collapse between elements that are part of the flex layout (chapters 4 and 5).
- 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.

3.6 Spacing elements within a container

The interplay between the padding of a container and the margins of its contents can be tricky to work with. Let's add some styles to the social links at the end of your page and work through problems that might arise.

You'll style the two links to social media pages and another, less important link. Your goal is for this section to look like figure 3.18.

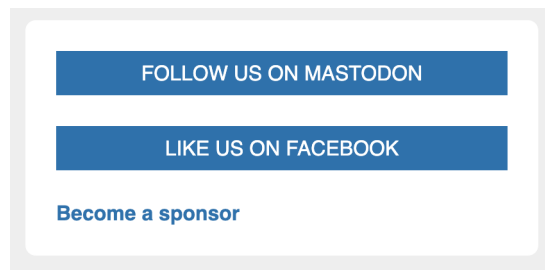


Figure 3.18 The goal layout: social links with properly spaced contents

Let's start with the two social links. They already have a `button-link` class that will be a good target for your CSS selector.

You'll apply styles for the links' 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 the following CSS to your stylesheet.

Listing 3.12 Setting size, fonts, and colors for the sidebar links

```
.social-links {
  max-inline-size: 25em;
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

.button-link {
  display: block;
  padding: 0.5em;
```

← Reduces the width of the container

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

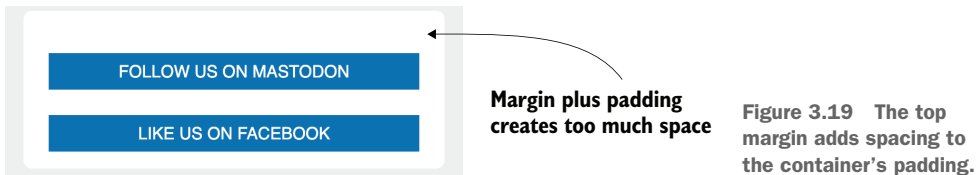
```

color: #fff;
background-color: var(--brand-color);
text-align: center;
text-decoration: none;
text-transform: uppercase;
}

```

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 container's padding. If you add `margin-top: 1.5em` to both links, you'll get the result shown in figure 3.19.



Now you'll have extra space at the top of the container. The first link'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 a number of ways. Listing 3.13 shows one of the simpler fixes. It uses the adjacent sibling combinator (+) to target only `button-links` that immediately follow other `button-links` as siblings under the same parent element. Now the margin only appears between two buttons.

Listing 3.13 Using an adjacent sibling combinator to apply a margin between buttons

```

.button-link {
  display: block;
  padding: 0.5em;
  color: #fff;
  background-color: var(--brand-color);
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}
.button-link + .button-link {
  margin-block-start: 1.5em;
}

```

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

This appears to work (figure 3.20). The first button no longer has a top margin, so the spacing is even.

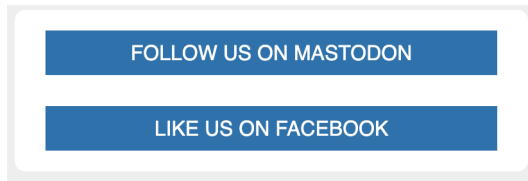


Figure 3.20 Even spacing is applied around the buttons.

Using the `+` combinator like this is a helpful pattern that you can use in a variety of scenarios to space a series of elements within a container. The same approach also works to apply a left margin between a series of `inline` or `inline-block` elements. See appendix A for a summary of all the various combinators.

3.6.1 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 3.14 Adding a different type of link to the sidebar

```
<aside class="social-links">
  <a href="/mastodon" class="button-link">
    Follow us on Mastodon
  </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 3.21 shows how the link will look *before* you fix the margin

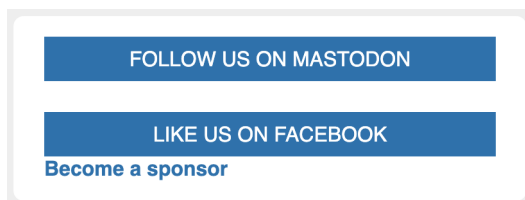


Figure 3.21 Spacing is off between the second button and the bottom link.

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

Listing 3.15 Adding styles for the sponsor link

```
.sponsor-link {  
  display: block;  
  color: var(--brand-color);  
  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.

3.6.2 Creating a more general solution

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: `* + *`. He wrote an article explaining this selector at <https://mng.bz/Y70z>.

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—`.button-link + .button-link`—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 that aren’t the first child of their parent. (Alternatively, you could use the selector `:not(:first-child)`, which functions essentially the same.)

I’ll show you how to use the lobotomized owl to add top margins to elements throughout your page. You’ll combine it with a new class name, `stack`, that you can add to any container where you want to vertically space all child elements. The result is shown in figure 3.22.

Add listing 3.16 to your stylesheet. This uses another combinator, the child combinator (`>`), to limit the lobotomized owl to only direct descendants of the `stack`. This little bit of code can do a lot of work on your page.

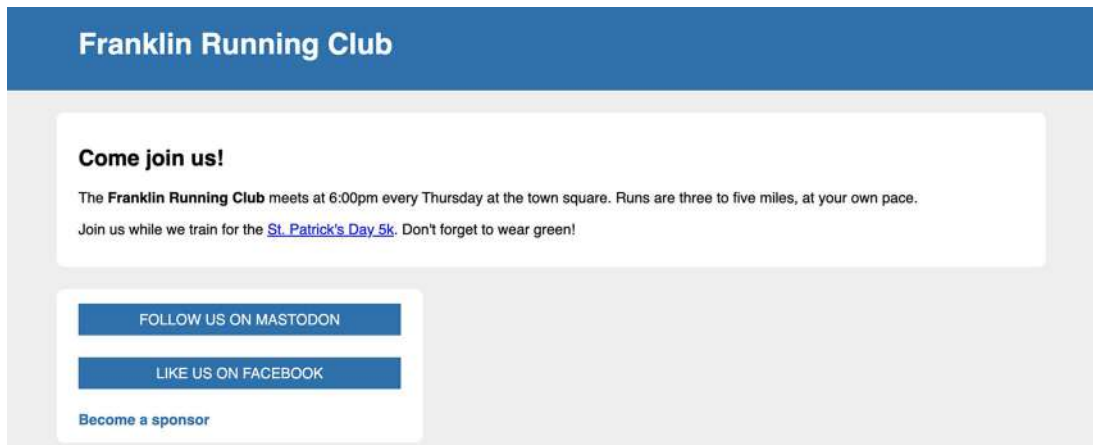


Figure 3.22 All elements are neatly spaced.

Listing 3.16 Creating a stack to vertically space elements in a container

```
.stack > * + * {
  margin-block-start: 1.5em;
}
```

← Targets all children in the stack except the first one

Then you can add the `stack` class to any containers where you want to space out child elements. First, add it to the `<body>` (to add space between the `<header>` and `<div class="container">`). Then, add it to `<div class="container">` to add space between the main content and the social links container. Finally, add it to the social links container to space out all three links within.

With this spacing added, your page is complete. The full stylesheet is given in the following listing.

Listing 3.17 Final stylesheet

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

:root {
  --brand-color: #0072b0;
  --column-width: 1080px;
}

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

```
.page-header {
  color: #fff;
  background-color: var(--brand-color);
}

.page-header h1 {
  max-inline-size: var(--column-width);
  margin: 0 auto;
  padding: 1em 1.5rem;
}

.container {
  max-inline-size: var(--column-width);
  margin-inline: auto;
}

.main {
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

.social-links {
  max-inline-size: 25em;
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

.button-link {
  display: block;
  padding: 0.5em;
  color: #fff;
  background-color: var(--brand-color);
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}

.sponsor-link {
  display: block;
  color: var(--brand-color);
  font-weight: bold;
  text-decoration: none;
}

.stack > * + * {
  margin-block-start: 1.5em;
}
```

Summary

- In normal document flow, inline elements are placed side by side, line wrapping as necessary. Block-level elements are each placed on their own line, filling the width of their container by default.

- Logical properties refer to the sides or sizes of an element in terms of the document flow, rather than their explicit cardinal directions.
- The box model describes the way margin, border, padding, and content define the size of an element. Using `box-sizing: border-box` makes the behavior of this sizing more intuitive.
- The height of elements is determined organically based on their contents. Explicitly setting a height can lead to overflow problems.
- Margins can collapse with other margins outside a container, leading to undesirable spacing between elements. Applying `overflow: auto` is one way to prevent this behavior.
- The “lobotomized owl” selector is a useful tool for adding space between block-level elements.

Part 2

Mastering layout

CSS provides several tools you can use to control the layout of a web page. In part 2, we'll look at the most important of these tools, from flexbox to grid 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. After that, you'll learn how to build responsive designs, so your page layout reacts to accommodate your users' various screen sizes.

4

Flexbox

This chapter covers

- Laying out elements of a page using flexbox
- Flex containers and flex items
- Main axis and cross axis
- Element sizing in flexbox
- Element alignment in flexbox

In chapter 3, we looked at normal document flow—the way the browser lays out elements on the page by default. In the next few chapters, we’ll explore the other forms of layout available to you, particularly flexbox, grid, and positioning. With these tools, you will have far more options available; you will be able to structure pages in almost any way imaginable.

Flexbox—formally Flexible Box Layout—is a method for laying out elements on the page. It’s primarily used for arranging elements in a row or column. It also provides a simple solution to the historically troublesome problems of vertical centering and equal-height columns.

If flexbox has one weakness, it’s the overwhelming number of options it provides. It introduces more than a dozen new properties to CSS, including some shorthand properties. These 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 ease into the topic by covering the most essential of these properties first.

I’ll show you the basic principles of the flexbox layout that you’ll need to understand, followed by practical examples. You don’t need to learn all the new properties to use flexbox. I’ve found that only a few of them do most of the heavy lifting. 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 reference guide that you can return to when you need it.

4.1 *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 affect only 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 4.1.

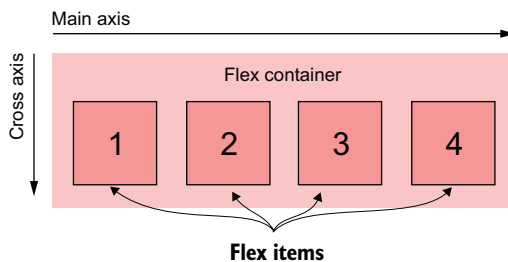


Figure 4.1 A flexbox container and its elements

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).

Conceptually, the main axis and cross axis are similar to inline and block directions. As you’ll see shortly, the flexbox properties make consistent use of the “start”

and “end” terminology. The direction of these axes can be changed—for instance, where you want elements to flow in another direction. I’ll show you how to do this later in the chapter.

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 new properties. To try it out, you’ll build the page shown in figure 4.2.

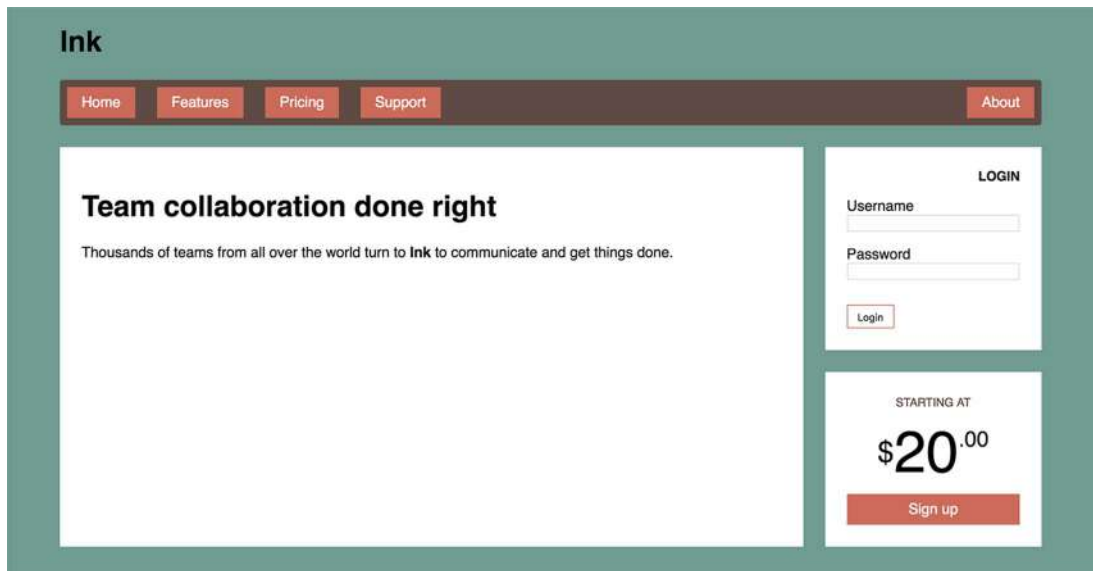


Figure 4.2 Finished page with a flexbox layout

I’ve structured this page to illustrate a number of ways to use flexbox. We’ll use flexbox for the navigation menu across the top, to lay out the three white boxes, and for the stylistic “\$20.00” on the bottom right.

Start a new page and link it to a new stylesheet, as shown in the following listing. Add this markup to your page.

Listing 4.1 Markup for the page

```
<!doctype html>
<html lang="en-US">
<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 stack">
      <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>
</html>

```

Navigation
menu

Large
main tile

← Sidebar containing
two stacked tiles

This HTML includes a link to styles.css. Create a file with that name and enter the CSS shown in the next listing. Most of these styles should look familiar after the last chapter.

Listing 4.2 Base styles for the page

<pre>*, ::before, ::after { box-sizing: border-box; }</pre>	Global box-sizing fix
<pre>body { margin: unset; background-color: #709b90; font-family: Helvetica, Arial, sans-serif; }</pre>	Sets green background color and sans-serif font for the page
<pre>.stack > * + * { margin-block-start: 1.5em; }</pre>	A “stack” for spacing elements
<pre>.container { max-inline-size: 1080px; margin-inline: auto; }</pre>	Double container to center page contents

Now that the page is set up, let’s start laying out some things with flexbox. You’ll start with the navigational menu at the top.

4.1.1 Building a basic flexbox menu

For this example, you’ll want the navigational menu to look like figure 4.3. Most of the menu items will align to the left, but you’ll move one over to the right side.



Figure 4.3 Navigation 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. The following shows 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 apply the colors. Figure 4.4 shows the result for these steps.



Figure 4.4 Menu with flexbox and colors applied

In the markup, you've given the `<ul>` a `site-nav` class, which you can use to target it in the styles. Add these declarations to your stylesheet.

Listing 4.3 Applying flexbox and colors to the menu

```
.site-nav {
  display: flex;
  padding: unset;
  list-style-type: none;
  background-color: #5f4b44;
}

.site-nav > li > a {
  background-color: #cc6b5a;
  color: white;
  text-decoration: none;
}
```

Makes `site-nav` a flex container and its children flex items

Removes the left padding and list bullets in the user-agent styles

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; 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.

4.1.2 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 the menu links. After this step, your menu will look like figure 4.5.

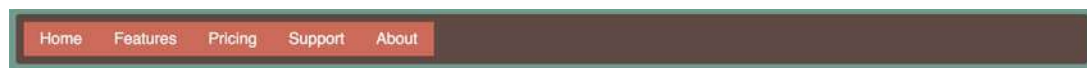


Figure 4.5 Menu with padding and link styles added

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 `<li>` 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 `<li>` 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 the next listing. This will fill out the padding of the menu.

Listing 4.4 Adding padding to the menu and its links

```
.site-nav {
  display: flex;
  padding: 0.5rem;
  list-style-type: none;
  background-color: #5f4b44;
}

.site-nav > li > a {
  display: block;
  padding: 0.5em 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 here.

Next, you'll need to add space between the menu items. You can do this with margins, but flexbox has a special property just for this called `gap`. For example, applying `gap: 1rem` will put 1 rem of space between each of the flex items.

Additionally, flexbox allows you to use `margin: auto` to fill all available space between flex items. You can also use this to push the final menu item all the way to the right side. After applying these changes, the menu will be complete (figure 4.6).



Figure 4.6 Gap and margins apply spacing between flex items.

See the styles for this in listing 4.5, where you'll apply a gap. This is stored in a custom property for reuse later. These changes also apply an `auto` margin to the left side of the last button, which pushes the last button all the way to the right. Add the following listing to your stylesheet.

Listing 4.5 Using margins to space the items

```

:root {
  --gap-size: 1.5rem;
}

.site-nav {
  display: flex;
  gap: var(--gap-size);
  padding: 0.5rem;
  list-style-type: none;
  background-color: #5f4b44;
}

.site-nav > .nav-right {
  margin-inline-start: auto;
}

```

← Adds a 1.5 rem gap between each of the flex items

← auto margins inside a flexbox will fill the available space.

You applied the `auto` margin to only one element (About). If you were to apply it to the Support menu item instead, it would shift both links to the right.

There are more options for spacing flex items, including the ability to spread them out evenly across the entire flex container. We'll take a look at these options in a bit, but the styles you've used here are appropriate for this nav menu.

4.2 *Flex item sizes*

When it comes to sizing flexbox elements, you can use the familiar `width` and `height` properties, but flexbox provides more options for sizing than these 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 (normally the width). In listing 4.6, you'll apply a flex layout to the main area of the page, and then you'll use the `flex` property to control the size of the columns. After these changes, your main area will look like figure 4.7.



Figure 4.7 Main area with a flex layout applied to the white tiles

Add the styles in listing 4.6 to your stylesheet. This 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 4.6 Applying a flexbox layout to the main container

<pre>.tile { padding: 1.5em; background-color: #fff; }</pre>	Adds a background color and padding to the three tiles
<pre>.flex { display: flex; gap: var(--gap-size); }</pre>	Applies a flexbox layout and gap to the main container

Now your content is divided into two columns: on the left is the larger area for the primary content of the page (`column-main`), and on the right is a login form and a small pricing box (`column-sidebar`). You haven't done anything yet to specify the width of the two columns, so they size themselves naturally based on their content. On my screen (figure 4.7), 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.

The `flex` property, which is applied to flex items, gives you several 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 listing to your stylesheet.

Listing 4.7 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.

4.2.1 *Flex basis*

The *flex basis* defines a sort of starting point for the size of an element—an “initial main size” (referring to its size along the main axis). 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 4.8 illustrates this.

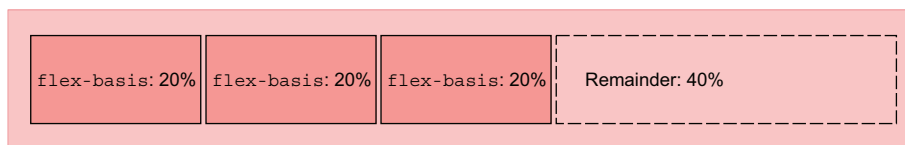


Figure 4.8 Three flex items with a flex basis of 20%, giving each an initial main size (width) of 20%

Once this initial main size is established for each flex item, they may need to grow or shrink to fit (or fill) the flex container along the main axis. That's where `flex-grow` and `flex-shrink` come in.

4.2.2 *Flex grow*

Once `flex-basis` is computed for each flex item, they (plus any gap or 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 4.8).

The remaining space (or remainder) will be consumed by the flex items based on their `flex-grow` values, which are always specified as nonnegative integers. If an item has a `flex-grow` of 0, it won't grow larger than its flex basis. If any items have a non-zero `flex-grow` value, 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 4.9).

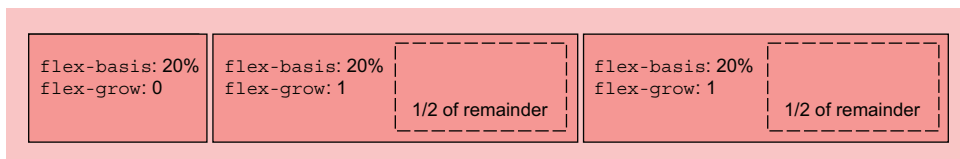


Figure 4.9 Flex items with `flex-grow` values will grow to fill available space (remainder) of the flex container.

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 4.10).

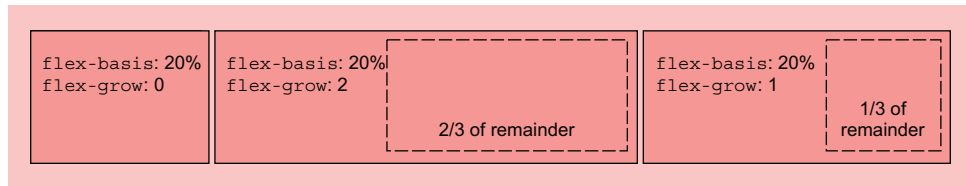


Figure 4.10 Items with a higher `flex-grow` value consume a higher proportion of the remaining available width.

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 gap 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 4.11).



Figure 4.11 The two columns fill the flex container’s width.

Favor the use of the shorthand `flex` property as you did here, instead of declaring only `flex-grow`. Unlike most shorthand properties, the values 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 factor of 1, flex shrink factor of 1, and a flex basis of 0%. These are most commonly what you’ll need.

4.2.3 *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 4.12).

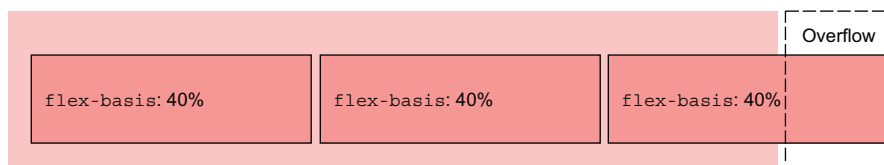


Figure 4.12 Flex items can have an initial size exceeding that of the flex container.

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. (The initial main size is also taken into consideration, so larger elements will shrink more than smaller ones.)

As an alternate approach to your page, you could achieve similar column 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 gap would overflow by 1.5 em. Give both columns a `flex-shrink` of 1, and some width 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 4.8 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 get effectively the same result as before (listing 4.7). Either one suits your purposes for this page.

NOTE If you look at the nitty-gritty details, there's a slight discrepancy between the results of listing 4.7 and listing 4.8. 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 4.7 is slightly wider than that in listing 4.8. If you need your measurements to be precise, grid is a better approach (chapter 5).

4.2.4 Some practical examples

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,” which is something libraries such as Bootstrap do, to provide a series of reusable classes defining columns of preset sizes. Figure 4.13 illustrates some layouts you can build with flexbox.

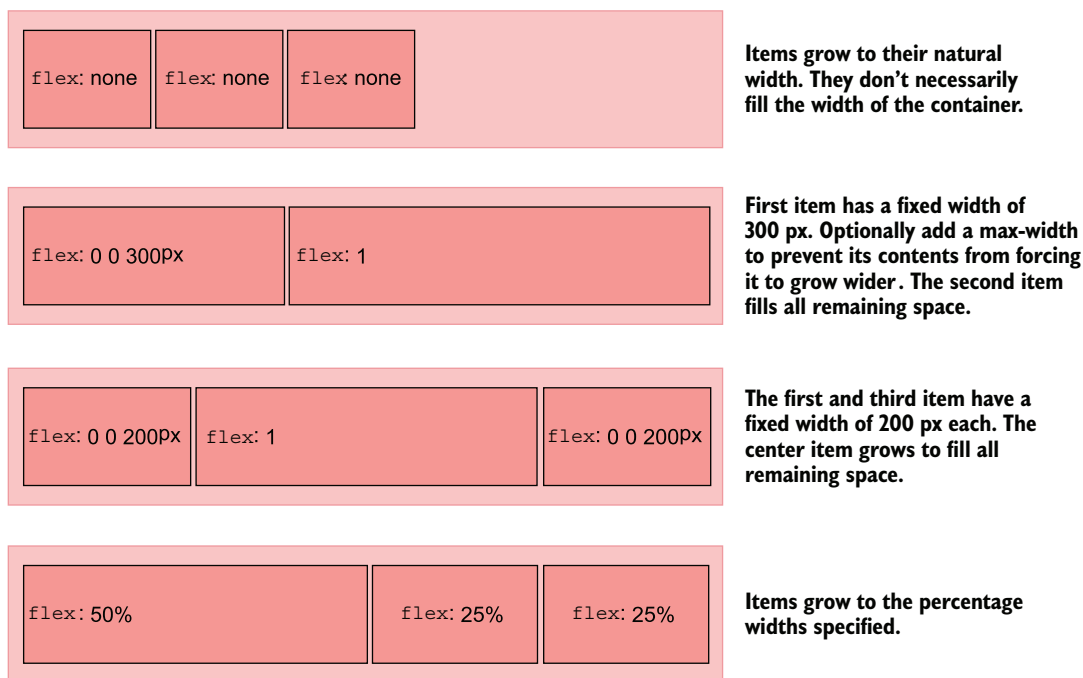


Figure 4.13 Some ways you can define item sizes using flex

The third example illustrates what was once called the “Holy Grail” layout. This is a layout that was once notoriously difficult in CSS. The two sidebars are a fixed width, whereas the 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. As you might imagine, you can mix and match these layouts in any number of ways, with any number of flex items.

4.3 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 in the inline direction, as you’ve done. Specifying `flex-direction: column` causes the flex items to stack vertically (in the block direction) instead. Flexbox also supports `row-reverse` to flow items right to left, and `column-reverse` to flow items bottom to top (figure 4.14).

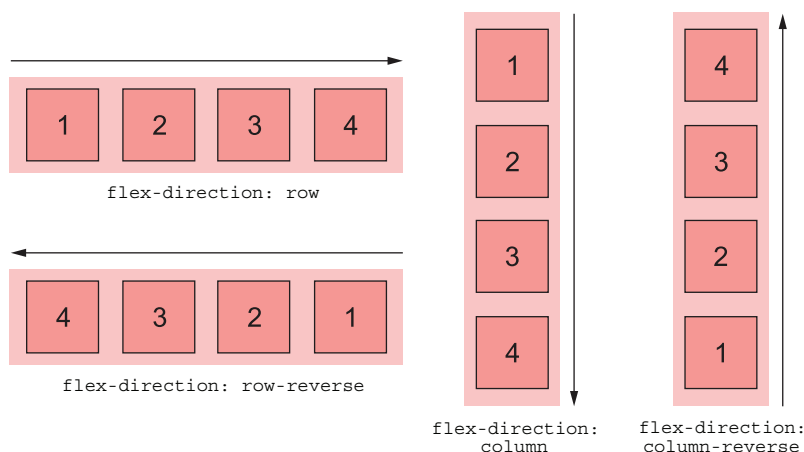


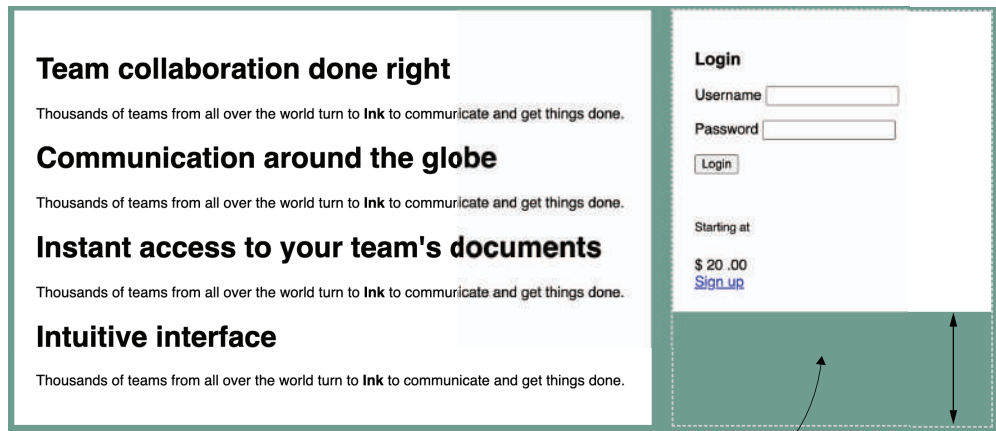
Figure 4.14 Changing the flex direction changes the main axis. The cross axis also changes to remain perpendicular to the main axis.

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 the 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 4.15.

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 4.15 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 4.16. 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).



Tiles should fill the height of their container

Figure 4.15 The main tile grows beyond the height of the tiles in the right column (dashed line indicates the size of column-sidebar).

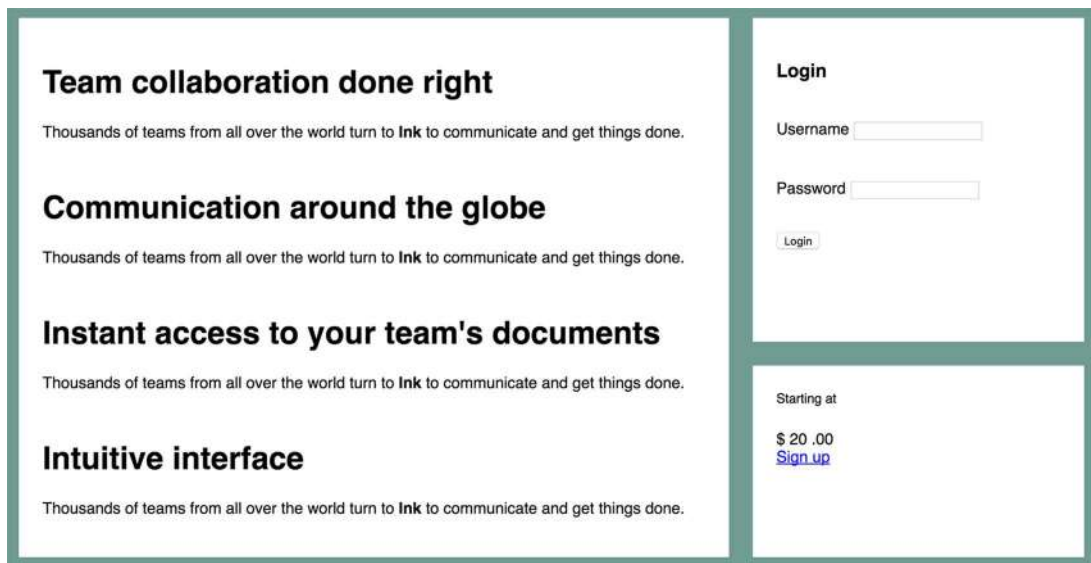


Figure 4.16 The ideal layout: tiles in the right column align with the large tile on the left.

4.3.1 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 4.9 Creating a flex column on the right

<pre>.column-sidebar { flex: 1; display: flex; flex-direction: column; gap: var(--gap-size); }</pre>	A flex item for the outer flexbox and a flex container for the new inner one
<pre>.column-sidebar > .tile { flex: 1; }</pre>	← 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 the following (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—recall that 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 still 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.

4.3.2 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. Afterward, the form should look like figure 4.17.

Figure 4.17 Login form

The `<form>` has the class `login-form`, so you’ll use that to target it in your CSS. Add the code in the following listing to your stylesheet. This will style the login form in three parts: the heading, the input fields, and the button.

Listing 4.10 Login form styles

```
.login-form h3 {
  margin: 0;
  font-size: 0.9em;
  font-weight: bold;
  text-align: end;
  text-transform: uppercase;
}
```

Makes the
heading bold,
right-aligned,
and all caps

```
.login-form input:not([type=checkbox]):not([type=radio]) {
  display: block;
  inline-size: 100%;
}
```

← Styles all text-
like inputs (not
checkboxes or
radio buttons)

```
.login-form button {
  margin-block-start: 1em;
  border: 1px solid #cc6b5a;
  background-color: white;
  padding: 0.5em 1em;
  cursor: pointer;
}
```

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 several 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 an exclusion-based approach, stating what you don't want to target. You could alternatively use an inclusion-based 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 an inline size 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` or `inline-size` 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.

4.4 *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: figures 4.18 and 4.19 list all the properties that may be applied to a flex container, and figure 4.20 lists all the properties for flex items. Many of these properties also apply to grid layout—which I'll cover in the next chapter.

Property	Values (initial values in bold)
flex-direction - Specifies the direction of the main axis - The cross axis will be perpendicular to the main axis.	row  row-reverse  column  column-reverse 
flex-wrap - Specifies whether flex items will wrap onto a new row inside the flex container - Items will wrap onto a new column if flex-direction is column or column-reverse.	nowrap  wrap  wrap-reverse 
flex-flow	Shorthand for <flex-direction> <flex-wrap>
gap	Specifies space between each flex item. Shorthand for <row-gap> <column-gap>.
justify-content Controls how items are positioned along the main axis	flex-start  start  left  flex-end  end  right  center  space-between  space-around  space-evenly 

Figure 4.18 Flex container properties

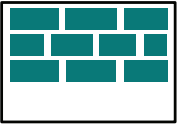
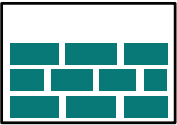
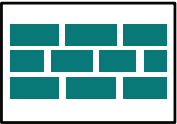
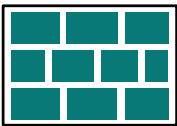
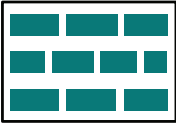
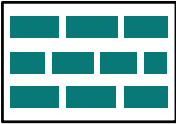
Property	Values (initial values in bold)	
align-content - Controls the spacing of the flex rows along the cross axis - Ignored if flex wrap is not enabled - Ignored if items don't wrap	flex-start start  flex-end end  center  stretch 	space-between  space-around  space-evenly 
place-content	Shorthand for <align-content> <justify-content>	
align-items	flex-start start self-start  flex-end end self-end  center 	stretch  baseline 
place-items	Shorthand for <align-items> <justify-items>. justify-items is used only in grid layout (chapter 5) and is ignored in flexbox layouts.	

Figure 4.19 Flex container properties (continued)

Property	Values (initial values in bold)
flex-grow - An integer that specifies the “growth factor” - Determines how much the item will grow along the main axis to fill unused space	
flex-shrink - An integer that specifies the “shrink factor” - Determines how much the item will shrink along the main axis, if needed, to prevent overflow - Ignored if the container has flex wrap enabled	
flex-basis Specifies the initial size of the item before flex-grow or flex-shrink is applied	<length> or <percent>
flex	Shorthand for <flex-grow> <flex-shrink> <flex-basis>
align-self - Controls how the item is aligned on the cross axis - Overrides the container's align-items value for the specific item(s) - Ignored if the item has an auto margin set on the cross axis	auto  center  flex-start start self-start  stretch  flex-end end self-end  baseline 
place-self	Shorthand for <align-self> <justify-self>. justify-self is used only in grid layout (chapter 5) and is ignored in flexbox layouts.
order An integer that moves a flex item to a specific position among its siblings, disregarding source order in the HTML	

Figure 4.20 Flex item properties

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 a `gap` and/or `flex-direction` on the container
- Declare `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—it can be difficult to keep all the `align-*` and `justify-*` property names straight. When you find you do need them, return here as a reference. Most of these final options are fairly straightforward.

4.4.1 *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 4.3. Let's look at some others.

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 several new keywords: `flex-start`, `flex-end`, `center`, `space-between`, `space-around`, and `space-evenly`. A value of `flex-start` (the default behavior) 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 there is a `gap` or 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 values `start` and `end` are logical values that align items left or right, depending on the writing mode, regardless of the direction of the main axis. The `left` and

right values are absolute, always aligning left or right, respectively, independent of writing mode or flex direction.

There are three ways to evenly space flex items, each providing a different amount of space before the first item and after the last item. The value `space-between` puts the first flex item flush to the beginning of the main axis and the last item flush to the end. The value `space-around` adds space outside the first and last items that is equal to half the space between items. And `space-evenly` adds space outside the first and last items that is equal to the space between items.

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-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`, `start`, `end`, `center`, `stretch` (the initial value), `space-between`, `space-around`, and `space-evenly`. These values apply spacing similar to the way described earlier for `justify-content`. There is also a `place-content` property, which is shorthand for `align-content` and `justify-content`.

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, 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).
- `start` and `end` are logical values that align the items based on the writing mode of the flex container.
- `self-start` and `self-end` are logical values that align each item based on its writing mode; these only differ from `start` and `end` if a flex item has a different writing mode than its flex container.
- `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 baseline is the line where the bottom edge of the text sits.

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. You can also add `first` or `last` to this, to specify which line of text to align. For example, `align-items: last baseline` will align the last line of text in each flex item to one another.

TIP It’s easy to confuse the names of the properties `justify-content` and `align-items`. I remember them by thinking of styling text in a word processor: you can “justify” text to spread it horizontally from edge to edge. And, much like `vertical-align`, you can “align” inline items vertically.

There is also a `place-items` property, which is shorthand for `align-items` and `justify-items`. Note that `justify-items` is used only in grid layout (chapter 5) and is ignored on flex containers.

4.4.2 Understanding flex item properties

I’ve described `flex-grow`, `flex-shrink`, `flex-basis`, and their collective shorthand, `flex` (section 4.2). 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`, `start`, `end`, `self-start`, `self-end`, `center`, `stretch`, and `baseline`.

There is also a `place-self` property, which is shorthand for `align-self` and `justify-self`. Note that `justify-self` is only used in grid layout (chapter 5) and is ignored on flex containers.

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 in which 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 the 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.

4.4.3 Using alignment properties

I'll show you how to use a few 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 4.21.

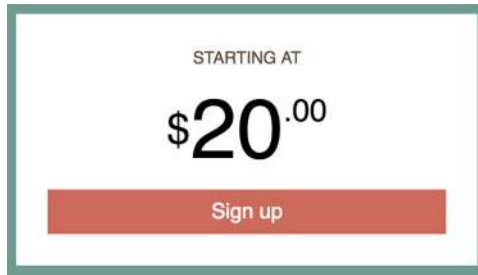


Figure 4.21 Stylized text using flexbox

The markup for this section is already in your page. It's as follows:

```
<div class="tile centered">
  <small>Starting at</small>
  <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, 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 the following listing to your stylesheet.

Listing 4.11 Setting styles for the cost tile

```
.centered {
  text-align: center;
}

.cost {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Centers flex items on both the main and cross axes

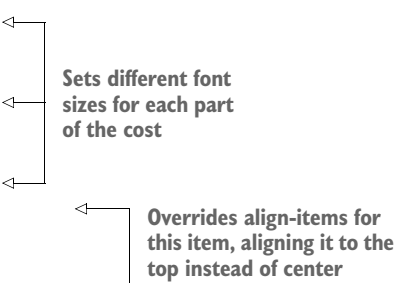
```

    line-height: 0.7;
  }

  .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: 0.5em 1em;
    text-decoration: none;
  }

```



Sets different font sizes for each part of the cost

Overrides align-items for this item, aligning it to the top instead of center

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: 0.7`. 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.

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 add flexbox only 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.

Summary

- Children of a flex container become flex items. The browser arranges flex items in a row or column.
- The `flex` shorthand property specifies `flex-grow`, `flex-shrink`, and `flex-basis` values to specify how the size of a flex item is determined.
- Flexboxes can be nested to piece together more complicated layouts and to fill the heights of naturally sized boxes.
- Flexbox automatically creates columns of equal height.
- Flexbox provides several properties that allow you to align and space flex items in countless ways.
- The `align-items` or `align-self` properties can vertically center a flex item inside its flex container.

5

Grid layout

This chapter covers

- Using grid to build page layouts
- Understanding grid layout options
- Placing items using grid lines and named grid areas
- Explicit and implicit 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 a piece of the picture. It has a big brother: a specification called the Grid Layout Module (grid). Together, these two specifications provide a full-featured layout engine.

In this chapter, I'll show you how you can start using grid layout. 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 fill only 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 5.1.

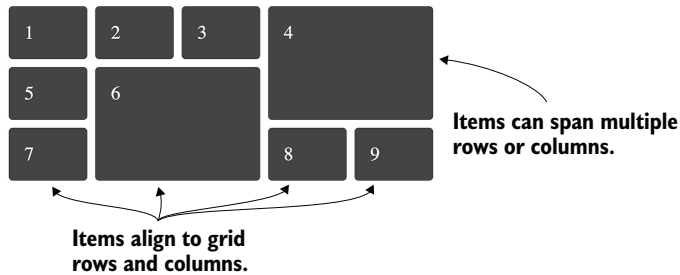


Figure 5.1 Boxes in a sample grid layout

5.1 *Building a basic grid*

Grid is extremely versatile, so I will walk you through several examples in this chapter to illustrate its capabilities. To get started, you'll build a very basic example. You'll lay out six boxes in three columns, as shown in figure 5.2. The markup for this grid is shown in listing 5.1.



Figure 5.2 A simple grid you'll construct 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 5.1 A grid with six items

```
<div class="grid">
  <div class="a">a</div>
  <div class="b">b</div>
  <div class="c">c</div>
  <div class="d">d</div>
  <div class="e">e</div>
  <div class="f">f</div>
</div>
```

← The grid container

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 the following listing to your stylesheet.

Listing 5.2 Laying out a basic grid

```
.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
  gap: 0.5em;
}

.grid > * {
  background-color: darkgray;
  color: white;
  padding: 2em;
  border-radius: 0.5em;
}
```

This code renders six equal-sized boxes in three columns (figure 5.2). Several 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 as `display: grid`.

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 much like the `flex-grow` factor in flexbox. The declaration `grid-template-columns: 1fr 1fr 1fr` declares three columns of 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.

The `gap` property defines the amount of space to add to the gutter between each grid cell, just like with flexbox. You can optionally provide two values to specify vertical and horizontal spacing individually (for example, `gap: 0.5em 1em`).

NOTE When the grid specification was first finalized, the `gap` property was named `grid-gap`, so some early examples of grid use `grid-gap`, which works the same way. The more general `gap` property was added later, and flexbox was updated to support it as well.

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.

5.2 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 5.3:

- *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 `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 grid cell is a single space on the grid, where a horizontal grid track and a vertical grid track overlap.
- *Grid area*—A grid area is a rectangular area on the grid made up of one or more grid cells. The area is between two vertical grid lines and two horizontal grid lines.

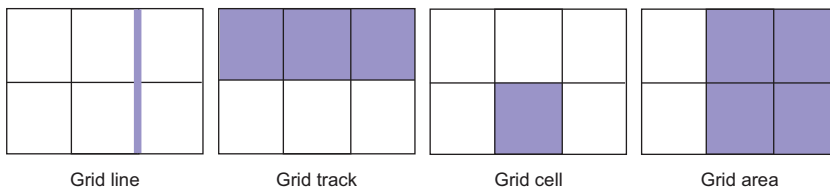


Figure 5.3 The parts of a grid

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 layout. The design is shown in figure 5.4. 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.

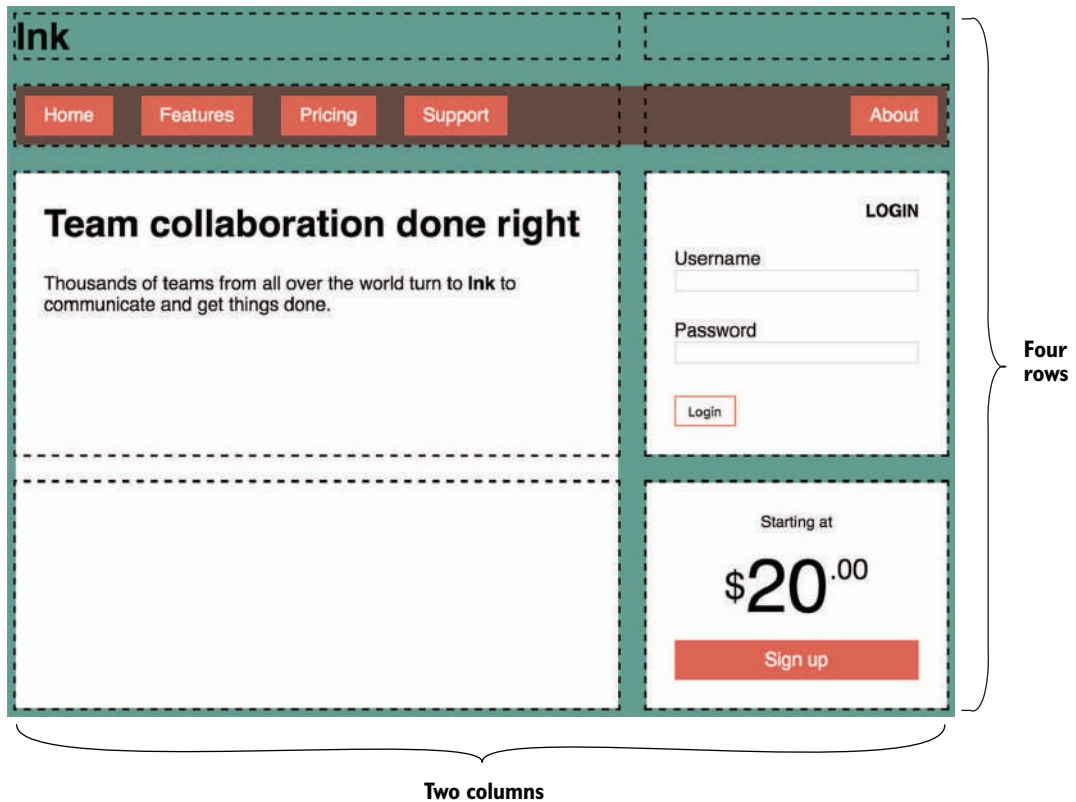


Figure 5.4 Page layout created with grid. The dashed lines are added to indicate the location of each grid cell.

TIP Your design doesn't need to fill every cell of the grid. Leave a cell empty where you want to add whitespace.

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, you'll use a different HTML structure: you'll 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 4) to match the following listing.

Listing 5.3 HTML structure for a grid layout

```
<body>
  <div class="container">
    <header>
      <h1 class="page-heading">Ink</h1>
    </header>
    <nav>
      <ul class="site-nav">
```

The “container” becomes your grid container.

Each grid item must be a child element of the grid container.

```

    <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="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 stack">
  <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 a grid layout to the page and put each section in place. We'll pull in a lot of styles from the version in chapter 4 momentarily, but first, let's get a general shape of the page in place. After building the basic grid, the page will appear as in figure 5.5.

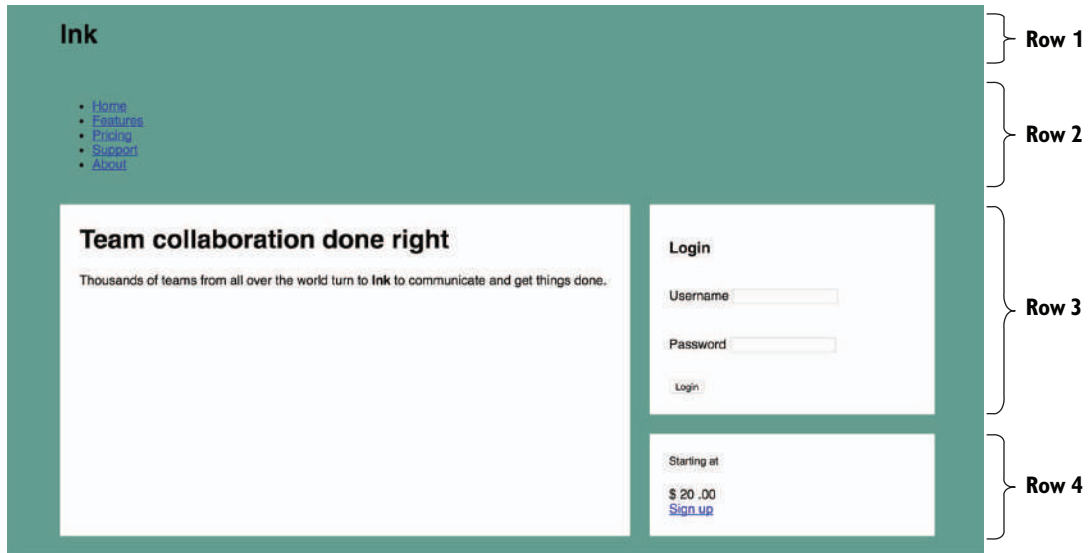


Figure 5.5 Page with basic grid structure in place

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 5.4 Applying a top-level page layout using grid

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

:root {
  --gap-size: 1.5rem;
}

body {
  background-color: #709b90;
  font-family: Helvetica, Arial, sans-serif;
}

.stack > * + * {
  margin-block-start: 1.5em;
}
```

```

.container {
  display: grid;
  grid-template-columns: 2fr 1fr;
  grid-template-rows: repeat(4, auto);
  gap: var(--gap-size);
  max-inline-size: 1080px;
  margin-inline: 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;
}

```

Defines two vertical grid tracks

Defines four horizontal grid tracks of size auto

Spans from vertical grid line 1 to grid line 3

Spans exactly one horizontal grid track

Places other grid items between various grid lines

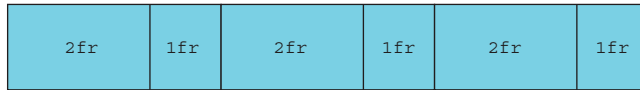
This listing provides a number of new concepts. Let's take them one at a time.

You've 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 5.6 illustrates the resulting columns.


```
grid-template-columns: repeat(3, 2fr 1fr);
```



```
grid-template-columns: 1fr repeat(3, 3fr) 1fr;
```



Figure 5.6 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.

5.2.1 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 5.7. The CSS uses these numbers to indicate where each item should be placed.

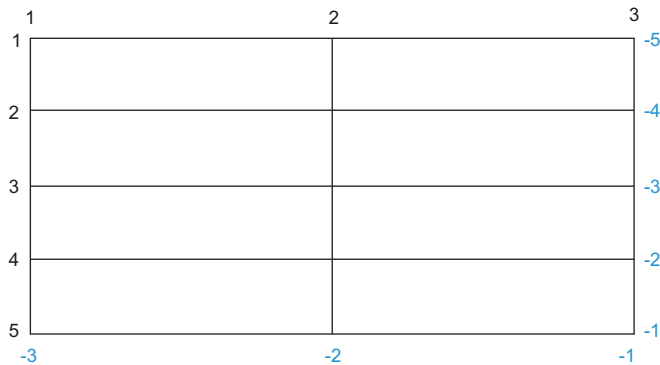


Figure 5.7 Grid lines are numbered beginning with 1 on the top left. Negative numbers refer to the position from the bottom right.

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 vertical 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.

On your page, several grid items are placed as follows:

```
.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 places 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 places 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.

5.2.2 *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 comes 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 5.8.

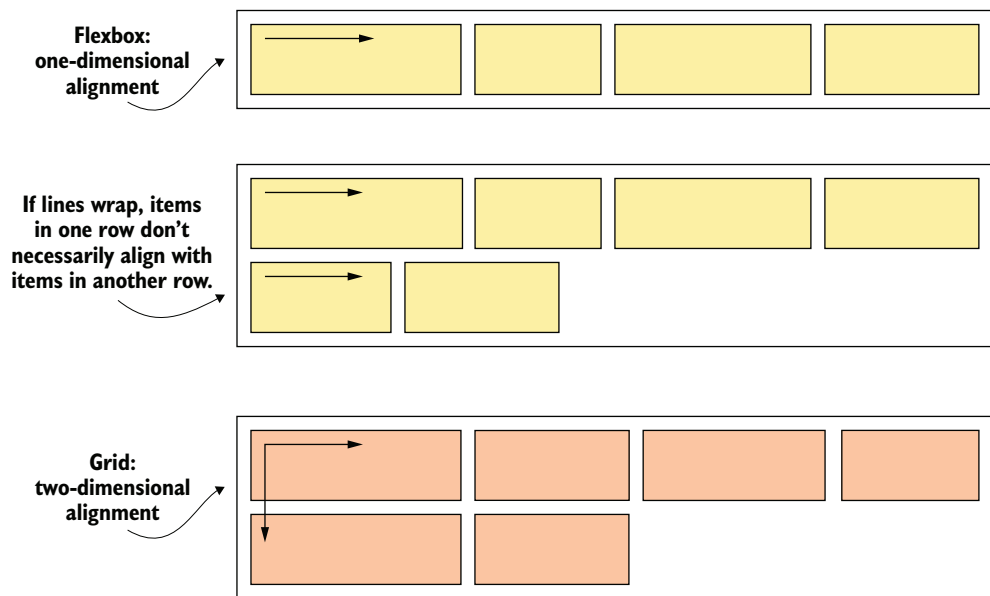


Figure 5.8 Flexbox aligns items in one direction, while grid aligns items in two directions.

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 a grid, however, you are first and foremost describing a layout, then placing items into that structure. While the content of each grid item can 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.

You've positioned the main regions of the page using a grid because you want the contents to adhere to the grid as it is defined. But for some other items on the page,

such as the navigation menu, you 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 5.9 shows the page with a top navigation 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.

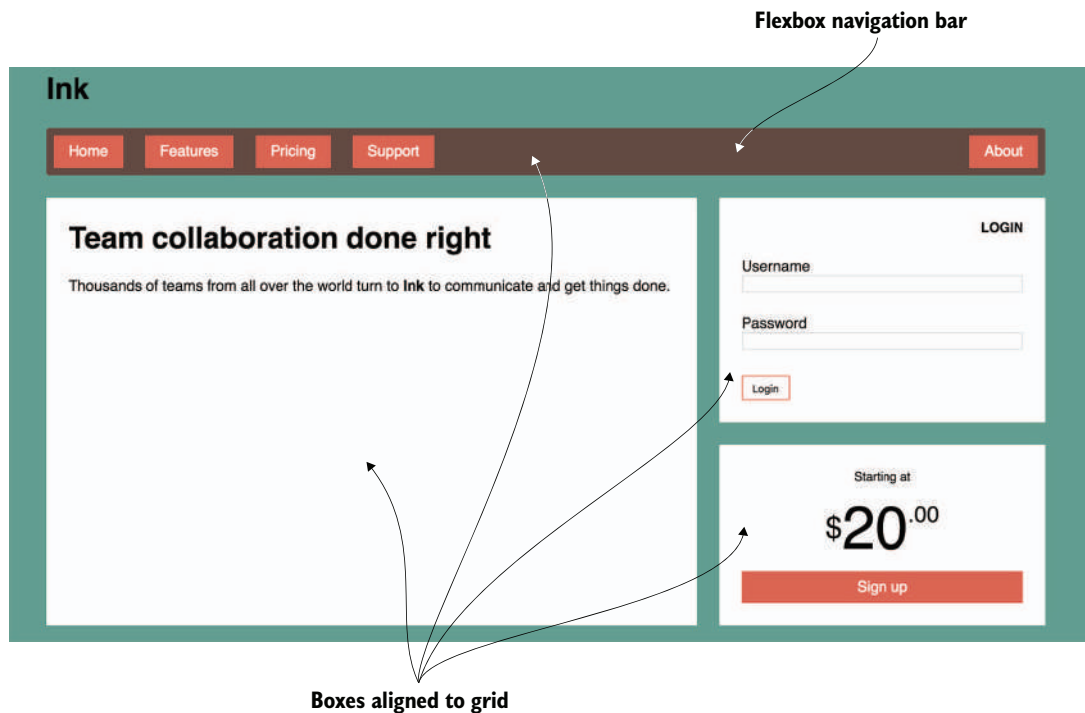


Figure 5.9 Fully styled page

The styles to do this are identical to those from the styles in chapter 4, minus the high-level layout you've applied using grid (listing 5.4). I've repeated them in the following listing. Add this to your stylesheet.

Listing 5.5 Remaining styling for the page

```
.page-heading {
  margin: 0;
}
```

```

.site-nav {
  display: flex;          ← The flexbox menu
  gap: var(--gap-size);
  margin: 0;
  padding: 0.5em;
  background-color: #5f4b44;
  list-style-type: none;
}

.site-nav > li > a {
  display: block;
  padding: 0.5em 1em;
  background-color: #cc6b5a;
  color: white;
  text-decoration: none;
}

.site-nav > .nav-right {
  margin-inline-start: auto;
}

.login-form h3 {
  margin: 0;
  font-size: 0.9em;
  font-weight: bold;
  text-align: right;
  text-transform: uppercase;
}

.login-form input:not([type="checkbox"]):not([type="radio"]) {
  display: block;
  width: 100%;
}

.login-form button {
  margin-block-start: 1em;
  border: 1px solid #cc6b5a;
  background-color: white;
  padding: 0.5em 1em;
  cursor: pointer;
}

.centered {
  text-align: center;
}

.cost {
  display: flex;          ← Stylized "cost"
                           using flexbox
  justify-content: center;
  align-items: center;
  line-height: 0.7;
}

.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: 0.5em 1em;  
  text-decoration: none;  
}
```

When your design calls for an alignment of items in two dimensions, use a 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.

NOTE Both grid and flexbox prevent margin collapsing between grid items or flex items. With the addition of a gap, margins added by user-agent styles can sometimes produce too much space between items. This is why you had to reset margins to 0 in a couple of places on this page.

5.3 *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 other. Let's look at both.

5.3.1 *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 key-word 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 listing 5.6 uses named grid lines to lay out the page. This produces the same result as the approach in listing 5.4. Update this portion of your stylesheet to match.

Listing 5.6 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);
  gap: var(--gap-size);
  max-inline-size: 1080px;
  margin-inline: 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 {
  grid-column: right;
  grid-row: 3 / 4;
}

.sidebar-bottom {
  grid-column: right;
  grid-row: 4 / 5;
}
```

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

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 5.10.

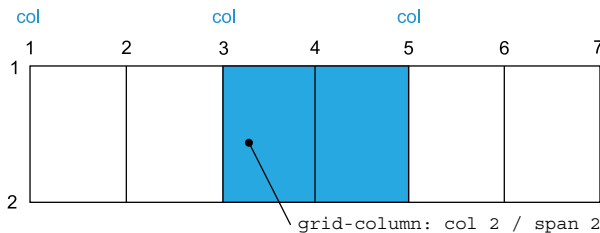


Figure 5.10 Placing a grid item at the second “col” grid line, spanning two tracks (col 2 / span 2)

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 place an item in the second set of columns (`grid-column: col 2 / span 2`).

5.3.2 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-areas` property on the grid container and a `grid-area` property on the grid items.

The code in listing 5.7 shows an example of this. Again, this code produces exactly the same result as the previous layout (listings 5.4 and 5.6). It's an alternate syntax that can be used instead. Update your stylesheet to match these styles.

Listing 5.7 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: var(--gap-size);
  max-inline-size: 1080px;
```

Assigns each grid cell
to a named grid area

Defines grid track
sizes as before


```

margin-inline: auto;
}

header {
  grid-area: title;
}

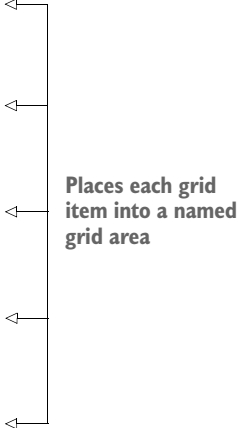
nav {
  grid-area: nav;
}

.main {
  grid-area: main;
}

.sidebar-top {
  grid-area: aside1;
}

.sidebar-bottom {
  grid-area: aside2;
}

```



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 a favorite for a lot of developers, and it shines when you know exactly where you want to place each grid item.

5.4 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 retrieved from 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 5.11 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 2 and 3), additional tracks are added to include it.

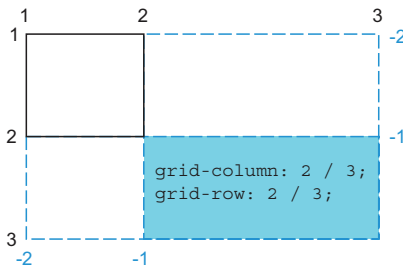


Figure 5.11 If a grid item is placed outside the declared grid tracks, implicit tracks will be added to the grid until it can contain the item.

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 5.12. 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.

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 in listing 5.8. Add it to the page.



Figure 5.12 A series of photographs laid out in a grid using implicit grid rows

Listing 5.8 Markup for a portfolio

```
<!doctype html>
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
<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>
</body>
</html>

```

The featured class will make certain images larger.

This markup includes 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 5.13). After that, you'll enlarge the “featured” images and apply a few other finishing touches.



Figure 5.13 Items automatically placed in grid cells from left to right

The styles for this are shown in listing 5.9. 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 two new concepts: `auto-fill` and the `minmax()` function, which I'll explain in a moment. Add the styles in the following listing to your stylesheet.

Listing 5.9 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-inline-size: 100%;
}

.portfolio figcaption {
  padding: 0.3em 0.8em;
  background-color: rgb(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

Semitransparent black

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 5.13, 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 <https://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 width of the entire grid container is filled.

5.4.1 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 of some grid items could result in gaps in the grid. Figure 5.14 illustrates

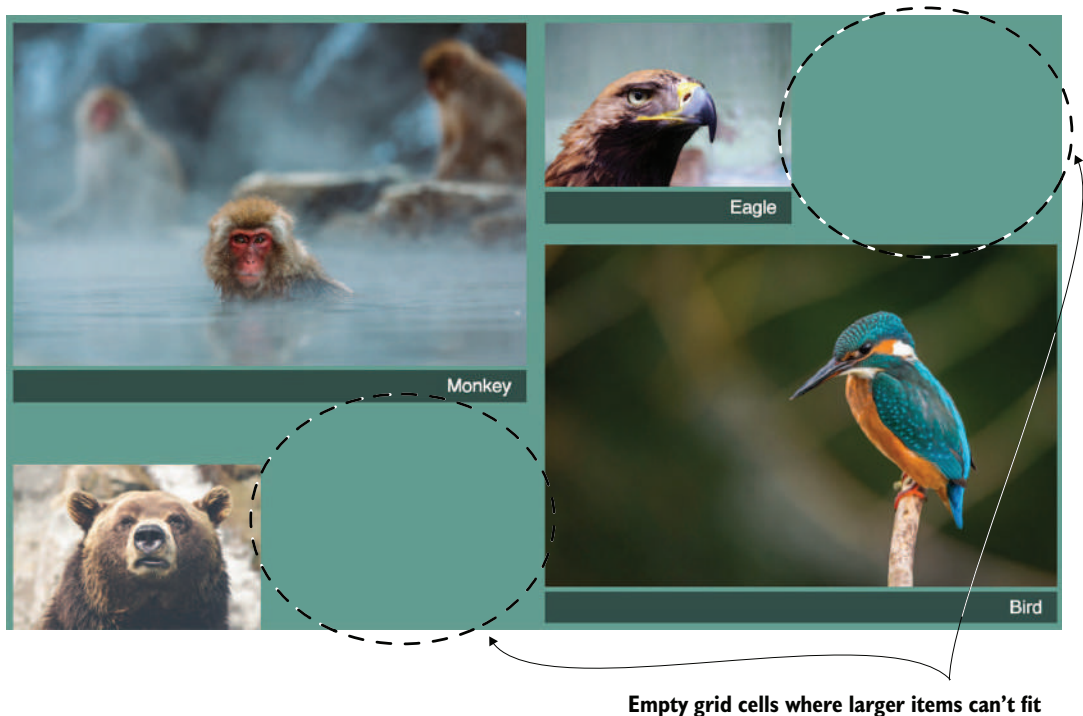


Figure 5.14 Increasing the size of some grid items introduced gaps in the layout where the large items don't fit.

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 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 5.15.

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.

Update your stylesheet with the changes indicated in the next listing. This enlarges featured images to fill two grid tracks in each direction and applies a `dense` auto-flow.

Listing 5.10 Enlarging featured images

```
.portfolio {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  grid-auto-rows: 1fr;
  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.

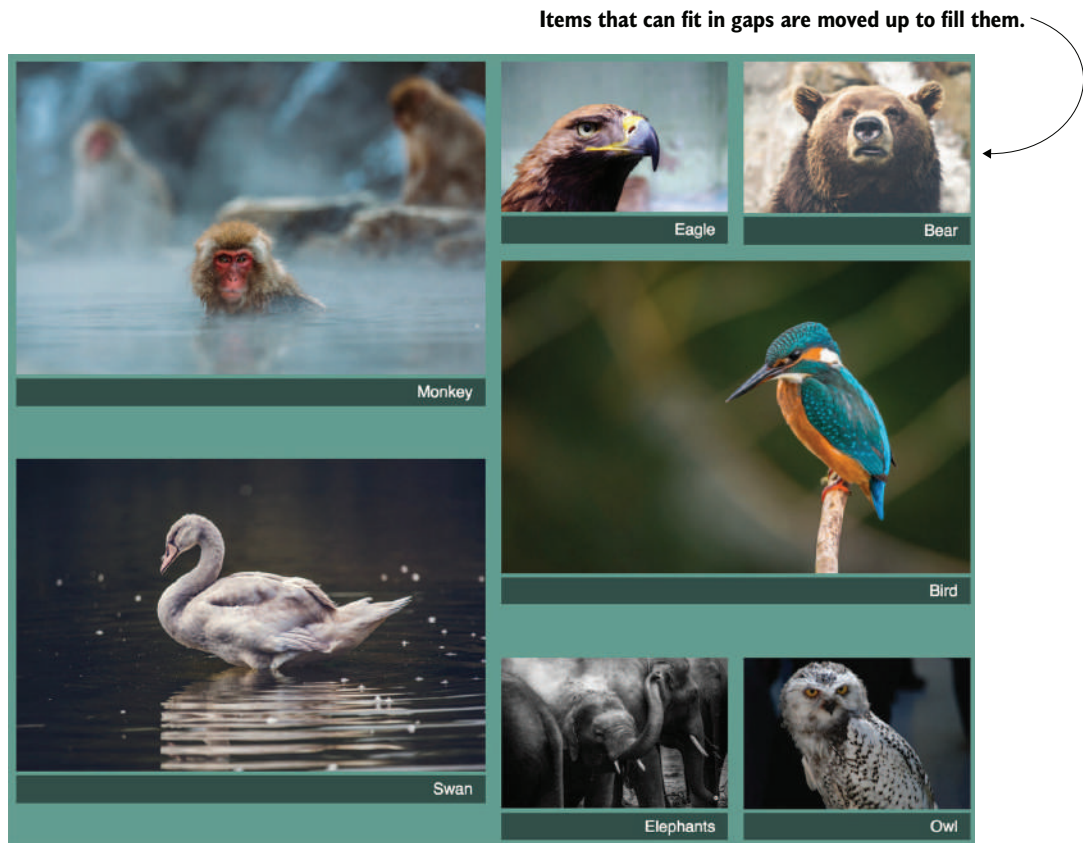


Figure 5.15 Using dense grid-auto-flow allows small grid items to backfill gaps in the grid.

Depending on your viewport size, your screen may not match figure 5.12 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.

5.4.2 *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 5.16.



Figure 5.16 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 5.11, 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 `<img>` has an `object-fit` value of `fill`, meaning the entire image will be resized to fill the `<img>` 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 5.17). 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`.

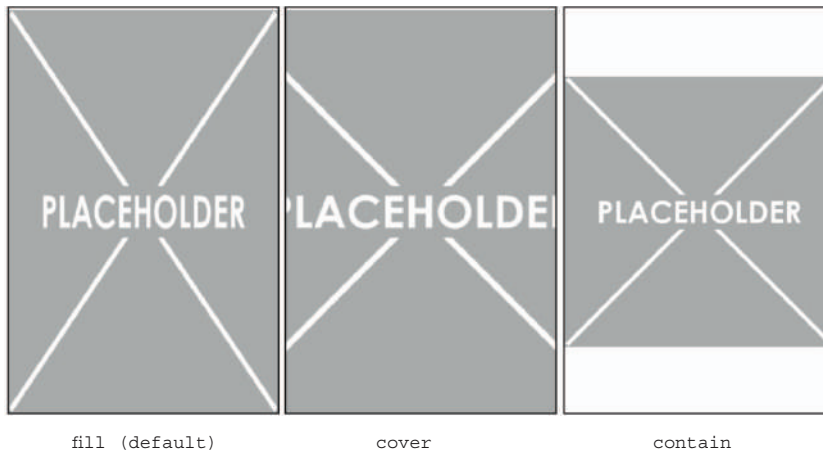


Figure 5.17 Using `object-fit` to control how an image is rendered in its box

There's an important distinction to make here: there is the box (determined by the `<img>` 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 5.18. For a more detailed look at this property, see <https://mng.bz/y8Kd>.

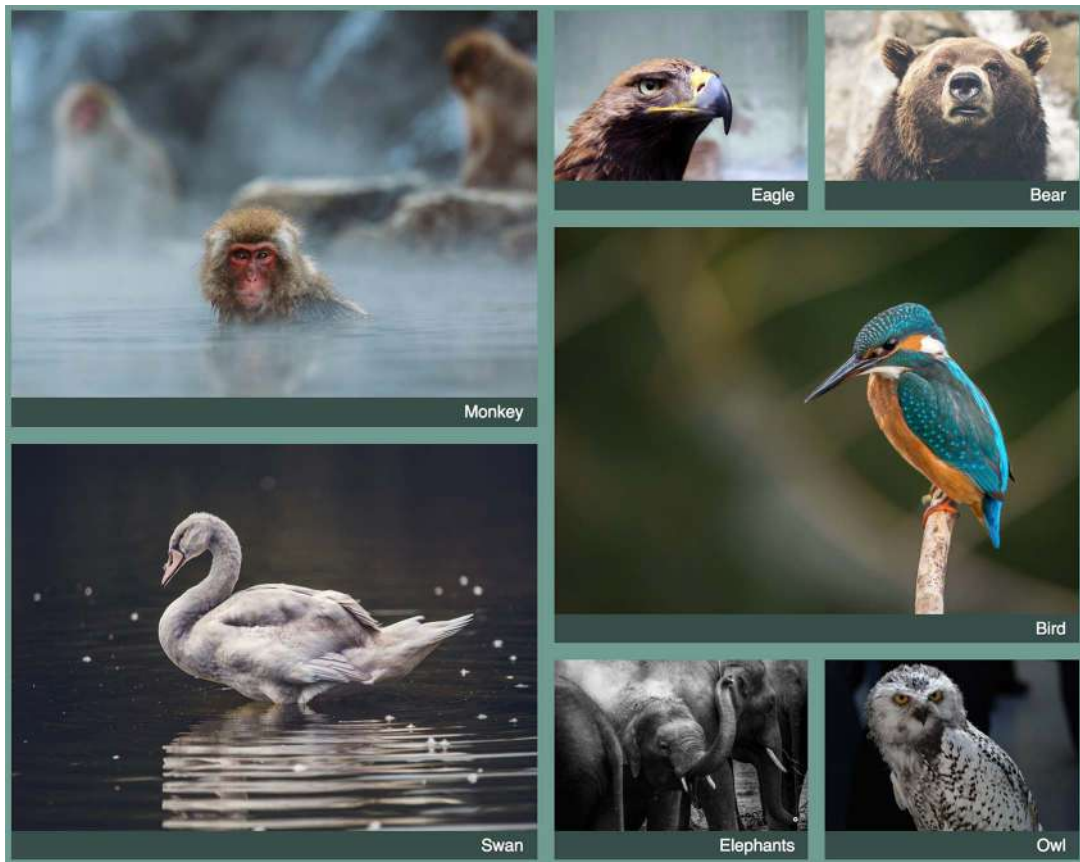


Figure 5.18 All images now fill their grid areas and align cleanly.

Now the top and bottom edges of all the images and their captions align in each grid track. The code for this is shown in the following listing. Add it to your stylesheet.

Listing 5.11 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;
  object-fit: cover;
  max-inline-size: 100%;
}
```

Makes each grid item
a vertical flexbox

Uses flex grow to make the image fill
the available space in the flex container

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.

5.5 Subgrid

In the previous examples, the grids have been restricted to two levels of the DOM: a grid container and its children. For some designs, you will need to align items that span a greater range of elements. For instance, you may need to align two elements that share a common grandparent element or give them an equal size. See figure 5.19 for an example of one such design.

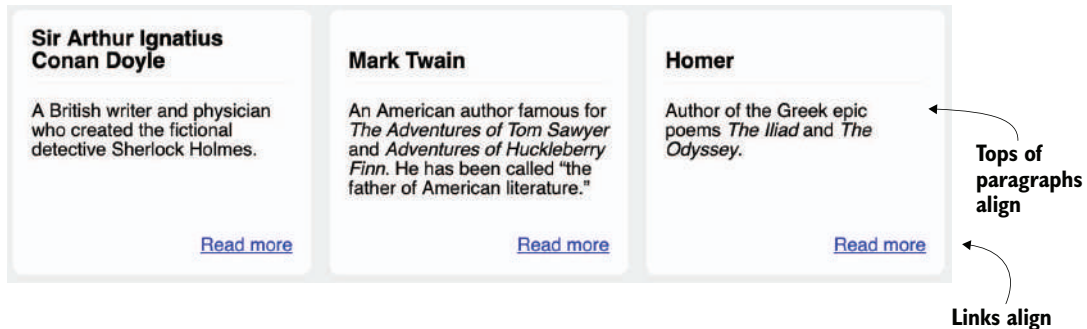


Figure 5.19 Cards with multiple aligned elements

In this design, the cards belong to a grid, so they appear aligned in a row, and each has the same height. But the elements within the cards also align with one another. In the cards with shorter headings, extra space at the top brings them in alignment with the longer heading. The paragraphs of each card begin at the same height, and the Read More links are aligned along the bottom.

The best way to accomplish this is with a newer feature called *subgrid*. With subgrid, you can place a grid within a grid and then position the inner grid's items on the grid lines of the parent grid. On this page, each card spans three rows, and the contents of each card align to each of those rows. Figure 5.20 shows the grid lines and gaps highlighted.

You can see here how the same grid defines the positions of the cards as well as the elements inside the cards.

TIP Figure 5.20 is a screenshot of grid rows as shown in Firefox using DevTools. In most browsers, you can debug a grid like this by clicking the small Grid label that appears beside the element in the Inspector pane. There is also a similar feature for flexbox.

Be aware that subgrid is a newer feature of CSS, and it is not as broadly supported as grid. It has been available in Firefox and Safari for some time, but Chrome and

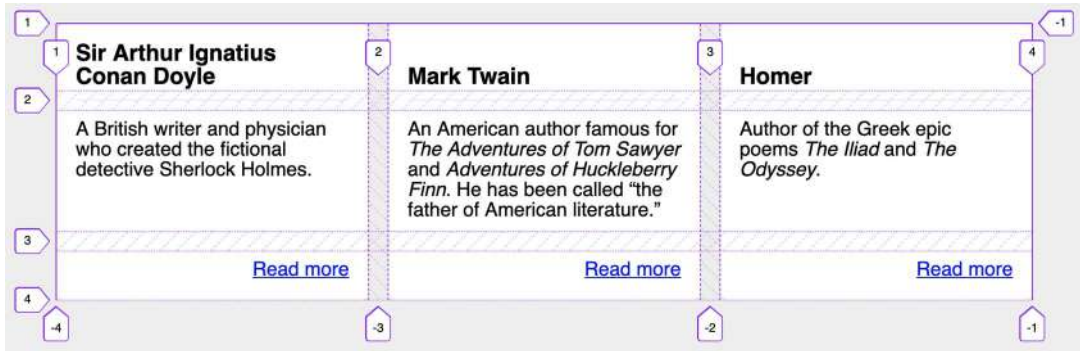


Figure 5.20 Grid lines and gaps shown

Edge added support in late 2023. See <https://caniuse.com/css-subgrid> for the latest support information.

I'll show you how to build this layout. Begin by starting a new page and adding the markup as shown in the following listing.

Listing 5.12 HTML for three author cards in a grid

```
<!doctype html>
<html lang="en-US">
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <div class="author-bios">
    <div class="card">
      <h3>Sir Arthur Ignatius Conan Doyle</h3>
      <div>
        <p>
          A British writer and physician who created
          the fictional detective Sherlock Holmes.
        </p>
      </div>
      <div class="read-more"><a href="/conan-doyle">
        Read more
      </a></div>
    </div>
    <div class="card">
      <h3>Mark Twain</h3>
      <div>
        <p>
          An American author famous for <i>The
          Adventures of Tom Sawyer</i> and
          <i>Adventures of Huckleberry Finn</i>.
          He has been called "the father of
          American literature."
        </p>
      </div>
    </div>
  </div>
```

Diagram annotations:

- Main grid container**: Points to the `<div class="author-bios">` element.
- Card elements**: Points to the `<div class="card">` elements.

```

    </div>
    <div class="read-more"><a href="/mark-twain">
      Read more
    </a></div>
  </div>
  <div class="card">
    <h3>Homer</h3>
    <div>
      <p>
        Author of the Greek epic poems <i>The
        Iliad</i> and <i>The Odyssey</i>.
      </p>
    </div>
    <div class="read-more"><a href="/homer">
      Read more
    </a></div>
  </div>
</div>
</body>
</html>

```



As you can see, this layout primarily deals with aligning three levels of DOM elements: the main grid container (`<div class="author-bios">`), its grid items (three `<div class="card">`s), and the children elements of the cards.

Begin with some basic styles, including background colors and spacing, as well as the top-level grid to put the cards in place. This is shown in listing 5.13. These should mostly be familiar styles at this point, with the possible exception of the grid, if you're still getting comfortable with that.

Listing 5.13 Laying out the author cards in a grid

```

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

.author-bios {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 1em;
  max-inline-size: 800px;
  margin-inline: auto;
}

.card {
  padding: 1rem;
  border-radius: 0.5rem;
  background-color: #fff;
}

.card > h3 {
  margin-block: 0;
  padding-block-end: 0.5rem;
}

```



```

border-block-end: 1px solid #eee;
}

.read-more {
  margin-block-start: 1em;
  text-align: right;
}

```

With these styles in place, the cards are laid out in a row, and they look pretty close to the design you want (figure 5.21). The last thing left to do is apply the subgrid.

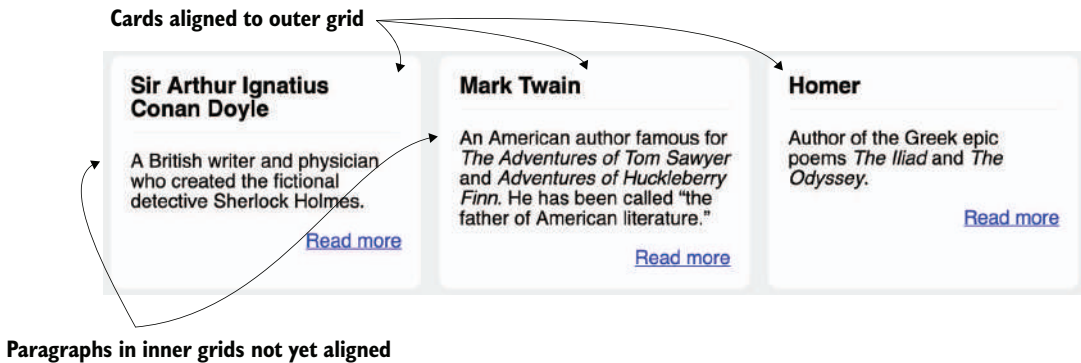


Figure 5.21 Three author tiles styled before adding subgrid

To add a subgrid to the page, you must first apply `display: grid` to the grid items, to create an inner grid inside the first one. Then you can use the keyword `subgrid` on the inner grid’s `grid-template-rows` and/or `grid-template-columns`, to indicate it should use its parent grid’s gridlines.

The CSS for this is shown next. Update your stylesheet to include these changes.

Listing 5.14 Adding a subgrid for the card contents

```

.card {
  display: grid;
  gap: 0;
  grid-row: span 3;
  grid-template-rows: max-content auto max-content;
  grid-template-rows: subgrid;
  padding: 1rem;
  border-radius: 0.5rem;
  background-color: #fff;
}

.card > h3 {
  align-self: end;
  margin-block: 0;
}

```

← Makes a card a grid container

← Makes the card span three rows

← Sets a fallback row sizing for browsers that don't support subgrid

← Uses the parent grid's rows

← Aligns the heading to the bottom of the grid row


```
padding-block-end: 0.5rem;
border-block-end: 1px solid #eee;
}
```

Because subgrid is a relatively new feature, I’ve supplied a fallback definition here for `grid-template-rows` that will approximate the subgrid behavior (it will push the Read More links to the bottom, but it will not align the bottoms of the headings). Alternatively, you could place these styles inside a feature query that checks for subgrid support: `@supports (grid-template-rows: subgrid) { ... }`.

This example uses subgrid to align to the parent grid’s rows, but you can also use it to align to columns using `grid-template-columns: subgrid`, as long as the parent grid defines them. You can also do this with both rows and columns. You can also continue the pattern down the DOM tree, nesting a subgrid inside another subgrid.

5.5.1 *Additional options*

In a subgrid, line numbers, line names, and grid area names are inherited from the parent grid, so you can use them to place items in the subgrid; for example, this will place all the card titles on the second row:

```
.card > h3 {
  grid-row: 2;
}
```

You can even supply new grid line names to the subgrid that are not present in the parent grid. Do this by providing a series of line names in brackets after the `subgrid` keyword:

```
.card {
  grid-template-rows: subgrid [line1] [line2] [line3] [line4];
}
```

You can refer to these the same as grid lines defined on a normal grid, as shown earlier in section 5.3.1. For example, `grid-row: line3 / line4` will place an item in the subgrid between the third and fourth horizontal grid lines.

NOTE One grid feature that may be coming in the future is a *masonry* layout, which is a layout that is popular for photo albums but requires the help of JavaScript to achieve. A masonry layout places grid items in equal-width columns but allows each item to be its natural height, so grid rows don’t necessarily align. See <https://www.matuzo.at/blog/2023/100daysof-day72/> for an overview from developer Manuel Matuzović.

5.6 *Alignment properties*

The grid module makes use of several of the same alignment properties that flexbox uses, as well as a few new ones. I’ve covered most of these in the previous chapter, but let’s 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.

There are also three alignment properties: `align-content`, `align-items`, `align-self`. These control vertical placement. I remember this by thinking about the `vertical-align` property from inline alignment. These properties are each illustrated in figure 5.22.

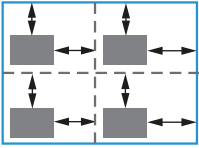
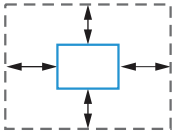
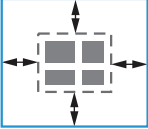
Properties	Applies to	Aligns
<code>justify-items</code> <code>align-items</code>	Grid container	Items within grid areas 
<code>justify-self</code> <code>align-self</code>	Grid item	Item within grid area 
<code>justify-content</code> <code>align-content</code>	Grid container	Grid tracks within container 

Figure 5.22 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 snippet:

```
.grid {
  display: grid;
  height: 1200px;
  grid-template-rows: repeat(4, 200px);
}
```

It explicitly sets the height of a grid container to 1200 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
- `end`—Places grid tracks to the bottom/right of the grid container
- `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 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

For detailed examples of all these justify/alignment properties, visit <https://gridbyexample.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.

Finally, there are shorthand properties for placement. These properties—`place-content`, `place-items`, and `place-self`—allow you to declare `align-*` and `justify-*` values at the same time. For instance, `place-content: center start` is equivalent to `align-content: center; justify-content: start`.

Because there's a lot to grid layout, I've only 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 defines a row- and column-based layout for your page so that items can be placed in relation to one another.
- Grid works in conjunction with flexbox to provide a complete layout system.
- Placement of grid items can be done using one of three syntaxes: numbered grid lines, named grid lines, and named grid areas. In any given situation, you can use whichever is the most intuitive for you and the needs of the layout.
- `auto-fill`/`auto-fit` and implicit grid placement work well when laying out a large or unknown number of grid items.
- Subgrid can align more deeply nested elements than a simple set of parent-child elements.
- The same alignment properties from flexbox apply to a grid.

Positioning and stacking contexts

This chapter covers

- The types of element positioning: fixed, relative, absolute, and sticky
- Building modal dialogs and dropdown menus
- Understanding z-index and stacking contexts
- Drawing a simple triangle with CSS

We've now looked at multiple ways to control the layout of the page, from normal document flow to flexbox and grid. 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 of which many developers only have a cursory understanding. Without a complete grasp of 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.

6.1 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. The same is true for `top`, `bottom`, and the viewport height.

You can also use the shorthand counterpart, `inset`, to specify the location of the elements. This property can conveniently specify all four sides at once. For example, `inset: 0` is equivalent to `top: 0; right: 0; bottom: 0; left: 0`. The `inset` property is shorthand for the following related logical properties:

- `inset-block-start`
- `inset-block-end`
- `inset-block` (shorthand for `inset-block-start` and `inset-block-end`)
- `inset-inline-start`
- `inset-inline-end`
- `inset-inline` (shorthand for `inset-inline-start` and `inset-inline-end`)

These shorthand properties tend to be useful when you would otherwise need a verbose set of declarations for the `top`, `right`, `bottom`, and `left` properties. The `inset` property itself also accepts up to four values to set all four values individually.

6.1.1 Creating a modal dialog with fixed positioning

You’ll use these properties to build the *modal* dialog box shown in figure 6.1. This dialog will pop up in front of the page content, masking it from view until the dialog is closed.

DEFINITION A modal dialog box is a window that appears in front of the main page. While it is present, the UI behind it is disabled, and the user must interact with the modal in some fashion to return to the main page.

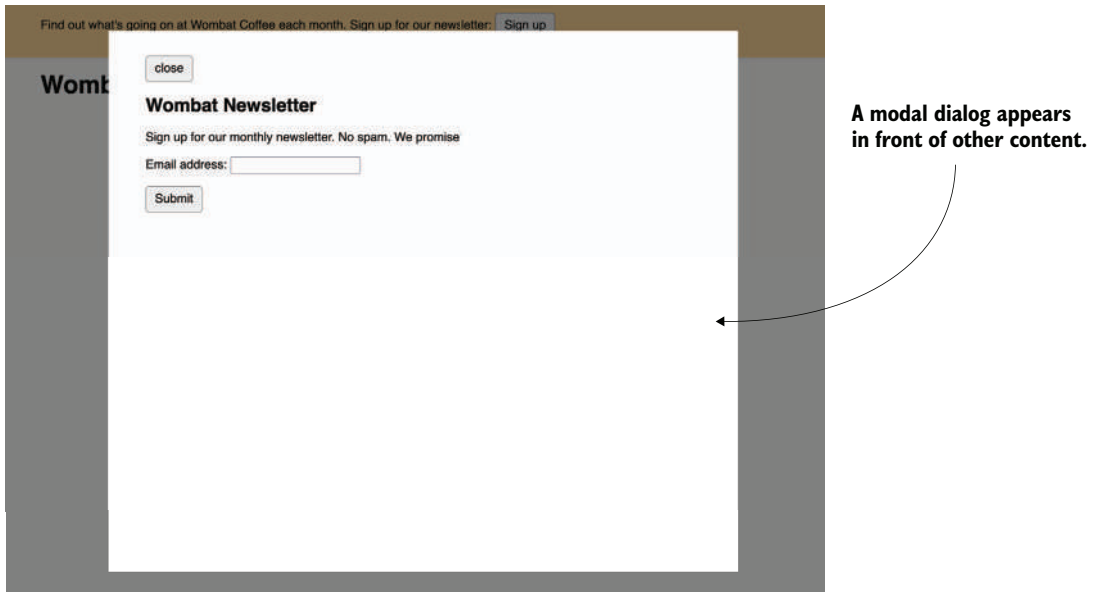


Figure 6.1 A modal dialog box

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 6.1) 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` to open the modal.

Create a new page and add listing 6.1 inside the `<body>` element. It places the content inside two containing elements and uses a `<script>` tag with some JavaScript to provide basic functionality. Note that the modal won't yet be fully working until you add CSS.

Listing 6.1 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" type="button">Sign up</button>
    </p>
  </div>
</header>
<div class="modal" id="modal" role="dialog" aria-modal="true">
```

Button to
trigger the
modal

Container for
the modal

```

<div class="modal-backdrop"></div>
<div class="modal-body">
  <button class="modal-close" id="close" type="button">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>
    <p><button type="submit">Submit</button></p>
  </form>
</div>
</div>
<main class="container">
  <h1>Wombat Coffee Roasters</h1>
</main>

<script type="text/javascript">
  const button = document.getElementById('open');
  const close = document.getElementById('close');
  const modal = document.getElementById('modal');

  button.addEventListener('click', function (event) {
    modal.classList.add('is-open');
  });

  close.addEventListener('click', function (event) {
    modal.classList.remove('is-open');
  });
</script>

```

Modal contents

A “backdrop” to obscure content behind the modal

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 opens 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 JavaScript here adds and removes an `is-open` class from the modal, which you use to toggle its visibility in the CSS.

NOTE HTML now has a `<dialog>` element that provides a lot of modal dialog behavior automatically. I am not using it in this example to more fully illustrate how fixed positioning behaves. For more on the dialog element, see my article at <https://mng.bz/MZ6B>.

The CSS to begin styling the page is shown next. I want to point out a few things in these styles before moving on to positioning the modal. The body has a `min-height` of 200 vh here. This is to force scrolling on the page; it will be helpful to see how the positioning examples you will be adding throughout the chapter interact with scrolling. Alternatively, you could add several paragraphs of content inside the `<main>` element until the page scrolls naturally.

Add the next listing to your stylesheet. It includes basic styling for the top banner and buttons.

Listing 6.2 Initial page styles

```
body {
  font-family: Helvetica, Arial, sans-serif;
  min-height: 200vh;
  margin: 0;
}

button,
input {
  font: inherit;
}

button {
  padding: 0.5em 0.7em;
  border: 1px solid #8d8d8d;
  background-color: #eee;
  border-radius: 5px;
  cursor: pointer;
}

.top-banner {
  padding: 0.5em;
  background-color: #ffd698;
  box-shadow: 0 1px 5px rgb(0 0 0 / 0.1);
}

.top-banner-inner {
  width: 80%;
  max-inline-size: 1000px;
  margin-inline: auto;
}

.container {
  width: 80%;
  max-inline-size: 1000px;
  margin: 1em auto;
}

.modal {
  display: none;
}

.modal.is-open {
  display: block;
}
```

Forces the page height to enable scrolling (for demo purposes only)

Overrides user-agent fonts applied to form elements

Hides the modal by default

Displays the modal while the is-open class is present

You have also added `font: inherit` to buttons and input fields. This overrides user-agent styles that apply some default font settings to form elements. This is a common step that I recommend adding to your stylesheets. Finally, the modal is removed from

the page with `display: none` until the `is-open` class is added, at which point it will be added back to the page by the `display: block` declaration in the following ruleset.

To style the modal itself, you'll use fixed positioning twice. First you'll use it on the `modal-backdrop` with the `inset` set to 0. This makes the backdrop fill the entire viewport. You'll give it a background color of `rgb(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'll use fixed positioning is on the `modal-body`. You'll position each of its sides inside the viewport: 3 em from the top and bottom edges and 20% from the left and right sides. You'll 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.

Add the styles shown in the following listing to your stylesheet.

Listing 6.3 Adding modal styles

<pre>.modal-backdrop { position: fixed; inset: 0; background-color: rgb(0 0 0 / 5); }</pre>	Uses a semitransparent backdrop to obscure the rest of the page while the modal is open
<pre>.modal-body { position: fixed; inset-block: 3em; inset-inline: 20%; padding: 2em 3em; background-color: white; overflow: auto; }</pre>	Positions the main part of the modal

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.

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.

6.1.2 Preventing the screen from scrolling while the modal dialog is open

Scrolling the page while the modal dialog is open is helpful for observing how fixed positioning works. However, it is not a great user experience. When a modal is in front of the main content of the page like this, the user usually expects to be able to interact with only the modal, not the other content.

You can fix this by applying `overflow: hidden` to the body while the modal is open. Add the following styles to your stylesheet:

```
body.no-scroll {  
  overflow: hidden;  
}
```

Then edit the JavaScript as shown in the following listing to apply these styles to the body by adding and removing the `no-scroll` class on the body.

Listing 6.4 Updating JavaScript to disable page scrolling

```
<script type="text/javascript">  
  const button = document.getElementById("open");  
  const close = document.getElementById("close");  
  const modal = document.getElementById("modal");  
  
  button.addEventListener("click", function (event) {  
    modal.classList.add("is-open");  
    document.body.classList.add("no-scroll");  
  });  
  
  close.addEventListener("click", function (event) {  
    modal.classList.remove("is-open");  
    document.body.classList.remove("no-scroll");  
  });  
</script>
```

Now when you open the modal, the page will no longer scroll. Close the modal, and scrolling will become reenabled.

For interactive elements on the page, this pattern of adding and removing classes to and from elements is very powerful. It often allows you to keep your JavaScript relatively simple, leaving the key logic for visual treatment in your stylesheet.

TIP In some browsers, you can accomplish this without the use of JavaScript by using the `:has()` pseudo-class. Instead of using the selector `body.no-scroll`, you can use `body:has(.modal.is-open)`. This will target the body element only while the modal has the `is-open` class applied to disable page scrolling. Firefox added support for the `:has()` pseudo-class in late 2023, so it may not be a viable approach quite yet. See appendix A for more on `:has()`.

6.1.3 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-navigation 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 example, 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.

6.2 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 inset properties (including `top`, `right`, `bottom`, and `left`) place the edges of the element within its containing block.

6.2.1 Absolutely positioning the Close button

To see how this works, you'll reposition the Close button to the top right corner of your modal dialog. After doing so, the modal will look like figure 6.2.



Figure 6.2 Close button positioned at top-right corner of modal dialog

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 the following listing.

Listing 6.5 Absolutely positioned Close button

```
.modal-close {
  position: absolute;
  top: 0.3em;
  right: 0.3em;
  padding: 0.3em;
}
```

**Positions the
button relative to
its containing block**

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, 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.

6.2.2 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 `×`, as in figure 6.3.



Figure 6.3 Close button replaced with an `×`

Your first temptation may be to remove the word `close` from your markup and replace it with an `×`, 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 `×`. 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 `×` to the button's `::after` pseudo-element and absolute positioning to center it within the button. Update your button's styles to match listing 6.4.

TIP Instead of the letter *x*, I recommend the unicode character for the multiplication sign. It's more symmetric and usually more aesthetically 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 6.6 Replacing the Close button with an $\times$

```
.modal-close {
  position: absolute;
  top: 0.3em;
  right: 0.3em;
  padding: 0.3em;
  border: 0;
  font-size: 2em;
  height: 1em;
  width: 1em;
  text-indent: 10em;
  overflow: hidden;
  background-color: transparent;
}

.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 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 $\times$ 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 DevTools 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 pop-up menus, tooltips, and "info" boxes. I'll show you how to use it to build a dropdown menu, but to do that, we'll need to first take a look at its companion, relative positioning.

6.3 *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 inset properties, if applied, will shift the element from its original position, but they won't change the position of any elements around it. See figure 6.4 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.

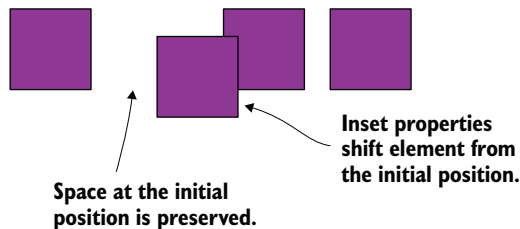


Figure 6.4 The second element shifted using relative positioning

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.

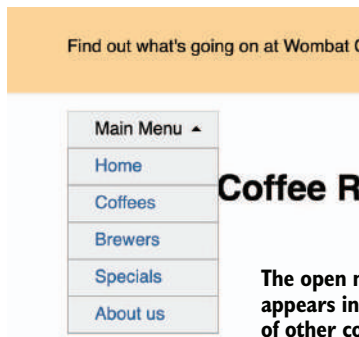
NOTE Unlike fixed and absolute positioning, you cannot use the inset properties to change the size of a relatively positioned element. Those values will only shift the position of the element up or down, left or right. You can use `top` or `bottom`, but not both together (`bottom` will be ignored); likewise, 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.

6.3.1 Creating a dropdown menu

I'll show you how to use relative and absolute positioning to create a dropdown menu. Initially, it'll be a simple rectangular button, but when the user clicks it, it pops open a list of links like those shown in figure 6.5.

Listing 6.5 shows the markup for this menu. Add it to your HTML at the beginning of the `<main class="container">` element. This code includes a container that you'll use to center the content and align it with the banner's content. I've included the `<h1>` on the page to illustrate how the popup appears in front of other content.



The open menu appears in front of other content.

Figure 6.5 Dropdown menu

Listing 6.7 Adding the dropdown menu's HTML

```
<main class="container">
  <nav>
    <div class="dropdown" id="dropdown">
      <button type="button" class="dropdown-toggle"
        id="dropdown-toggle">Main Menu</button>
      <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>
```

Container for the dropdown

The toggle button will always be visible.

Div to show and hide as the menu is opened and closed

The dropdown container includes two children: the toggle button, which is always visible, and the dropdown menu, which 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.

You'll need a bit of JavaScript to open and close the menu when the toggle button is pressed. Add the script shown in the following listing to the page to add this functionality.

Listing 6.8 JavaScript to open and close the dropdown menu

```
<script type="text/javascript">
  const dropdownToggle = document.getElementById("dropdown-toggle");
  const dropdown = document.getElementById("dropdown");
```

```
dropdownToggle.addEventListener("click", function (event) {
    dropdown.classList.toggle("is-open");
});
</script>
```

Similar to the modal you built earlier in the chapter, this JavaScript adds and removes an `is-open` class on the dropdown. Based on that, the CSS will be able to define the opened and closed styles.

Next, you'll apply relative positioning to the dropdown container. This establishes the containing block for the absolutely positioned menu within. Add these styles to your stylesheet, as shown in the following listing.

Listing 6.9 Displaying the open dropdown menu

```
.dropdown {
    display: inline-block;
    position: relative;
}

.dropdown-toggle {
    padding: 0.5em 1.5em;
    border: 1px solid #ccc;
    background-color: #eee;
    border-radius: 0;
}

.dropdown-menu {
    display: none;
    position: absolute;
    left: 0;
    top: 2.1em;
    inline-size: max-content;
    min-inline-size: 100%;
    background-color: #eee;
}

.dropdown.is-open .dropdown-menu {
    display: block;
}

.submenu {
    padding-inline-start: 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: 0.5em 1.5em;
```

Establishes
the containing
block

Hides the
menu initially

Positions the dropdown
menu below the toggle
button

Reveals the
menu when
active

```
background-color: #eee;
color: #369;
text-decoration: none;
}

.submenu > li > a:hover {
  background-color: #fff;
}
```

Now when you click the Main Menu toggle button, the dropdown menu pops open beneath it. Clicking the toggle again will close it. Note that similar functionality could be accomplished using a `:hover` pseudo-class instead, but that approach would not be compatible with keyboard navigation or touchscreen devices. (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.)

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` of 100% ensures it will 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.

TIP When a dropdown menu is on the right edge of the screen, it often makes sense to position it using `right: 0` rather than `left: 0`. That way, wide menu items will extend to the left of the toggle button instead of overflowing outside the right edge of the viewport.

Accessibility is an important concern when building web pages, and this is doubly so for interactive elements such as dropdown menus. At a bare minimum, always ensure you can fully interact with your page via the keyboard. For this dropdown menu, you can use the Tab key to focus on the menu's toggle button and the space bar to open and close the menu. When the menu is open, you can use Tab to cycle through the menu items. (On macOS, you may need to enable keyboard navigation in your System Preferences.) For more on building accessible dropdowns, see <https://mng.bz/aEjm>.

6.3.2 *Creating a CSS triangle*

There is one finishing touch you can add 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. You'll 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 6.6. I've made each side of the border a unique color to highlight where one edge ends and the next begins.



**Thick borders
join at a
diagonal edge.**

Figure 6.6 An element with thick borders

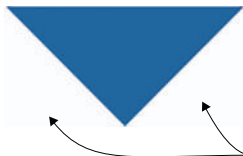
Notice what happens in the corners where the borders from two edges meet: they form a diagonal edge. Now observe what happens if you shrink the element to a height and width of zero (figure 6.7). The borders all come together and join in the middle.



**An empty content
area creates
triangular borders.**

Figure 6.7 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 6.8—a simple triangle.



**Transparent borders
on other sides leave
a single visible triangle.**

Figure 6.8 Triangle formed from an element's top border

Apply styles to the `dropdown-label::after` pseudo-element to make a triangle and position it using absolute positioning. Add the following listing to your stylesheet.

Listing 6.10 Absolutely positioning a triangle on the dropdown label

```

.dropdown-toggle {
  padding: 0.5em 2em 0.5em 1.5em;
  border: 1px solid #ccc;
  background-color: #eee;
  border-radius: 0;
}

.dropdown-toggle::after {
  content: "";
  position: absolute;
  right: 1em;
  top: 0.9em;
  border: 0.3em solid;
  border-color: black transparent transparent;
}

.dropdown.is-open .dropdown-toggle::after {
  top: 0.6em;
  border-color: transparent transparent black;
}

```

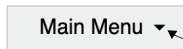
← Increases the right padding to make space for the arrow

Positions the element on the right side of the toggle button

Uses the top border to form a down arrow

← When open, 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 6.9.



Triangle built from a border

Figure 6.9 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 0.9 em to 0.6 em) helps the up arrow to optically appear in the same spot as the down arrow.

Alternatively, 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. You can also inline a small SVG in the HTML, but I find if I'm not already using an icon pack of some sort in a project, this CSS-only approach can be convenient. 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 into CSS, see <https://css-tricks.com/the-shapes-of-css/>. These are often not needed, especially with the rise of SVG icons, but they illustrate the versatility of CSS.

6.4 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 6.10. Notice the dropdown you added to the page now appears in front of the modal.

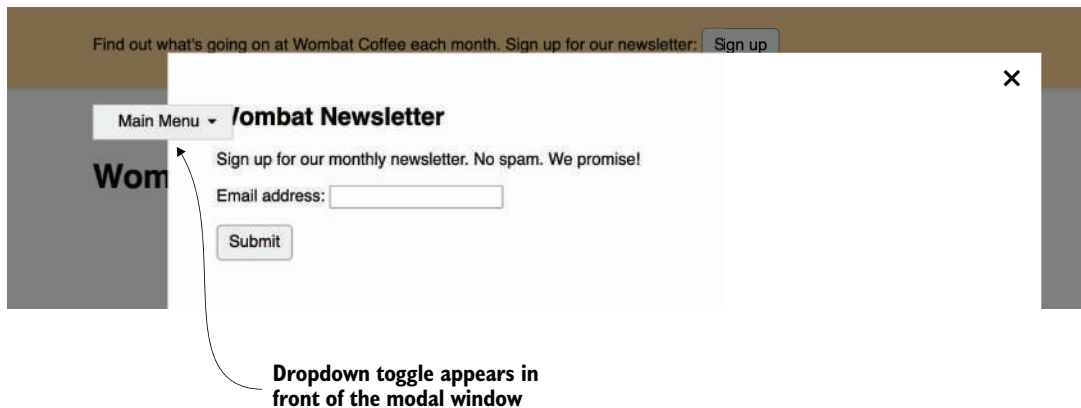


Figure 6.10 The modal dialog incorrectly appears behind the dropdown menu.

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.

6.4.1 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 6.11. 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.



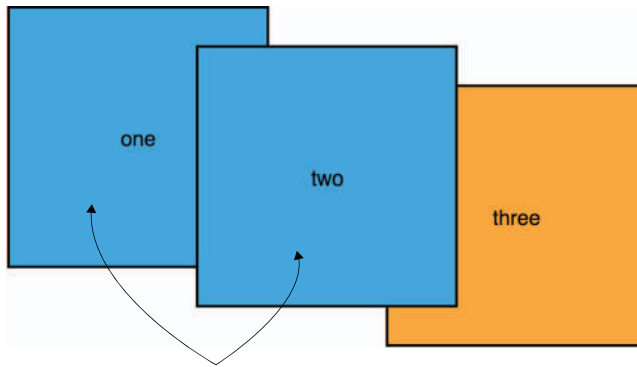
Figure 6.11 Three elements stacking normally

This behavior changes when you start positioning elements. The browser first paints all nonpositioned elements; then it paints the positioned ones. By default, any positioned elements appear in front of any nonpositioned elements. In figure 6.12, 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.

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.



Positioned elements appear in front of statically positioned ones.

Figure 6.12 Positioned elements are painted in front of static elements.

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 with the `z-index` property.

6.4.2 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. 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 following listing.

Listing 6.11 Bringing the modal to the front with `z-index`

```
.modal-backdrop {
  position: fixed;
  inset: 0;
  background-color: rgb(0 0 0 / 0.5);
  z-index: 1;
}
```

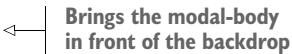
```
.modal-body {
  position: fixed;
  inset-block: 3em;
  inset-inline: 20%;
  padding: 2em 3em;
  background-color: white;
}
```

← Brings the backdrop in front of elements without a `z-index`

```

overflow: auto;
z-index: 2;
}

```



Brings the modal-body
in front of the backdrop

Z-index seems straightforward, but using it introduces two gotchas. First, z-index works only 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.

6.4.3 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.

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 I'll show you how to put together a minimal scenario to illustrate this. In a fresh HTML page, add the markup in the following listing.

Listing 6.12 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 6.13).

The styles for this scenario are in listing 6.10. 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.

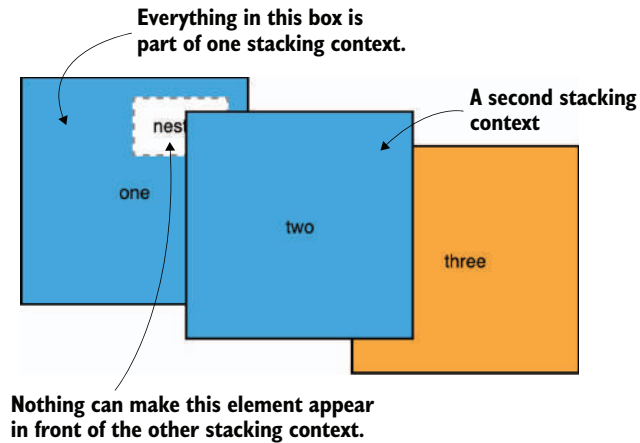


Figure 6.13 The entire stacking context is stacked together relative to other elements on the page.

Listing 6.13 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;
  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 is not the only way to create a stacking context. An opacity below 1 creates one, as do the transform or filter properties. Fixed positioning and sticky positioning (section 6.5) always create a stacking context, even without a z-index. 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
- Nonpositioned elements
- Positioned elements with a z-index of auto and their children
- Positioned elements with a positive z-index and their children

Using 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 999,999. After this happens a few times, it's anybody's guess what the z-index for another new component should be.

To keep this under control, use custom properties 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 reassess 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 do this only when you need to stack elements in front of one another.

6.5 Sticky positioning

The last type of positioning is *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. A common use case for this is sidebar navigation.

Return to 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 6.14.

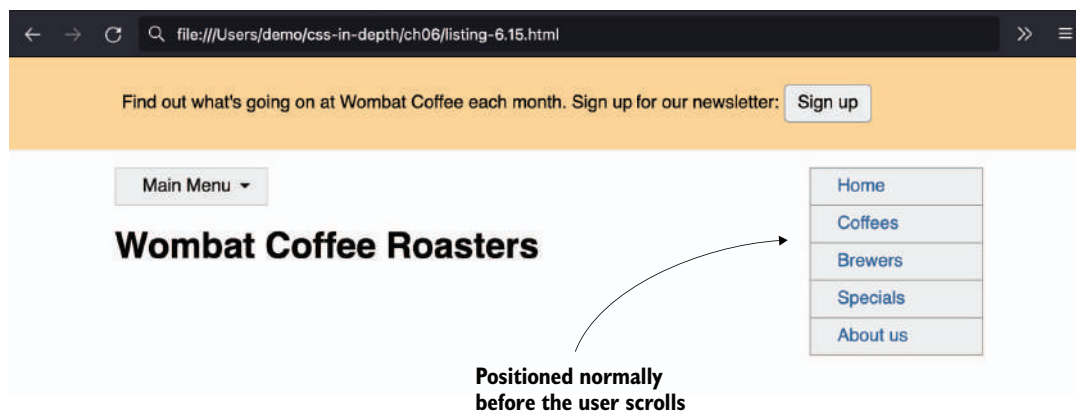


Figure 6.14 The sticky-positioned sidebar is initially positioned normally.

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, 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, the page will look like figure 6.15.

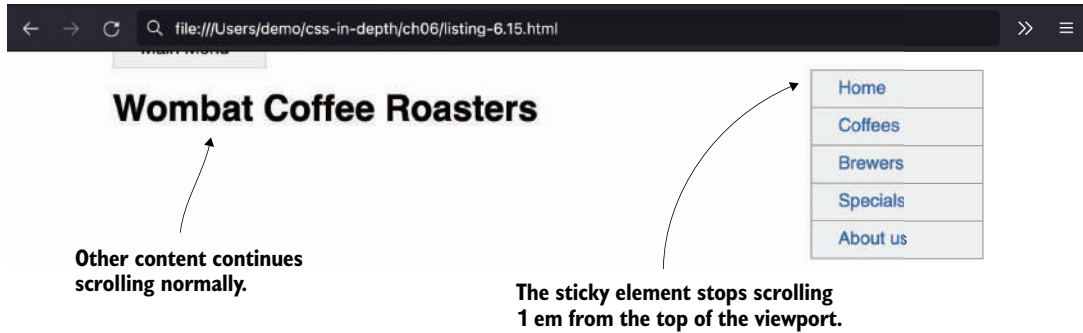


Figure 6.15 The sidebar remains “stuck” to the viewport.

To illustrate this, 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 drop-down menu and page heading) into a left column and adds a right column with an “affix” menu.

Listing 6.14 Changing to a two-column layout with a sidebar

```
<main class="container">
  <div 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>
  </div>

  <aside class="col-sidebar">
    <div class="affix">
      <ul class="submenu">
```

← Wraps the existing content in a col-main for the main column

Adds a second column with an affix element inside it

```

        <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>
</main>

```

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 the styles in the following listing to your stylesheet.

Listing 6.15 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 takes only 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 for it to stick into place, which is why I artificially increased its height by adding a `min-height` to its flex container. If the parent element were to scroll all the way out of the viewport, the sticky element would resume scrolling again when the bottom of its parent reaches it.

Summary

- Fixed positioning places elements relative to the viewport.
- Relative positioning places elements relative to their initial static position. Other elements on the page will continue to flow around their initial positions.
- Absolute positioning places elements relative to their nearest positioned ancestor element.
- Sticky positioning allows elements to stay on screen as the rest of the page scrolls. They will stay contained within their parent element.
- `z-index` only works on positioned elements, and using it creates a new stacking context.



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 a mobile-first approach when styling a page
- Using 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 very large 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 it at. 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, <https://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.

There are three key principles of responsive design:

- *A mobile-first approach to design*—This means you have a plan for the mobile version before you begin constructing the desktop layout.
- *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.
- *The use of fluid layouts*—This approach allows containers to scale to different sizes based on the width of the viewport.

I touched on some of these ideas in chapter 2, but this chapter takes a deeper look at these three principles. You'll start by building a responsive page, and I'll unpack each of those as we go. After that, we'll also take a look at images, which require special considerations on responsive sites.

7.1 **Mobile first**

The first principle of responsive design is *mobile first*. As mentioned, this means planning your mobile layout before you jump in and build the desktop layout. 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 fully interactive website and 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 7.1 shows the page you’re going to build. You guessed it—that’s the mobile design.

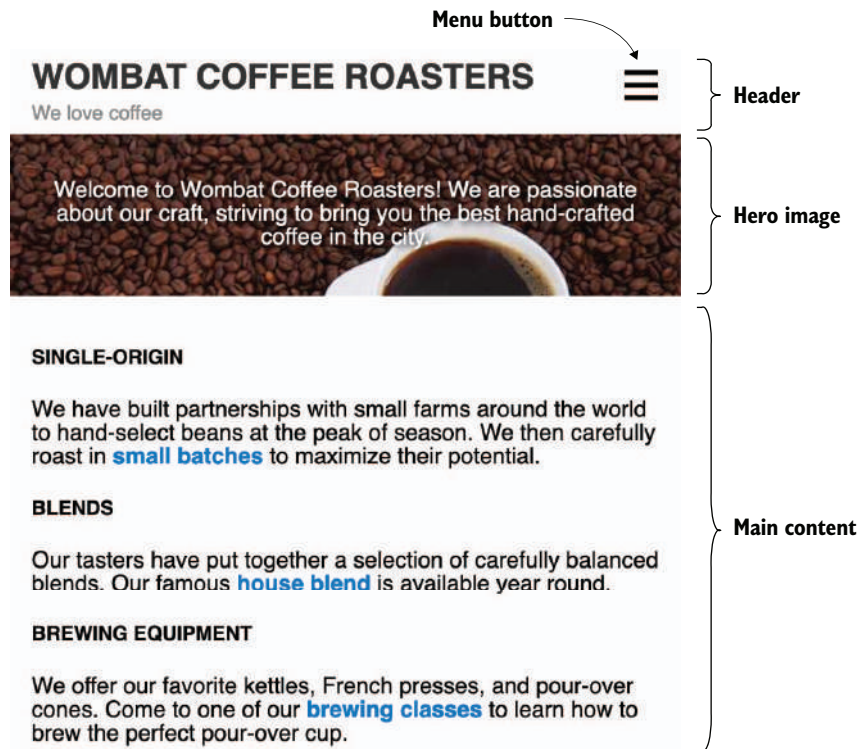


Figure 7.1 Mobile page design

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 7.2). 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



Figure 7.2 Mobile page with the menu opened by clicking or tapping the hamburger icon

want to quickly find store hours or another specific piece of information, like a price or an 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, 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. However, it's a striking part of the design, so I think in this case 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.

NOTE 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, initially writing the CSS needed only for smaller screens. Before diving into that, however, it's helpful to keep in mind your overall design as you will want it on larger screens. This can help guide some of your decisions regarding the structure of the code.

After you put the mobile styles in place, you'll add a medium and large *breakpoint* to the page. You'll do this by using media queries to layer on additional styles that will only apply on larger screen sizes.

DEFINITION A *breakpoint* is a particular point—a browser width or height—at which the page styles change to provide the best possible layout for the screen size.

I'll look at the specifics involved in adding these breakpoints a bit later in the chapter. At this stage, it's helpful to know that those will be added soon and to consider the general changes needed to the page's layout for larger screen sizes. Figure 7.3 shows the page design on a medium layout.

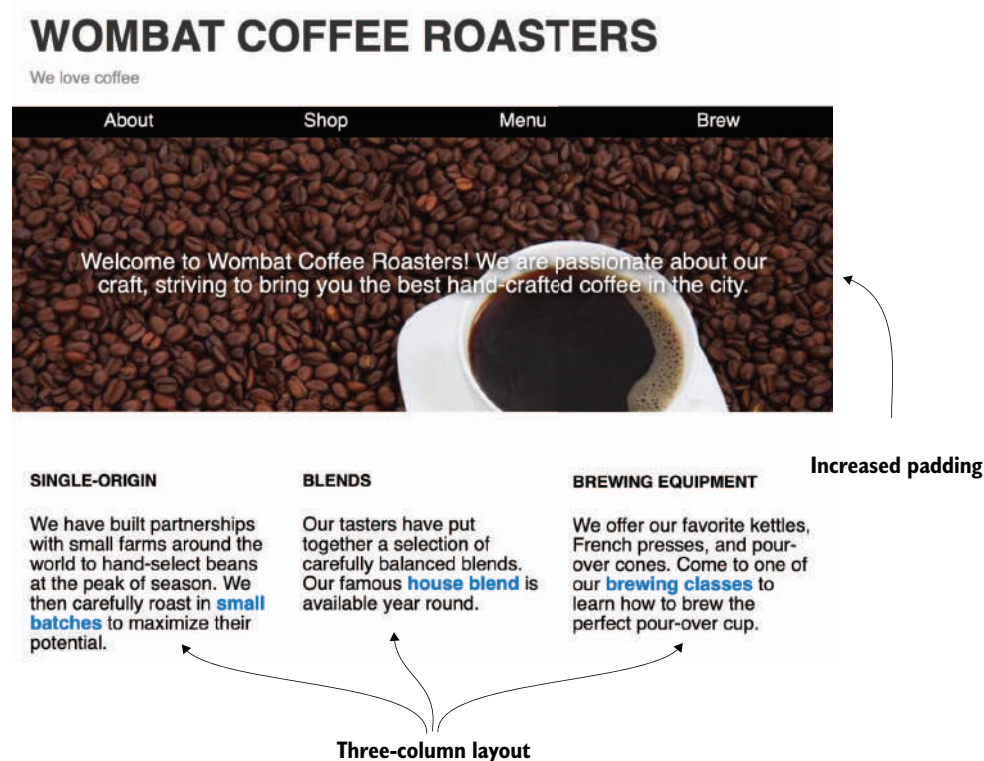


Figure 7.3 Your page on a medium-sized viewport

At this viewport size, you have 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 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 7.4.

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. Add listing 7.1 to the HTML.

The markup looks much the same as that for a nonresponsive design. I've made a couple of considerations for the mobile design, which I'll highlight momentarily.

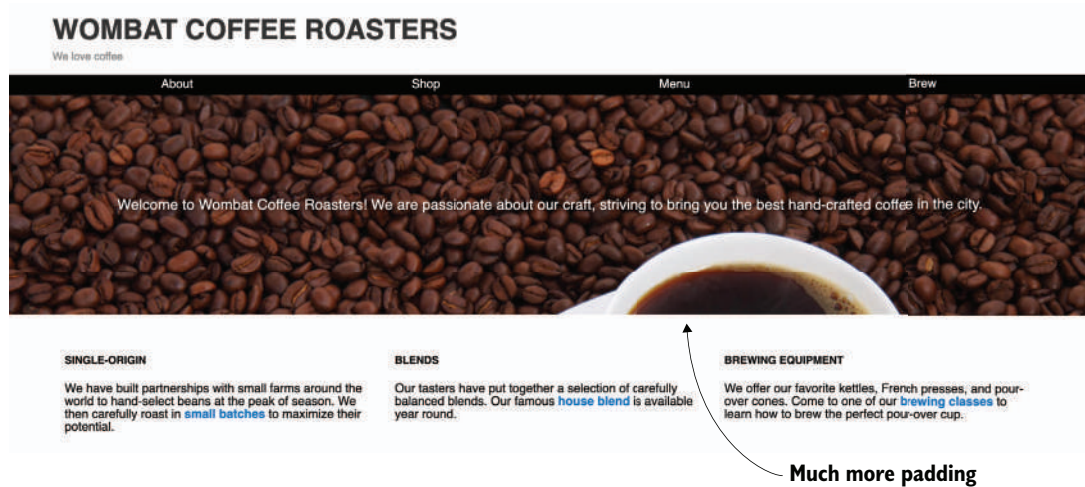


Figure 7.4 Page on a large viewport

Listing 7.1 Page markup for a responsive design

```

<!doctype html>
<html lang="en-US">
<head>
  <meta charset="UTF-8">
  <title>Wombat Coffee Roasters</title>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
<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>

```

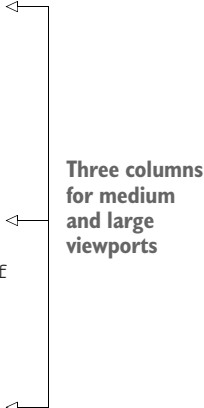
Defines the hamburger button for a mobile menu

Main menu that will be hidden by default on mobile devices

```

<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 class="main">
  <section class="column">
    <h2 class="subtitle">Single-origins</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">
    <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>
</main>
</body>
</html>

```



Three columns
for medium
and large
viewports

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. The `main` and `column` classes are in place to allow for the desktop design. (When building new pages, 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 7.5. 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.

These styles are given in listing 7.2. 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.



Figure 7.5 First set of styles applied

Listing 7.2 Adding initial styles for the page

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

:root {
  font-size: clamp(0.9rem, 0.5svw + 0.6em, 1.125rem);
}

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;
}
```

Base font size
scales slightly
with the viewport
(see chapter 2)

```

.page-header {
  padding: 0.4em 1em;
  background-color: #fff;
}

.title > h1 {
  color: #333;
  text-transform: uppercase;
  font-size: 1.5rem;
  margin-block: 0.2em;
}

.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;
}

main {
  padding: 1em;
}

.subtitle {
  margin-block: 1.5em;
  font-size: 0.875rem;
  text-transform: uppercase;
}

```

Page header and title

Adds the hero image to the page

A dark text shadow helps light text remain readable in front of a complex background.

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`) is the blur radius, indicating how much to blur the shadow. Finally, `#000` defines the color of the shadow.

7.1.1 Creating a mobile menu

At this point, you're left with the most complicated part of the page: the menu. When you're done, it'll look like figure 7.6.



Figure 7.6 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 6.9). Initially, you'll hide the `menu-dropdown`; then, you'll add some 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 downside to a hamburger menu

Hamburger menus have become a popular approach to mobile design. 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 navigation 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 your HTML (listing 7.1), 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 the following listing to your stylesheet to style the menu.

Listing 7.3 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;
}

.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;
}
```

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

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 will be added via JavaScript when the menu is open; the styles here will cause the menu to become visible while the class is present. Figure 7.7 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, though they are difficult to see at this stage).



Figure 7.7 Hamburger button working

The JavaScript in the next listing adds and removes the `is-open` class when the toggle button is pressed. Add it to your page inside the `<head>` tag.

Listing 7.4 JavaScript for dropdown functionality

```
<script type="module">
  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)

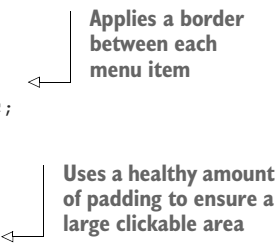
Toggles `is-open`
class on the menu

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 needs to change only one class name.

Now that the dropdown works, the `nav-menu` needs some styling. Add the following listing to your stylesheet.

Listing 7.5 Styling the navigation 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;  
}  
  
.nav-menu > li > a {  
  display: block;  
  padding: 0.8em 1em;  
  color: #fff;  
  font-weight: normal;  
}
```



Applies a border between each menu item

Uses a healthy amount of padding to ensure a large clickable area

Again, this is nothing new. Because the menu is a list (`<ul>`), 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 to tap precisely on a tiny button or link.

7.1.2 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 listing 7.6. This indicates to the browser that you have intentionally considered the responsive behavior of your site.

Listing 7.6 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" rel="stylesheet" type="text/css" />
</head>

```

**Viewport
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 all 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://mng.bz/ppa5> (Chrome) or <https://mng.bz/OZnK> (Firefox).

Other options are available for the viewport meta tag, 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 option that developers sometimes 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://mng.bz/Y7po>.

7.2 *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 the following:

```

@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 larger font size of 2.25 rem will only be applied to the title's `h1` 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.

Using px vs. em in media queries

You can also use many units to define your media queries, though px and ems are the most common. Instead of `@media (min-width: 560px)`, for instance, you could use `@media (min-width: 35em)`. Rem generally behaves the same as em.

The value of an em in a media query is based on the browser's default font size. This is the keyword `medium`, which typically equals 16 px. However, if a user has changed their default font size in the browser settings, this may change the meaning of em. This can be a beneficial behavior in your design, increasing the size of your page's breakpoints in accordance with an increased default font size. Just be aware this behavior is not consistent across all browsers (Safari is the notable exception).

In the past, developers, including myself, have recommended the use of ems in media queries due to browser inconsistencies. However, in recent versions of browsers, px has been more consistent. For that reason, I'm comfortable using px in many cases and switching to ems when I prefer the breakpoints to scale with an increased default font size.

I'll walk you through adding some media queries to your page. Find the `.title` styles in your stylesheet and insert the media query in the following listing to add some responsive behavior to the page header.

Listing 7.7 Adding a breakpoint to the page title styles

```
.title > h1 {  
  color: #333;  
  text-transform: uppercase;  
  font-size: 1.5rem;  
  margin-block: 0.2em;  
}  
  
@media (min-width: 560px) {  
  .title > h1 {  
    font-size: 2.25rem;  
  }  
}
```

Targets viewports
at 560 px or wider

Overrides the mobile
font size (1.5 rem)
with a larger one

Now the title has two different font sizes, depending on the viewport size. It'll be 1.5 rem for viewports below 560 px 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 font size applied to the page using `clamp()` (listing 7.2). Once it reaches a width of 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 reuse the same few breakpoints in multiple media queries throughout your stylesheet. We'll discuss how to choose these breakpoints later in the chapter.

7.2.1 *Understanding types of media queries*

You can further refine a media query by joining the two clauses with the keyword `and`:

```
@media (min-width: 320px) and (max-width: 560px) { ... }
```

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: 320px), (min-width: 560px) { ... }
```

This example targets both viewports 320 px and narrower and those 560 px and wider. The keyword `or` was added in the W3C Media Queries Level 4 specification that is equivalent to this comma; this is relatively new, but it works in recent versions of Chrome, Firefox, and Safari.

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. The following are some examples:

- `(min-height: 320px)`—Targets viewports 320 px and taller.
- `(max-height: 320px)`—Targets viewports 320 px 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.
- `(pointer: coarse)`—Targets devices that don't have a precise pointing method (such as a mouse). In general, this will target touchscreen devices, where the user's finger cannot accurately target tiny buttons; use this to enlarge these click targets.

- (pointer: fine)—Targets devices that do have a precise pointing method, such as a mouse.

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 useful in cases 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.

One important thing to note is that you cannot access custom properties inside a media query definition. This means something like `@media (min-width: var(--breakpoint))` is invalid because `var()` can only be used when assigning a value in a property declaration.

RANGE SYNTAX IN MEDIA QUERIES

There is a new syntax for media queries using less-than and greater-than signs. For example, instead of `@media (min-width: 800px)`, you can use `@media (width >= 800px)`. You can also provide an upper and lower bound for the query, so `@media (800px < width < 1200px)` will apply only if the viewport width is between 800 px and 1200 px.

This syntax offers two benefits over the more classic approach. First, it is a little more intuitive to read, especially when defining both upper and lower bounds; `(800px <= width <= 1200px)` is a little easier to make sense of than the more verbose `(min-width: 800px)` and `(max-width: 1200px)`.

The second benefit comes from the option to use less-than-or-equal-to (`<=`) and greater-than-or-equal-to (`>=`) signs. With classic media queries, imagine a use case where you have one `max-width: 30em` media query and another `min-width: 30em` query. If the viewport happens to be precisely 30 em wide, both media queries will apply, which might produce unwanted results. With the range syntax, you have the option to make the query inclusive or exclusive of the specified size by including or excluding the equal sign. The following are a few more examples of this syntax:

```
@media (480px <= width < 1200px) { ... }
@media (800px < height) { ... }
@media (width = 1600px) { ... }
```

The latest versions of all major browsers support this syntax, though it is still relatively new. Check <https://caniuse.com/css-media-range-syntax> for the latest browser statistics.

LIGHT AND DARK THEMES

You can also use media queries to detect when the user has their operating system set to dark mode. You can use this to give your page either a light or dark background to match the user's preference:

```
@media (prefers-color-scheme: dark) {  
  :root {  
    --theme-font-color: white;  
    --theme-background: #222;  
  }  
}
```

Or you can target a light theme:

```
@media (prefers-color-scheme: light) {  
  :root {  
    --theme-font-color: #222;  
    --theme-background: #eee;  
  }  
}
```

I find setting custom properties similar to these examples a relatively straightforward way to change the entirety of the page. Just be sure to consistently use the custom properties throughout your stylesheet.

In a real-world example, you will certainly need more than these two variables to accomplish this. We'll take a deeper look at managing color variables in a later chapter.

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 remove background images (to save on ink) and hide 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 the screen only, use `@media screen`.

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 nonessential parts of the page, such as navigation 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.

7.2.2 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 the next listing. (You don't need to add this to your page yet.)

Listing 7.8 General structure of responsive CSS

```
.title {  
  ...  
}                                     ← Mobile styles; applied  
                                     to all breakpoints  
  
@media (min-width: 35em) {           ← Medium breakpoint;  
  .title {                           overrides select  
    ...                               mobile styles  
  }  
}  
  
@media (min-width: 50em) {           ← Large breakpoint; overrides  
  .title {                           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 sufficient 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 a mobile-first approach. They should be an exception and not the rule.

Next, I'll walk you through adding 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. You'll do this in stages. In listing 7.9, 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 navigation menu, so you'll also hide the hamburger button and reveal the menu at all times (listing 7.10). Finally, you can shift the main content to a three-column layout (listing 7.11). Afterward, the page will look like figure 7.8.

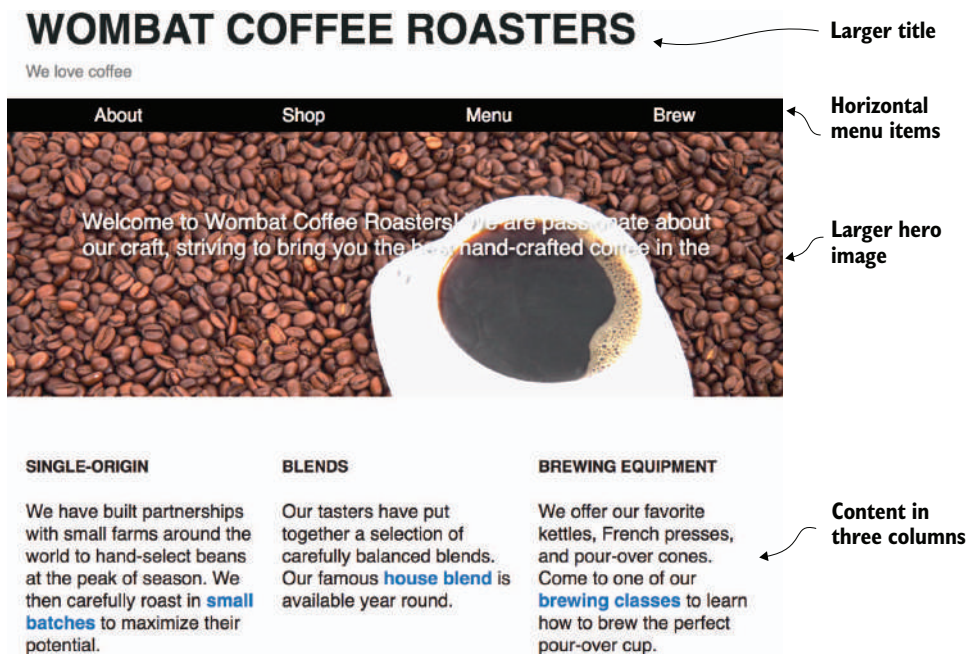


Figure 7.8 Page at medium breakpoint

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 7.9. Add these to your stylesheet.

Listing 7.9 Padding and font adjustments for the medium breakpoint

```

.page-header {
  padding: 0.4em 1em;
  background-color: #fff;
}

@media (min-width: 560px) {
  .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: 560px) {
  .hero {
    padding: 5em 3em;
    font-size: 1.2rem;
  }
}

main {
  padding: 1em;
}

@media (min-width: 560px) {
  main {
    padding: 2em 1em;
  }
}

```

← | Increases header padding

← | Increases hero image padding and font size

← | 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 when the viewport reaches 560 px wide.

TIP Place your media queries after any styles they need to override. This way, the styles in the media query can override the other styles with the same selector specificity.

Next, the menu 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. Add the media query blocks from listing 7.10 to your stylesheet following the styles you already have for `.menu` and `.nav-menu`. Again, be sure these media queries appear after the mobile styles for these elements, so they can override values as needed.

Listing 7.10 Restructuring the nav menu for a medium breakpoint

```

@media (min-width: 560px) {
  .menu-toggle {
    display: none;
  }

  .menu-dropdown {
    display: block;
    position: static;
  }
}

@media (min-width: 560px) {
  .nav-menu {
    display: flex;
    border: 0;
    padding-inline: 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 has 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-level element. You won't need to override the `top`, `left`, and `right` properties from the mobile styles, as those now have no effect on 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.

7.2.3 Adding responsive columns

The final change to make for the medium breakpoint is the introduction of multiple columns. This is done exactly like multicolumn layouts you've built in previous chapters. This time, you 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 `main` class where you anticipated a three-column layout. Add the following listing to your stylesheet to define those styles now.

Listing 7.11 Three-column layout inside a media query

```
@media (min-width: 650px) {  
  .main {  
    display: grid;  
    grid-template-columns: 1fr 1fr 1fr;      | Defines a three-  
    gap: 1.5em;                               | column grid  
    max-inline-size: 1400px;                 | Limits the width of the grid  
    margin-inline: auto;                     | and centers it horizontally  
  }  
}
```

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 grid container with grid items forming the columns.

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, images beside text, or any other items that feel cramped on a narrow screen.

Notice I used a slightly wider breakpoint for listing 7.11. I chose 650 px here because this is the point where the three columns started to feel overcrowded. In this case, below 650 px, the columns were too narrow. You don't need to feel locked into the same breakpoints everywhere.

Web designer Brad Frost has compiled a list of responsive page layouts 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.

MINIMIZING MEDIA QUERIES

A stylesheet with a lot of large media query blocks can become complex. This is especially true if you end up with a combination of `min-width` and `max-width` queries targeting the same portions of the page. With multiple rules at various screen sizes, it can be difficult to keep it all in your working memory at once.

When possible, I try to keep media query blocks short. One way to do this is to use custom properties outside of a media query and change them as needed inside a media query. For example, consider using a custom property to define a common gap size that you enlarge for bigger breakpoints:

```

:root {
  --gap-size: 1rem;
}

@media (min-width: 560px) {
  :root {
    --gap-size: 1.5rem;
  }
}

```

You can use this custom property throughout the page, and this one simple media query will increase its size accordingly anywhere it's used. Another way to keep media query blocks short is to use a grid layout, applied outside a media query. Then you can redefine the grid columns or rows inside a media query based on the screen size.

In many cases, you might not need media queries at all, 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.

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 14 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.

Media queries have been a staple of responsive design for years, but there is another related tool called a *container query*. This allows you to change the styles of an element based on the size of its container element, rather than the page as a whole. We'll take a look at container queries in chapter 10.

7.3 *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 `inline-size: 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 fluid terms 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.

7.3.1 Adding styles for a large viewport

Next, you'll 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 a grid to allow the columns to achieve columns, each one-third of the viewport width.

The final, large viewport layout of your page is shown in figure 7.9. 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.

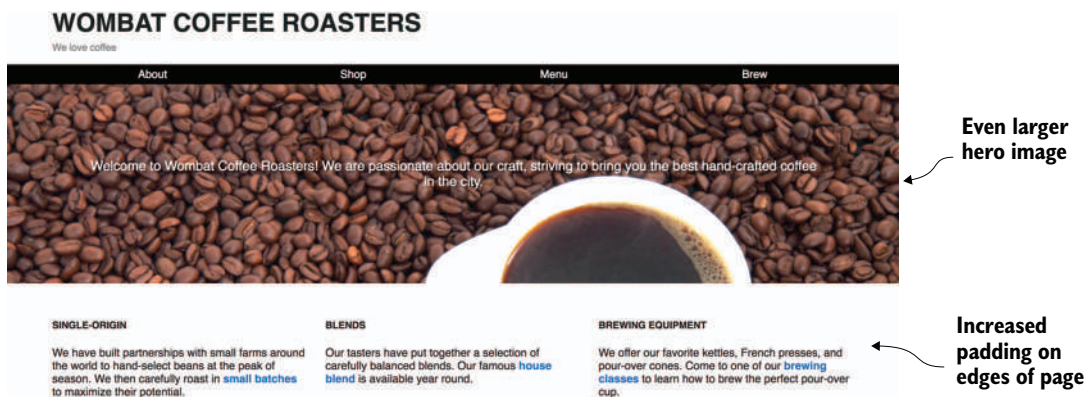


Figure 7.9 Page layout on a large viewport

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 7.12.

Add all the (min-width: 800px) 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 7.12 Increasing paddings for large breakpoints

```
@media (min-width: 800px) {  
  .page-header {  
    padding: 1em 4em;  
  }  
}  
  
@media (min-width: 800px) {  
  .hero {  
    padding: 7em 6em;  
  }  
}  
  
@media (min-width: 800px) {  
  main {  
    padding: 2em 4em;  
  }  
}  
  
@media (min-width: 800px) {  
  .nav-menu {  
    padding-inline: 4em;  
  }  
}
```

Increases left and right padding on edges of page to 4 em

Increases hero padding on all sides for a larger hero image

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.

7.3.2 Dealing with tables

Tables are particularly problematic for fluid layouts on mobile devices. If a table has more than a handful of columns, it can easily overflow the screen width (figure 7.10).

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 and 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 7.11 shows an example of this.

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 7.10 Right edge of a table clipped on a mobile device

Nicaragua
Matagulpa
Dark chocolate, almond
\$13.95
Ethiopia
Yirgacheffe
Sweet tea, blueberry
\$15.95
Ethiopia
Nano Challa
Tangerine, jasmine
\$14.95

Figure 7.11 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 following listing. (You can apply this to any `<table>` markup to see the results.)

Listing 7.13 Forcing a responsive table layout on a mobile device

```
table {
  inline-size: 100%;
}

@media (max-width: 480px) {
  table, thead, tbody, tr, th, td {
    display: block;
  }
```

Makes all table elements block display

```
thead tr {
  position: absolute;
  top: -9999px;
  left: -9999px;
}

tr {
  margin-block-end: 1em;
}
```

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.

7.4 *Responsive images*

In responsive design, images need special attention. Not only do you need to fit them on the screen, but 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.

7.4.1 *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 smaller size, there's no sense in sending an extremely large image; the browser will have to downscale it to make it fit.

Use responsive techniques to serve an appropriate image to a user based on their screen size. For the hero image on your page, this looks like the CSS shown in listing 7.14. Add this to your stylesheet.

Listing 7.14 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;
}
```

← Uses the smallest image on mobile devices


```

@media (min-width: 560px) {
  .hero {
    padding: 5em 3em;
    font-size: 1.2rem;
    background-image: url(coffee-beans-medium.jpg);
  }
}

@media (min-width: 800px) {
  .hero {
    padding: 7em 6em;
    background-image: url(coffee-beans.jpg);
  }
}

```

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.

7.4.2 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 `<img>` tag? For inline images, a different approach is necessary: the `srcset` attribute (short for "source set").

This attribute allows you to specify multiple image URLs for one `<img>` tag, indicating the resolution of each. The browser will then figure out which image it needs and download that one.

Listing 7.15 Responsive srcset image

Supplies a normal `src` for browsers that don't support `srcset` (for example, IE and Opera Mini)

```



```

URL of each image and its width

All modern browsers support `srcset`, but older browsers 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 2 x pixel density, it can download a higher-resolution image accordingly.

For a closer look at responsive images, visit <https://mng.bz/GZjM>. 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-inline-size: 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. Understanding these principles will help you to keep responsive needs in mind as you build out pages.

This approach has proven effective for quite some time. However, there is one important tool to add to this list: container queries. This is a recent feature of CSS that developers have been looking forward to for several years. I'll lay some helpful groundwork in the next couple of chapters, then return to the topic of responsive design in chapter 10 with a look at this feature.

Summary

- Approaching a design with a mobile-first mentality makes responsive design easier to implement. Scaling a mobile page up to larger screens is generally easier than scaling a desktop page down to mobile.
- Media queries allow you to apply CSS conditionally based on the viewport size or other aspects of the user's environment. Use this to progressively enhance the page at larger and larger viewports.
- Fluid layouts will always fit the screen at any browser size.
- Responsive images help to accommodate the bandwidth limitations of mobile devices.
- A meta viewport tag tells browsers that your site has been designed responsively. This will ensure your styles apply on smaller devices as you expect.

Part 3

Modern code organization

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, we'll look at some tools that have recently been added to the language to provide more explicit control over the cascade and help with code organization in general. I'll show you how to write your CSS so that it's logically organized and easily maintainable in the future.

Cascade layers and nesting

This chapter covers

- Organizing CSS in a logical manner
- Controlling the cascade using layers
- Using the `:is()` and `:where()` pseudo-classes
- Grouping related styles with nesting

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, flexbox, and grid. You’ll need these skills, particularly when you first set out on a new project. In the world of software development, however, there’s more to programming than just making your code work. It’s also important to organize code in a way that’s understandable and allows for future changes without making it more brittle. In CSS, this surfaces an entirely new set of difficulties.

In the past, the only way developers have been able to address these problems is with strictly prescribed methodologies and naming patterns, but CSS has some brand-new features to help us in this regard. These are first-class features that provide better means of code organization and explicit control over the cascade.

In part 3, we’ll discuss these features, and I’ll teach you how to use them to keep these problems in check. We’ll look at the architecture of CSS. These chapters are a

little different from the rest of the book, focusing less on the declarations in your stylesheet and more on the selectors you choose and manipulating the cascade to prioritize the styles you want. I'll also cover the overall structure of a stylesheet at a high level, so it stays organized and understandable.

While these tools are useful on projects of any size, they primarily exist to address problems that emerge as a website or application grows in size and complexity. In that context, they prove invaluable.

Many of the features in these chapters are new to CSS. Some are ready to use now, but others might be a little further out. I am writing this book with an optimistic eye toward the near future, and I expect these all to be the norm before long. As I cover each of these features, I'll point out the current browser support, but be sure to check for more current statistics when you read this. Two good resources for this are <https://developer.mozilla.org/> and <https://caniuse.com/>.

8.1 Manipulating the cascade with layers

When it comes to the cascade, the browser simply applies a series of rules to our styles to ensure one rule predictably overrides another if they both happen to target the same element. Selector specificity and source order are a predictable way to determine this, but in truth they don't always accurately convey your intent as to the priority of a CSS rule. It's rather common to find places in your stylesheet where you would prefer a lower-specificity rule take precedence. For example, most stylesheets apply basic link styles with CSS along these lines:

```
a:any-link {  
  color: var(--brand-blue);  
  font-weight: bold;  
}
```

This selector, with a tag and a pseudo-class, has a specificity of 0,1,1. Elsewhere on the page, there may be a link that's made to look like a large call-to-action button:

```
<a class="button" href="/about">Read more</a>
```

The problem arises when you try to style that button with a simple `.button` selector; this has a specificity of 0,1,0, which is lower than `a:any-link` (0,1,1). So any styles within cannot override the color or font weight that's applied to all links on the page. The classic approach to fixing this is to tweak the specificity of one or both selectors to achieve the desired result.

8.1.1 Defining layers

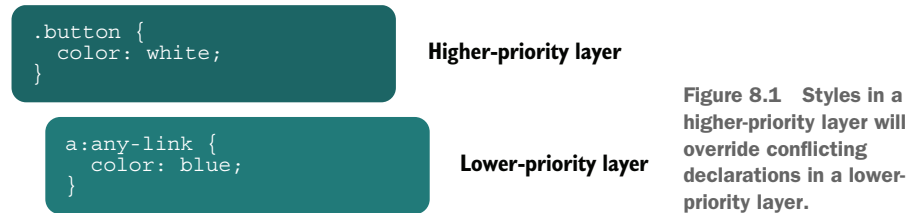
A better way to manage this with modern CSS is by using *cascade layers*. Cascade layers are a new criterion added to the cascade, which allows you to partition your styles into a series of groups called *layers*. You can specify the priority order of these

layers, so all the styles in layer A will always override the styles in layer B, regardless of selector specificity.

Cascade layers started shipping in all major browsers in early 2022, so it has fairly broad support. However, you should check statistics at <https://caniuse.com/css-cascade-layers> for the latest support statistics. A small percentage of users don't update browsers right away, and it is unfortunately difficult to apply graceful degradation to layers—all rules inside a layer will be ignored by a browser that does not support the feature.

Depending on when you are reading this and your target audience, that may or may not be acceptable. Be sure to check the latest statistics when deciding whether to use layers on your site. If you want to use layers but are concerned about browser support, see <https://mng.bz/z89w> for details on a tool that can be used to mimic layers in your stylesheet.

Figure 8.1 illustrates styles from two different layers. Any styles in a higher-priority layer will override those from a lower-priority layer, regardless of selector specificity. In this example, declarations with the `.button` selector on a higher layer will override those with an `a:any-link` selector on a lower one.



Layers are defined using the `@layer` at-rule. Styles on a layer are defined inside the curly braces of the corresponding `@layer` rule. An example of these is shown in listing 8.1, which defines a global layer followed by a theme layer. Layers defined later in the stylesheet take precedence over those defined earlier, so anywhere these styles conflict on the page, declarations in the theme layer will take precedence over the global layer, regardless of selector specificity.

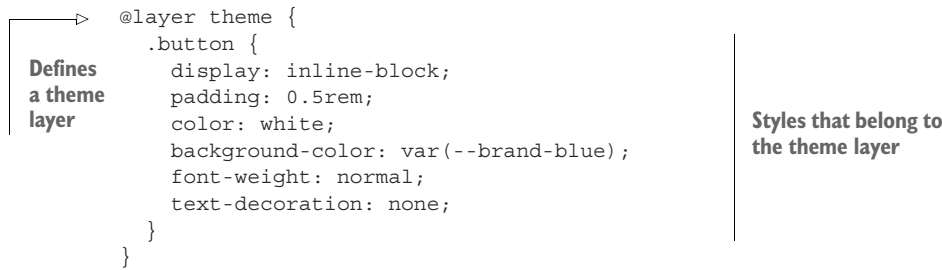
Listing 8.1 Cascade layers syntax

```
@layer global {
  :root {
    --brand-blue: #0063cc;
  }

  a:any-link {
    color: var(--brand-blue);
    font-weight: bold;
  }
}
```

← Defines a global layer

Styles that belong to the global layer



In this case, `<a class="button">` would have a normal font weight and white text on a blue background.

In terms of the cascade, layers are considered after origin and inline styles but before specificity. Within the same layer, all the normal rules of specificity still apply. Figure 8.2 shows the cascade flow chart from chapter 1. This illustrates how two conflicting declarations on the same element are resolved.

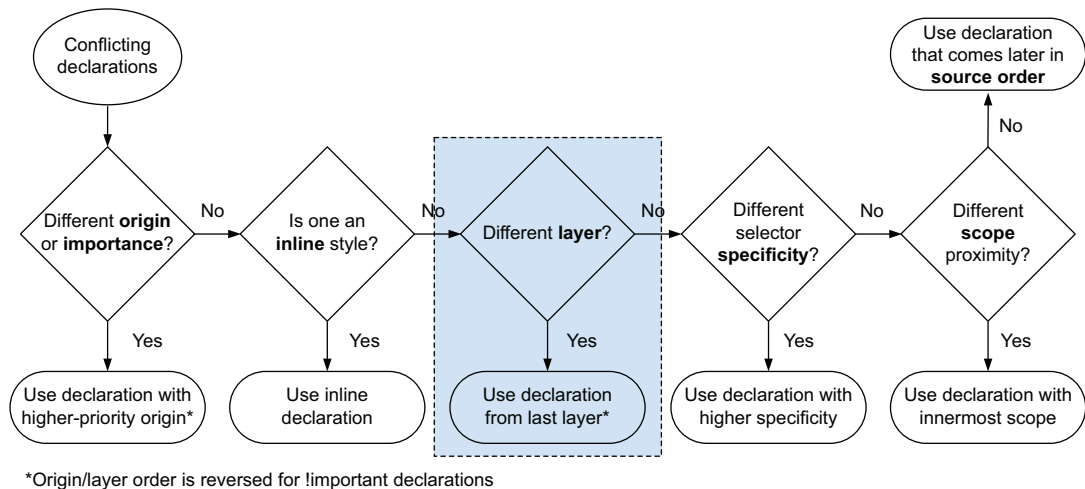


Figure 8.2 Cascade layers are considered before selector specificity in the cascade.

WARNING Styles that are not defined inside a layer will take precedence over styles in a layer; they are effectively treated as belonging to their own highest-priority layer.

The layer names you provide can be any identifier you wish consisting of alphanumeric characters, hyphens, and underscores. They must not begin with a decimal digit. Any characters beyond these must be escaped with a backslash (for example, `@layer theme\$`). This means you can name them and structure them however you need to for your site’s specific needs. Later in the chapter, I’ll show a common pattern that can help organize your layers in a meaningful way.

ANONYMOUS LAYERS

You are not required to name your layers. You can create *anonymous layers* by omitting the layer name. Listing 8.2 shows the same styles as the previous listing but with layer names removed. This produces the same result.

Listing 8.2 Anonymous cascade layers

```
@layer {
  :root {
    --brand-blue: #0063cc;
  }

  a:any-link {
    color: var(--brand-blue);
    font-weight: bold;
  }
}

@layer {
  .button {
    display: inline-block;
    padding: 0.5rem;
    color: white;
    background-color: var(--brand-blue);
    font-weight: normal;
    text-decoration: none;
  }
}
```

← **First anonymous layer**

← **Second anonymous layer**

In this example, the second layer still takes precedence over the first because it is defined later in the stylesheet.

On simple sites, where you need only a few layers, using anonymous layers might be useful, but I generally avoid this approach. Using names allows you to reference the same layer again, so you can add styles to the same layer in different parts of the stylesheet. For this reason, naming your layers generally provides a little more flexibility, as you'll see shortly. I also find it helpful to use names to document the purpose of each layer, which helps keep the code a little more logically organized.

TIP When using the `@import` at rule to import an external stylesheet, you can assign the stylesheet to a layer. For example, `@import "bootstrap.css" layer(bootstrap);` imports `bootstrap.css` to a layer named `bootstrap`. This behaves as if all the contents of the imported stylesheet were wrapped in a `@layer` at-rule. Unfortunately, there is not yet any equivalent when linking to a stylesheet via an HTML `<link>` tag; if you want to import this way directly from HTML, consider putting an `@import` rule inside a `<style>` tag.

8.1.2 Layer order and priority

Layers provide us with a means of indicating the priority of various parts of our stylesheet. A stylesheet typically starts with some general styles for the page as a whole, like font settings and colors, then progresses to styles for specific parts of the page, like the header or menus.

Using layers helps in this process two ways. First, it gives you more explicit control over the priority of your styles in a more flexible way than selector specificity. And second, well-named layers help document the intent of a set of styles, which can help when making changes in the future. Styles in a layer named `reset`, for instance, will likely focus on overriding certain user-agent styles, while a layer named `global` is probably establishing fonts and colors for the whole page.

Cascade layers are prioritized based on the order layers first appear in the stylesheet. If you reference a layer twice, its priority is not changed by the second reference. For example, consider styles structured as in listing 8.3. This will still produce the same behavior as before; the `global` layer is referenced first, so it receives a lower priority than the `theme` layer, including the `global` styles that appear in the second `@layer global` { ... } block.

Listing 8.3 Referencing the same layer twice

```
@layer global {           ← Establishes the
  :root {                 global layer
    --brand-blue: #0063cc;
  }
}

@layer theme {
  .button {
    display: inline-block;
    padding: 0.5rem;
    color: white;
    background-color: var(--brand-blue);
    font-weight: normal;
    text-decoration: none;
  }
}

@layer global {           ← Adds more
  a:any-link {             styles to the
    color: var(--brand-blue);
    font-weight: bold;
  }
}
```

Referencing the same layer name a second time allows you to add more styles to the layer that was already established earlier in the stylesheet.

Generally, it's best to have a plan for which layers you will need and the priority order those layers should have. When you have such a plan, you can declare multiple layers up front, before you go about adding styles to them. This is done by comma

separating each layer name in a single `@layer` at-rule. You do not need to add curly braces when doing this. The following line of CSS declares four layers in increasing order of precedence:

```
@layer reset, global, theme, components;
```

In this example (see figure 8.3), any styles that are added to the global layer will override conflicting declarations that exist in the reset layer. Any styles added to the theme layer will override styles in the reset or global layers. And finally, styles in the components layer will override all three prior layers. This must be added before any styles are added to the layers, as a layer's priority is determined by the first reference to it that appears in the stylesheet.



Figure 8.3 Four layers in priority order

I highly recommend declaring all your layers like this at the very beginning of your stylesheet. It makes their priority order explicit and easy to find. Then, throughout your stylesheet, nest your styles into these predefined layers with successive `@layer` code blocks.

Any styles in your stylesheet that are not in a layer take higher precedence than those that are; they are effectively treated like the highest-priority layer. For this reason, I recommend you place all your styles inside a layer.

TIP The high priority of unlayered styles can be useful for temporary debugging or prototyping. Once you've got the exact rules you need figured out, you can then organize them into the appropriate layers.

USING !IMPORTANT IN LAYERS

In chapter 1, we looked at the three stylesheet origins: user-agent, user, and author. I also showed that for styles marked with `!important`, the priority level of the origins is reversed. While author styles take precedence over user-agent styles, important user-agent styles take precedence over important author styles.

Layers also exhibit this behavior. Consider this line of code again, which defines four layers in increasing priority order:

```
@layer reset, global, theme, components;
```

In this case, styles on the components layer will take precedence over styles on the global layer, but important styles on the global layer will take priority over important styles on the components layer, as illustrated in figure 8.4.

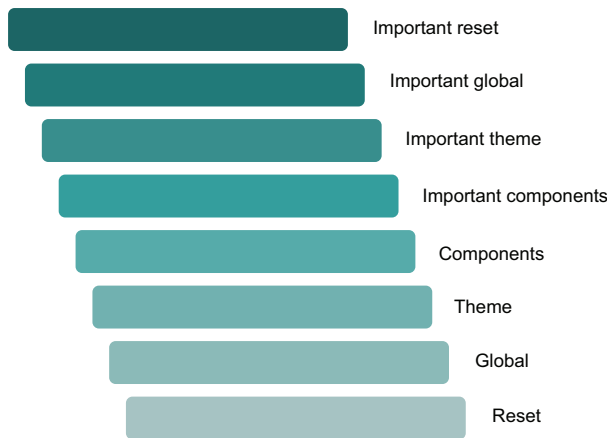


Figure 8.4 Layer priority order is reversed for styles marked `!important`.

You will likely not make heavy use of this feature, but it can be useful in certain cases. Listing 8.4 shows one example. This uses a `prefers-reduced-motion` media query to detect when users have enabled a reduced motion setting in their browser or operating system. These specific declarations will come into play when we take a look at transitions and animations in the final chapters of the book, but essentially, they disable motion for users who have indicated specific sensitivity to it.

Listing 8.4 Using `!important` to reduce motion

```
@layer reset {
  @media (prefers-reduced-motion: reduce) {
    *,
    *::before,
    *::after {
      animation-duration: 0.01ms !important;
      animation-iteration-count: 1 !important;
      transition-duration: 0.01ms !important;
      scroll-behavior: auto !important;
    }
  }
}
```

These declarations are defined on the lowest-priority layer, `reset`, because that's conceptually where they belong. But because they have the `!important` annotation added, they will receive higher precedence than anything on later layers, even if those styles make use of `!important`.

In the past, we have avoided the use of `!important` in CSS because of the difficulties it could cause. But now, with the control we have over cascade layers, it can be a very useful tool in moderation.

NESTED LAYERS

Sometimes you will find you want a more granular way to organize your layers. This is where nested layers come into play.

You can create nested layers by embedding one layer inside the block of another layer. This pattern looks like the following:

```
@layer modules {  
  @layer layout { ... }  
  @layer theme { ... }  
}
```

In this snippet, the modules layer is further broken down into layout and theme layers. The same rules of layer precedence take place within the modules layer, so the theme layer takes precedence over the layout layer. Styles that are within the modules layer but not in a nested layer take precedence over both layout and theme nested layers.

You can also define these nested layers with a dot notation. This code is equivalent to the previous listing:

```
@layer modules.layout { ... }  
@layer modules.theme { ... }
```

This dot notation makes it easier to reference a nested layer again later in the stylesheet. You can continue to nest layers multiple levels deep, if needed, either by further nesting `@layer` rules or with a longer dot notation such as `@layer modules.layout.base`.

One benefit of nested layers is that you can import a third-party stylesheet into a layer without negating any layers that may be defined in that stylesheet. Any layers defined in that stylesheet will become nested within the layer you specify in your `@import` statement.

8.1.3 The *revert-layer* keyword

Early in the book, we looked at four important keywords that can be declared for any property: `initial`, `inherit`, `unset`, and `revert`. Cascade layers come with one more of these values: `revert-layer`.

As I covered in chapter 1, the `revert` keyword allows you to remove any values applied to a property in the author styles, restoring them to the default user-agent styles. Similarly, the `revert-layer` keyword allows you to remove any values applied in the current layer, restoring them to values specified in previous layers.

This is useful when you've defined a set of global styles that you want applied to the entire page and have then overridden some of those styles on a higher-priority layer. The `revert-layer` keyword allows you to revert to those global styles if needed.

To see this in action, add listing 8.5 to a stylesheet and link it to an HTML page. Like before, this code defines some link styles on the global layer, and then on the theme layer, it defines some special button-like styles. Finally, it uses `revert-layer` to revert links inside a `.blog-content` block back to the global styles.

Listing 8.5 Using `revert-layer` to restore global link styles

```

@layer global, theme;

@layer global {
  :root {
    --brand-blue: #0063cc;
  }

  a:any-link {
    color: var(--brand-blue);
    font-weight: bold;
  }
}

@layer theme {
  .button {
    display: inline-block;
    padding: 0.5rem;
    color: white;
    background-color: var(--brand-blue);
    font-weight: normal;
    text-decoration: none;
  }

  .blog-content a:any-link {
    display: revert-layer;
    padding: revert-layer;
    color: revert-layer;
    background-color: revert-layer;
    font-weight: revert-layer;
    text-decoration: revert-layer;
  }
}

```

Reverts all links inside
.blog-content to the
global styles

With this code, all links inside a `<div class="blog-content">` container will appear with the defined global styles, regardless of whether the `button` class is present.

THE ALL PROPERTY

Reverting multiple properties like this can be tedious. In these cases, there is a useful shorthand property, `all`. You can use this to revert all properties at once:

```

@layer theme {
  .blog-content a:any-link {
    all: revert-layer;
  }
}

```

The `all` property is not specific to layers, but this is a useful place for it. This property can only be set to the five special keyword values `initial`, `inherit`, `unset`, `revert`, and `revert-layer`.

8.2 A recommended organization for cascade layers

Cascade layers provide a huge degree of flexibility. The downside to this is it can make it difficult to decide how exactly you want to organize and prioritize your code.

Layers are a new feature, so there is still a lot of room for experimentation, but you will find some common themes in use across the industry. The following is a list of layers I like to use:

```
@layer reset, theme, global, layout, modules, utilities;
```

Other developers have recommended very similar patterns as well; I've based this list on a recommendation from frontend engineer Stephanie Eckles. Figure 8.5 illustrates how these layers stack on one another, with the later layers taking priority over the earlier ones.

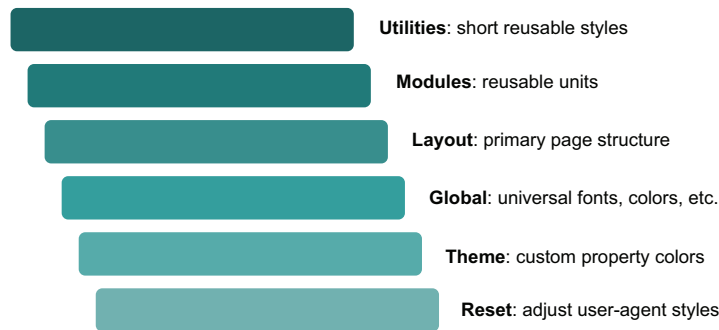


Figure 8.5 A useful template for layer organization

Let's take a look at these layers one at a time to see how they can help organize your code. I'll start with the lowest-priority layer, reset, and work up from there. Not every project will require every layer, so you will sometimes want to omit some of these depending on your needs.

If you want to follow along, start a new stylesheet and add the styles as we go. I won't be walking you through building a specific page in this chapter, as my focus is more on the way the code itself is organized. I still encourage you to put the styles all together in one place to see how they fit together and to experiment with the examples in the browser.

8.2.1 Reset layer

In the early days of CSS, some user-agent styles differed quite a bit between different browsers, so it became necessary to include a *reset* stylesheet on the page to establish predictable styles for links, lists, and headings. Several of these were available for use online; one of the most popular of these was called *normalize.css*.

Browsers are much more consistent now, so this is not necessary for fixing inconsistencies, but it can still be useful to reset some properties to more desirable defaults.

This includes things like setting border box sizing (chapter 3), adjusting button and input fonts, and removing the small margin that’s on the `<body>` element by default.

In your example stylesheet, add the following listing. This establishes the priority order for all our layers and sets a number of common reset styles.

Listing 8.6 Resetting layer styles

```
@layer reset, theme, base, layout, modules, utilities;

@layer reset {
  *,
  *::before,
  *::after {
    box-sizing: border-box;
  }

  body {
    margin: unset;
  }

  button,
  input,
  textarea,
  select {
    font: inherit;
  }

  img,
  picture,
  svg,
  canvas {
    display: block;
    max-inline-size: 100%;
    height: auto;
  }

  @media (prefers-reduced-motion: reduce) {
    *,
    *::before,
    *::after {
      animation-duration: 0.01ms !important;
      animation-iteration-count: 1 !important;
      transition-duration: 0.01ms !important;
      scroll-behavior: auto !important;
    }
  }
}
```

Establishes the priority order for all layers

Fixes box sizing model

Removes body margin

Fixes font behavior for form elements

Makes sizing of images and similar elements more predictable

These are styles I recommend adding to all your projects. I’ve covered most of these earlier in the book. This should serve as a helpful reference, with my recommendations all in one place. Feel free to add other fixes as you see fit.

The reset layer is generally a layer without too much in it. Think of it as “fixing” the user-agent styles to the behavior you like. Any site-specific design decisions will come

in later layers. You should be able to copy this layer almost entirely across various projects, making only minor adjustments if needed.

8.2.2 Theme layer

The next thing you'll want in your stylesheet is to define custom properties, particularly the colors you'll be using. Conceptually, it makes sense to have these early in the stylesheet, since you'll reference them throughout the rest of your code. It also makes sense that the priority level is low so you can override them easily if needed in specific parts of the page.

Beyond custom properties, two other properties should be considered for inclusion on this layer. The first is `accent-color`. This property customizes the accent color for some elements, such as checkboxes and radio buttons that are selected, range inputs (`<input type="range">`), and progress bars (`<progress>`). Setting this to match other colors on your site provides a nice touch.

The second property is `color-scheme`. This can be set to either `light` or `dark`, to indicate whether the site is a light mode or a dark mode site. The browser can use this to determine reasonable default colors for form inputs and scrollbars (the specifics of how this occurs may vary across browsers). Add the following listing to your stylesheet to set some theme properties for the page.

Listing 8.7 Theme layer styles

```
@layer theme {
  :root {
    --brand-color: #0063cc;
    --background-color-1: #edf3fa;
    --background-color-2: #c6cdd5;
    --foreground-color-1: #39434d;
    --foreground-color-2: #5a636d;
    --font-main: "Helvetica Neue", "Nimbus Sans", Arial, sans-serif;
    --font-heading: Georgia, sans-serif;

    accent-color: var(--brand-color);
    color-scheme: light;
  }
}
```

This is just a small sampling to give an idea of how this layer can be used. In most real-world sites, you will need far more colors. On my personal blog, for instance, I have over 50 custom properties defined at this point in my stylesheet, and applications I've worked on comprise hundreds. I'll go over some practical advice for naming and managing colors in a later chapter; this is where most of it will take place in a stylesheet.

Not every custom property needs to reside on this layer. This is a home for site-wide custom properties, but specific parts of the page might need other custom properties that aren't globally used across the page. In those cases, it often makes more sense to put those custom properties along with the code that uses them. For example,

if you have a sidebar that uses its own `--image-radius` property, you can define that along with the rest of the styles for the sidebar; this will likely be on a higher-priority layer, such as the layout or modules layer.

8.2.3 Global layer

The third layer is the global layer (sometimes called the *base* layer). This is where you can set all the default values you want applied throughout your website. This includes things like the default font styles, background colors and font colors, headings, and link styles.

Most of the selectors on this layer will be tag selectors, because you'll be using them to globally apply default styles to various HTML elements. The following listing shows some global styles. Add this to your stylesheet.

Listing 8.8 Global layer styles

```
@layer global {
  :root {
    font-size: clamp(1rem, 0.4rem + 0.8svw, 1.2rem);
  }

  body {
    font-family: var(--font-main);
    background-color: var(--background-color-1);
    color: var(--foreground-color-2);
  }

  a:any-link {
    color: var(--brand-color);
  }

  h1 {
    font-family: var(--font-heading);
    font-size: 2.2rem;
  }

  h2 {
    font-size: 1.125rem;
  }

  @media (min-width: 768px) {
    h1 {
      font-size: 3rem;
    }

    h2 {
      font-size: 2rem;
    }
  }
}
```

The styles on this layer begin to make use of the custom properties you established on the theme layer. These styles also establish some font sizes, including a little bit of

responsive behavior to make fonts larger on a wider viewport. Other items that you might style on this layer include form inputs and labels, code blocks, blockquotes, and a more comprehensive treatment of link states like `:hover` and `:visited`.

8.2.4 Layout layer

It's generally easiest to style a web page from the outside in; put the major sections of the page roughly in place before focusing on fine-tuning the elements within. This is the primary purpose of a layout layer.

This layer is focused on the high-level layout of the page. This includes things like putting the header, footer, and sidebars in place. If you need to place a series of items in a large grid or define columns, the styles for those can go here as well.

On many sites, you'll have a few different page layouts. For instance, the home page might have a different layout than an article, which in turn differs from a search results page. You can define the general page structure for each of these on the layout layer.

Listing 8.9 shows a brief example of some layer styles. Add these to your stylesheet to see them in context. These include some styles for a home page layout and an article page layout.

Listing 8.9 Layout layer styles

```
@layer layout {
  #homepage {
    display: grid;
    grid-template-areas:
      "header header"
      "main sidebar"
      "footer footer";
    grid-template-columns: 1fr 300px;
    gap: 1rem;
  }

  #homepage > header,
  #homepage > footer {
    grid-column: span 2;
  }

  #article > main {
    max-inline-size: 1000px;
    margin-inline: auto;
  }
}
```

You can change which layout is applied to a page by adding the `homepage` ID or `article` ID to the `<body>` or a high-level `<div>` on the page. This way the same stylesheet can be reused across multiple pages.

In the past, we've generally avoided using IDs in selectors like this, because of the high selector specificity. But with cascade layers, this isn't a concern. Any styles we add to the following layers will be able to override these rules if needed because of the

layer priority. If you wish to use a class name instead of an ID, a common convention I like to use is to prefix the class name with an *l*- (the *L* stands for *layout*). For example, the ID selectors in the previous listing could be replaced with `.l-homepage` and `.l-article` class selectors. The *l*- prefix helps me differentiate at a glance between these classes and classes in the modules layer.

It's important to note that an ID must be unique on the page according to the rules of HTML. So if you have layout styles that might need to be repeated on a page, you should use a class name to style them.

8.2.5 Modules layer

The next layer is devoted to reusable units of the page called *modules*. These are also commonly referred to as components, blocks, or objects. These modules include things like dropdown menus, modals, banner images, navigation bars, or information cards. These styles will provide most of the colors, spacing, and typography rules for the various parts of the page. The vast majority of your styles belong on this layer.

As a general rule, each module should have a unique name and should be relatively small. They should be built in such a way that they can be added anywhere on the page. The next listing shows some example styles for a `nav-menu` module and a `card` module. Add this listing to your stylesheet.

Listing 8.10 Modules layer styles

```
@layer modules {
  .nav-menu {
    margin-block: unset;
    padding-inline: unset;
    border: 1px solid #ccc;
    list-style: none;
  }

  .nav-menu > li + li {
    border-top: 1px solid #ccc;
  }

  .nav-menu > li > a {
    display: block;
    padding: 0.8em 1em;
    color: inherit;
    font-weight: normal;
  }

  .nav-menu > li > a:hover {
    color: var(--brand-color);
    background-color: white;
  }

  .card {
    padding: 1rem;
    border-radius: 0.5rem;
  }
```

← Nav menu module

← Card module

```

    background-color: #fff;
  }

  .card > h3 {
    align-self: end;
    margin-block: 0;
    padding-block-end: 0.5rem;
    border-block-end: 1px solid #eee;
  }
}

```

Building a site out of reusable modules like this is a large topic unto itself. We'll take a much closer look at this in the following chapter.

Because this is the layer containing the most code, it can be useful to break these styles down further with nested layers. Layers are still a relatively new feature, so there aren't any specific commonly used strategies for this yet.

8.2.6 Utilities layer

Sometimes you'll need a class to do one simple, very specific thing to an element. This could mean centering text or hiding an element, 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. Listing 8.11 shows some utility classes on the utility layer. Each performs a specific action: center text, hide an element, and add a border radius. Add these to your stylesheet.

Listing 8.11 Utility layer styles

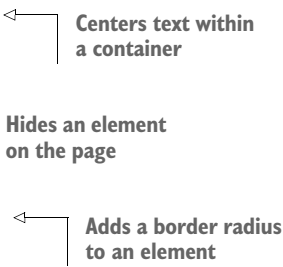
```

@layer utilities {
  .text-center {
    text-align: center;
  }

  .hidden {
    display: none;
  }

  .border-radius {
    border-radius: 1rem;
  }
}

```



Centers text within a container

Hides an element on the page

Adds a border radius to an element

In the past, I've used `!important` for most utility classes. But with layers, I can ensure they are high priority by putting them on a high-priority layer. 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.

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.

8.2.7 *Additional layers*

Some developers like to add a seventh layer called `states`. This is for styles that are dynamically added to and removed from the page depending on the state or status of various modules. This can be things such as loading states or the open and closed states of a dropdown menu.

I personally prefer to keep these on the same layer as modules—or on a sublayer therein. We'll take a closer look at these styles in the following chapter.

Again, this specific set of layers is a recommendation, not a hard rule. As you develop your own patterns, feel free to adapt this approach to your needs. You may develop something you like better, or maybe certain projects will benefit from a different approach entirely.

8.3 *The `:is()` and `:where()` pseudo-classes*

Two relatively new CSS pseudo-class selectors help with code organization, `:is()` and `:where()`. Let's take a look at these, beginning with `:is()`.

This pseudo-class can be used to write large selectors more concisely. To illustrate, first consider the following CSS without `:is()`:

```
.contact-form input,
.contact-form textarea,
.contact-form select,
.contact-form button {
  padding: 5px 10px;
  border: 1px solid var(--brand-color);
  border-radius: 5px;
}
```

The large selector list here is repetitive and a bit cumbersome to read. `:is()` allows you to do the same thing in a much cleaner way, by merging this all into a single selector. The following snippet is equivalent to the previous listing:

```
.contact-form :is(input, textarea, select, button) {
  padding: 5px 10px;
  border: 1px solid var(--brand-color);
  border-radius: 5px;
}
```

Put formally, the `:is()` pseudo-class is a function that takes a series of selectors as arguments and matches any elements that match any of the given selectors.

The specificity of `:is()` is derived from whichever argument has the highest specificity. Therefore, `:is(input, textarea, select, button)` has a specificity of 0,0,1, but `:is(input, #login-password)` has a specificity of 1,0,0. This specificity is set regardless which of the selector arguments matches a given element.

WARNING The :is() selector does not work with pseudo-element selectors. Therefore, :is(div::before, div::after) is invalid. Pseudo-classes, on the other hand, are still safe to use; for example, :is(:first-child) is valid.

8.3.1 More forgiving selectors

Another benefit of :is() is that it is forgiving of invalid or unrecognized selectors. Consider the following selector list:

```
input.invalid,
input:user-invalid {
  border: 1px solid red;
}
```

The :user-invalid pseudo-class is relatively new and has limited browser support, so this entire ruleset will be ignored in some browsers, even when the other selector matches. This restriction sometimes catches developers off guard, so :is() has been intentionally designed to be more forgiving of invalid selectors. The previous code can be written as follows to achieve a more predictable result:

```
input:is(.invalid, :user-invalid) {
  border: 1px solid red;
}
```

With this version, a browser that doesn't support :user-invalid will still honor the .invalid portion of the selector. This can be a more graceful way to add progressive enhancement for new pseudo-classes in your stylesheet.

8.3.2 Manipulating specificity

The :where() pseudo-class is functionally identical to :is(), with one difference: :where() always has a specificity of zero. This means :where() can be used to reduce the specificity of otherwise high-specificity selectors.

Perhaps you want to select based on an ID but don't want it to override other styles on the same layer. You can write a selector like :where(#login-form) input. This has a specificity of 0,0,1—0,0,0 from :where(#login-form) combined with 0,0,1 from input. You can chain multiple :where() pseudo-classes together.

In an example from earlier in the chapter, we wanted a .button selector to override a:any-link. This can be accomplished by using :where(), as shown in the next listing. The button styles will override the link styles when applied to the same element.

Listing 8.12 Using :where() to manipulate specificity down

```
a:where(:any-link) {
  color: var(--brand-blue);
  font-weight: bold;
}
```

← Specificity of 0,0,1

```

.button {
  display: inline-block;
  padding: 0.5rem;
  color: white;
  background-color: var(--brand-blue);
  font-weight: normal;
  text-decoration: none;
}

```

← Specificity of 0,1,0

Before the advent of cascade layers, this was a very useful technique to keep the specificities of long selectors in check. It's needed less frequently now, but it can still be useful, particularly for managing styles that live on the same layer.

In a similar way, you can use an `:is()` selector to drive up the specificity of a selector by adding an arbitrarily high-specificity argument. For example, `:is(.button, #fake-id)` creates a selector with a specificity of 1,0,0. However, I encourage you to keep specificity low when possible, so I generally discourage this approach.

8.4 Nesting

Nesting is a feature of CSS that allows you to place one style rule inside of another so that the selector of the child rule is relative to that of the parent rule. This allows you to reduce some repetition in code and group related selectors together. If you've worked with preprocessors like Sass and Less, this will look familiar (though there are a couple of minor differences, which I'll cover shortly).

WARNING Nesting is a brand-new feature. As of late 2023, it has been enabled in all major browsers, but some users take a while before installing upgrades. See <https://caniuse.com/css-nesting> for current information on usage statistics before deciding whether to use nesting on a production site.

To illustrate, I'll show you an example written both with and without nesting. Listing 8.13 shows three related rulesets written in the classic, nonnested way. Notice that all three selectors begin with the same `.card` class selector.

Listing 8.13 Related CSS selectors without nesting

```

.card {
  padding: 1rem;
  border-radius: 0.5rem;
  background-color: #fff;
}

.card > h3 {
  margin-block: 0;
  border-block-end: 1px solid #eee;
}

.card .card-body {
  padding-block: 1em;
}

```


Nesting allows this to be done more cleanly. Listing 8.14 shows the same code, rewritten to use nesting. The `.card` class selector has been removed from the second and third selectors, and, instead, those entire rulesets are nested inside the first.

Listing 8.14 Nested CSS selectors

```
.card {                                ← Parent selector
  padding: 1rem;
  border-radius: 0.5rem;
  background-color: #fff;

  > h3 {                               ← Selects .card > h3
    margin-block: 0;
    border-block-end: 1px solid #eee;
  }

  .card-body {                         ← Selects .card .card-body
    padding-block: 1em;
  }
}
```

The selectors for the nested rulesets are joined to the selector of the outer ruleset by the browser, so they still target the same elements as the previous listing. This provides two benefits. First, it removes the repetition of `.card` in all three selectors, making future changes to that class name a little simpler and less prone to typos. Second, it has explicitly grouped these related styles all into the same set of braces, so it's clear these styles all work together. The indentation is optional, but it is generally done to emphasize the structure of the code.

Selector specificity is determined by the final assembled selector, so the specificities of the parent and child selectors are added together. The `.card-body` selector, joined with the parent `.card` selector, has a specificity of 0,2,0. In other words, nested code in this example has the same specificity as its unnested equivalent.

TIP Try to avoid nesting selectors too many layers deep, as this can build up the selector specificity much higher than is necessary.

You can continue to nest selectors several levels deep by adding more rulesets inside the child rulesets. However, this builds up the length of the final assembled selectors, increasing selector specificity in the same way writing a longer selector does. A common rule of thumb is to limit nesting to about three levels deep. When you get deeper than that, it's probably time to think about reorganizing the code a bit. This is a topic I'll look at in greater depth in the next chapter.

8.4.1 Using the nesting selector

Nesting can also be used to indicate other structures beyond the default ancestor-descendant relationship. Consider the following code, without nesting:

```
.modal {  
  display: none;  
}  
.modal.is-open {  
  display: block;  
}
```

In this CSS, the `modal` class is repeated, but if you were to try to nest this code, the default behavior of nesting would not produce the same result. Specifically, if you were to nest `.is-open` inside a `.modal` block, the result would be equivalent to the selector `.modal .is-open`, with a space between the two (a *descendant combinator*), rather than the joined compound selector `.modal.is-open`.

However, you can modify the way the outer selector is joined to the inner one by using an ampersand (&) in the inner selector. This is called the *nesting selector*. The nesting selector is a way to refer to the outer selector and specify where it is merged into the inner selector. The following listing shows an example of this, where the inner selector of `&.is-open` is equivalent to `.modal.is-open`, with no space between the two class selectors.

Listing 8.15 Using & to create a compound selector

```
.modal {  
  display: none;  
  
  &.is-open {  
    display: block;  
  }  
}
```



This is particularly useful when you want to apply different styles for multiple variations or states of an element on the page. You can use `&:hover` as the inner selector to target the element in a hover state or `&::after` to target its `::after` pseudo-element.

WARNING Unlike in Sass, you cannot use the nesting selector to merge partial class names. In Sass, a `.card` parent with an `&large` inner selector is converted to a selector with a single class like `.cardlarge`, but in CSS it is interpreted more like the compound selector `large.card`, targeting a `<large class="card">` element (even though `<large>` is not a valid element).

You can also use the `&` to change the selector order by placing it later in the nested selector. The following listing shows an example of this. In this case, the nested selector targets `.homepage .card`—a card when it is inside a `.homepage` element.

Listing 8.16 Using the & symbol in nested CSS

```
.card {  
  padding: 1rem;  
  border-radius: 0.5rem;  
  background-color: #fff;
```

```
.homepage & {  
  background-color: #ccc;  
}  
}
```



Using this approach flips the code around in a way. Instead of styling an element that’s a child of the outer selector, you’re styling the outer element in a specific context.

WARNING Typically, I try to avoid nesting with `&` at the end of the selector when possible. This is not an absolute rule, but styling elements based on their context like this can make code difficult to maintain for the long term. I’ll explain the reasons for this in the next chapter and provide alternate ways to achieve the same thing.

An obsolete restriction

In early versions of the nesting specification, there was a restriction that nested selectors had to begin with a symbol such as `&`, `.`, or `#` and could not begin with a letter identifier—essentially, they could not be a tag selector, so the following was deemed invalid:

```
.card {  
  h3 { ... }  
}
```

This restriction was added because browser vendors had concerns about the performance of their language-parsing algorithms and the ability to disambiguate between a nested selector and a regular property declaration. However, browsers found an efficient way to handle this. This restriction has been removed from the specification, and this sort of nesting is in fact valid.

I point this out only because many early tutorials teach this restriction, and the first implementations in Chrome and Safari do enforce it. This behavior is expected to change before this book goes to print or soon thereafter. In the meantime, you can work around this restriction by prepending an `&` to the inner selector (e.g., `& h3`).

These options open up a wide range of new ways to organize CSS selectors in your stylesheet. Deciding how exactly to do that can be a complex topic. In the next chapter, I’ll look at some practical advice on how to best do that and what patterns to avoid.

8.4.2 Understanding the nuances of the nesting selector

Nesting is a peculiar feature of CSS because technically it adds no new functionality to the language. It’s what’s known as *syntactic sugar*: it provides a way to write code that is “sweeter” for human use. It’s easier to read or express certain ideas, but behind the scenes, it translates (or *desugars*) directly to equivalent code written without nesting.

It's important to know that any time you use `&` to reference the parent selector, it is desugared by wrapping the parent selector in an `:is()` pseudo-class function. This allows for the parent ruleset to use a selector list with more than one selector. Consider the following example:

```
button,
input,
textarea,
select {
  &.invalid {
    border: 1px solid red;
  }
}
```

This nested selector is desugared by the browser to `:is(button, input, textarea, select).invalid`. By interpreting the nesting selector with `:is()`, the translation is straightforward for the browser, even for complex, deeply nested selectors.

NESTING AFFECTS SPECIFICITY

While the behavior of this is straightforward in a simple example, it's important to be aware of in certain scenarios for two reasons. The first reason is the way it can affect selector specificity. Because `:is()` derives its specificity from the highest specificity of its arguments, the nested selector does as well. If the parent selector has a selector with high specificity, any nested selectors that use `&` will receive that specificity as well. For example, consider the following nested selector:

```
input,
#login-form button {
  &:focus {
    border-color: var(--brand-color);
  }
}
```

In this case, the browser interprets the nested selector as `:is(input, #login-form button):focus`, which has a specificity of 1,1,1, even if conceptually only the `input:focus` part of the selector (seemingly 0,1,1) matches the element in question.

NESTING AFFECTS WHICH ELEMENTS ARE TARGETED

The second ramification of this is it can lead to selectors that target more elements than you might expect. Consider the following nested selector:

```
.card .title {
  footer & { ... }
}
```

This nested selector targets a title inside a card inside a footer, matching HTML such as the following:

```
<footer>
  <div class="card">
```

```

    <h3 class="title">Franklin Running Club</h3>
  </div>
</footer>

```

However, it's easy to miss that it also targets the title in markup like in the following:

```

<div class="card">
  <footer>
    <h3 class="title">Franklin Running Club</h3>
  </footer>
</div>

```

The nested selector desugars to `footer :is(.card .title)`. The selected title is a child of a footer, and it is a child of the card—but the parentage between the footer and the card is ambiguous, so they can be nested in either order and the selector will match.

This difference is easy to miss, and you will likely go a long time before it surfaces in such a way that it causes a bug on a page you're building. But you need to be aware of it when it does.

8.4.3 Nesting media queries and other at-rules

Nesting can also be used to add `@media` queries inside a ruleset. I find this very convenient because I like keeping related code grouped together. This way, the responsive code for each segment of the page stays together.

Listing 8.17 shows an example of this. Note that the declarations inside the nested media queries don't require any nested selectors; they are applied directly to the parent selector. This way, you only need to write the selector once to apply responsive styles to it.

Listing 8.17 Nested media query

```

h1 {
  font-family: var(--font-heading);
  font-size: 2.2rem;

  @media (min-width: 800px) {
    font-size: 2.6rem;
  }

  @media (min-width: 1200px) {
    font-size: 3rem;
  }
}

```

**Medium
breakpoint
styles for h1**

**Large
breakpoint
styles for h1**

This is equivalent to the following nonnested code:

```

h1 {
  font-family: var(--font-heading);
  font-size: 2.2rem;
}

```

```

@media (min-width: 800px) {
  h1 {
    font-size: 2.6rem;
  }
}

@media (min-width: 1200px) {
  h1 {
    font-size: 3rem;
  }
}

```

Similar to media queries, you can nest `@layer` and `@supports` at-rules as well. The next listing shows an example of each of these. These all apply to the parent `input` selector.

Listing 8.18 Nested `@layer` and `@supports` rules

```

input {
  @layer base {
    font: inherit;
    border: 2px solid #999;
  }

  @layer modules {
    padding: 5px 10px;

    @supports selector(:user-invalid) {
      border-color: green;

      &:user-invalid {
        border-color: red;
      }
    }
  }
}

```



Nested layer rules for inputs



Input styles for browsers that support the `:user-invalid` pseudo-class

As in listing 8.17, this example includes some declarations that are nested in the various at-rules but no child selector. Again, these apply to the element specified by the parent selector.

The `@scope` and `@container` at-rules may also be nested. I will cover these rules in the following chapters.

ALWAYS PLACE NONNESTED DECLARATIONS BEFORE NESTED BLOCKS

In all of my examples thus far, I have made a point of putting nonnested declarations at the beginning of the outer code block. I recommend you do the same thing for two reasons.

The first reason is that this is how the browser will always interpret them. If you place nonnested declarations after a nested block, the browser will actually reverse the order when determining the source order for purposes of the cascade.

In listing 8.19, for example, it appears that the 2.2 rem font size would override the 2.6 rem in the nested media query. This is not the case. When the media query matches, the 2.6 rem will take precedence in the cascade, just as if the nonnested declaration appeared before the media query.

Listing 8.19 Nonnested declarations before nested blocks

```
h1 {  
  @media (min-width: 800px) {  
    font-size: 2.6rem;  
  }  
  font-size: 2.2rem;  
}
```



The browser moves this declaration up above the @media block, reversing the source order.

If you have experience with preprocessors, this is the same behavior seen with preprocessor nesting.

The second reason to add nonnested declarations first is it allows them to be interpreted by browsers that don't support nested blocks. An older browser can parse and apply code like `h1 { font-size: 2.2rem;` before it encounters a nested selector it doesn't understand. This usually allows for better graceful degradation rather than leaving the page largely unstyled in older browsers.

Summary

- Cascade layers provide explicit control over CSS priority. Using layers provides a more reasonable means to manipulate which styles take precedence compared to using selector specificity and source order alone.
- Use these six layers as a helpful template to organize your stylesheets: reset, theme, global, layout, modules, and utilities. Adapt this as needed based on the particular needs of your project.
- The `:is()` pseudo-class allows you to select from among a list of given selectors. This can help shorten otherwise repetitive and cumbersome selectors.
- The `:where()` pseudo-class behaves like the `:is()` pseudo-class but always has zero specificity. Use this when you need to keep selector specificity at a minimum to prevent overriding other styles.
- Nesting allows you to place descendant selectors or at-rules inside a parent selector block.

Modular CSS and scope

This chapter covers

- Organizing CSS into reusable modules
- Preventing escalating selector specificity
- Common naming conventions for CSS organization
- Using `@scope` for better control of the cascade
- Using a pattern library to document your modules

When you make changes to an existing stylesheet, those changes can affect any number of elements on any number of pages across your site. Here'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 the previous chapter, we looked at layers, which are the first step in controlling the priority of the various styles on the page. I recommended defining several layers on your page. The largest of these, the modules layer, is where the bulk of your styles should reside. This chapter is about organizing that layer into manageable, modular pieces of code.

9.1 Defining modules

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 on 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. 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 many developers don't think to apply it to CSS. 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. Using a modular approach is a well-proven technique.

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 I'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.

DEFINITION *Encapsulation* is the grouping together of related functions and data to compose 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 a module, and each module will be responsible for styling a small number of DOM elements. CSS also has a brand-new feature, `@scope`, that can help immensely in this process.

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.

NOTE If you work in a JavaScript framework such as React, Vue, or SolidJS, this is a familiar concept, as your JavaScript is organized similarly into components. When organizing your CSS into modules, it is quite possible your CSS modules will align one to one with your JavaScript components, but this correlation is not strictly necessary.

9.1.1 *Modules and global styles*

Before we dive into writing modular styles, you'll need some basic styles in place. This includes common resets and global styles such as fonts. Because these styles affect everything on the page, you'll need them present before you start building any specific modules.

Create a new page and stylesheet, and add the reset and global styles shown in the following listing to the CSS. The reset layer has been copied from listing 8.5 in the previous chapter.

Listing 9.1 Adding reset and global styles

```
@layer reset, global, modules;

@layer reset {
  *,
  *::before,
  *::after {
    box-sizing: border-box;
  }

  body {
    margin: unset;
  }

  button,
  input,
  textarea,
  select {
    font: inherit;
  }

  img,
  picture,
  svg,
  canvas {
    display: block;
    max-width: 100%;
    height: auto;
  }
}

@layer global {
  body {
    font-family: Helvetica, Arial, sans-serif;
  }
}
```

A full website will likely have more layers, as you saw in chapter 8, but this will be sufficient for taking a closer look at modules.

Layers are not a prerequisite for writing modular CSS. If you are avoiding layers for browser support reasons, you can still follow these principles without them.

9.1.2 A simple module

You'll begin with a simple module for brief notification messages. Because each module needs a unique name, you'll call this one "message." It will have a bit of color and a border to capture the user's attention (figure 9.1).



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 the padding, border, border radius, and colors. To apply these styles to the message module, add the following listing to the end of your stylesheet.

Listing 9.3 Implementing the message module

```
@layer modules {
  .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 the 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 system notification, 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. Creating a module 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.

9.1.3 *Variations of a module*

Consistency is good, but sometimes you'll want to intentionally deviate from it. The message module is nice, but you 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.

BEM class name conventions

Double-hyphen and double-underscore class names are common conventions that are part of a methodology called *BEM*. BEM is an acronym for Block-Element-Modifier.

In BEM, a *block* refers to the main element of a module, which is given a descriptive, unique class name, such as `message`. A BEM *element* refers to a child (or, occasionally, grandchild) element that is part of the module; these are denoted by the block name followed by a double underscore and the element name—for example, `media__image` to reference an image element inside a media module. And a *modifier* refers to a class name added to the block when creating a variant; these are denoted by the block name followed by a double hyphen—for example, `message--error`.

BEM is one of many similar methodologies. I am using some aspects of it in this chapter because it is one of the most well known. Other popular approaches include OOCSS, SMACSS, ITCSS, and SUITCSS. These each have slightly different rules and naming conventions, but they all aim for the same thing: organizing your styles in a modular way.

You'll create three modifiers for your module: success, warning, and error. Add the next listing to your stylesheet. This uses the BEM double-hyphen convention to indicate the modifiers belong to the message module.

Listing 9.4 Message module with modifier classes

```
@layer modules {
  .message {
    padding: 0.8em 1.2em;
    border-radius: 0.2em;
    border: 1px solid #265559;
    color: #265559;
    background-color: #e0f0f2;
  }

  .message--success {
    color: #2f5926;
    border-color: #2f5926;
    background-color: #cfe8c9;
  }

  .message--warning {
    color: #594826;
    border-color: #594826;
    background-color: #e8dec9;
  }

  .message--error {
    color: #59262f;
    border-color: #59262f;
    background-color: #e8c9cf;
  }
}
```

← **Base message module**

← **The success modifier changes the message to green.**

← **The warning modifier changes the message to yellow.**

← **The error modifier changes the 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 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 look at another module with some variants. You'll make a button module with modifiers 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.

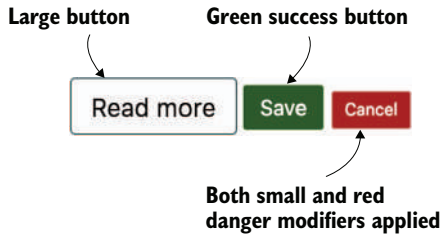


Figure 9.2 Buttons with various size and color modifiers applied

The styles for these buttons are given in listing 9.6. It includes the main button module, as well as four modifier classes: two size modifiers and two color modifiers. Add these to your stylesheet in the modules layer.

Listing 9.6 Button module and modifiers

```
@layer modules {
  .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;
  }

  .button--danger {
    border-color: #e8c9c9;
    color: #fff;
    background-color: #a92323;
  }

  .button--small {
    font-size: 0.8rem;
  }

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

← Basic button styles

← Green success color variant

← 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 CSS for a module together in the same place, including its modifier classes. 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.

The following listing shows some HTML, mixing and matching the modifiers to create various buttons. Add this anywhere on 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 might have similar names.

AVOIDING CONTEXT-DEPENDENT SELECTORS

Imagine you're maintaining a website that has light-colored dropdown menus. One day, you find you need the dropdown menu in the page header 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 `.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? If a coworker needs to make the changes, will they be able to find 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 have 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 problems 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 on the page you need it.

Never use descendant selectors to alter a module based on its location on 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.

9.1.4 *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.

Next, you'll build a more complex module. This one will be a media module (figure 9.3), a name coined by Nicole Sullivan for this common pattern in CSS.

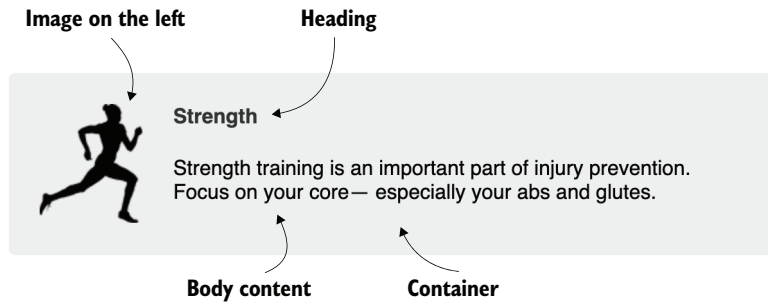


Figure 9.3 Media module made with four elements

This module consists of several elements: it has a `div` container that includes an image and a body, and inside the body is the title and some text. 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 and then the name of the subelement—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 next. Add these to your stylesheet.

Listing 9.8 Media module with subelements

```
@layer modules {
  .media {
    display: flex;
    gap: 1.5em;
    padding: 1.5rem;
    background-color: #eee;
    border-radius: 5px;
  }

  .media__image {
    align-self: start;
  }

  .media__body {
    overflow: auto;
    margin-block-start: 0;
  }

  .media__body > h4 {
    margin-block-start: 0;
    color: #333;
  }
}
```

← Main container

← The image and body subelements

← 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.

You did use a direct descendant combinator for the title. You could use the class `media__title` (or even `media__body__title` to fully indicate its location 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, as shown in the following listing.

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>
```

Diagram illustrating the structure of the media module markup:

- Image subelement**: Points to the `<img>` tag.
- Body subelement**: Points to the `<div class="media__body">` tag.
- Title subelement**: Points to the `<h4>Strength</h4>` tag.

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-inline-size` to the image so it doesn't crowd out the body).

USING VARIANTS AND SUBELEMENTS TOGETHER

You can also create variations of the module. Now, it's trivial to make a version where the image sits on the right instead of the left (figure 9.4).

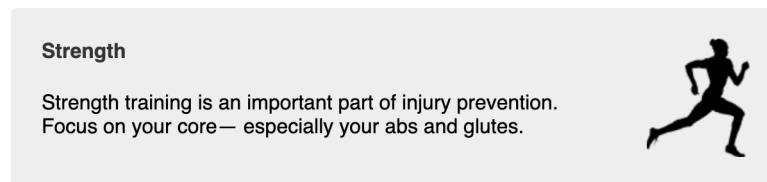


Figure 9.4 Media module with the right variant

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, inside the modules layer, to see this version.

Listing 9.10 Defining a right variant for the media module

```
@layer modules {
  .media--right {
    flex-direction: row-reverse;
  }
}
```

This reverses the order of the flex items, placing the image on the right.

For some module variants, you might need to adjust the styles of child elements. You can do so with descendant selectors—for example, `.media--right .media__image`.

AVOIDING GENERIC TAG NAMES IN MODULE SELECTORS

In the media module, you used the selector `.media__body > h4` to target the title element. You were 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 to add a `menu__item` class to each and every item in a list, though opinions vary on this topic.

You should avoid targeting based on overly generic tag types, especially `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 involves hunting down every use of that module in your markup and making the change there.

9.2 Modules composed into larger structures

Your modules should each be responsible for one thing. The message module is responsible for making a message noticeable. The 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, you'll build a dropdown menu (figure 9.5). This will appear like the one you built in chapter 6 (section 6.3.1).

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 have a rule of thumb: if I must 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?



Figure 9.5
Dropdown menu

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.

9.2.1 Dividing multiple responsibilities among modules

I’ll show you how to 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 listing 9.11 onto 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.

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="module">
  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 subelement 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 subelements 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, producing `<div class="dropdown is-open">` when the toggle button is clicked and reverting back to `<div class="dropdown">` when it is clicked again.

The styles for the dropdown are shown in listing 9.12. 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

```
@layer modules {
  .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;
    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;
  }
}
```

Establishes containing block for the absolutely positioned drawer

Draws the triangle using borders (chapter 6)

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. Several things are going on here, but it's not so much that 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.

NOTE 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 SASS `@use` statements. It would look something like the following:

```
@use 'reset';
@use 'global';
@use 'message';
@use 'button';
@use 'media';
@use 'dropdown';
```

The preprocessor would then bring in your reset and global styles from `reset.scss` and `global.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 next. Add these to your stylesheet.

Listing 9.13 Menu module styles

```
@layer modules {
  .menu {
    margin-block: 0;
    padding-inline-start: 0;
    list-style-type: none;
    border: 1px solid #999;
  }

  .menu > li + li {
    border-block-start: 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 6. Each `<li>` is a subelement of the module, so I didn't feel the need to add a double-underscore class to each and every one. The direct descendant selector `.menu > li` is specific enough.

This module is completely standalone. It has no dependency on 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.

It takes some experience to develop a feel for organizing your code into modules. In this example, there may not be a strict reason to separate the dropdown module from the menu module—the two truly are meant to work together, and in a real-world project you may not have a concrete need to support different menu types in a dropdown.

In this case, I saw that they were conceptually easy to separate. This makes each module smaller, and smaller modules tend to be easier to understand and update in the future. Sometimes a module will naturally be larger; if splitting it up proves to be difficult, don't do it until there is a concrete need to do so.

9.2.2 *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.

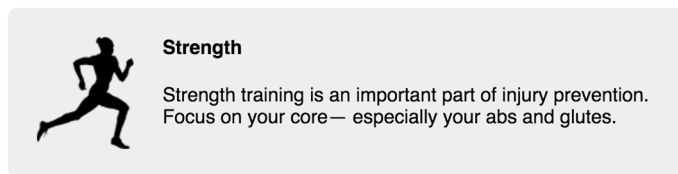


Figure 9.6 Media module with an image and a running tip

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 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 in. 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 have 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 have 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`.

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

(continued)

to modular CSS. In search of a solution, they turned to JavaScript. They devised an approach now known as *CSS in JS*.

Instead of relying on class-naming conventions, this approach requires using JavaScript to manufacture class names that are guaranteed unique. A number of JavaScript libraries have emerged to do this work, the most popular of which are Styled Components, vanilla-extract, and one called (confusingly) CSS Modules. Many of these are tied to a particular JavaScript framework or toolset, such as React or WebPack.

This approach is 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 React. Going down this road involves some tradeoffs, and it locks you into a particular toolset. It's not a perfect solution but has proven successful for some.

9.3 CSS scope

There is a brand-new feature in CSS that will be immensely helpful in writing modular styles called *scope*. While BEM and other similar methodologies provide conventions that aim to keep your styles organized in a modular way, it takes a certain level of discipline to enforce its use and requires education across a development team to ensure all the developers use and understand the same approach consistently.

Scope, on the other hand, provides an alternate approach that is enforced by the browser and controlled by the cascade itself. Scope, defined using the `@scope` at-rule, allows you to restrict styles so they only apply to a specified portion of the page. It comes with some new features that are particularly well suited for building modular styles.

WARNING Scope is the newest feature of CSS I cover in this book. It shipped in Chrome version 118, in October 2023—though it has been available behind the Experimental Web Platform features flag for some time (see chapter 1)—and in the Safari tech preview shortly thereafter. It is still unclear when Firefox will follow suit, but it is likely to have a significant effect on how we write CSS when it does. Therefore, I think its inclusion in this book is warranted.

As I've shown, BEM and other similar methodologies rely on the use of carefully crafted class names to prevent styles from multiple modules from targeting the same element and conflicting with one another. Scope can be used to control or prevent these conflicts, rendering the extra attention on class names unnecessary.

The `@scope` at-rule is followed by a selector, which determines where the scope applies. Brackets then surround styles that only apply within that scope. A basic example is shown in listing 9.14. This shows the media module from earlier, but it has been rewritten to use scope. BEM-style class names are also simplified or, in

some cases, removed entirely. Edit that portion of your stylesheet to match the following listing.

Listing 9.14 The media module written with scope

```
@layer modules {
  @scope (.media) {
    :scope {
      display: flex;
      gap: 1.5em;
      padding: 1.5rem;
      background-color: #eee;
      border-radius: 5px;
    }

    :scope.right {
      flex-direction: row-reverse;
    }

    img {
      align-self: start;
    }

    .body {
      overflow: auto;
      margin-block-start: 0;
    }

    h4 {
      margin-block-start: 0;
      color: #333;
    }
  }
}
```

The scope is restricted to inside a .media element.

Targets the .media element itself

Targets elements inside .media

This includes a special pseudo-class selector, `:scope`, which targets the root element of the scope—the element that is matched by the scope selector. In this case, that’s the `.media` element. Notice that the `.media` selector only appears in the scope definition. When you want to style that element directly, you should use the `:scope` selector inside the scope block.

All the other selectors are scoped within the `.media` element, so a `.media` selector is no longer needed as part of these selectors. Selectors inside of a scope block (such as `img` or `.body`) can’t target other elements elsewhere on the page. You are free to reuse those class names or element types anywhere on your page, and these selectors will target only those that are descendants of a `.media`.

The selector in the `@scope` rule does not add any specificity to the selectors within its curly braces. The `img` selector, while superficially similar to `.media img`, has a specificity of only 0,0,1 (not 0,1,1). Similarly, the `.body` selector has a specificity of 0,1,0.

With these changes, the markup for a media module can now be simplified for the shorter class names. Update the corresponding HTML to match the following listing.

Listing 9.15 Simplified media module markup

```
<div class="media">
  
  <div class="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 class name is no longer needed.

The class name is shortened from `media__body`.

Be sure to use Chrome 118 or higher to view this example (or, if on a lower version of Chrome, enable Experimental Web Platform features on the <chrome://flags/> page).

Developer and CSS Working Group member Miriam Suzanne, who is the primary author of the scope specification, likens scope to a sense of “belonging.” The standard selector `.media .body` selects `.body` elements that happen to be a descendant of a `.media` element. However, what you typically want is not all the body elements within a media module but specifically the body that belongs to the media module. Scope is an explicit way to enforce this.

At this point, it may not be obvious how this provides a benefit over the previous approach. That becomes apparent when you start nesting multiple scopes on the page, one within another. At that point, two key features of scope become important: proximity and an inner scoping limit. Let’s look at each of these features.

9.3.1 Scope proximity

Proximity is a new criteria in the cascade. When one scope is nested inside of another, both scopes might have selectors that target the same element. When this occurs, the selector with the highest specificity takes precedence, as normal. However, if both selectors have the same specificity, the rule of proximity dictates that the styles of the nearest scope (the inner one) will take precedence.

Figure 9.7 shows the cascade flow chart from chapter 1. I have highlighted the step where scope proximity is considered by the cascade. Crucially, proximity is considered before the source order when resolving conflicting declarations.

To see how this works, you’ll add another module to the page. It will be a container with a light-blue background, intended to contain a portion of the page that’s meant to stand out from the content around it. I’ll call it a highlight-block. This module will apply a blue text color to any headings within (`<h1>` through `<h4>`). Add the code in listing 9.16 at the end of your stylesheet beneath the media module.

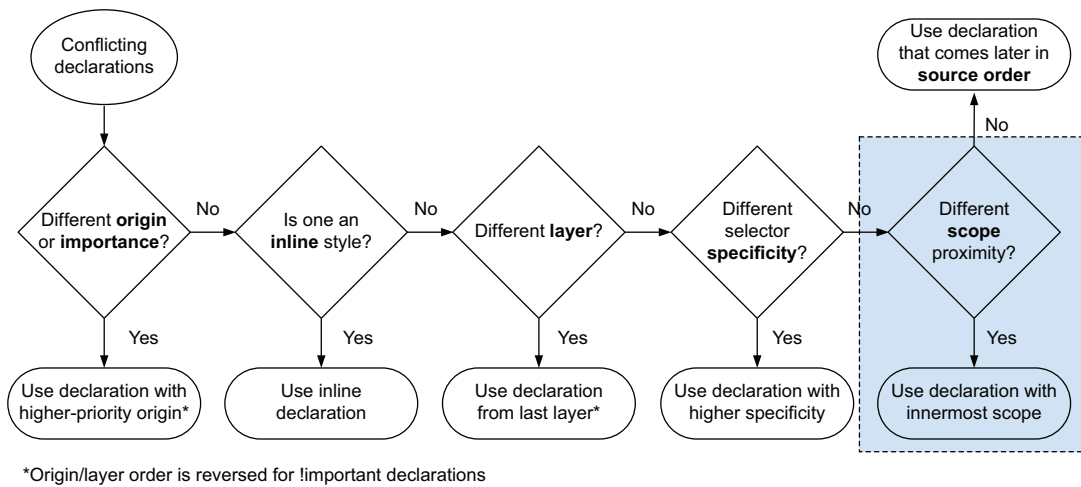


Figure 9.7 Scope proximity is considered after selector specificity but before source order.

Listing 9.16 The highlight-block module

```

@layer modules {
  @scope (.highlight-block) {
    :scope {
      padding: 1rem 1.5rem;
      background-color: #b3cbe6;
    }

    h1,
    h2,
    h3,
    h4 {
      color: #264b73;
    }
  }
}

```

I’ve kept this module very simple. The key thing here is the way it interacts with another module with a different scope. If you place a media module inside this, it will look like the screenshot in figure 9.8. Both modules have an `h4` selector, so both target the Strength heading in the media module. But because the media module is the innermost scope, its styles take precedence, making the heading dark gray rather than blue.

In your HTML, add a div with the `highlight-block` class so that it contains the media module. This makes the highlight block an outer scope and the media module a scope nested within. This markup is shown in listing 9.17. This also includes a little example text to illustrate how the highlight-block module works on its own.

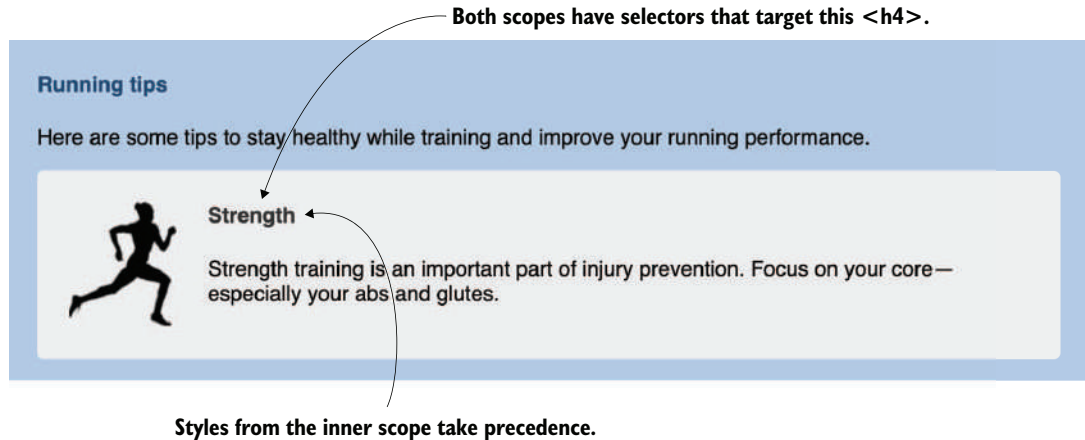


Figure 9.8 Nested scopes

Listing 9.17 Wrapping the media module with a highlight block

```

<div class="highlight-block">                                ← Outer scoped element
  <h4>Running tips</h4>
  <p>
    Here are some tips to stay healthy while training
    and improve your running performance.
  </p>
  <div class="media slot">                                    ← Inner scoped element
    
    <div class="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>

```

I've also added a `slot` class to the media module here, which is unused at this point. You'll make use of this shortly.

The whole idea behind modular CSS is to construct a library of building blocks in the CSS that can then be arranged in any order necessary to assemble the whole page. When you are building a module, such as the media module, you don't know all the places it could be used in the future. The authors of the site may want to place it inside all sorts of different containers. But when that happens, you will want its styles to take precedence over the styles of the outer modules.

When you are writing the CSS, you don't necessarily know what this order of precedence will be. Scope proximity is a way to ensure a module will not be adversely affected by any of its containing modules. With scopes, the actual structure of the HTML is able to influence the cascade. This is often what we want out of the page, especially in complex applications that are composed of dozens of nested modules.

9.3.2 Scoping limit

While proximity helps with prioritizing styles between modules, you often know where the inner boundary of a module is. Instead of relying on proximity to sort out conflicting styles, it's even better to avoid the conflicting styles altogether. This can be done by specifying a *scoping limit*.

Add a scoping limit to a scope by using the `to` keyword and a second selector—for example, `@scope (.highlight-block) to (.slot)`. Think of the `to` keyword as “until.” This scopes from `.highlight-block` as an upper bound to `.slot` as a lower bound. This is sometimes referred to as *donut scoping*, because the part of the DOM that's in scope has a hole in it, like a donut.

Update your stylesheet to include this change, as shown in the following listing.

Listing 9.18 Adding a scoping limit

```
@layer modules {
  @scope (.highlight-block) to (.slot) {
    :scope {
      padding: 1rem 1.5rem;
      background-color: #b3cbe6;
    }

    h1,
    h2,
    h3,
    h4 {
      color: #264b73;
    }
  }
}
```

Selectors in the scope will not target `.slot` or its contents.

Proximity only resolves declarations between two scopes when they conflict—if they specify values for the same property on the same element. It does not prevent an outer scope from applying additional styles beyond those. In the earlier example (listings 9.16 and 9.17), the Strength heading received its color from the media module due to proximity precedence. But if the media module did not specify a color, the blue color from the outer scope would have still applied.

Using a scoping limit is important for modules that are expected to contain other modules. This way, you know whatever selectors you add to the module will never reach into inner modules and change things unexpectedly. So, for instance, the dropdown module you built earlier could be rewritten with `@scope (.dropdown)`

to `(.drawer > *)`. This allows it to style its `<div class="drawer">` but nothing that resides within it.

Listing 9.19 shows the dropdown module rewritten using scope and an inner scoping limit. I'll leave it as an exercise for you to update the HTML and JavaScript to work with these changed class names; you can also modify the menu module to use a scope as well.

Listing 9.19 Dropdown module written using scope

```
@layer modules {
  @scope (.dropdown) to (.drawer > *) {
    :scope {
      display: inline-block;
      position: relative;
    }

    .toggle {
      padding: 0.5em 2em 0.5em 1.5em;
      border: 1px solid #ccc;
      font-size: 1rem;
      background-color: #eee;
    }

    .toggle::after {
      content: "";
      position: absolute;
      right: 1em;
      top: 1em;
      border: 0.3em solid;
      border-color: black transparent transparent;
    }

    .drawer {
      display: none;
      position: absolute;
      left: 0;
      top: 2.1em;
      min-width: 100%;
      background-color: #eee;
    }

    :scope.is-open .toggle::after {
      top: 0.7em;
      border-color: transparent transparent black;
    }

    :scope.is-open .drawer {
      display: block;
    }
  }
}
```

These selectors might even be made simpler. The toggle button is probably the only `<button>` that will ever appear in this scope, so you could probably select by the tag name rather than requiring a class name.

If you have a lot of experience with BEM, these selectors may feel odd. After all, those methodologies require heavy use of class names to prevent collisions (and to keep specificity predictable). But with proper use of scope, the chance of that happening is eliminated. You can add a `.title` selector to dozens of modules and know that they'll each only ever target the title element that belongs to them.

You can define a scope such that portions of the module are restricted with a scoping limit and others are not. Do this by using one `@scope` rule with a scoping limit and a second similar `@scope` rule without one. Consider the following snippet, which follows this pattern:

```
@scope (.tabs) to (.tabs__pane) {  
  button {  
    padding: 0.4rem 1.4rem;  
    color: #ccc;  
    background-color: white;  
  }  
}  
  
@scope (.tabs) {  
  h2 {  
    font-size: 1rem;  
    font-weight: bold;  
    color: #369;  
  }  
}
```

The first scope here, with a scoping limit, styles buttons that belong to the tabs module. The second scope, with no scoping limit, affects all h2 headings within the module, at any depth. You can use both approaches together, allowing a module to selectively influence only certain aspects of content within.

Note that a scoping limit does not stop style inheritance. Things like font styles of an outer module can still be inherited by nested modules.

Scope compared to Shadow DOM

Shadow DOM is used when defining web components as a means of isolating them from the outer portion of the web page. Shadow DOM has sometimes been offered as a way to provide scoping for styles, though I believe it is only useful for this in specific circumstances. Shadow DOM is primarily a JavaScript-driven feature, so it is outside the scope of this book. But I want to note the differences between it and scope, as users of shadow DOM will often ask about this when introduced to scope.

The portion of a web page in the shadow DOM is strongly isolated from the rest of the page. In a sense, it is like a hole is cut out of the page, removing it from the cascade. The shadow DOM must provide its own styles; the page outside cannot style it directly. This is useful if you need to bundle up a component for use on multiple websites and it must look identical on all of them. A widget for embedding social media posts on blogs would be a good example of this. These details are defined in the HTML and JavaScript and cannot be done by your stylesheet.

(continued)

Scope, on the other hand, incorporates your modules into the cascade. It is more suited for situations where the entire page is branded together cohesively. It allows the outer page to influence styles within a scope while giving you fine-grained control over how it does and does not do so. Modules like those defined in this chapter are a good example of this: they can be written in the same stylesheet as the rest of your page and can rely on a known set of global styles and custom properties.

When all your modules are scoped, the only naming collisions you need to be concerned with are the class names on the root elements of the modules. I recommend distinguishing these classes as special cases somehow. One option is to give all module class names a common prefix like `m-`, resulting in unique class names like `m-media` or `m-dropdown`. Or you can use a custom attribute rather than a class name, such as `<div data-scope="dropdown">`. If you use the latter approach, you can use attribute selectors to define your scopes. For example:

```
@scope ([data-scope="dropdown"]) to ([data-scope])
```

The inner bound here will match any nested elements that have a `data-scope` attribute. If you place a `<div data-scope="menu">` inside a `<div data-scope="dropdown">`, the dropdown module will have an inner bound of the menu module, so its styles will not reach in and alter the other module. If you use this approach consistently for all modules, then modules will be scoped to only style content outside any nested modules they contain.

Since this is a brand-new feature of CSS, there are no established patterns in the industry yet. I encourage you to experiment and see what works for you.

9.3.3 *Implicit scope*

Scoped styles can also be added on the page as a `<style>` tag inside the scope's root element. By using the `@scope` rule without any selector, it will implicitly scope to the parent element. In the following listing, for instance, the scoped styles are scoped to `<div class="wrapper">`.

Listing 9.20 Implicitly scoped styles

```
<div class="wrapper">
  <style>
    @scope {
      .message {
        padding: 0.8em 1.2em;
        border-radius: 0.2em;
        border: 1px solid #265559;
        color: #265559;
        background-color: #e0f0f2;
      }
    }
  </style>
</div>
```

```
</style>
<div class="message">Save successful</div>
</div>
<p class="message">This is out of scope.</p>
```

This approach can be helpful for one-off content that you need to embed on the page. That could be a short demo of something in a blog post or possibly markup embedded in the page as part of a JavaScript-driven framework.

This is likely not the default way you'll use scope. Modules tend to be designed for reuse, and you don't want to have to repeat the CSS each time you add a certain module to the page. But in some particular circumstances, it might happen to be more convenient.

9.3.4 Scope and layers

It might seem peculiar that specificity still takes precedence over scope. After all, if we want an inner scope to override an outer scope; why let specificity get in the way of that? But between specificity, the ability to define an inner scoping limit, and cascade layers, we have a huge amount of flexibility.

If you don't want a module to affect the styles of child modules, use a scoping limit. But if you do, you can manipulate the selector specificity of both modules to decide which should have precedence. You also aren't restricted to keeping all styles on the same cascade layer.

In all the examples I've shown, scoped styles have all been applied on the same layer, but this is only one possible approach. You may want to consider breaking the modules layer up into sublayers. You can define the shape and layout of modules on a `modules.base` layer and their colors and fonts on a higher-priority `modules.ui` layer. Styles on a higher-priority layer will always override those on a lower one (based on the order layers appear, as covered in chapter 8).

Again, this is a new feature, so the industry hasn't solidified around any particular patterns here. You can think of layers and scope as two separate dimensions, like rows and columns on a grid. You can choose where to place your styles in that grid, giving you a lot of options to dictate which styles have the highest precedence.

9.4 Pattern libraries

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 assemble 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, a dropdown, or a navigation menu and you've already created one, you already have the styles ready for it. You'll simply add to your HTML the needed elements and class names, 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 and 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 common on large projects to put together documentation that provides this inventory. This documentation is sometimes called a *pattern library*. 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.

How you go about building this pattern library will be highly dependent on what technology you are familiar with or what your team uses to build your application. You might consider a tool like Storybook (<https://storybook.js.org/>) if you're using one of the JavaScript frameworks it supports, or maybe you'll prefer to build a set of hand-built HTML pages that index and showcase all your stylesheet's modules.

9.4.1 *Using a CSS-first workflow*

When you need to build new modules for your site or application, I recommend adding it to your pattern library first, before you use it on the page. 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 should look something like the following:

- 1 When building a page, have a sketch or mockup or some general idea of 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 so 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 the module 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 10 different pages on your site with 10 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.

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. Organizing CSS into modules is the means of accomplishing this. A module in the CSS dictates the structure of HTML needed to make it work on the page.

9.4.2 Refactoring and breaking changes

Sometimes you'll need to make changes to a module in a way that is no longer compatible with an older version. 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 structure.

But often I find the best course of action is to deprecate the module (indicating so in the pattern library) 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 version my CSS using a three-numbered *semver*. When the version number changes, it communicates to developers the nature of those changes.

DEFINITION *Semver*, short for Semantic Versioning, is 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 <https://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 existing HTML, 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 several 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 as part of a Node.js 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 version 3.0.0 with breaking changes, but the web application can continue using the old version until developers are ready 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 regard to which modules are still valuable and which are no longer needed.

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 write new CSS, adding more to the end of the stylesheet, overriding earlier rules with ever-increasing selector specificity until the stylesheet is an unmaintainable tangle of code.

By taking care to organize your CSS in a modular way, 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 a pattern library helps developers keep tabs on everything going on within the stylesheet.

Summary

- A module is a small, reusable piece of styling that is not dependent on its context in the page. The styles for a module should be grouped in one place in your stylesheet, following the programming principle of encapsulation. Modular CSS will never reach into another module and change its appearance.
- Variant classes allow you to provide multiple versions of the same module.
- Large constructs should be broken down into smaller modules. Assemble your pages by piecing together a number of modules.
- A naming convention such as double hyphens and double underscores can be used to make your modules' structure easier to understand at a glance.

- The `@scope` at-rule is a brand-new feature that restricts a set of styles to a specific portion of the page. This is a perfect tool for keeping modular styles scoped to the intended module. Supplying an inner scoping limit prevents these styles from targeting descendant scopes. Keep an eye on emerging browser support for this feature.
- For large projects, you can construct a pattern library to document and inventory your modules. Use this as a reference for available styles when assembling new pages.

10

Container queries

This chapter covers

- Defining containers and using the @container at-rule
- Making modules responsive based on their container's size
- Using container-relative units
- Using style queries to change styles based on the values of custom properties

We've now looked at organizing CSS into easy-to-reuse modules. This sort of approach makes it possible for large teams to use the same CSS when building large-scale websites and web applications. However, designing a module that can be used anywhere on the page can be difficult in the context of responsive design. You will often find you want to place a module in a narrow column, such as a sidebar. While the viewport itself may be large, the space available for that particular module is not. So a regular @media query is not always a practical approach for making the module responsive.

A new feature of CSS, *container queries*, provides a more versatile option. There are two types of container queries: container size queries and container style queries.

Container size queries allow you to modify the styles of elements on the page based on the width of their *container* element. Container style queries allow you to modify styles based on a custom property of the container.

DEFINITION In the context of container queries, a *container* is a specific ancestor element that contains the element in question. This may be its parent element or be further up the DOM tree. The container element is determined differently for container size queries and container style queries, as you will see shortly.

In this chapter, I'll briefly illustrate this limitation of media queries and when to use container queries as part of your responsive designs. Then we'll dive into some real-world examples and unpack this long-awaited feature of CSS.

10.1 A basic example of a container query

Let's take a closer look at the media module we built in chapter 9 (figure 10.1). You'll add it to a simple two-column page and make the page responsive so the two columns stack above one another in a smaller viewport. This will illustrate how media queries aren't always enough to control the responsive nature of a page, especially when it comes to modular CSS.

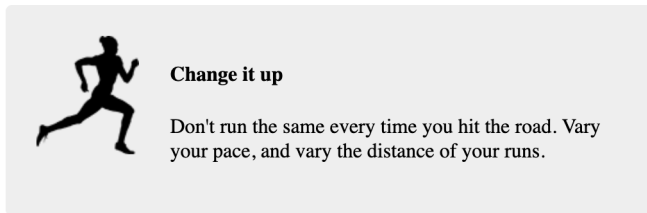


Figure 10.1 A media module as built in chapter 9

Our first listing shows the markup for the page. The media module is placed inside an `<aside>`, which will be a narrow sidebar column on the page.

Listing 10.1 Simple two-column page with a media module

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <link href="style.css" rel="stylesheet" />
  </head>
  <body class="l-page">
    <main>
      <h1>Franklin Running Club</h1>
    </main>
```

Viewport
meta tag

```

<aside>
  <h3>Running tips</h3>
  <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>
</aside>
</body>
</html>

```

Media module

At this point, there is not much content in the main column. You'll add more there later in the chapter. For now, we'll just focus on the media module in the sidebar.

Next, start the stylesheet and add the CSS shown in listing 10.2. At this stage, it uses a media module like the one you built in the previous chapter, with some media queries added to help make it responsive. There is also a media query applied to the page layout as a whole, so the sidebar stacks beneath the `<main>` on small viewports. This technique is essentially the same one you used in chapter 7.

Listing 10.2 Two-column layout and media module

```

@layer reset, layout, modules;

@layer base {
  img {
    display: block;
    max-width: 100%;
  }
}

@layer layout {
  .l-page {
    margin-inline: 1rem;
  }

  @media (min-width: 800px) {
    .l-page {
      display: grid;
      gap: 1rem;
      grid-template-columns: 1fr 30%;
    }
  }
}

@layer modules {
  .media {
    padding: 1.5rem;
  }
}

```

Two-column layout
for large viewports

```

background-color: #eee;
border-radius: 5px;
}

.media_image {
  margin-inline: auto;
}

@media (min-width: 450px) {
  .media {
    display: flex;
    gap: 1.5em;
  }

  .media_image {
    align-self: start;
    margin-inline: revert;
  }
}

```

← Moves text beside the image above 450 px viewport

When you load up the page in the browser, the media module appears in the narrower right column. At very small viewports, the columns are stacked, and the inner contents of the media module are also stacked, so its image appears above its text. This is shown in figure 10.2. I've added `margin-inline: auto` to the image on small breakpoints so it is centered horizontally.

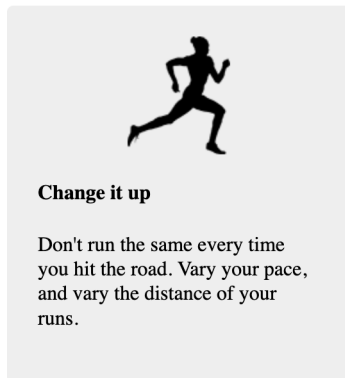


Figure 10.2 Media module stacked responsively on small viewports

In this example, everything works in large viewports, and everything responds nicely in small viewports. However, something less appealing occurs when the viewport is somewhere in the middle. When the viewport is just a little wider 800 px, the page is in the two-column layout, but the sidebar isn't quite wide enough to fit the media module nicely. The media module's text is squashed into a very narrow space, as shown in figure 10.3.

Franklin Running Club

Running tips



Change it up

Don't run the same every time you hit the road. Vary your pace, and vary the distance of your runs.

Figure 10.3 The sidebar isn't wide enough for the media module in this layout.

The problem with our media query approach here is that, while the viewport itself is wide enough to show the media module in a side-by-side mode, we've placed it in a container that is not.

You could in theory add additional media queries to address this, so when the viewport is between 800 px and about 1100 px and the media module is inside a sidebar, it receives the “mobile” stacked layout. But to do this would require quite a bit more code, and it would also require you to be on the alert for other places in the website where a media module might appear in a narrow container. And, importantly, this violates one of the key principles of modular design: you should be able to design your module without concern for the context in which it might be used.

A much better approach to this avoids media queries altogether, instead using container queries. When it comes to a module like this, the size of the viewport is irrelevant. What matters is the size of the container that holds the module.

10.1.1 Using container size queries

Using container size queries requires two steps: defining a *container*, then querying the container with the `@container` at-rule. First I'll walk you through a basic use case to improve the media module, and then we'll take a closer look at some additional options you have when defining containers.

NOTE Container queries are supported in the latest versions of all major browsers as of early 2023, but be aware of users that might put off upgrades when considering whether to use this feature. See <https://caniuse.com/css-container-queries> for recent browser statistics.

The term *container* has long been used in CSS to refer to an ancestor element, usually one that is given some sort of sizing or background color to define a region of the

page. This is still the case in the context of container queries. But in the case of container size queries, we must explicitly define the container by setting the `container-type` property. You may optionally also set a `container-name` on the same element:

```
container-name: layout;
container-type: inline-size;
```

This gives the container a name of `layout` and specifies that you will be querying it based on its inline size or width. We'll take a closer look at the values you can use for these properties shortly, but this is enough to build a basic working example.

Both of these properties can be set at once by using the shorthand `container` property. This is often the easiest way to set both values:

```
container: layout / inline-size;
```

With this container defined on the page, you can use `@container` to query its width and then style any element within the container accordingly. The syntax for this is similar to a media query—for example, `@container layout (min-width: 450px)` or with the range syntax `@container layout (width >= 450px)`. The `layout` here references the container name defined on the container. This query allows you to adapt the styles of an element when the selected container meets the size criteria specified—in this case, if the container is wider than or equal to 450 px.

The following listing shows these in context in the stylesheet. Update your CSS to include these changes.

Listing 10.3 Adding container queries

```
@layer layout {
  .l-page {
    margin-inline: 1rem;
  }

  .l-page > * {
    container: layout / inline-size;
  }

  @media (min-width: 800px) {
    .l-page {
      display: grid;
      gap: 1rem;
      grid-template-columns: 1fr 30%;
    }
  }
}

@layer modules {
  .media {
    padding: 1.5rem;
    background-color: #eee;
  }
}
```

Makes each
column a
container

```

    border-radius: 5px;
  }

  .media__image {
    margin-inline: auto;
  }

  @container layout (width >= 450px) {
    .media {
      display: flex;
      gap: 1.5em;
    }

    .media__image {
      align-self: start;
      margin-inline: revert;
    }
  }
}

```

← Queries the width of the layout container

Resize the browser to see these changes (or use the responsive design mode in your browser’s DevTools). Now the media module effectively has four breakpoints. It’s stacked in a narrow window, horizontal from 450–800 px wide, stacked again until about 1530 px (while the sidebar is narrow), and, finally, horizontal at viewports wider than that.

More importantly, the code for the sidebar isn’t tangled up with the code for the media module. You can make changes to either one and the browser will automatically determine which layout is correct for the module in any given context.

WARNING One important stipulation results from the nature of containers: a container query cannot be used to style the container itself. Selectors inside a container query can only target descendant elements of the container.

As with media queries, you can use the `not` keyword to reverse the query to apply only when the specified condition is false, for example, `@container layout not (width >= 450px)`.

10.2 A closer look at containers

When you defined the previous container, you gave it both a name and a container type. Let’s look a little closer at how these properties behave and what you can do with them. While the previous example was straightforward, there are some particular restrictions you’ll need to understand when working with containers; understanding the ramifications of the container type is particularly important.

10.2.1 Container types

For more than a decade, browser vendors did not think this feature could ever work, primarily because they were concerned about producing an infinite loop. Imagine a scenario where you query for a container with a height less than 300 px; when the query matches, you increase the font size of an element within it to 2 em. This, in

turn, will increase the height of the element, thus the height of its container and the container query may no longer match—so the font size would drop back to its previous value, the query would match again, and so on.

Container queries have been designed in such a way that this sort of infinite loop is not possible. To avoid this type of problem, CSS introduced the concept of *containment*. This provides a means of partially isolating a subsection of the DOM from the rest of the page.

In the context of container queries, this means you cannot use a container query to alter the size of the container being queried—and its size must be explicitly established in a way that does not depend on its child elements.

The specifics of this limitation depend on what value is set for the container type. The `container-type` property can be set to one of three values: `normal`, `inline-size`, and `size`. The initial value is `normal`, which means the element is not a query container, and you can't write a container size query against it.

NOTE You can explicitly control the containment of an element with the `contain` property. This is generally not necessary as setting the `container-type` property applies the necessary containment for you. You can read more about containment at <https://mng.bz/WEg4>.

THE INLINE-SIZE CONTAINER TYPE

Setting the value to `inline-size`, as I have done in examples thus far, allows you to query a container only by its inline size (its width). You cannot use height-based container queries against this type of container. For most practical uses, this is the type of container you will need. The height of the container will still be determined normally based on the height of its contents, and you can adapt your styles based on the width of space available. Nothing inside the container can alter its inline size, but that's how inline sizing works anyway.

In normal document flow, block-level elements will have a width that fills all available space. This provides a known width for container queries to query against. In some other cases, you may need to add additional rules to define the width explicitly. This is notably the case when the container is a flex item; you will need to ensure it has either a `flex-basis` or `flex-grow` value specified, or the container will have a width of zero, and your queries won't work as expected.

THE SIZE CONTAINER TYPE

If you set the container type to `size`, the container will have full-size containment in both inline and block directions. This means the browser cannot determine the height of the container by its children, as they are expected to query against the container's height.

When you use this setting, you need to ensure that the container height is controlled explicitly somehow. This can be done by setting `height` or `min-height` directly or by styles outside the container that establish its height, such as a grid or flexbox. Its height may also be established with the `inset` properties if the container is positioned

with absolute or fixed positioning. When none of these things establish the container's height, it will result in a height of zero, and queries against this height will probably not do what you want.

Generally speaking, you should use a container type of `inline-size` in most cases. When the container in question is part of a grid or flexbox that definitively establishes its height, then it is okay to use a container type of `size` if you want to write container queries against that height, but most responsive design is done in terms of inline size, so this is often not necessary.

10.2.2 Container names

Specifying names for your containers allows you to choose which container you wish to query. You have a lot of freedom in how you go about this. You can give a container a unique name, or you can reuse the same name for multiple containers.

A good approach is to use a common name, such as `layout`, that you apply to most or all containers on the page. In most cases, this will probably be the container you want to query for, since you will want your modules to respond based on the width available to them. Container queries will query against the closest container ancestor they find with a matching name as they look up the DOM (figure 10.4).

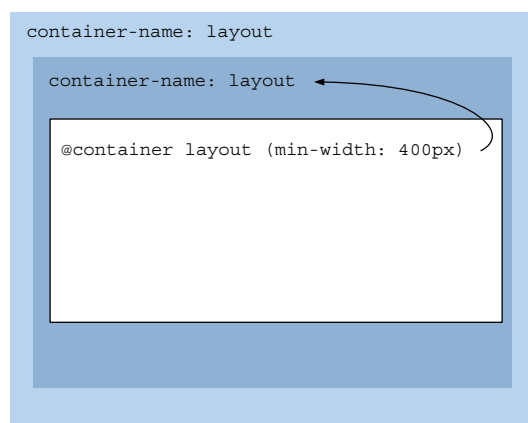


Figure 10.4 A container query evaluates against the nearest container with a matching name.

When you encounter special cases where you need a different container, you can specify a different name. You can even apply multiple names to the same container. For example, `container: layout sidebar / inline-size` applies the names `layout` and `sidebar` both to the container. When you do this, any `@container` query with either name will match against the container.

A container query will look up past any containers that do not have a matching name, as illustrated in figure 10.5. Most often, you will want the nearest container, but you can use this technique when you want to respond to a specific container.

Adding a name to a container is optional. You can apply `container-type: inline-size` to establish a container without a name and omit the name when querying it (for

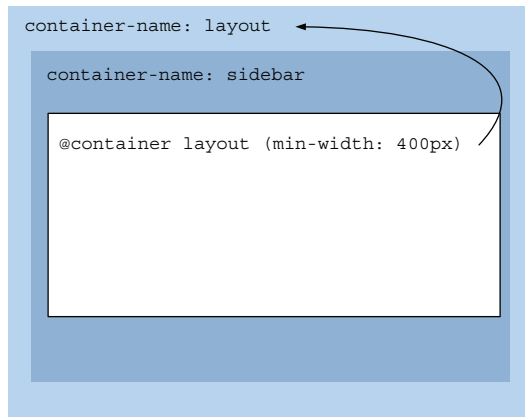


Figure 10.5 A container query ignores containers that don't have a matching name.

example, `@container (min-width: 450px)`). When you use a container query without specifying a name, the browser will look up the DOM tree until it reaches the first container it finds.

NOTE A container name is required when using the `container` shorthand property. To define a container without a name, you must instead use the `container-type` property.

Although you can omit the name from a query (for example, `@container (width > 400px)` is perfectly valid), I recommend getting in the habit of naming all your containers. A query without a name will match the first container it finds as it traverses up the DOM, regardless of any names that container has.

10.2.3 Containers and modular CSS

When it comes to building modular CSS for your site, you should have a strategy for ensuring you define containers in a predictable way so your modules can reliably query against them. Whenever a module has a containing element that may house other modules, I prefer to make that element a container.

In the media module, for example, the `media__body` is meant to hold content, potentially including other modules. So in this case, I would define `media__body` as a container, so that anything placed inside it can respond accordingly.

On your page, update your stylesheet to include the changes shown in listing 10.4. This defines the container. It also adds a `flex-grow` value to the media body to ensure it has a known width.

Listing 10.4 Making the media body a container

```
@layer modules {
  .media {
    padding: 1.5rem;
    background-color: #eee;
  }
}
```

```

    border-radius: 5px;
  }

  .media__image {
    margin-inline: auto;
  }

  .media__body {
    container: layout / inline-size;
    flex-grow: 1;
  }
}

```

Now this module is ready to hold any other modules you might need to nest inside it. You can even nest multiple media modules within one another, and they will each respond just as they need to—this is entirely impractical, but it is fun to see in action. When you take this approach, you’ll also need to ensure any layout styles also define containers, as you’ve already done for the `.l`-page layout (listing 10.3).

One alternative to this approach is to define modules in such a way that they come with their own container. For instance, wrap the media module with a `<div class="media-container">` and style that with `container-type: inline`. However, I don’t like the extra markup this requires for many modules, so this is not an approach I use very often.

When working with modules, container queries are almost universally a better choice than media queries. As long as you consistently define containers throughout your page, you can generally use them for all your modules instead.

10.3 Container-relative units

Container queries come with several container-relative units. These are conceptually similar to the viewport-relative units I covered in chapter 2 except they are defined in terms of the query container. There are six of these units:

- `cqw`—One percent of the container width
- `cqh`—One percent of the container height
- `cqi`—One percent of the container’s inline size
- `cqb`—One percent of the container’s block size
- `cqmin`—The smaller of `cqi` or `cqb`
- `cqmax`—The larger of `cqi` or `cqb`

Just as you cannot query against the block size of an `inline-size` container, you cannot use block-direction units (`cqb` and `cqh`, assuming a horizontal writing mode) for an `inline-size` container. If there is a `size` type container at a further level up the DOM, they will measure against that; otherwise, they will fall back to the behavior of viewport units.

I’ll show you how to use these in some container queries on your page. Your page will have a series of detail cards for some road races. Then you’ll add a grid layout to

the page and set some of the detail cards to be larger than others. This will illustrate some of the ways container-relative units respond to different container sizes. The end result is shown in figure 10.6.

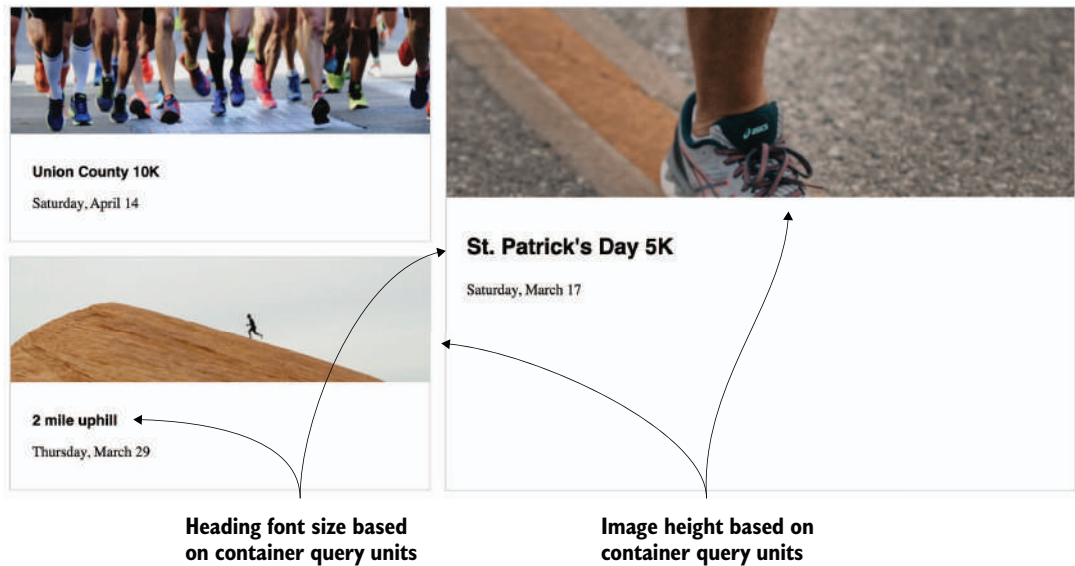


Figure 10.6 Road race detail cards defined using container query units

First, you'll add some markup for these cards to the page you've started. You should have a `<main>` with only an `<h1>` inside it and an `<aside>` after it (listing 10.1). Add the code shown in listing 10.5 to your page so your `<main>` matches the listing. This includes a race-grid made up of some race-detail cards with images. Leave the `<aside>` as it is for now.

Listing 10.5 Markup for race grid and detail cards

```
<main>
  <h1>Franklin Running Club</h1>
  <div class="race-grid">
    <div class="race-detail">
      
      <div class="race-detail__body">
        <h3>Union County 10K</h3>
        <p class="rate-detail__date">Saturday, April 14</p>
      </div>
    </div>
    <div class="race-detail">
      
      <div class="race-detail__body">
        <h3>St. Patrick's Day 5K</h3>
        <p class="rate-detail__date">Saturday, March 17</p>
      </div>
    </div>
  </div>
</main>
```

```

    </div>
  </div>
  <div class="race-detail">
    
    <div class="race-detail__body">
      <h3>2 mile uphill</h3>
      <p class="rate-detail__date">Thursday, March 29</p>
    </div>
  </div>
</div>
</main>

```

As you can see here, each detail has an image and a body. Inside the body is a heading and a date.

Next, you'll add the CSS to lay out the grid and style the cards. These styles are shown in the next listing. This uses `cqi` units for two things: the size of the image and the size of the heading text. Add the following code to your stylesheet, inside your existing modules layer.

Listing 10.6 Responsive styles for detail cards

```

@layer modules {
  .race-grid {
    display: grid;
    grid-template-columns: 2fr 3fr;
    grid-auto-rows: 1fr;
    gap: 1rem;
  }

  .race-grid > :nth-child(3n - 1) {
    grid-row: span 2;
  }

  .race-detail {
    container: layout / inline-size;
    border: 1px solid #ccc;
  }

  .race-detail > img {
    inline-size: 100%;
    max-block-size: 30cqi;
    object-fit: cover;
  }

  .race-detail__body {
    padding: 1rem 1.5rem;
  }

  .race-detail__body > h3 {
    font-size: clamp(1rem, 3.5cqi, 1.8rem);
    font-family: Helvetica, sans-serif;
  }
}

```

Makes every third detail card span two grid rows

Establishes container for the card

Limits image height to 30% of the container width

Allows images to be clipped to prevent stretching

Defines font size as 3.5% of the container width

This code makes use of the `:nth-child()` pseudo-class to target every third grid item, beginning with the second one; this coincides with all grid items in the wider, right-hand column based on the way the grid is defined. This works because `:nth-child(3n)` would target the third item, sixth item, ninth item, and so on. Subtracting 1 from this with `:nth-child(3n - 1)` instead targets the second, fifth, eighth, and so on. See appendix A for more about the `:nth-child()` selector.

The `cqi` units here make the heading size grow directly in proportion to the width of the detail card, equal to 3.5% of the container size, with a 1 rem minimum and 1.8 rem maximum defined in the `clamp()` function. Similarly, the image height is responsive, set to 30% of the container's width. These vary responsively not only between the smaller cards on the left and the larger cards on the right but also in response to the overall viewport width because the grid itself changes size.

Notice that while this code uses container query units, it does not require any actual `@container` queries. A container query is only needed to apply styles conditionally based on a container size, but in this case, the styles should always apply.

10.4 Container style queries

The term *container query* has long been used to refer to the idea of responding to a container based on its size. But there is a second, newer concept of responding to a container based on other aspects of its styles beyond its size. In contrast to container size queries, these are called *container style queries*.

These allow you to apply CSS conditionally based on custom property values. The syntax uses a `style()` function and looks like the following:

```
@container style(--color-theme: dark) {  
  /* conditional styles */  
}
```

When an element inherits the custom property value specified, the selectors inside the code block can select its descendant elements and apply conditional styles to them—in this case, if a `--color-theme` property has the value `dark`.

You do not have to explicitly define a container to use style queries. This is because there is no need for size containment when you are not querying based on the container size.

WARNING Chrome and Edge browsers added support for style queries in March 2023, and as of mid-2024, Firefox and Safari have not yet added support. This feature is still considered experimental and may change slightly in the future. See <https://mng.bz/8wQZ> for the current support information.

With broad browser support still to come, this feature is not yet ready for widespread use except in cases of progressive enhancement. However, it is a very useful tool, and many developers greatly anticipate the ability to use it universally. Let's take a look at how style queries can be beneficial.

10.4.1 Decoupling a module from its container

Sometimes you will want a module to appear slightly differently in a certain context on a page. To illustrate this on your page, you'll use a style query to modify a race detail card when it is placed in a certain context. You will identify this context by setting a `--sidebar` custom property.

Figure 10.7 shows a race detail card that has been added to the right sidebar. In this sidebar, the `--sidebar` property is set, and the detail responds based on that.

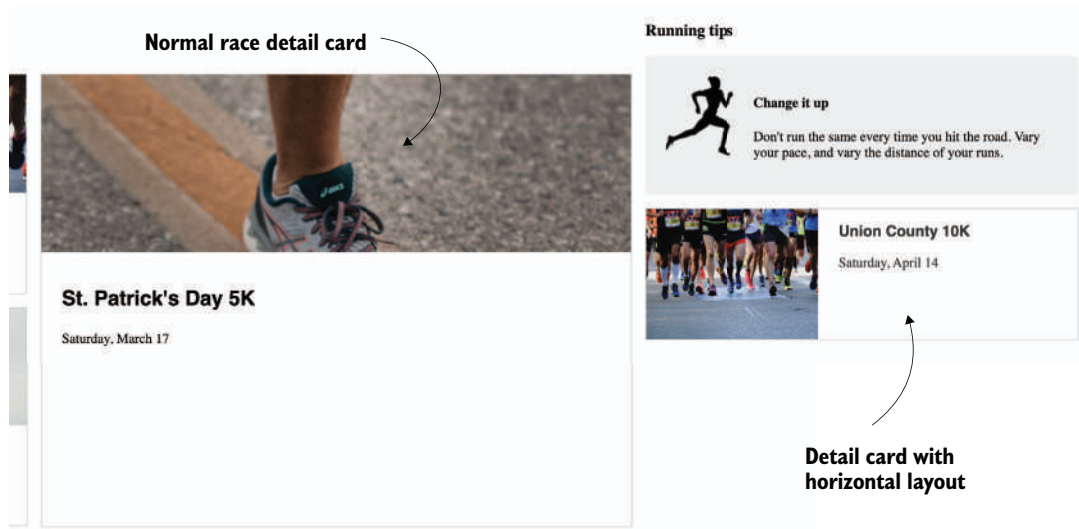


Figure 10.7 Race detail card changed to a horizontal layout in the sidebar

First, copy one of the race-detail cards into your page's `<aside>`. On a real website, you would likely not duplicate it on the same page like this. Instead, it may appear in the sidebar on a separate page, but this will serve to illustrate the concept.

Next, you'll add the CSS to set the `--sidebar` property and then responsively change the layout of the race detail based on that property. Add listing 10.7 to your stylesheet, adding to the layout and modules layers. This queries the `--sidebar` property, only applying the container styles when this property is set to `true`.

Listing 10.7 Using a container style query to change module layout

```
@layer layout {
  .l-page > aside {
    --sidebar: true;
  }
}

@layer modules {
  @container style(--sidebar: true) {
```

← Sets the `--sidebar` property in the sidebar

← Only applies styles when the `--sidebar` property is true

```

.race-detail {
  margin-block: 1em;
  display: flex;
}

.race-detail > img {
  flex: 0 0 30cqi;
}

.race-detail__body > h3 {
  margin-block-start: 0;
}
}
}

```

The code inside the style query uses flexbox to change the card to a horizontal layout, with an image width 30% of the inline size of the container. It also adjusts margins slightly. These layout changes only take effect in the sidebar of the page, based on the `--sidebar` property.

THE BENEFITS OF STYLE QUERIES OVER OTHER APPROACHES

Without style queries, it is often possible to achieve similar results in a couple of ways, but these approaches usually require making some tradeoffs.

You could style the module directly based on its location in the DOM. A selector such as `.l-page > aside .race-detail` would do this. But, as I covered in chapter 9, this violates the principles of modular CSS, because the `race-detail` is now coupled to the `l-page`. Now changes to one might necessitate changes to the other.

Furthermore, other contexts may arise in the future where you want the same behavior. If you had a sidebar in a separate page layout, this selector would no longer work, and you would have to add an additional selector for something like `.l-search-results > .sidebar .race-detail`. That selector specificity is creeping up in an undesirable way.

The second way you could do this is with the introduction of a module variant. You could change the markup in the sidebar to be `<div class="race-detail race-detail--horizontal">` and use the second class to define a variant much like the examples in chapter 9.

There is nothing explicitly wrong with this approach—in some cases, it might be the best solution; for example, when you want the author of the HTML to be able to use different variants as they choose, this provides a way. Sometimes, however, you will want the CSS rather than HTML to make that decision. A style query does exactly that, without coupling the module styles to a specific containing element.

10.4.2 Reducing code duplication

Sometimes you might have styles that you want applied when one of several things occurs. In these instances, style queries provide a sort of single entry point so you don't have to duplicate large blocks of code: you can set a custom property via JavaScript, a media query, or a class name on an element. Then a style query can apply the same styles in all of these circumstances, based on the custom property.

A common situation where this occurs is on sites that allow toggling between light mode and dark mode. Ideally, you will use a `prefers-color-scheme` media query to initially display the page in the correct mode while also allowing the user to toggle the mode via some JavaScript-controlled inputs.

Let's take a look at implementing a dark mode on your page to see how style queries can simplify the process. I'll show you in three steps. First, you'll use a `prefers-color-scheme` media query to set a `--color-theme` custom property. You will be able to override the value of the property if a `data-theme` attribute is present on the `<html>` element. With that in place, you will add some buttons that set the `data-theme` attribute to allow the user to toggle the theme. Finally, you can use style queries where needed to adjust modules for a dark theme. Figure 10.8 shows the final result with dark mode enabled.

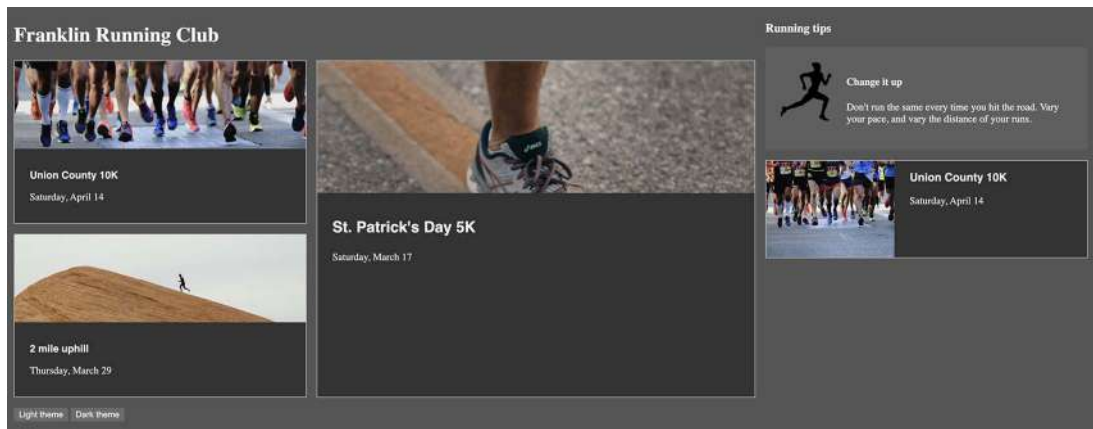


Figure 10.8 Dark mode enabled using style queries

The CSS in listing 10.8 shows the first step. This will enable dark mode based on the user's operating system settings or the presence of a `data-theme` attribute in the HTML. Add this to your stylesheet.

Listing 10.8 Enabling dark mode via a custom property

```
@layer base {
  :root {
    --color-theme: light;
  }
  @media (prefers-color-scheme: dark) {
    :root {
      --color-theme: dark;
    }
  }
}
```

Sets light or dark theme
based on system settings


```

[data-theme="light"] {
  --color-theme: light;
}
[data-theme="dark"] {
  --color-theme: dark;
}

@container style(--color-theme: dark) {
  body {
    background-color: #555;
    color: white;
    color-scheme: dark;
  }
}

```

← Overrides system settings
if the user has explicitly
set a theme value

← Applies dark theme
colors to the page

By using a style query here, you’ve removed the need to duplicate the final ruleset. Because you need to apply these styles both based on a media query and based on an HTML attribute, this is the only way to accomplish these changes without more complicated code. Consider an alternative without style queries. It would look something like this instead (don’t add this to your stylesheet):

```

@layer base {
  body {
    background-color: white;
    color: black;
    color-scheme: light;
  }
  @media (prefers-color-scheme: dark) {
    body {
      background-color: #555;
      color: white;
      color-scheme: dark;
    }
  }
  [data-theme="light"] body {
    background-color: white;
    color: black;
    color-scheme: light;
  }
  [data-theme="dark"] body {
    background-color: #555;
    color: white;
    color-scheme: dark;
  }
}

```

Compared to listing 10.8, this alternative requires a significant amount of repetition for otherwise relatively simple code, all to ensure the correct color theme applies. Furthermore, when you need to make adjustments to the modules of the page based on dark mode—something you’ll do shortly—it would require repeating this pattern

each time. This would make the stylesheet considerably more complicated. Most developers would consider this approach untenable.

Now, with a style query in place to avoid this cumbersome repetition, you can add some buttons to the page that allow the user to select a theme. The HTML and JavaScript for this are shown in the following listing. Add these to your page after the `<aside>` but before the closing `</body>` tag.

Listing 10.9 Controls allowing the user to select light or dark mode

```
<footer>
  <button id="light-theme">Light mode</button>
  <button id="dark-theme">Dark mode</button>
</footer>
<script type="text/javascript">
  const lightThemeButton = document.getElementById("light-theme");
  const darkThemeButton = document.getElementById("dark-theme");

  lightThemeButton.addEventListener("click", () => {
    document.documentElement.setAttribute("data-theme", "light");
  });
  darkThemeButton.addEventListener("click", () => {
    document.documentElement.setAttribute("data-theme", "dark");
  });
</script>
```

These buttons will set the appropriate `data-theme` attributes on the `<html>` element. Combined with the CSS from listing 10.8, this will now allow the user to set the color theme for the page.

The last step is to make some adjustments to the modules on the page as needed when dark mode is enabled. Light mode already works as expected, since that was the default theme you built the page for.

Listing 10.10 shows these changes. Add these to your stylesheet. I have omitted the existing CSS from this listing for brevity, but as a best practice, you should place this code so the styles for the media module are located with other media module styles and the styles for the race-detail module are located with the other race-detail styles. (Be sure both of these style queries appear after the light mode styles so they can override them as expected.)

Listing 10.10 Dark mode adjustments for the modules on the page

```
@layer modules {
  @container style(--color-theme: dark) {
    .media {
      background-color: #5f5f5f;
    }
  }

  @container style(--color-theme: dark) {
    .race-detail {
      background-color: #333;
    }
  }
}
```

```
        color: #eee;
    }
}
```

Now the page's dark mode is complete. When the user initially loads the page, the theme will be set based on their operating system settings, and clicking the light mode or dark mode button will change the theme.

Some of this can be simplified even further, with a carefully planned use of custom properties to define all the relevant background colors and text colors. These can be changed all at once near the beginning of the stylesheet based on a style query. In the next chapter, I'll show you some strategies for managing color variables in this way.

As a further enhancement, you could also add more JavaScript to ensure the setting is remembered for future visits, perhaps using the `localStorage` API. However, I'm keeping the JavaScript at a minimum here to stay within the scope of this book.

Unfortunately, container style queries are not yet supported broadly enough for everyday use. Feel free to use them for small enhancements to the page, and keep an eye on the latest browser support statistics when deciding whether to rely on them for essential features on your page.

Future style queries

In the future, it is possible that browsers may support style queries for CSS values beyond just custom properties. The initial concept included this idea—for example, querying based on font color with `@container style(color: black)`. The scope of the specification was reduced to only custom properties because that seems to meet most practical needs, but style queries could expand to include this someday.

Container size queries have brought a long-awaited improvement to responsive design. Container style queries promise to take that even further. These two features are an important enhancement to the language in the past few years.

It's important to keep browser support in mind when using these features. Container size queries are still very new, and container style queries are still awaiting initial support in some browsers. Use these features to add progressive enhancement, when you are comfortable with the behavior of your fallback styles for users on older browsers. Define as much of your styles outside the `@container` at-rule as possible to help define these fallback behaviors.

Summary

- Container size queries allow styles to respond to the size of their container. This is often a more practical approach than traditional media queries.
- Applying a container type to an element establishes a container. Use `@container` to query the container's size, and style its child elements accordingly.

- Elements inside an `inline-size` or `size` type container may not affect its size in the measured dimension(s).
- Container-relative units size things as a percentage of their container's inline or block size. These are conceptually very similar to viewport-relative units.
- The `style()` function in a container query allows you to apply conditional styles based on the value of a custom property on the container. This is a new feature with emerging browser support.

Part 4

Visual enhancements

By this point in the book, you should have your mental model of CSS established. You should understand how the language works, how to approach styling problems in the browser, and how to avoid common pitfalls. From here, it's time to have some fun. You should be competent in making your pages functional; now it's time to make them look beautiful. 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 these four chapters, we'll look at bringing together everything you've learned so far, and I'll point out some important considerations when it comes to design. These are small details that can have a big impact on the look and feel of your site.

11

Color and contrast

This chapter covers

- Converting a designer mockup into HTML and CSS
- Using contrast to draw attention to the most important parts of a page
- Selecting and manipulating colors
- Understanding color space and the various CSS color notations
- An in-depth look at OKLCH color
- Accessibility concerns with color

An important part of web development involves taking a mockup from a designer and bringing it to life with CSS and HTML. 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 reuse. 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 on 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 this case, it's beneficial to have a basic understanding of design principles so you can implement them yourself.

Throughout the next two chapters, we'll consider a page mockup as a designer might provide and convert it into code. This chapter will focus primarily on the aspects of color. Chapter 12 will focus on typography and spacing. Converting a designer's mockup will provide a useful structure for examining these topics practically. Color itself is a large topic, and much has changed in the past several years.

As we go, I'll highlight some considerations a designer may have made in regard to these things. 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 11.1.

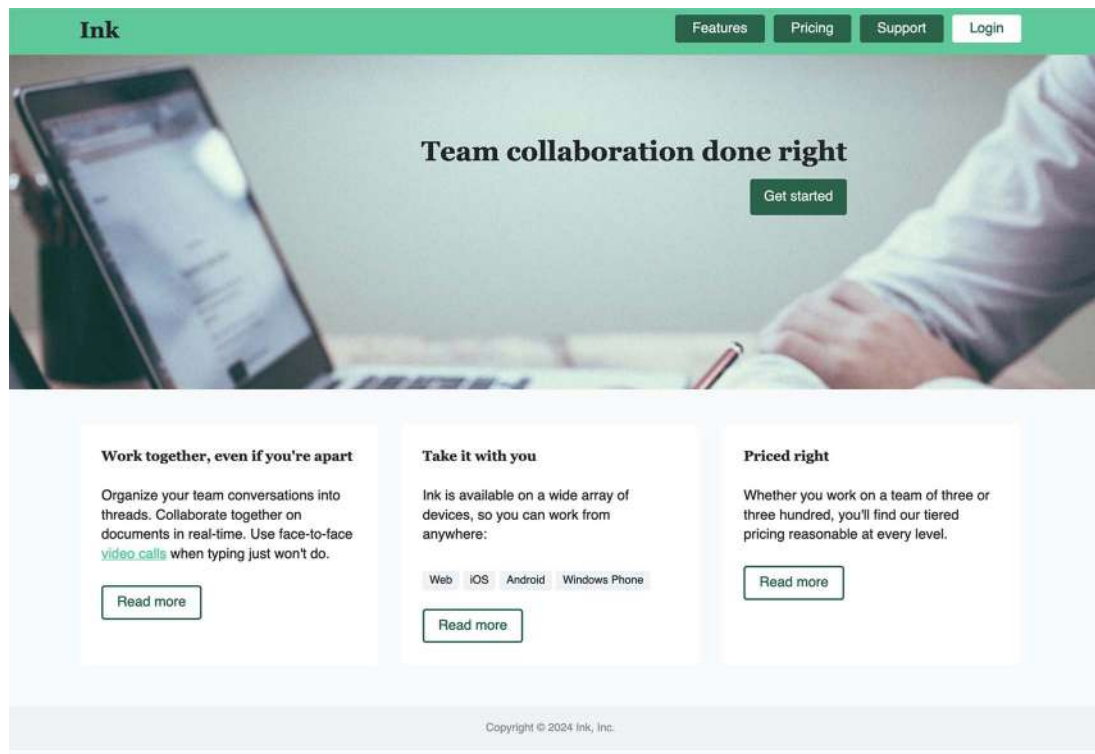


Figure 11.1 The page design for Ink collaboration software

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.

11.1 Communicating with contrast

When you look at the screenshot in figure 11.1, 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—the 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 11.1, 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. There are a few different splashes of color on the page, but they're all the same hue of green, with varying darkness. This is one reason why modular CSS is so important (chapter 9)—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.

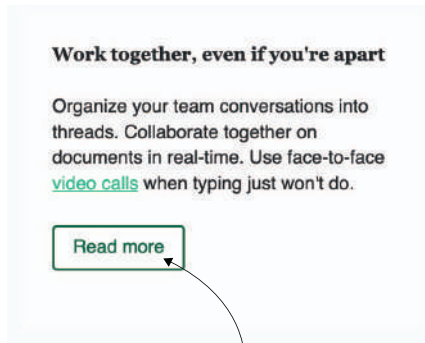
Some key ways you can establish contrast is by using color, spacing, or size. If several items are light but one is dark, you'll notice the dark item first. When one item is surrounded by a lot of unused space (called *whitespace*), that item stands out. And large elements stand out amid a series of 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 whitespace, 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 11.2) and within each of the text columns at the bottom of the page (figure 11.3). These elements are not as strong as the slogan, but they attract attention within their respective regions. The footer, the least important content on the page, is smaller and of lesser contrast.



Login button stands out

Figure 11.2 The lighter Login button draws more attention than the three darker green buttons.



Visual contrast draws attention

Figure 11.3 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, paragraphs of text, ads, 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.

11.1.1 *Establishing patterns*

To establish patterns, your designer may be meticulous about things that seem unimportant to you, such as precise spacing between certain elements or using the same border radius or same box shadow throughout multiple different components; some even care about the spacing between letters and lines of text.

An example mockup is shown in figure 11.4, 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 several small 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 a key aspect of a cohesive design: color choice. (Controlling spacing and 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 11.4. 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.

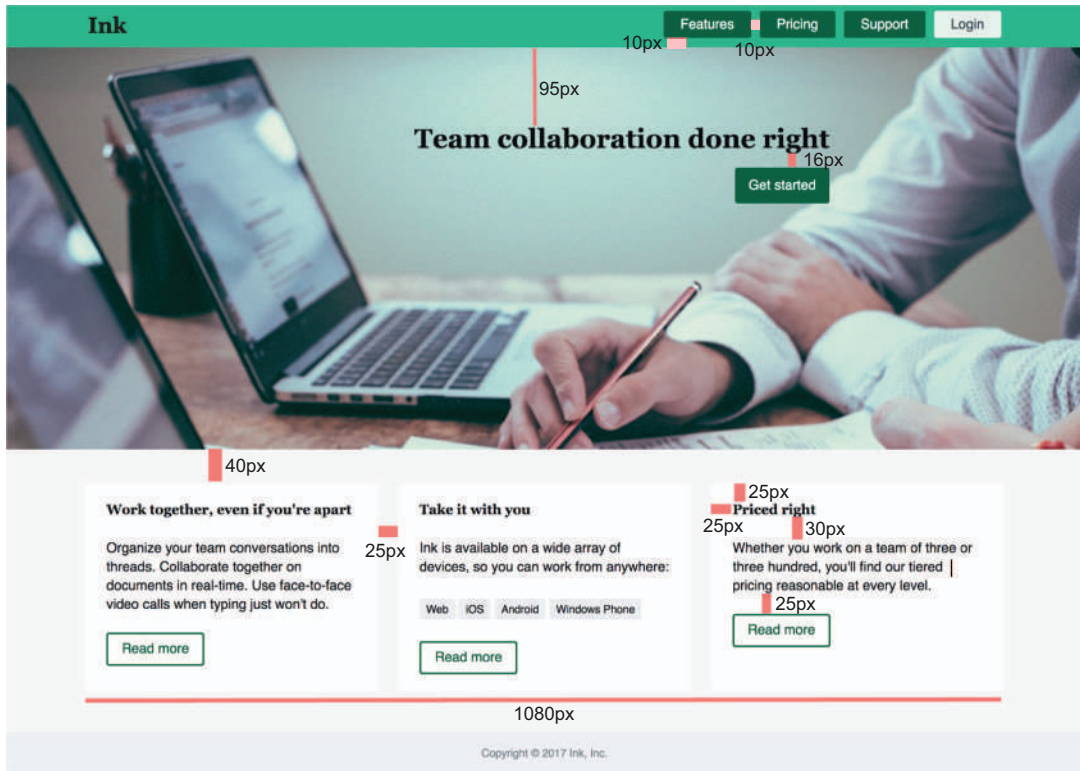


Figure 11.4 A mockup of the page design with annotated measurements

11.1.2 Implementing the design

Create a new page and link it to a new stylesheet. Then copy the markup shown in this first listing. I've broken up the page design into various modules, which you'll style throughout the rest of the chapter.

Listing 11.1 Page markup

```
<!doctype html>
<html lang="en-US">
<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>
```

Top navbar container

```

        <li class="top-nav__featured"><a href="/login">Login</a></li>
    </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; 2024 Ink, Inc.
    </div>
</footer>
</body>
</html>

```

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 subelements of a module,

such as `hero__inner`, and double hyphens indicate variants of a module, such as `button--cta` (see chapter 9 for more on BEM). We'll work our way through these modules. Our first stop is to take a closer look at the colors they use.

11.2 Defining color

When a designer delivers a design, typically you'll get a large document with several sections. A large portion of the documentation will likely consist of full-page mockups similar to that shown in figure 11.4. But before that, a designer will establish some basics. The document 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 will include a color palette for the site.

The color palette may look something like that shown in figure 11.5. 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.

	#076448	Brand green		#868e96	Gray
	#099268	Dark green		#1f3f5	Light gray
	#20c997	Medium green		#f8f9fa	Extra-light gray
	#212529	Text color		#####	White

Figure 11.5 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 your 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 `#####`), 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 custom properties. Otherwise, you'll be typing the values countless times and will, almost certainly, make mistakes.

You'll put together some initial 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 11.6 won't look like much yet, but the colors will start to look closer to the mockup.

- [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 11.6 Page with initial styles and some colors in place

Add the styles in the following listing to your stylesheet.

Listing 11.2 Global styles, including color variables

```
@layer reset, theme, global, layout, modules, utility;
```

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

```
body {  
  margin: unset;  
}
```

```
button,  
input,  
textarea,  
select {  
  font: inherit;  
}
```

```
img,  
picture,  
svg,  
canvas {  
  display: block;  
  max-width: 100%;  
}
```

```
@layer theme {  
  :root {
```

```

--brand-green: #076448;
--dark-green: #099268;
--medium-green: #20c997;
--text-color: #212529;
--gray: #868e96;
--light-gray: #f1f3f5;
--extra-light-gray: #f8f9fa;
--white: #fff;
}
}

@layer global {
  body {
    font-family: Helvetica, Arial, sans-serif;
    line-height: 1.4;
    background-color: var(--extra-light-gray);
    color: var(--text-color);
  }

  h1,
  h2,
  h3,
  h4 {
    font-family: Georgia, serif;
  }

  a {
    color: var(--medium-green);
  }
  a:visited {
    color: var(--brand-green);
  }
  a:hover {
    color: var(--brand-green);
  }
  a:active {
    color: var(--red);
  }
}

```

Assigns each color to a custom property

Sets heading fonts

Uses custom properties to assign color where needed

Placeholder for active links. You'll need to define this red color later.

As in previous chapters, you've used custom properties for these colors. 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 adjust the brand color late in the process. It was trivial to update the variable in one place, but it would have meant updating 100 or more spots in the code without it.

I've also left a placeholder here for active links. You'll come back and define the `--red` color shortly. Before you do that, you'll 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 11.7). We won't fuss much with the spacing yet.

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

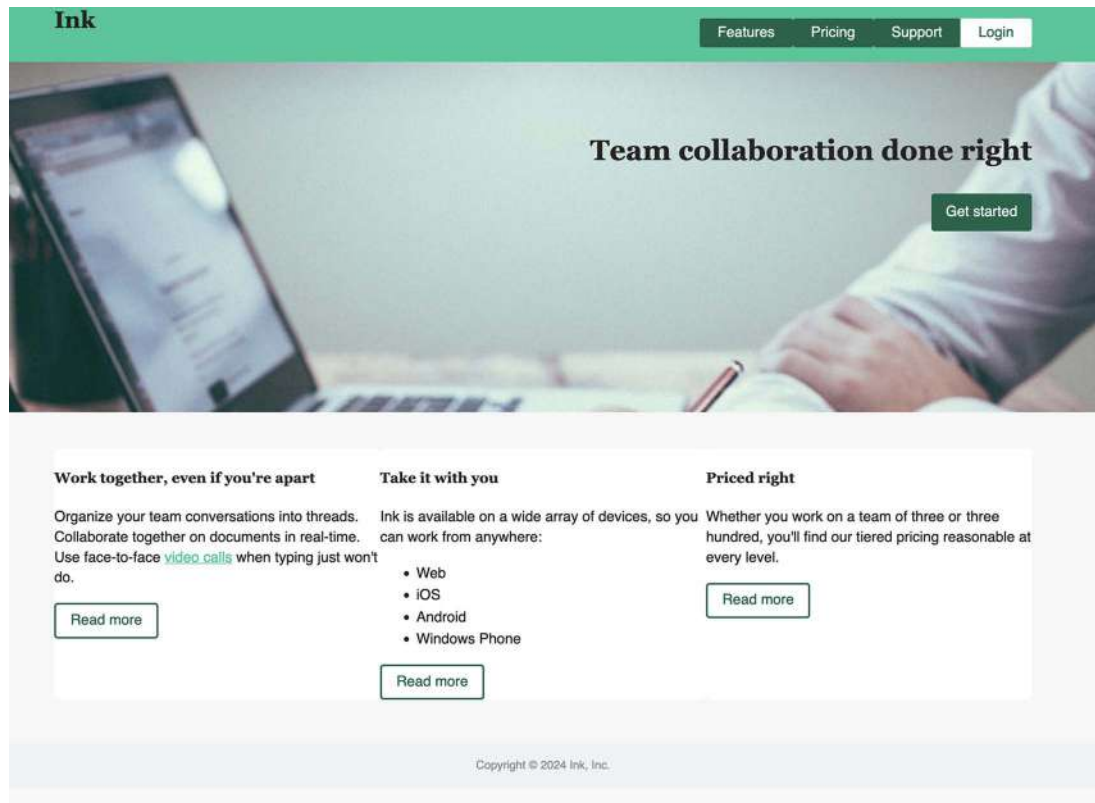


Figure 11.7 Page elements roughly positioned and the initial styles applied

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 11.3 Header styles

```
@layer modules {
  .nav-container {
    background-color: var(--medium-green);
  }
  .nav-container__inner {
    display: flex;
    justify-content: space-between;
    max-inline-size: 1080px;
    margin-inline: auto;
  }
}
```

Centers the contents and
restricts their width to 1080 px


```

.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);
}
}

```

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`. This uses the double-container pattern to center the inner element (see chapter 3 for a review of this technique). This allows the background color to bleed to the edges of the page, while the width of the 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, you'll 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 following listing to your stylesheet.

Listing 11.4 Hero image and button styles

```

@layer modules {
  .hero {
    background: url(collaboration.jpg) no-repeat;
    background-size: cover;
    margin-block-end: 40px;
  }
}

```

```

.hero__inner {
  max-inline-size: 1080px;
  margin-inline: auto;
  padding-block: 50px 200px;
  text-align: end;
}
.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. In this chapter, you'll get everything in about the right place; in the next chapter, I'll circle back and point out key concerns as you more precisely conform to the designer's mockup.

You'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. You 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 11.8.

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 buttons, so you don't need to add anything for those.

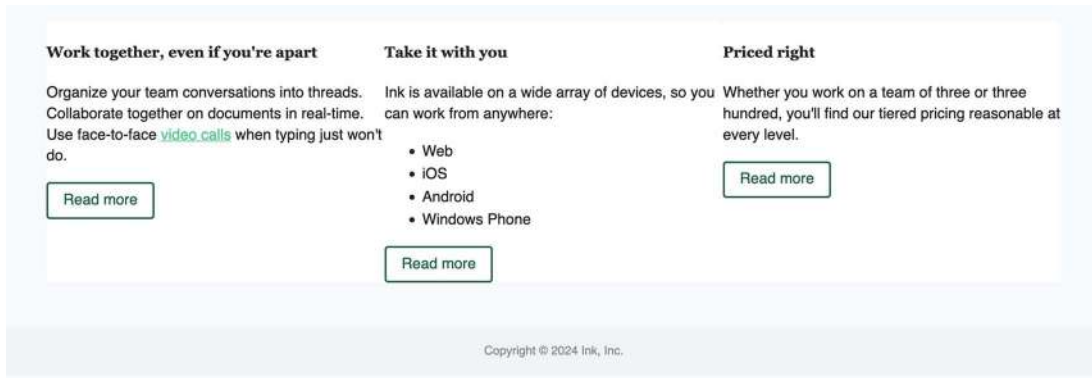


Figure 11.8 Rough styles for the main portion of the page

Listing 11.5 Three columns and tiles

```
.container {
  margin-inline: auto;
  max-inline-size: 1080px;
}

.tile-row {
  display: flex;
}
.tile-row > * {
  flex: 1;
}

.tile {
  background-color: var(--white);
  border-radius: 0.3em;
}

.page-footer {
  margin-block-start: 3em;
  padding-block: 1em;
  background-color: var(--light-gray);
  color: var(--gray);
}
.page-footer__inner {
  margin-inline: auto;
  max-inline-size: 1080px;
  text-align: center;
  font-size: 0.8rem;
}
```

Makes each
column equal
width

Same 1080 px
constrained width as
with other page sections

Again, this listing uses the double-container pattern, restricting the width to 1080 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."

11.2.1 Gamuts and color spaces

The colors in the previous 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 has some limitations.

Hex colors are rather cryptic. It's difficult to see a hex color like #076448 and know at a glance what color that actually represents. Additionally, it can represent colors only within a limited *gamut*.

DEFINITION A *gamut* is a range of color. It's often used to indicate how many colors can be displayed on a device or represented by a particular color notation, such as hex notation.

Color was once a relatively simple topic in CSS, but as monitors and smartphones have improved, they have begun to offer wider gamuts of color than previous generations of hardware. It's helpful to have a basic understanding of this when comparing the various color notations we can use in CSS.

A decade ago, most consumer devices were limited to a gamut of color called Standard RGB, abbreviated sRGB. This gamut includes the range of colors you are familiar with seeing on screens, but it includes only a portion of all colors the human eye can perceive. Higher-end devices now display a wider range of color, a gamut called Display p3. The p3 gamut includes an approximately 25% wider range of colors than sRGB and is available now on most smartphones and high-end laptops, among others. A select few monitors today support an even wider gamut called Rec2020. This gamut covers about 75% of all visible color.

Figure 11.9 shows a common visualization indicating the breadth of color available in each of these gamuts. You'll notice there's a lot of visible color outside the range of sRGB.

Older color notations in CSS can only express colors in the sRGB gamut. Even as hardware displays expanded to support more colors, we have not been able to make use of the wider color range without using newer color syntaxes. This is one of the main reasons CSS has adopted new color notations in the past few years.

TIP You can use a media query to detect the color gamut support of the user's browser and hardware. For example, `@media (color-gamut: p3)` will apply styles only if p3 color is supported. Other supported values are `srgb` and `rec2020`.

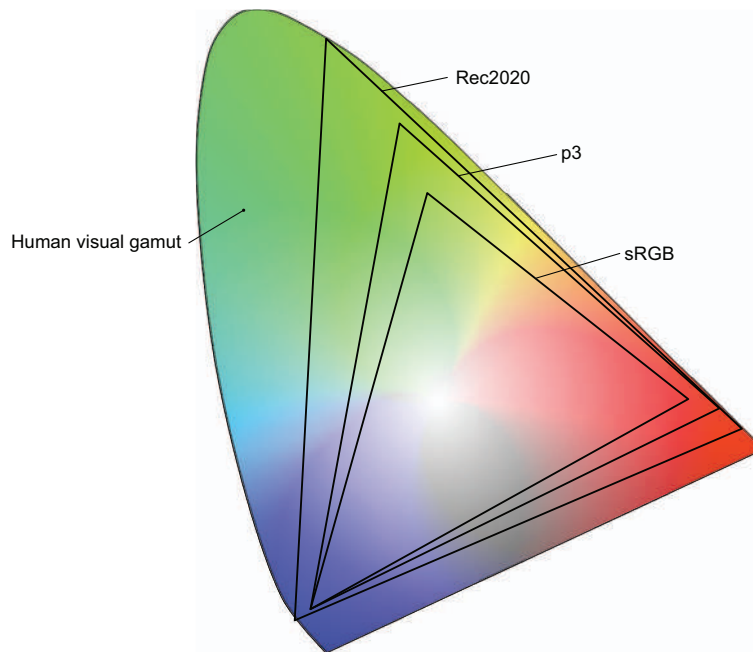


Figure 11.9 Representation of display gamut color ranges

While a gamut indicates a range of colors available, a *color space* is a particular arrangement of that color. This is usually mapped onto a 3D space to help us visualize colors and conceptually navigate them. The same gamut of color can be mapped in different ways onto multiple different color spaces.

Most color spaces use either a rectangular or cylindrical arrangement. Figure 11.10 shows a rectangular RGB color space, where the red, green, and blue values each map to x, y, and z Cartesian coordinates. Figure 11.11 shows a cylindrical HSL color space, where the hue is mapped in a circular arrangement about a central axis using polar coordinates. (These images are provided at https://en.wikipedia.org/wiki/HSL_and_HSV by SharkD. Licensed under CC BY-SA 3.0 license.)

Cylindrical color spaces tend to be more intuitive to work with. In an RGB color space, it's hard to imagine what will happen to an orange color by increasing blue and decreasing green. However, increasing color lightness or rotating to a different hue can be more predictable. Let's take a closer look at the various color spaces and their associated notations in CSS.

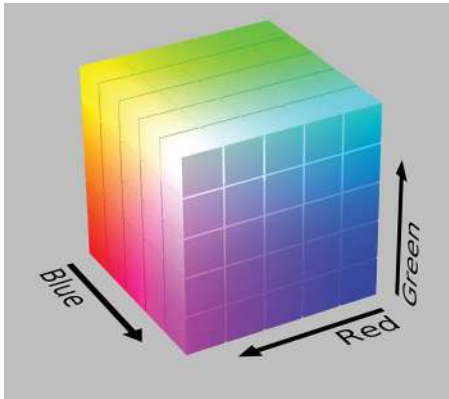


Figure 11.10 RGB rectangular color space

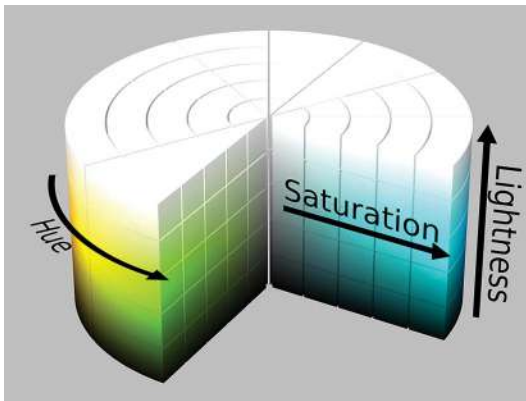


Figure 11.11 HSL polar color space

11.2.2 CSS color notations

CSS now has support for many color notations, including the `rgb()` and `hsl()` functions, which are well supported across browsers.

Many newer notations aren't yet quite as well supported: `hwb()`, `lab()`, `lch()`, `oklab()`, and `oklch()`. These were added to browsers between late 2022 and mid-2023. As of mid-2024, you may want to define fallback values using other notations, though support will continue to rise as users upgrade their browsers. See <https://mng.bz/EZKo> for the latest statistics.

Each of these notations corresponds to a particular color space. Together, these provide an overwhelming number of options. I'll briefly explain how each of these notations works and how they relate to one another. At the end, I'll show several examples of these notations side by side.

My preferred notations are `hsl()`—for the best browser compatibility today—and `oklch()`—the most intuitive notation that will be my go-to in the near future, once browser support is sufficient.

RGB AND HEX COLOR

Hex color notation indicates red, green, and blue values, each as a two-digit hexadecimal value ranging from 00 to FF. Hexadecimal is a base-16 number system, using the letters A–F to represent values ten through fifteen. The color `#80c090` means 80 red, c0 green, and 90 blue (equal to the decimal values 128 red, 192 green, and 144 blue).

A newer eight-digit hex notation includes two additional digits to indicate an *alpha channel* to indicate transparency. This value ranges from 00 (fully transparent) to FF (fully opaque). So `#80c09088` is a semitransparent version of `#80c090`.

NOTE Hex notation can be shortened to a three-digit notation, where each of the three digits is doubled. `#c90`, for instance, is equal to `#cc9900`. There is also a corresponding four-digit notation that expands to the eight-digit notation with an alpha value.

The `rgb()` function is a way to represent the same thing using decimal rather than hexadecimal notation. Instead of 00 through FF, it uses the decimal equivalents 0 through 255: `rgb(0 0 0)` is pure black (equal to `#000`) and `rgb(136 0 0)` is brick red (equal to `#800`).

RGB and hex color are limited to the sRGB gamut. 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.

Hex color is great when you want a succinct notation that's easy to copy and paste between applications. RGB is equally cryptic but lacks the clean brevity. Most other notations are easier to understand and work with.

HSL COLOR

HSL is a type of notation intended to be more human readable. It stands for Hue, Saturation, and Lightness. The syntax looks like `hsl(198deg 73% 46%)`, which is equivalent to the hex color `#2097c9`.

In HSL, the first value, representing hue, is an angle. This is typically between 0 and 359 degrees. This indicates the degrees around the color circle, transitioning through red (0), yellow (60), green (120), cyan (180), blue (240), magenta (300), and back to red. Values outside this range are supported; their equivalent values can be found by adding or subtracting 360-degree increments. It may also be written as a number, without the `deg` unit.

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(198deg 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. HSL color is limited to the sRGB gamut.

A newer, no-comma notation for color functions

When first introduced, the `rgb()` and `hsl()` color functions required commas between the arguments. Instead of `rgb(136 0 0)`, the syntax was `rgb(136, 0, 0)`. For backward compatibility, the comma syntax is still supported for these color functions, but the other, newer color functions only accept a no-comma syntax, so that is the syntax I recommend using. This is well supported in all browsers.

`rgb()` and `hsl()` also have corresponding `rgba()` and `hsla()` functions that accept a fourth argument for an alpha channel for transparency. These functions are no longer necessary, as the newer syntax also supports the alpha argument following a slash. For example, `rgb(136 0 0 / 0.7)` defines a brick red with 70% opacity. All the color functions listed next support an opacity value in this way.

HWB COLOR

HWB stands for Hue, Whiteness, and Blackness. An example color with this notation is `hwb(198deg 12.5% 20.4%)`. This is equivalent to the blue HSL color described earlier.

The hue value in HWB color is equivalent to the hue in HSL; it can also be defined as either an angle or a number indicating a number of degrees.

The W and B values are percentages that define how much white or black to mix into the color. If both values are low, the color will be vivid. `hwb(0deg 100% 0%)` is pure white (the hue is irrelevant in this case), and `hwb(0deg 0% 100%)` is pure black. If W and B add up to 100% or more, the color will be a pure gray, its darkness determined by which of the two values is greater. HWB is limited to the sRGB gamut.

NEXT-GENERATION COLOR SPACES

The next four color notations are for wide gamut color spaces. Instead of being limited to sRGB, or any other gamut, these notations are capable of representing any color the human eye can see. This means they can specify colors that are outside the range of the hardware's capability. In this case, the browser will automatically round these to the closest available representation—as long as you stay within the expected range of values, this will usually result in an acceptable fallback color. You can use

this to specify a p3 color, and users without a p3 display will see the closest sRGB approximation.

Another benefit to these colors is that they strive to be perceptually linear, meaning the values are tailored to align with our human perception. In HSL, a green with a lightness value of 50% will appear a little bit brighter than a blue with a lightness of 50% because our eyes tend to perceive green as brighter. But in these next-generation colors, the colors are balanced so the same lightness value looks the same to our eyes across all hues. The same is true for values that control the vividness of the color.

Two of these notations, *LAB* and *LCH*, were created first, but their specifications have some inconsistencies, especially for some blue and purple hues that don't quite behave as they should. These are superseded by *OKLAB* and *OKLCH*, which are conceptually similar but behave more predictably. I'll explain how all four work, but I recommend sticking with *OKLAB* and *OKLCH*. Browser support for all four is nearly identical.

LAB AND OKLAB COLOR

LAB color and its successor OKLAB are both defined with three values: lightness, A axis, and B axis. The syntax for these looks like `lab(58% -19 -35)` or `oklab(65% -0.06 -0.1)`. While the numerical values for the two are different, they each work conceptually the same way.

The first value, lightness, is a percentage from 0% (black) to 100% (white). In LAB, this can also be represented with a number between 0 and 100, and in OKLAB it can be a number between 0 and 1.0—but this discrepancy can be confusing, so I suggest sticking with percent.

The A axis is a range from red to green, and the B axis is a range from blue to yellow. In OKLAB, these values each range from -0.4 to 0.4, or you can use percentages between -100% and 100% to indicate them. In LAB, these values range from -125 to 125.

These are rectangular color spaces, akin to wide gamut versions of RGB, but, similar to RGB, it's not a very intuitive color space to use.

LCH AND OKLCH COLOR

LCH and OKLCH are cylindrical color spaces like that of HSL. They are defined with three values for Lightness, Chroma, and Hue. The syntax for these looks like `lch(58% 39.8 241.5deg)` and `oklch(64% 0.12 233deg)`.

TIP The name OKLCH is a bit of a mouthful. I have found it easier to remember after web developer Eric Portis coined the term “Oklachroma” as a more friendly way to pronounce it.

The first value defines the perceived lightness. This is a percentage from 0% (black) to 100% (white). It may also be represented as a decimal value from 0 to 1.0. This is

very similar to the lightness in HSL but is more consistent across different hues to align with our perception.

The chroma value indicates how vivid the color is. A value of 0 is pure gray. This is fairly similar to saturation in HSL. Theoretically, this number is unbounded, but for all practical purposes, an OKLCH chroma does not exceed 0.5 (and it does not exceed 0.33 in the sRGB gamut). In LCH, chroma ranges from 0 to 230.

The final value is the hue. This is an angle around the color wheel. This is similar to hue in HSL, but the values have been adjusted. These range from red (30) through yellow (90), green (140), cyan (195), blue (260), magenta (330), and back to red.

Using RGB in wider gamuts with the `color()` function

There is a `color()` function that allows you to specify a gamut and color values. For example, `color(display-p3 0.3 0.59 0.77)` specifies red, green, and blue values in the Display p3 gamut. This color is equivalent to `oklch(64% 0.12 233deg)`.

I find this less practical than the previous notations for general use, as it uses an RGB-based paradigm, not to mention it is rather verbose. And because wide gamut notations such as OKLAB and OKLCH can be used to specify any color in the visible spectrum, they should be future proof if displays expand to support even wider gamuts in the future.

However, the `color()` function supports more obscure gamuts beyond sRGB, Display p3, and Rec2020, so it may be useful if you have a particular need to work within a specific gamut. See <https://mng.bz/IMQ6> for more on using this function.

In summary, each color notation is defined by these parameters:

- RGB—Red, green, blue (each 0–255)
- HSL—Hue (angle), saturation (percent), luminescence (percent)
- HWB—Hue (angle), whiteness (percent), blackness (percent)
- LAB—Lightness (percent), A axis (–125 red to 125 green), B axis (–125 blue to 125 yellow)
- OKLAB—Lightness (percent), A axis (–0.4 red to 0.4 green), B axis (–0.4 blue to 0.4 yellow)
- LCH—Lightness (percent), chroma (0–230), hue (angle)
- OKLCH—Lightness (percent), chroma (0–0.5), hue (angle)

Most of these notations also support the `none` keyword in place of one of their arguments. This can be helpful when one parameter is irrelevant. For example, `oklch(100% 0 0deg)` is white, so the hue value is essentially meaningless. This can instead be expressed as `oklch(100% 0 none)` or even `oklch(100% none none)`.

Figure 11.12 shows a side-by-side comparison of several colors and their various representations in several color spaces; I have rounded some values slightly.

blue #0000ff rgb(0 0 255) hsl(240deg 100% 50%) hwb(240deg 0% 0%) oklab(45% -0.03 -0.31) oklch(45% 0.31 264deg)	lavender #e6e6fa rgb(230 230 250) hsl(240deg 67% 94%) hwb(240deg 90% 2%) oklab(93% 0.01 -0.03) oklch(93% 0.03 286deg)	lightgray #d3d3d3 rgb(211 211 211) hsl(0deg 0% 83%) hwb(0deg 82.75% 17.25%) oklab(87% 0 0) oklch(87% 0 0deg)
gold #ffd700 rgb(255 215 0) hsl(51deg 100% 50%) hwb(51deg 0% 0%) oklab(89% -0.02 0.18) oklch(89% 0.18 95deg)	green #008000 rgb(0 128 0) hsl(120deg 100% 25%) hwb(120deg 0% 50%) oklab(52% -0.14 0.11) oklch(52% 0.18 142.5deg)	tan #d2b48c rgb(210 180 140) hsl(34deg 44% 69%) hwb(34deg 55% 17.5%) oklab(79% 0.02 0.06) oklch(79% 0.06 75deg)

Figure 11.12 Comparison of equivalent color values

These particular colors can also be referenced by the name given (blue, lavender, lightgray, etc). There are about 150 such named colors that are valid CSS values. See <https://mng.bz/NRoN> for a complete list of named colors. For an even deeper dive into gamuts and color spaces, I recommend the High Definition CSS Color Guide by Adam Argyle at <https://mng.bz/Dd0n>.

The best way to get familiar with OKLCH—or any other color space—is to play around with it. I’ll walk you through using it for colors on your page and show you some helpful tools when working with it.

11.3 Using OKLCH to work with color

Next, I’ll walk you through converting the hex colors to other notations. Many online resources will give you a color picker and the ability to convert between color notations. But the easiest place to do conversion is often in your browser DevTools because you always have them on hand. I’ll show you how to do this in Chrome.

You can follow this process to convert to whichever color syntax you prefer. I recommend using HSL if you are concerned about browser compatibility. But I will illustrate this process using OKLCH, as browser support is rapidly growing, and I am optimistic about its relevance in the near future. The process is the same either way.

NOTE As of mid-2024, Firefox DevTools don’t yet support conversion to some of the newer color notations. Screenshots in this section show Chrome DevTools.

With the page loaded, open the browser’s DevTools (press F12). 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 11.13).

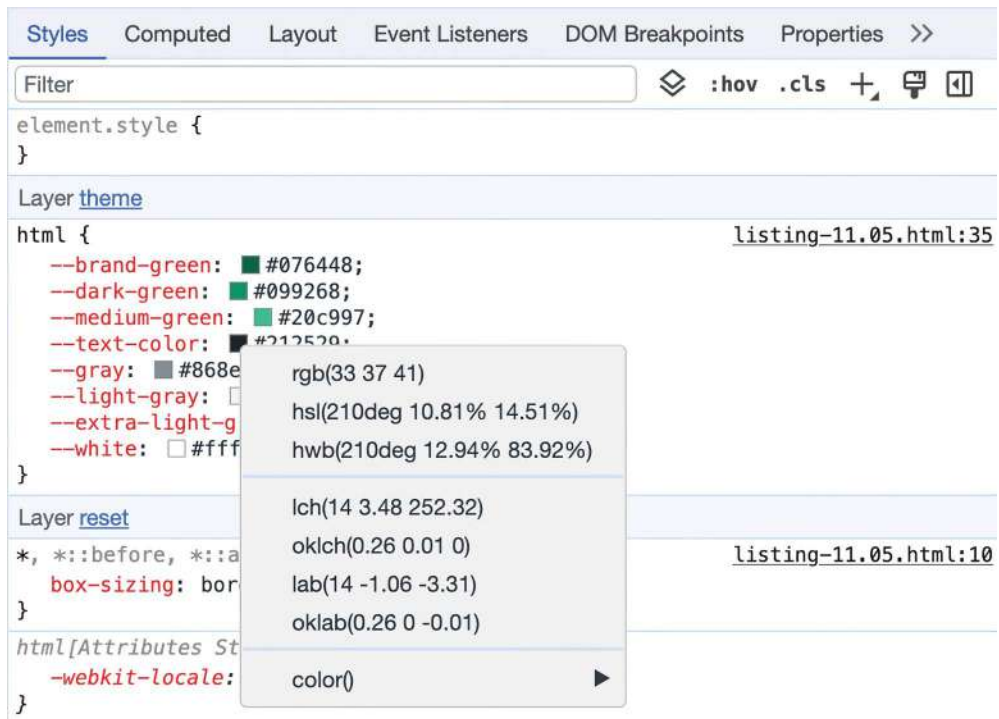


Figure 11.13 The applied colors are shown in the DevTools Styles panel. Shift-click the small square beside a color to switch color notations.

Beside each color is a small indicator displaying an example of the color. If you press and hold Shift and then click on this square, it allows you to convert the hex value to another notation. In Firefox and Safari, it cycles between color notations each time you click. In Chrome, it opens a dropdown allowing you to select a notation.

If you want to dig deeper, clicking the square opens a full color picker dialog (figure 11.14). This allows you to fine-tune colors, select from a palette, or cycle between color notations. It also includes an eyedropper to extract colors from the page. Firefox and Safari offer similar color pickers, although they aren't as full featured.

When the color picker is in a wide gamut color space, it will show a thin white line indicating the limit of the sRGB gamut. Colors to the left of this line are part of sRGB and will display accurately on an sRGB monitor. The more vivid colors to the right are beyond sRGB but part of Display p3—if you use one of these colors, an sRGB monitor will display the closest approximation it can, but it won't be exact.

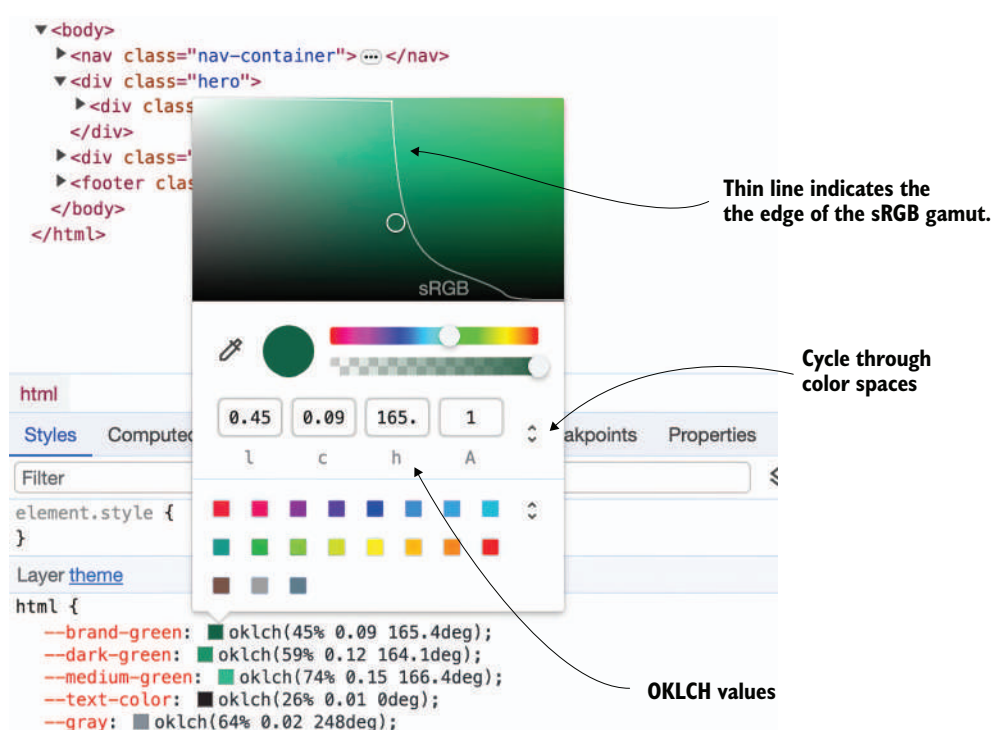


Figure 11.14 Use the color picker dialog to fine-tune colors.

11.3.1 Switching the stylesheet to OKLCH

Often, a hex color is all you need. But converting to HSL or OKLCH can help with fine-tuning colors or finding new colors to add to your site. I'll have you convert your colors to OKLCH and then make some observations about the site's colors.

TIP While many of the values in color notations can be represented as plain numbers without units, I personally prefer to use units when applicable. For example, I will write `oklch(45% 0.09 165.4deg)` rather than `oklch(0.45 0.09 165.4)`, because I find it is a little easier at a glance to interpret which value is which—especially a `deg` unit, which jumps out to me as the hue.

Copy the OKLCH values from your DevTools into your stylesheet, replacing the hex values. These changes are shown in the next listing—I have rounded some of the values for brevity, where there is virtually no discernible difference, and added units for clarity.

Listing 11.6 Converting hex colors to OKLCH

```

@layer theme {
  :root {

```

```

--brand-green: oklch(45% 0.09 165.4deg);
--dark-green: oklch(59% 0.12 164.1deg);
--medium-green: oklch(74% 0.15 166.4deg);
--text-color: oklch(26% 0.01 0deg);
--gray: oklch(64% 0.02 248deg);
--light-gray: oklch(96% 0.003 248deg);
--extra-light-gray: oklch(98% 0 0deg);
--white: oklch(100% 0 0deg);
}

```

Green colors all have similar hues.

Some of the gray colors aren't pure gray.

A couple of things become apparent when the colors are in OKLCH notation. First, you can now see that all three green colors have essentially the same hue. You may not know offhand that 165 is a teal green until you view it 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 OKLCH 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 `oklch(85% 0.12 165deg)`, then fine-tune the lightness and chroma in the browser's DevTools until it looks appropriate.

TIP If you have several similar colors, you can assign individual values, such as the hue angle, to a custom property, then use that in your color definitions—for example, `oklch(45% 0.09 var(--brand-hue))`.

You can also observe that the gray colors aren't pure gray: several have a small bit of chroma. 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 seldom happen in the real world, and our eyes expect to see some color, even if it's subtle.

Again, you don't necessarily need to convert every color to OKLCH in a real-world project. But when the need arises, go for it. Often this makes finding related colors easier.

I will continue to use OKLCH color in examples throughout the rest of the book. For real-world projects, be sure to keep an eye on the latest browser support at https://caniuse.com/mdn-css_types_color_oklch and provide fallback values in an older format until you are comfortable with the support statistics for your target audience.

11.3.2 *Naming color variables*

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. The same is sometimes true of the primary brand color as well. On your page, for instance, you may eventually find you need another green beyond the three you have. 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 lightness. This way, you can

insert another value between two existing ones when needed. It is common now for sites to have a long list of custom properties defining colors, such as `--blue-10` through `--blue-90` and `--gray-10` through `--gray-90`.

On large-scale sites or complex apps, it is common to assign these “raw” color variables in turn to a second set of values, such as a series of numbered foreground and background colors, as in the following listing (you do not need to add this to the page you’ve been building in this chapter).

Listing 11.7 Abstracting raw colors from purpose-based variables

```
:root {
  --gray-10: oklch(10% 0.01 165deg);
  --gray-20: oklch(23% 0.01 165deg);
  --gray-30: oklch(35% 0.01 165deg);
  --gray-40: oklch(42% 0.01 165deg);
  --gray-50: oklch(53% 0.01 165deg);
  --gray-60: oklch(65% 0.01 165deg);
  --gray-70: oklch(78% 0.01 165deg);
  --gray-80: oklch(90% 0 165deg);
  --gray-90: oklch(98% 0 165deg);

  --background-4: var(--gray-90);
  --background-3: var(--gray-80);
  --background-2: var(--gray-70);
  --background-1: var(--gray-60);
  --background-neutral: var(--gray-50);
  --foreground-1: var(--gray-40);
  --foreground-2: var(--gray-30);
  --foreground-3: var(--gray-20);
  --foreground-4: var(--gray-10);
}
```

You may also add variables assigning the colors to borders, shadows, and other reused design elements. Then, throughout the stylesheet, use these secondary variables rather than using the raw color variables. This level of indirection allows you to adjust specific colors in one place and adjust what those colors are used for in another place. If you need to change the common border color, for instance, you can do so without also changing the same gray color that’s used as the background of a dropdown menu.

This is especially useful with a light mode and dark mode, where you can invert the order of the background and foreground colors (perhaps inside a style query, as in chapter 10) while leaving the raw color values the same.

There are countless strategies for handling color variables, and there are no hard and fast rules. I’m always fascinated to look at the CSS on sites by developers and designers I respect; everyone has their own approach. I suggest you try what makes sense to you and adapt as you get a feel for it.

11.3.3 Selecting new colors for the palette

Consider the scenario where you find you 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 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.

11.3.4 Deriving colors from others on the page

The simplest way to find 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 OKLCH or HSL colors, finding a complementary color is easy: add or subtract 180 to the chroma value—but be prepared to adjust this. The 360 degrees of hue in these two systems are spaced a little differently, and often other tools like Adobe's online palette generator (<https://color.adobe.com/create/color-wheel>) will give you yet another result as a complementary color.

Your primary brand green has an OKLCH hue of 165. Adding 180 to this gets us a hue of 345, which is red bordering on magenta. You can also subtract 180 to find its complement, which results in a hue of -15. This is equivalent to hue 345, so `oklch(45% 0.09 -15deg)` will render the same as `oklch(45% 0.09 345deg)`. But I prefer to keep my values between 0 and 360 so they are in more familiar territory.

Now that you have a hue, you need a chroma and lightness. The page's regular link color is medium green—`oklch(74% 0.15 166.4deg)`—so you can start there and adjust things until they look good. The hue of 345 is a little too magenta for my taste, so I'll change it to 10deg. After testing this, it feels a little light to me, so I'll reduce the lightness by about 10%. This gives a color of `oklch(64% 0.15 10deg)`.

I arrived at this color purely by trial and error in my browser DevTools, after adding code similar to listing 11.8 and tweaking it. Of course, if you have the opportunity to work with a designer on color selection, that's even better.

Add this color to the stylesheet. Edit the indicated portion of your code to match listing 11.8. This assigns the color to the custom property `--red`, which is already being referenced by your link styles.

Listing 11.8 Adding the red color to the palette

```
@layer theme {
  :root {
    --brand-green: oklch(45% 0.09 165.4deg);
    --dark-green: oklch(59% 0.12 164.1deg);
    --medium-green: oklch(74% 0.15 166.4deg);
    --text-color: oklch(26% 0.01 0deg);
```



```

--gray: oklch(64% 0.02 248deg);
--light-gray: oklch(96% 0.003 248deg);
--extra-light-gray: oklch(98% 0 0deg);
--white: oklch(100% 0 0deg);

--red: oklch(64% 0.15 10deg);
}

```

← Assigns the red color variable

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 Force State `> :active` (or Change Pseudo-class `> active` in Firefox) from the context menu (figure 11.15). This will force the browser to display the element's active styles.

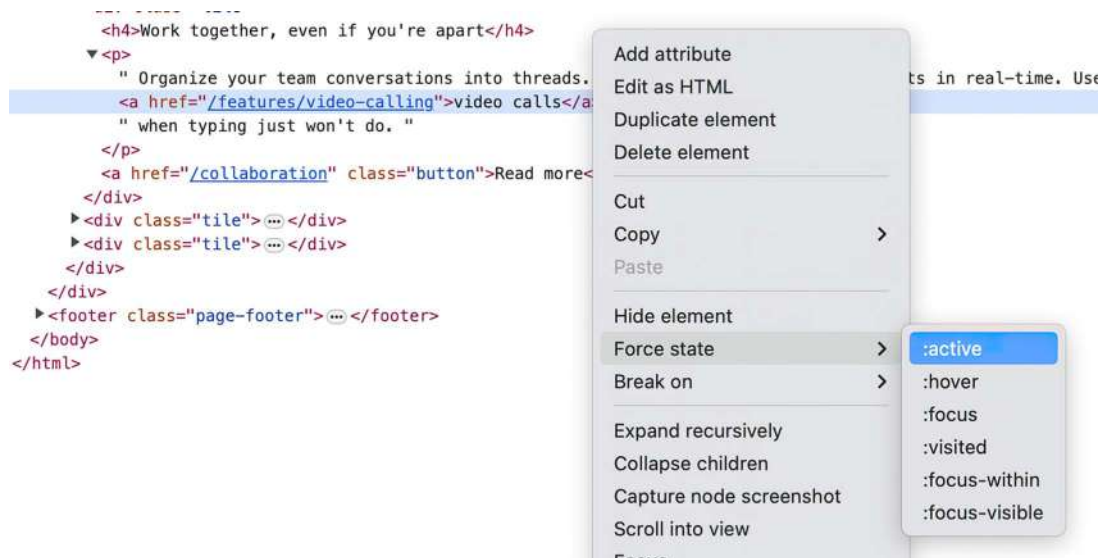


Figure 11.15 DevTools lets you force an element into active, hover, focus, and other pseudo-class states so you can preview how those styles will appear.

With the element forced to an active state, you can see how the red color looks. When needed, you can edit styles live in DevTools and see how it affects this active element.

Choosing colors that look good is never an exact science, but working in a polar color space can help make it easier. Try colors that are complementary to colors already on your page. Play with the lightness and chroma 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/>.

USING A COLOR PICKER

Sometimes you will want a better color picker than the one built into DevTools, especially when you are working on your own design rather than working with a designer. Several are available online. If you are working with OKLCH, I highly recommend trying out <https://oklch.com>. It's invaluable not only for selecting color but for familiarizing yourself further with the OKLCH color space.

A screenshot of this color picker is shown in figure 11.16. You can see the selected color swatch on the top left and sliders for lightness, chroma, hue, and alpha values on the right. The gradients each show a 2D slice of the 3D color space, seen from each primary direction. You can click in these areas or move the sliders to adjust the current color. It also allows you to paste almost any color syntax into the textbox on the left, which will then convert to OKLCH.

This picker is especially helpful for ensuring you select colors that are within the sRGB or Display p3 gamut. If you've used color pickers in the past, you are probably used to seeing a rectangular gradient of color. But OKLCH is a wider color space—far wider than any hardware is capable of displaying, so a rectangular representation of the colors available includes a lot of colors that can't be displayed on your screen. These areas are left blank in the color picker, leaving an irregularly shaped gradient. If you happen to be on a Display p3 or Rec2020 monitor, the limits of the smaller gamuts are indicated by a thin line. These details are annotated in figure 11.17.

If you happen to select a color outside the gamut, the color swatch in the top left of the page will indicate this, with a message like “P3 is unavailable on this monitor.” To the right of this, it will display the closest fallback color, which is what the browser will display if given the out-of-gamut color.

NOTE When you define a gamut color that is very far out of gamut (for example, one with a chroma of 5), the browsers may round this to a color with a shifted hue or lightness. The W3C has defined a carefully crafted algorithm to avoid this, but it is not yet implemented in any browser. Be sure to keep your OKLCH chroma below 0.5 to avoid this.

This tool is useful for understanding the limits of your monitor's gamut and for converting other color notations to OKLCH. You can copy the generated `oklch()` function on the left and paste it into your stylesheets.

MIXING WITH COLOR-MIX()

There are two brand-new features to help with color manipulation. Neither of these has enough browser support in mid-2024 to rely on without using fallback styles, but they will likely be essential in the future.

The first is the `color-mix()` function. This allows you to blend two colors together, producing a third, intermediary color. The function takes three parameters:

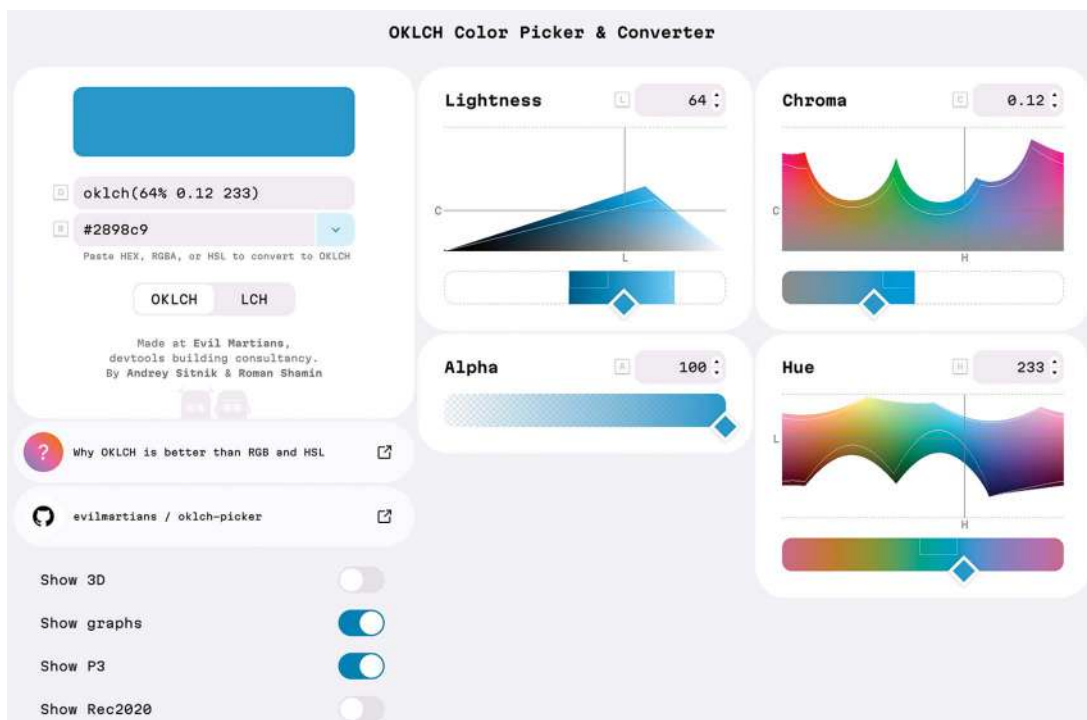


Figure 11.16 The oklch.com color picker

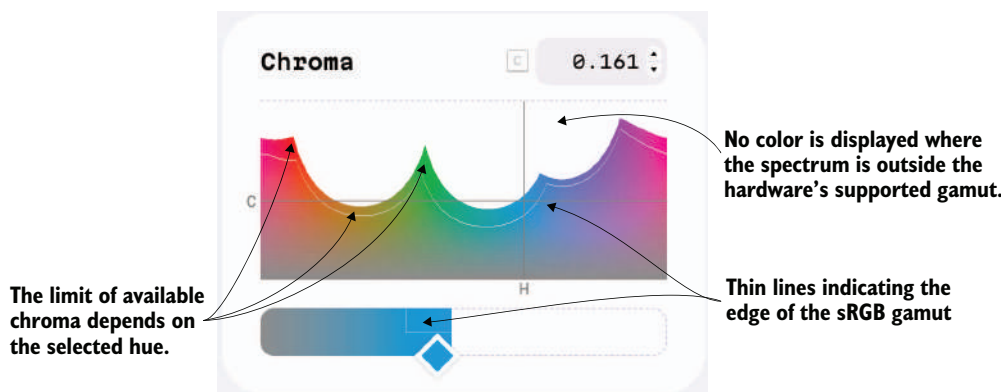


Figure 11.17 The color picker helps you choose colors within the desired gamut.

an *interpolation method* and two colors. For example, `color-mix(in oklch, blue, red)` blends blue and red, resulting in a purple.

The interpolation method determines which color space will be used when mixing colors. This can have a significant effect on the resulting color; the color “in between” two colors depends a lot on how the colors are all arranged spatially.

Available interpolation methods include `srgb`, `srgb-linear`, `lab`, `oklab`, `xyz`, `hwb`, `hsl`, `lch`, and `oklch`. Color interpolation is also an important subject when it comes to color gradients, and we’ll look at this in more depth in chapter 13.

In short, I find that a polar color space such as `hsl` or `oklch` is best when you want to blend two different hues to find an intermediate hue, but `srgb` seems better suited when blending with white or black to produce lighter or darker shades of the same color.

You can also add a percentage to one or both colors to skew the balance in one direction. `color-mix(in srgb, blue 75%, white)` will blend 75% blue with 25% white, producing a shade closer to blue than white. If you specify a percentage for both colors and the two sum to less than 100%, the resulting color will be partially transparent.

TIP Color mixing works with the `currentColor` keyword. `currentColor` is a special keyword that always equals the text color of the current element. You can use it with `color-mix()` to make similar or transparent colors based on the font color. For example, `border-color: color-mix(in srgb, currentColor, white 20%)` will make the border color a lighter version of the text color.

Using `color-mix()` is an excellent way to produce multiple shades of the same color. The following code, for example, defines a spectrum of nine green colors, ranging from dark to light, by mixing with various amounts of white and black:

```
:root {
  --middle-green: oklch(65% 0.15 166.4deg);
  --green-10: color-mix(in srgb, var(--middle-green), black 80%);
  --green-20: color-mix(in srgb, var(--middle-green), black 60%);
  --green-30: color-mix(in srgb, var(--middle-green), black 40%);
  --green-40: color-mix(in srgb, var(--middle-green), black 20%);
  --green-50: var(--middle-green);
  --green-60: color-mix(in srgb, var(--middle-green), white 20%);
  --green-70: color-mix(in srgb, var(--middle-green), white 40%);
  --green-80: color-mix(in srgb, var(--middle-green), white 60%);
  --green-90: color-mix(in srgb, var(--middle-green), white 80%);
}
```

This is supported in the latest version of all major browsers, so you can experiment with it today, though as of mid-2024, enough users are still on older versions of Chrome and Safari that I would avoid its use without fallbacks for a little while longer. See https://caniuse.com/mdn-css_types_color_color-mix for the latest browser support statistics.

DEFINING RELATIVE COLORS

The second color manipulation feature is called *relative color*. This is an expansion of existing color notation functions that allows manipulation of the individual parameters,

based on a given color. This is best explained with an example: the function `oklch(from green calc(1 / 2) c h)` produces a darker version of the color green. It does this by first taking color and breaking it down into its constituent L, C, and H values. By referencing l, c, and h as variables, they can be mathematically manipulated with the `calc()` function. In this case, the L component is reduced by half, producing a darker result.

Similarly, you can manipulate the h, s, and l values in `hsl()`, the r, g, and b values in `rgb()`, and so on. The originating color does not need to be specified in a matching color space. This can be useful if you have a hex color defined but want to manipulate it in another color space, such as OKLCH. The following example takes a light-green accent color and produces an equally vivid blue by manipulating the hue value:

```
--accent-green: #22cf58;
--accent-blue: oklch(from var(--accent-green) l c calc(h + 100deg));
```

I often find myself wishing I could take a color on my page and have a semitransparent version of the same color. This can be done easily with the relative color syntax. The following relative color leaves the l, c, and h values from `--brand-green` unaltered but adds an alpha channel for opacity:

```
--transparent-green: oklch(from var(--brand-green) l c h / 0.6);
```

As of mid-2024, relative color syntax is not supported in Firefox. See <https://caniuse.com/css-relative-colors> for up-to-date statistics.

11.4 Considering contrast for font colors

On your page, you probably noticed the font color is a dark gray and not a true black. In OKLCH notation, it has a lightness value of 26%, 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 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 11.1.

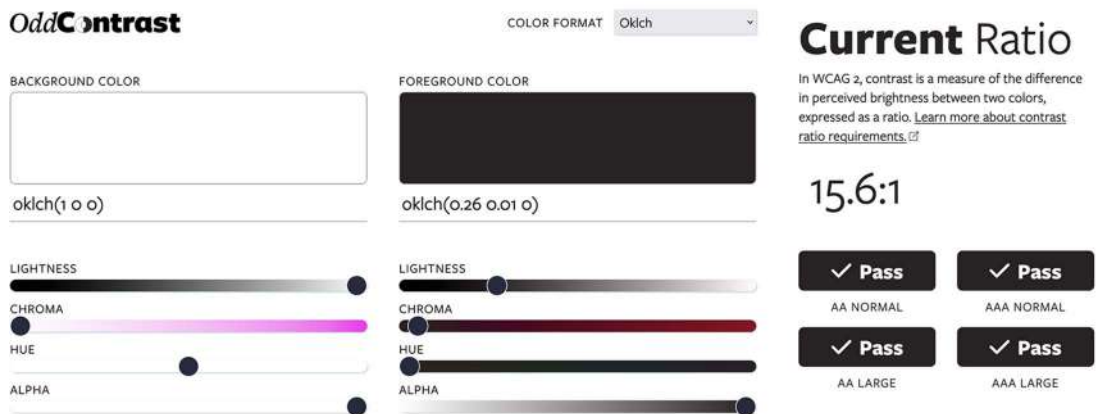
Table 11.1 WCAG text contrast recommendations

	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 specification also provides guidance for contrast and a number of design elements, such as interface components, graphics, and more.

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 that does it for me.

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 <https://www.oddcontrast.com>. This checker supports all the CSS color formats. Paste in your background color and your text color, and it shows you the computed contrast ratio (it will even convert the values to the currently selected format). The result is shown in figure 11.18.

**Figure 11.18** The background and text colors have a contrast ratio of 15.6:1.

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 this 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 11.19 shows two copies of the same paragraph. Even though the colors are the same in each, the perceived contrast is different.

**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.**

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.

400 (normal) font weight

100 (thin) font weight

Figure 11.19 A thin font face results in less visual contrast, even with the same color.

The paragraphs in figure 11.19 are both set using the Helvetica Neue typeface. The one on the left has a font weight of 400 (or normal); the one on 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.

Keep the page you began building in this chapter. In the following chapter, I'll show you how to finish building it with an eye on precise spacing between elements of the design and adding in web fonts.

Summary

- Visual contrast can be used to draw the user's attention to important parts of the page.
- Modern color syntaxes like `oklch()` and `oklab()` allow you to define wide gamut colors beyond those available with `hex`, `rgb()`, or `hsl()`.
- The browser will round out-of-gamut colors to the nearest available color on the user's hardware.
- HSL and OKLCH are cylindrical color spaces, which make working with colors easier and more understandable at a glance.
- The `color-mix()` function mixes two colors together, resulting in a third intermediary color.
- Using a consistent strategy for assigning color values to custom properties makes it easier to use those colors throughout the site and add new colors to the palette when needed.

12

Typography and spacing

This chapter covers

- Fine-tuning spacing between elements
- Using web fonts to give your page a unique feel
- Using the Google Fonts API
- Tuning font spacing: tracking, leading, and line height
- Web font performance concerns and optimizations
- Using variable fonts

In this chapter, you'll continue building the page you started in the previous chapter. You should now have the colors defined and the page layout roughly in place. Next, you'll look specifically at fine-tuning the spacing of elements on the page and replacing the fonts with more visually interesting web fonts. These may sound like two separate topics, but they interact with one another in some key ways. CSS includes a number of properties that control the layout, spacing, and size of fonts, so it will be beneficial to look at them and some other general spacing techniques together.

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.

12.1 Spacing

If you followed along in the last chapter, you should already have a page built, with colors and a rough layout. The HTML for this page is shown in listing 11.1, and the CSS was built up in listings throughout the rest of chapter 11, so your page should already match the listings from that chapter. Otherwise, you can copy listing 11.08.html from the example code repository (<https://github.com/CSSInDepth/css-in-depth-2/blob/main/ch11/listing-11.08.html>).

Next, you'll turn your attention to the precise spacing between elements, as provided by a designer. Depending on which tools the designer uses, this can be a tedious part of the development process. Modern design tools tend to simplify translation to the web, but 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 affect your vertical spacing.

12.1.1 Using *ems* vs. *px*

One important decision you'll have to consider 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 12.1). 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.

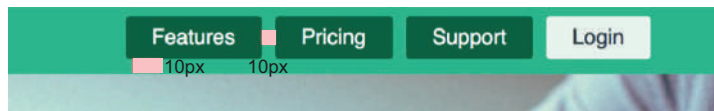


Figure 12.1 The specification calls for 10 *px* around and between each navbar item.

Throughout chapter 2, I discussed the benefits of using relative units, so it is worth considering which measurements you want to use relative units for and for which you will use *pixels*. You can be dogmatic about this and convert all of these distances to *ems* or *rems*, or you can be a little more practical and make a judgment call on a case-by-case basis for each design element of the page. If you think the space should resize when the user has a different font size setting, use *ems* or *rems*. Otherwise, *px* are probably fine.

In practice, this generally leads me to favor ems for smaller measurements, especially those immediately surrounding text or buttons. Larger gaps or space between boxes don't benefit as much from this responsive behavior, so I often leave those as px.

As I lead you through the spacing on this page, I'll point out my reasoning for choosing one or the other, but keep in mind this is a bit subjective, and there isn't always a single right answer.

In the design specification, the spacing measurements in the navbar call for 10 px around the menu items (figure 12.1). Because these are small design elements, primarily text with a little padding, I'll space these using ems.

Because the base font size is 16 px, you can do the math to calculate the em value, dividing the desired length by the base font size: 10 divided by 16 is 0.625, so the distances here will be 0.625 em. Now you're ready to add the declarations indicated here to your stylesheet. The relevant portions of the stylesheet are shown in the following listing, with the changes indicated in bold.

Listing 12.1 Using padding and margin to set navbar spacing

```
.nav-container {
  background-color: var(--medium-green);
}
.nav-container__inner {
  display: flex;
  justify-content: space-between;
  max-inline-size: 1080px;
  margin-inline: auto;
  padding: 0.625em 0;
}
/* ... */

.top-nav {
  display: flex;
  list-style-type: none;
  margin: unset;
  gap: 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. I used a flexbox gap for the horizontal spacing between nav items because I wanted each item to have the space between them, but this could be done with margins as well.

Next, you can look to the space below the hero image and between each of the three columns. Figure 12.2 shows these spacing measurements as given in the designer mock-ups. 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.

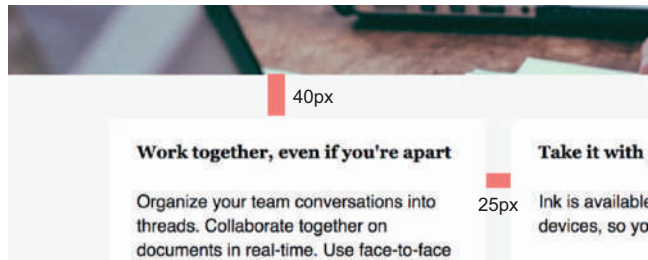


Figure 12.2 The page margins below the hero image (40 px) and between the columns (25 px)

In this case, I don’t see any reason these gaps need to scale for larger fonts, so I will keep them in px. These measurements are shown in listing 12.2. You already specified the 40 px margin beneath the hero image in the last chapter, but I’ve repeated it here for clarity. Add the `tile-row gap` highlighted in the following listing to your stylesheet.

Listing 12.2 Adding margins below the hero image and between the columns

```
.hero {
  background: url(collaboration.jpg) no-repeat;
  background-size: cover;
  margin-block-end: 40px;
}

/* ... */

.tile-row {
  display: flex;
  gap: 25px;
}
.tile-row > * {
  flex: 1;
}
```

← 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.

12.1.2 Factoring in line height

The mockup also defines spaces around the text. Figure 12.3 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.)

Applying the 25 px space around the text edges is a matter of adding padding to the tiles. I will choose to use ems here, because I would want this to respond to larger fonts to give “breathing room” for readability. Calculating $25/16$ gives 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

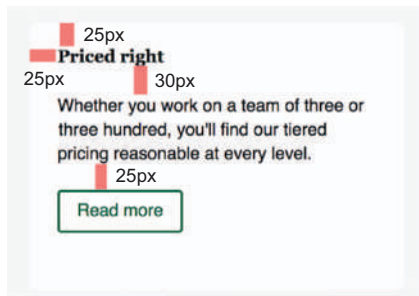


Figure 12.3 Desired spacing in the tile and around the text

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 these elements, 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 12.4. The text is 1 em high, but the line height extends a little further above and below the edges of the text.

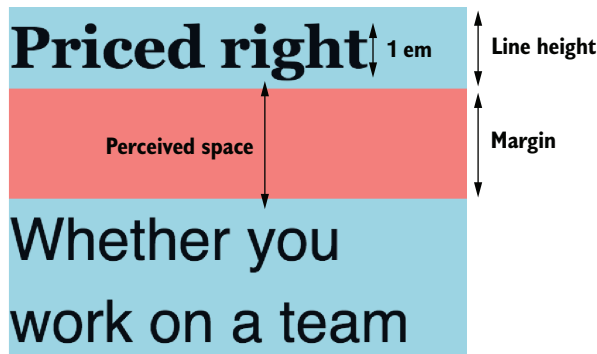


Figure 12.4 The line height defines the height of the content box.

On your page, you have a line height of 1.4. This is applied to the `<body>` element and inherited from there. Thus, an element with one line of text has a content 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 the heading and paragraph have the same line height and font size.) This produces a perceived space between the two of 36.4 px.

NOTE Print 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.

A designer won't usually fuss over a 1 or 2 pixel discrepancy, but an extra 6½ pixels might bother them. The difference will be even larger if you have a bigger line height or if 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 the spacings from the following listing to the relevant portions of your stylesheet.

Listing 12.3 Setting the tile and paragraph spacing

```
@layer global {
  p {
    margin-block: 1.5em;
  }
}

/* ... */

@layer modules {
  .tile {
    background-color: var(--white);
    border-radius: 0.3em;
    padding: 1.5625em;
  }
  .tile > h4 {
    margin-block: 0 1.5em;
  }
}
```

Adds margins to paragraphs

Adds padding inside the tiles

Adds the margin beneath the tile headings

You've applied the 1.5 em margins to the global styles, so all paragraphs will have the same spacing throughout the page. You repeated this measurement below the 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.

TIP An upcoming CSS property called `leading-trim` will be able to remove this extra space from the top and bottom of the content box, but this is not enabled in any browsers yet. You can read more about it at <https://mng.bz/67PR>.

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 12.5.

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.

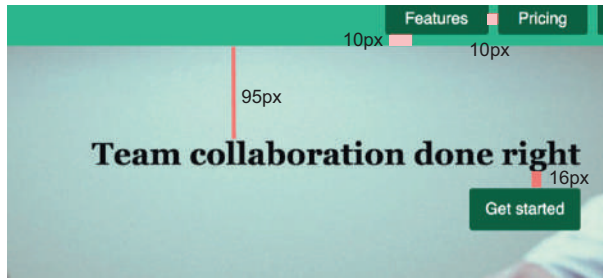


Figure 12.5 The design calls for 95 px above the slogan and 16 px beneath it (between it and the button).

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 add them to the stylesheet. Update the portion of the stylesheet shown in listing 12.4, adding or updating the bold declarations to refine these spacings. I've also included a minimum height on the `hero__inner` to ensure it fills a large portion of the screen, as shown in the mockup.

Listing 12.4 Positioning the slogan and button within the hero image

```
.hero {
  background: url(collaboration.jpg) no-repeat;
  background-size: cover;
  margin-bottom: 40px;
}
.hero__inner {
  max-inline-size: 1080px;
  margin-inline: auto;
  min-block-size: 40svh;
  padding-block: 89px;
  padding-inline-end: 12.5em;
  text-align: right;
}
.hero h2 {
  margin-block: 0 10px;
  font-size: 1.95rem;
}
```

← Replaces the rough estimate with the newly calculated spacing

← Defines the space between the slogan and the button

The top padding of the `hero__inner` provides the spacing above the slogan. You've included padding on the right edge, though the specific values weren't specified in the mockup. You then set the slogan's top margin to zero, so it won't add more space inside the `hero__inner`'s padding.

12.1.3 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 12.6). I've left this as a regular unordered list until now. You'll get these laid out inline as they appear in the mockup.

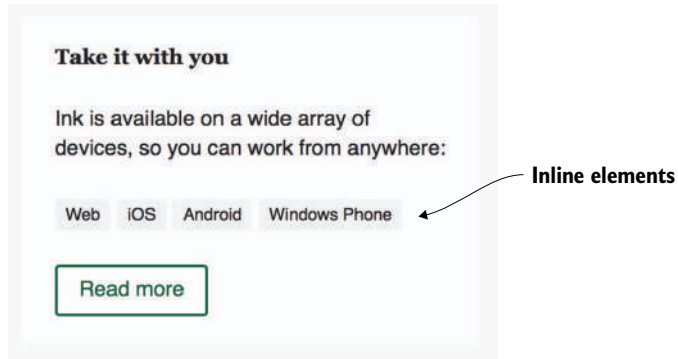


Figure 12.6 The list items need to be styled and made inline.

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 several flexbox layouts throughout the book, so I want to take a look at the concerns when using inline elements.

Many 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 12.7, which could occur at certain viewport widths or if the content changes down the road.

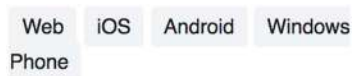


Figure 12.7 The list items overlap when they line wrap.

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. Listing 12.5 shows the CSS for a tag-list module, which defines styles for these tags. Add this code to your stylesheet in the modules layer. Go ahead and edit the line height to different values to see the effect it has.

Listing 12.5 Styling the tags

```
@layer modules {  
  .tag-list {  
    list-style: none;  
    padding-inline-start: unset;  
  }  
  .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 whitespace 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 as shown in the mockups in the last chapter. Your page should now match the full mockup as shown in figure 12.8.

We spent a lot of time focusing on very particular details. Many developers won't pay this much attention to detail when implementing a design, but for those who do, it pays off. These details are often 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 overuse color, but selectively put it in the places where you want to draw attention. Establish consistent patterns, then break those patterns to draw the user's eye to the most important things on the page.

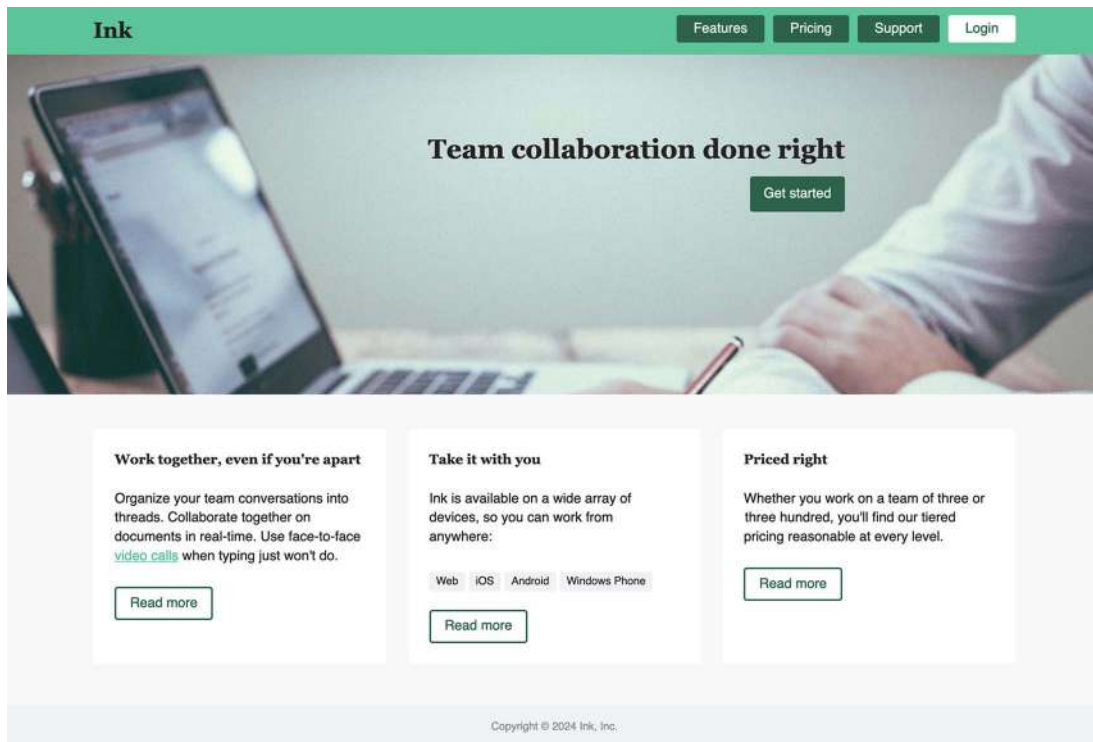


Figure 12.8 Fully designed page

12.2 Web fonts

Next, you'll take the page a step further by adding custom fonts. Up to this point, I haven't included these in the page mockups for the page you've been building, but a designer will almost certainly specify web fonts in their design.

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. Early 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 Helvetia 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 decision making.

You can use fonts that make a page feel playful or serious, trustworthy or informal. Look at the font examples in figure 12.9. 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.

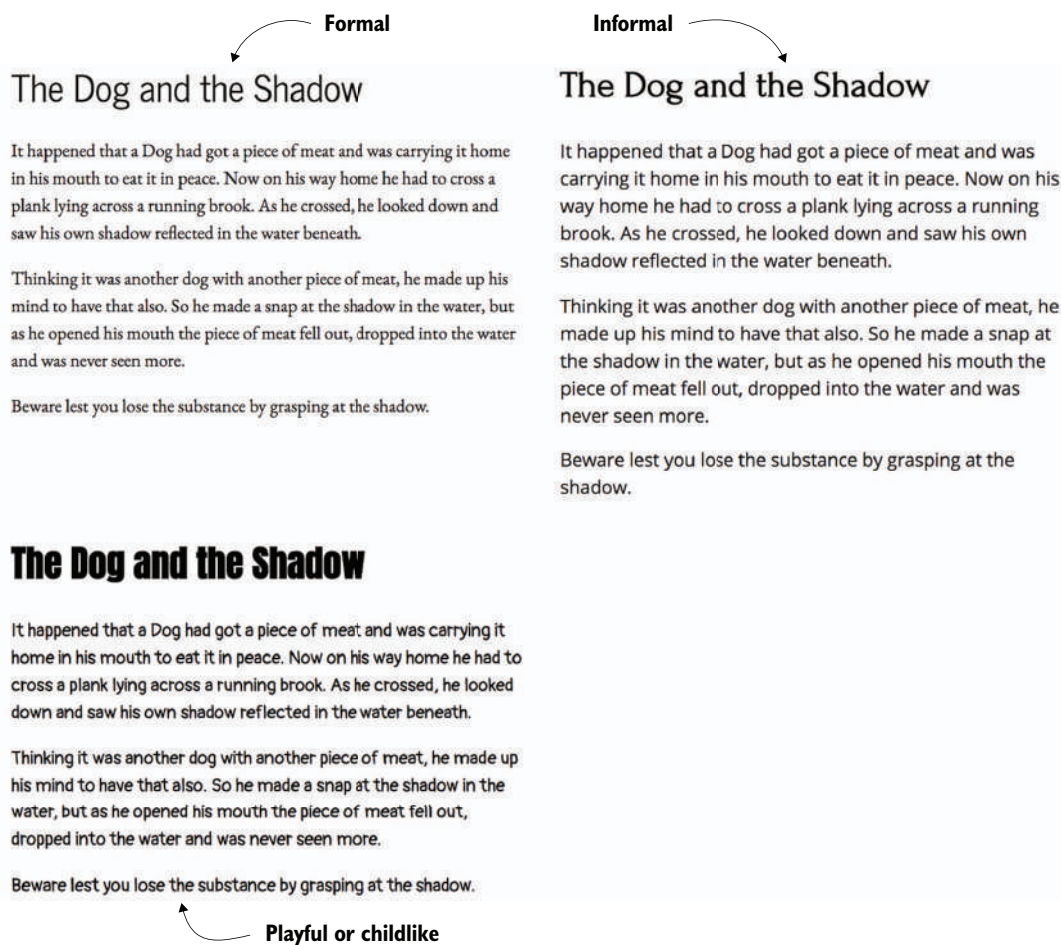


Figure 12.9 The fonts you use can have a marked effect on the feel of your site.

The easiest and perhaps most common way to use web fonts is through an online service. Common ones are Google Fonts (<https://fonts.google.com/>) and Adobe Fonts (<https://fonts.adobe.com/>).

Whether paid or free, these services take care of many concerns for you, including both technical (hosting) and legal (licensing) factors. They each offer a large library of typefaces to choose from. Sometimes, if you need a particular typeface, it might only be available for purchase, requiring you to host it yourself.

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 and how you can host your own fonts.

You'll take the page you built and enhance it with web fonts to improve the design. Afterward, the page will render as shown in figure 12.10. The Roboto font is the main body font used throughout most of the page, and Sansita is the font used in the headings.



Figure 12.10 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.

DEFINITION A *serif* is 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).

12.3 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 12.11). You can select fonts from this screen, or you can search for a particular font using the search bar at the top of the page or the filters on the left.

Clicking one of the fonts will show more information about it, including a list of the available weights (thin, regular, bold) and styles (regular or italics). On the top right of the screen is a Get Font button. Clicking this button adds it to your selected fonts. You can preview your selected fonts by clicking the small shopping bag icon in the top right.

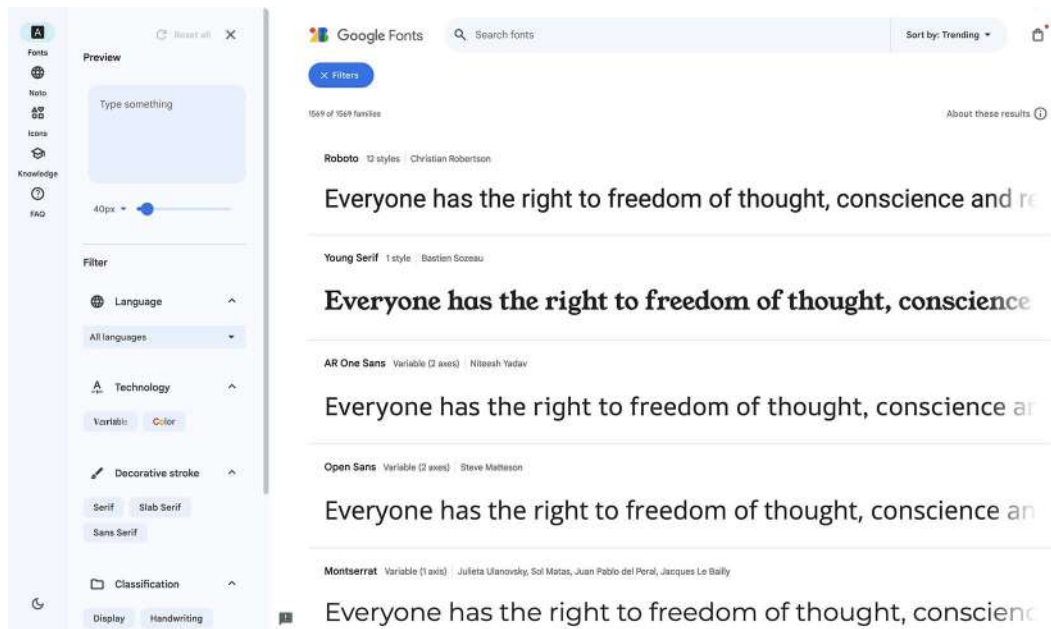


Figure 12.11 Font selection interface for Google Fonts

Because you know the fonts you want, you can search for them by name. In the search bar, type *Sansita*; this filters out the other fonts from the main view. Click the *Sansita* font to view its page. Click the *Get Font* button to add it to your selected fonts. This will add all font weights and styles; you will be able to refine selected styles shortly.

Next, navigate back to the Google Fonts homepage (click the Google Fonts icon at the top). In the search box, type *Roboto*. Google will pull up several typefaces, including *Roboto*, *Roboto Condensed*, and *Roboto Slab*. Click *Roboto* and add it to your selected fonts.

Now, the page will say “2 font families selected.” On the right, there are two buttons. The *Get Embed Code* button will provide you with the necessary CSS for embedding these fonts on your page (figure 12.12), while the *Download All* button will allow you to download the selected font files. Click the *Get Embed Code* button.

By now, you’re familiar with regular and bold font weights, but some typefaces support several different weights. For example, *Roboto* comes in six different weights, ranging from thin to black, as well as an italic variant of each. Notice that apart from names like thin, regular, bold, and black, weights can also be referred to by numerical values like 100, 400, 700, and 900.

NOTE The terms *typeface* and *font* are often conflated. *Typeface* refers to an entire family of fonts (such as *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*.

← Embed code

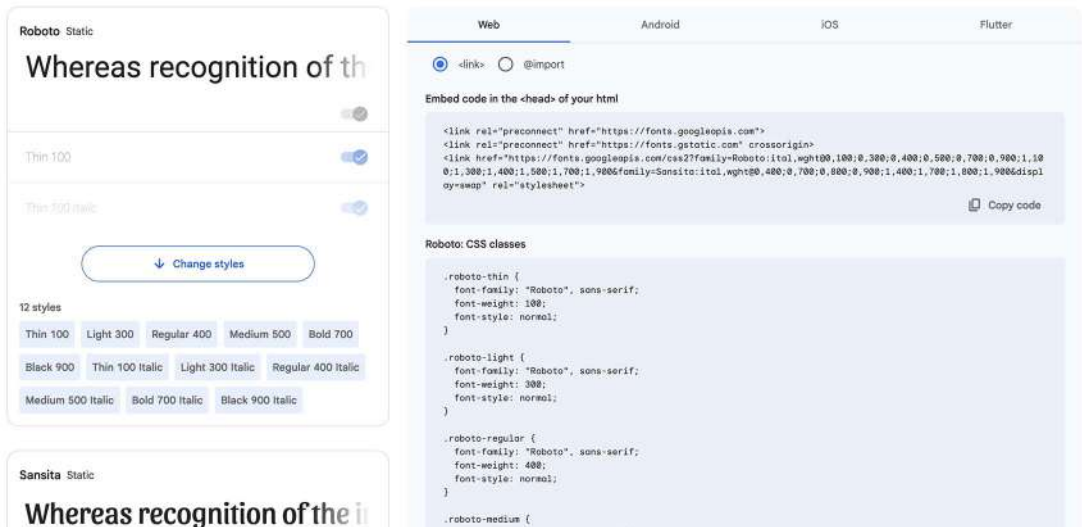


Figure 12.12 Currently selected fonts are listed on the left, and embed code is shown on the right.

In an ideal world, you could select all the font variants, giving you plenty of options to choose from in your page design. The more you add, however, the more the browser will have to download. Web fonts, after images, are one of the biggest offenders in slowing down page-loading time. A newer technology, called *variable fonts*, allows for embedding multiple variants in a single, smaller file. I'll show you how this works later in the chapter, but when that's not an option, you should be judicious, selecting only the fonts you need.

Initially, all font weights are selected. Beneath each selected font, click **Change styles** and deselect all weights apart from Roboto Light (300) and Sansita Extra-bold (800). These are the weights you'll be using for your page. You'll often need the italic version of the main body font as well, but it's not necessary for this example.

Copy the `<link>` tag and add it to your page's `<head>`, as shown in listing 12.6. This adds the stylesheet containing the font definitions to your page. Your page will now have two stylesheets: yours and the font stylesheet.

This also includes two preconnect `<link>` elements. These are a hint to the browser that it will need resources from these Google domains, so it can preemptively initiate connections as it parses the HTML, speeding up the font loading when it encounters them later in the stylesheet.

Listing 12.6 Stylesheet `<link>` tags for Google Fonts

```
<link rel="preconnect" href="https://fonts.googleapis.com" />
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
```

```
<link
  href="https://fonts.googleapis.com/css2?family=Roboto:wght@300&family=Sansita:wght@800&display=swap"
  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 12.13.

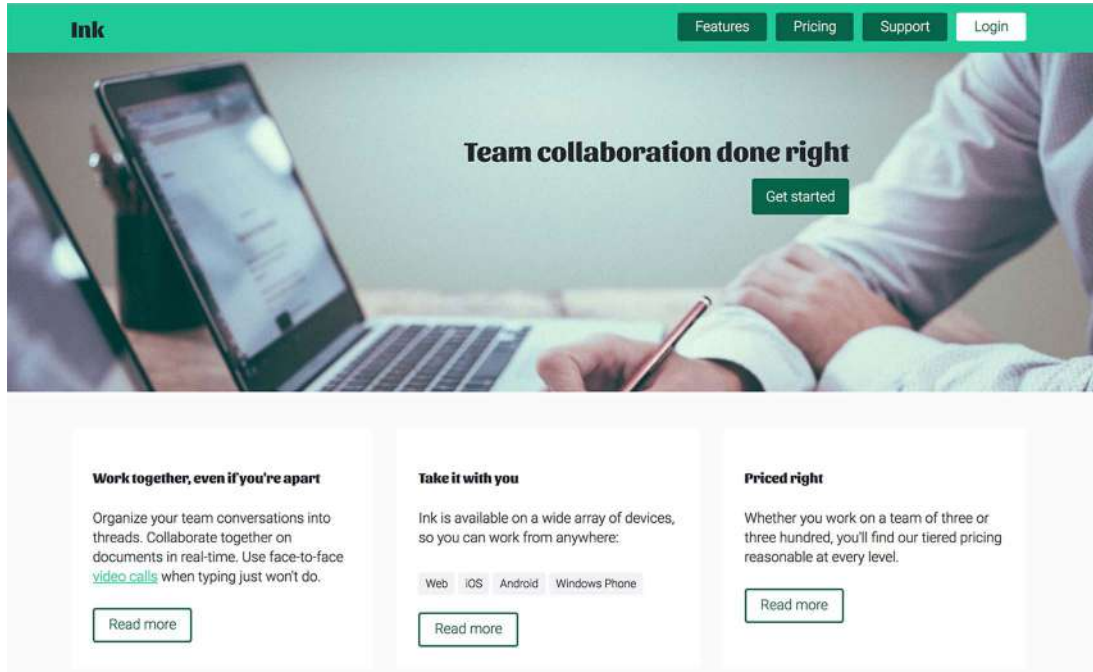


Figure 12.13 The page with Roboto and Sansita fonts applied

To use the fonts, you'll need to specify Roboto and Sansita using the `font-family` property. Next, you'll 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 home page link in the top-left corner. Change the corresponding portions of the code to match the changes in bold in the following listing.

Listing 12.7 Using the web fonts

```
@layer global {
  body {
    margin: 0;
    font-family: Roboto, sans-serif;
```

← Applies Roboto globally to the page


```

    line-height: 1.4;
    background-color: var(--extra-light-gray);
  }

  h1, h2, h3, h4 {
    font-family: Sansita, serif;
  }

/* ... */

@layer modules {
  .home-link {
    color: var(--text-color);
    font-size: 1.6rem;
    font-family: Sansita, serif;
    font-weight: bold;
    text-decoration: none;
  }
}

```

← Sets headings to the Sansita font

← Sets the home page 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 Adobe Fonts, 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.

12.4 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/css2?family=Roboto:wght@300&family=Sansita:wght@800&display=swap> in your browser to see Google's CSS. I've copied a portion of it into the following listing.

Listing 12.8 Google's font definition stylesheet

```

/* latin */
@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 300;
  font-display: swap;
  src: url(https://fonts.gstatic.com/s/roboto/v30/KFOlCnqEu92Fr1MmSU5fBBc4
    AMP6lQ.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+02BB-02BC, U+02C6,
        U+02DA, U+02DC, U+0304, U+0308, U+0329, U+2000-206F, U+2074, U+20AC,
        U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
}

/* latin */
@font-face {
    font-family: 'Sansita';
    font-style: normal;
    font-weight: 800;
    font-display: swap;
    src: url(https://fonts.gstatic.com/s/sansita/v11/QldLNTRRphEb_-V7JLmXWX5
    ➡ -w4dsz_k.woff2) format('woff2');
    unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6,
        U+02DA, U+02DC, U+0304, U+0308, U+0329, U+2000-206F, U+2074, U+20AC,
        U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
}

```

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` property gives the location where the browser will download this font.

The `unicode-range` property is not strictly necessary to make `@font-face` work. Google uses it for performance optimization: it places the most commonly used Latin characters in one, smaller font file and extended sets of other characters in additional files. This way, Latin characters can be downloaded quickly, and the extended set will only be downloaded by the browser if the page requires it. Extended character sets such as Cyrillic, Greek, and Vietnamese each receive their own `@font-face` rule in the Google stylesheet. The principles are the same, so I've left them out of listing 12.8 for simplicity.

It's important to know how to define your own `@font-face` rules, if you ever wish to use web fonts you purchase that aren't available on a hosted service. Often, I'll use Google Fonts to search for freely licensed typefaces I like but then download them and host them myself.

12.4.1 *Font formats and fallbacks*

These font files served by Google are all in a format called WOFF2. *WOFF* stands for Web Open Font Format. It's a compressed format designed specifically for use over a network. All modern browsers support WOFF2, but newer, better formats emerge periodically. When this happens, a robust solution should provide URLs for both WOFF2 and the newer font files. Listing 12.9 supposes a hypothetical WOFF3 format

and shows how to provide both options to the browser depending on support. (I'm using shorter URLs than Google's to make it more readable.)

Listing 12.9 A web font declaration with fallback to another format

```
@font-face {
  font-family: "Roboto";
  font-style: normal;
  font-weight: 300;
  src: local("Roboto Light"), local("Roboto-Light"),
        url(https://example.com/roboto.woff3) format('woff3'),
        url(https://example.com/roboto.woff2) format('woff2');
}
```

The first supported format listed will be used.

Fallback to WOFF2 for browsers that don't support WOFF3

This example also uses a special `local()` function. This tells the browser to first check if a font with the given name is installed on the user's operating system. If it is, it will use this rather than making a network request.

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. WOFF2 is now broadly supported, so this shouldn't be needed unless a newer format emerges.

12.4.2 Multiple variants of the same typeface

If you need multiple font styles of the same typeface, each needs its own `@font-face` rule. If, in the Google Fonts interface, you select both light and bold versions of Roboto, Google will give you a stylesheet URL that looks something like this: <https://fonts.googleapis.com/css2?family=Roboto:wght@300;700&display=swap>. Open this URL in your browser to see the code. I have copied a portion of it into the following listing.

Listing 12.10 Defining two different weights for the same typeface

```
/* latin */
@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 300;
  font-display: swap;
  src: url(https://fonts.gstatic.com/s/roboto/v30/KFOlCnqEu92Fr1MmSU5fBBc
        4AMP6lQ.woff2) format('woff2');
  unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6,
                U+02DA, U+02DC, U+0304, U+0308, U+0329, U+2000-206F, U+2074, U+20AC,
                U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
}
...
/* latin */
@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 700;
```

Roboto light

Roboto bold

```
font-display: swap;
src: url(https://fonts.gstatic.com/s/roboto/v30/KFOlCnqEu92Fr1MmWUlfBBc4
➡ AMP6lQ.woff2) format('woff2');
unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6,
               U+02DA, U+02DC, U+0304, U+0308, U+0329, U+2000-206F, U+2074, U+20AC,
               U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
}
```

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, producing a “faux bold” or “faux italic” font. This never looks as good as using a properly designed font. You should take care to only use font weights and styles for web fonts as they are specified in `@font-face` rules.

When you use Google Fonts or another font provider, you can use its interface and it will 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.

12.5 *Performance considerations*

When using web fonts, you need to consider performance. Font files are large. 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 will happen for that brief moment.

First, consider what happens if the browser uses an installed system font as a fallback while it waits for a web font to load. This scenario is illustrated in figure 12.14.

Your web fonts will likely take up a different amount of space onscreen 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.

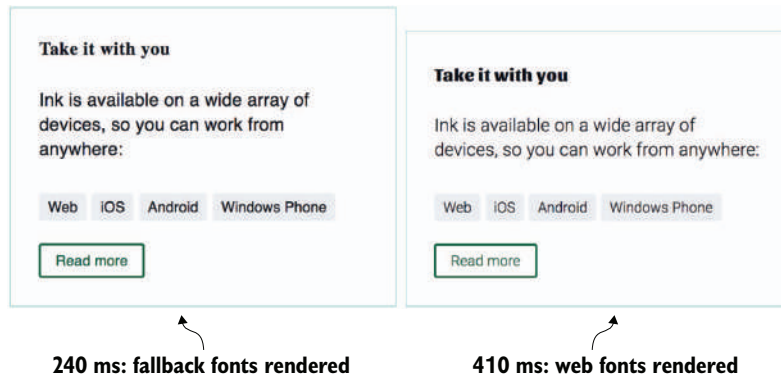


Figure 12.14 FOUT—Flash of Unstyled Text

To avoid this, browser vendors developed another approach. 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 results in a different acronym: *FOIT*, for Flash of Invisible Text (figure 12.15). Background colors and borders show up, but the text only appears during the second render, when web fonts are ready.

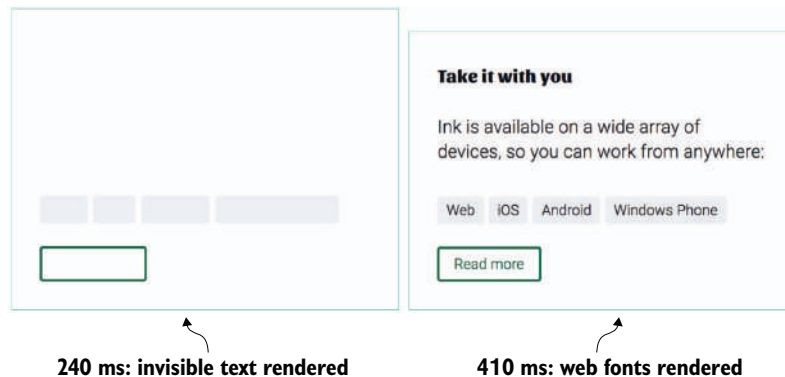


Figure 12.15 FOIT—Flash of Invisible Text

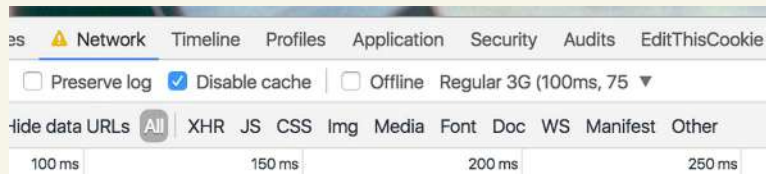
This approach solves one problem, but it creates another: What happens if the web fonts take a long time to load? Or fail to load altogether? In this case, the page remains blank, a shell of colored boxes that are entirely useless to the user. When this happens, it leaves you wanting the system font you would see during FOUT.

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 you can hope to do is to mitigate the problem as best you can.

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:



Network throttling tools in browser DevTools

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.

12.5.1 *The font-display property*

A `font-display` property allows you to specify how you want the browser to treat font loading. This property goes inside a `@font-face` rule. It specifies how the browser should treat web font loading.

In the Google font stylesheets (listings 12.8 and 12.10), you may have noticed that `font-display: swap` is specified in each of their font-face rules. 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 other values:

- `auto`—The default behavior, as defined by the browser (similar to `block` in most browsers).
- `block`—Displays invisible text while fonts load (a FOIT). This will time out after a short time and fall back to unstyled text, to prevent leaving the user with

permanently invisible text. The fallback text will then be replaced with the web font should it finish loading.

- `swap`—Displays the fallback font, then swaps in the web font whenever it's ready (a FOUT).
- `fallback`—A compromise between `block` and `swap`. For a brief time, 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 leave the fallback font in place if the font takes too long to load. Typically, this means the web font may not be used at all on slower connections.

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.

The stylesheet URL provided by Google Fonts has an `&display=swap` query parameter, which sets `font-display: swap` in the resulting stylesheet. You can edit this to any of the previous values to change this font display property.

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 topics relevant to CSS.

More font-relative units

CSS has a few more font-relative units to be aware of beyond `ems` and `rems`.

The line height unit, `1lh`, is equal to the line height of the current element. So if an element has a font size of 16 px and a line height of 1.5, `margin-block: 1lh` will apply a top and bottom margin of 24 px. All major browsers added support for this by the end of 2023. There is also an upcoming `1rlh` unit, which equals the line height of the root element (the `<html>`).

The character unit, `ch`, is equal to the width of the “0” character in the current font. This is particularly useful in combination with monospace fonts, as you can use something like `width: 80ch` to restrict a container to 80 characters of text.

The x-height unit, `ex`, is equal to the x-height of the current font: the height of lower-case letters without ascenders, such as the character “x.” This varies between different typefaces. This is usually equal to a value close to or between 0.4 em and 0.6 em, but some stylistic typefaces have very large or very short x-heights.

When using web fonts, it is usually best to avoid using `ch` and `ex` units for sizing large portions of the page. These values vary between different fonts, so it can cause a lot of layout shift as soon as the web fonts on the page finish loading.

Web fonts can look great and add a lot to the feel of a page, but they will never be as performant as fonts already available on the user's system. If you want to use system fonts, <https://modernfontstacks.com/> offers a helpful list of font stacks you can use that work across a wide range of operating systems.

12.5.2 *Variable fonts*

Some newer fonts are available as *variable fonts*. Instead of one font file for thin, one for regular, and one for bold, a single font file can include the entire range of font weights in a single file. A variable font file is typically much smaller in size than multiple individual font files for the same typeface. This makes them a more efficient way to add multiple font weights to the page.

Variable fonts work by defining one or more *variation axes* in the font file. A variation axis describes an allowable range for a particular aspect of the font. The most common axis is a “weight axis,” defining for the browser how to transform the font weights from light to bold. Font weight can range anywhere from 0 to 999.

An axis may also define a binary “on” or “off” choice. A typical example of this is an “italic axis,” which controls whether the font is displayed as normal or italic with no in-between.

Listing 12.11 shows a `@font-face` rule for the Open Sans variable font with a font weight ranging from 300 to 800. Notice the `font-weight` has two values, specifying the minimum and maximum values to use.

Listing 12.11 `@font-face` rule for a variable font

```
@font-face {
  font-family: "Open Sans";
  font-style: normal;
  font-weight: 300 800;
  font-display: swap;
  src: url(./open-sans.woff2) format("woff2");
}
```

Specifies font weights between 300 and 800

With this `@font-face` rule in place, you can use the Open Sans font family on the page with any font weight ranging from 300 (light) to 800 (extra bold). You are not limited to multiples of 100; you can use precise gradations such as `font-weight: 650` or `font-weight: 467` as long as it is within the range supported by the font file and the `@font-face` rule.

FONT VARIATION AXES

Specifying a variable font weight using the `font-weight` property is generally the simplest approach, but you can also accomplish the same thing with a lower-level syntax using the `font-variation-setting` property. Consider the following example. Using the Open Sans font from listing 12.11, the following two rulesets are equivalent:

```
p {  
  font-family: 'Open Sans', sans-serif;  
  font-weight: 400;  
}  
  
p {  
  font-family: 'Open Sans', sans-serif;  
  font-variation-setting: 'wght' 400;  
}
```

The `font-variation-setting` property references a special four-character string `'wght'` that indicates the font weight variation axis in the file. A variation axis is always defined using a four-character string. Variable fonts can use variation axes from among five *registered axes*. These axes are each mapped to a corresponding font property as follows:

- `font-weight` is controlled with the `'wght'` axes, ranging from 0 to 999. As shown previously, `font-weight: 400` is equivalent to `font-variation-setting: 'wght' 400`.
- `font-style: italic` is controlled with the `'ital'` axes. It can be set to either 0 for normal font or 1 for italics. Thus, `font-variation-setting: 'ital' 1` is equivalent to `font-style: italic`.
- `font-style: oblique` is controlled with the `'slnt'` axes. This controls the angle of the letterforms. This is typically a value between 0 (normal) and around 20 degrees of slant, but any value between -90 and 90 may be valid. Applying `font-variation-setting: 'slnt' 15` is equivalent to `font-style: oblique 15deg`. The slant value differs from italics in that it changes the angle of the letterforms but does not replace the letterforms with a different character altogether as italics typically do.
- `font-stretch` compresses or expands the width of letterforms (called *condensed* or *expanded* in typographic terms). This is controlled with the `'wdth'` axes. These values are expressed indicating a percentage above or below 100% (normal). Applying `font-variation-setting: 'wdth' 75` is equivalent to `font-stretch: 75%`.
- A new property, `font-optical-sizing` is controlled with the `'opsz'` axis. In many typefaces, vertical and horizontal lines are drawn with different thicknesses. However, when displayed at smaller font sizes, the thin lines (usually the horizontal portion of a letterform) can become difficult to see. Optical sizing is used to correct this by reducing the difference between wide and narrow lines at smaller font sizes. `font-optical-sizing` can be set to `none` or `auto`. However, fonts that support this axis allow fine-grained control via `font-variation-setting`. For example, `font-variation-setting: 'opsz' 16` sets the optical sizing as the font designer intended for a 16 px font size.

It is not common to find fonts that use more than one or two of these axes. For interactive examples of all these axes, see <https://mng.bz/ZEDP>.

Beyond the five registered axes, font designers have the ability to add *custom axes* to a font. A custom axis is defined with a four-character, capitalized string, such as 'GRAD'. Custom axes do not map to another property and can only be adjusted using `font-variation-setting`. If you encounter a font that uses a custom axis, see the documentation for that font for instructions on how to use it.

NOTE Some fonts come with a color palette built in, which allows for portions of the letterforms to be displayed in multiple colors. CSS properties can be used to modify these colors. This is a bit of a niche use case, so I won't cover it in this book, but you can read how to do this at <https://www.matuzo.at/blog/2023/100daysof-day76/>.

Many font-hosting services, including Google Fonts, offer a variety of variable fonts to select from. I recommend looking for these fonts when you can, especially when you anticipate needing multiple styles of the same typeface.

12.6 *Adjusting space for readability*

Let's return to your page. Now that web fonts Roboto and Sasita are loaded, you can 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 in 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 12.16 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.

The Dog and the Shadow

It happened that a Dog had got a piece of meat and was carrying it home in his mouth to eat it in peace. Now on his way home he had to cross a plank lying across a running brook. As he crossed, he looked down and saw his own shadow reflected in the water beneath.

Thinking it was another dog with another piece of meat, he made up his mind to have that also. So he made a snap at the shadow in the water, but as he opened his mouth the piece of meat fell out, dropped into the water and was never seen more.

Beware lest you lose the substance by grasping at the shadow.

The Dog and the Shadow

It happened that a Dog had got a piece of meat and was carrying it home in his mouth to eat it in peace. Now on his way home he had to cross a plank lying across a running brook. As he crossed, he looked down and saw his own shadow reflected in the water beneath.

Thinking it was another dog with another piece of meat, he made up his mind to have that also. So he made a snap at the shadow in the water, but as he opened his mouth the piece of meat fell out, dropped into the water and was never seen more.

Beware lest you lose the substance by grasping at the shadow.

The Dog and the Shadow

It happened that a Dog had got a piece of meat and was carrying it home in his mouth to eat it in peace. Now on his way home he had to cross a plank lying across a running brook. As he crossed, he looked down and saw his own shadow reflected in the water beneath.

Thinking it was another dog with another piece of meat, he made up his mind to have that also. So he made a snap at the shadow in the water, but as he opened his mouth the piece of meat fell out, dropped into the water and was never seen more.

Beware lest you lose the substance by grasping at the shadow.

Figure 12.16 The spacing of the text can have a marked effect on readability.

12.6.1 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 12.17 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.)

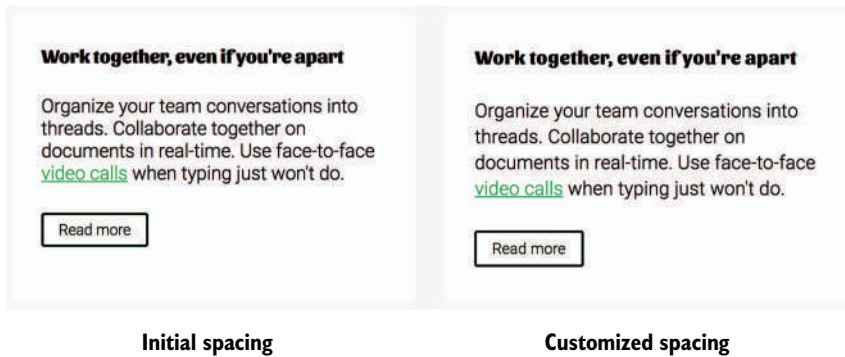


Figure 12.17 A tile from Ink's page with initial spacing values compared to purposely chosen values

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 readable.

Next, you can consider letter-spacing. If you are using a well-designed font, the default spacing probably does not require adjusting, but sometimes the readability can be improved, so I'll walk you through making changes to this. You may also want to adjust this property on small portions of the page for stylistic reasons, such as on buttons or headings.

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 very 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 in the following listing.

Listing 12.12 Setting letter spacing on the body element

```
@layer global {
  body {
    margin: 0;
    font-family: Roboto, sans-serif;
    line-height: 1.4;
```

Line height and letter spacing will be inherited by everything on the page.

```

letter-spacing: 0.01em;
background-color: var(--extra-light-gray);
color: var(--text-color);
}

```

← 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 I'll 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 $96 / 72 = 1.333$): $18 \text{ pt} \times 1.333 = 24 \text{ px}$. 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 thousandths of an em, so to convert it to ems, divide by a 1000: $100 / 1000 = 0.1 \text{ em}$.

12.6.2 Headings, small elements, and spacing

The spacing for headings won't always be the same as that for the body copy. After you have a 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 arise, and your styles should accommodate it. 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 12.18).

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.

For the body copy, spacing is focused on maximizing readability. But this matters less for headings and other elements with little content, such as buttons. In these

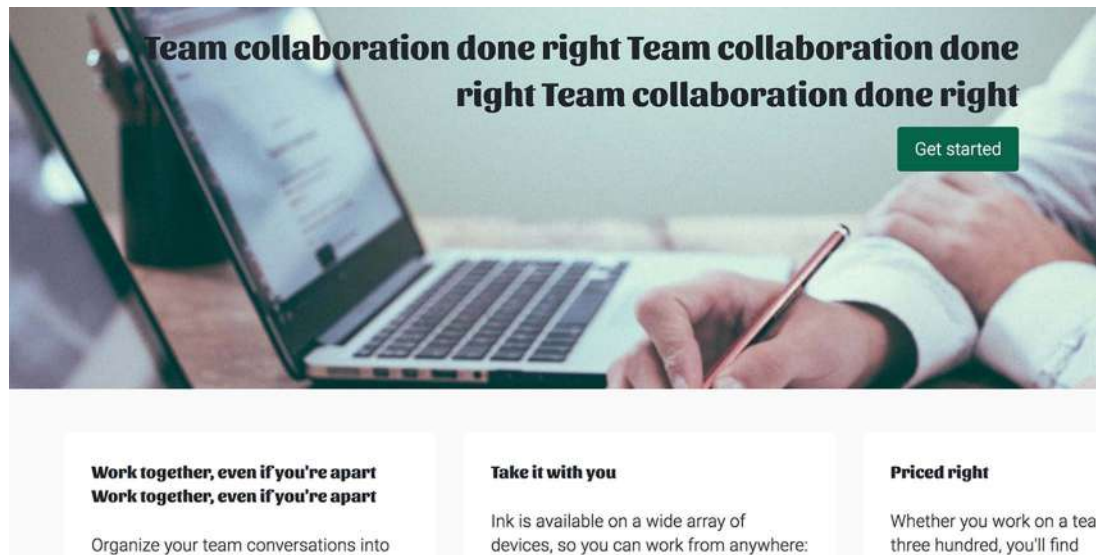


Figure 12.18 Forcing headings to line wrap to make sure their line height is appropriate

cases, you can get more creative. You can get away with broader spacing. You can also use negative letter spacing to reduce the space between characters. Figure 12.19 shows the slogan with a `letter-spacing: -0.02em` applied.



Figure 12.19 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). Apply this to the slogan text. Update the corresponding portion of your stylesheet to include the change in bold shown in the following listing.

Listing 12.13 Tightening up the spacing of the slogan

```
.hero h2 {
  margin-block: 0 10px;
  font-size: 1.95rem;
  letter-spacing: -0.02em;
}
```

A negative letter spacing compresses space between 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. You'll adjust those next. Figure 12.20 shows how they look now (top) and how they'll look after making the changes (bottom).

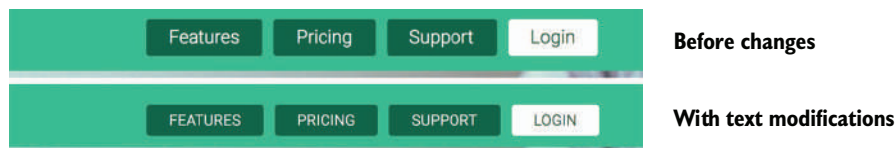


Figure 12.20 Changing the text properties can improve the look of the nav buttons.

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.

To implement these changes, add or update the declarations indicated in bold in the following listing.

Listing 12.14 Adjusting size and spacing of nav menu items

```
.nav-container__inner {
  display: flex;
  justify-content: space-between;
  align-items: end;
  max-inline-size: 1080px;
  margin-inline: auto;
  padding: 0.625em 0;
}

...

.top-nav a {
  display: block;
  font-size: 0.8rem;
  padding: 0.3rem 1.25rem;
  color: var(--white);
  background: var(--brand-green);
  text-decoration: none;
  border-radius: 3px;
  text-transform: uppercase;
  letter-spacing: 0.03em;
}
```

Aligns items in the nav container to the bottom

Decreases font size of nav links and buttons

Changes the padding values from em to rem

Capitalizes nav links and increases letter spacing

You've reduced the size of the navigational links, so they 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 fixes this, which is why this listing includes the `align-items: end` declaration.

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 here.

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. Only when something should be capitalized according to the rules of grammar (such as an acronym) should you 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.

Summary

- The line height of text contributes to the content box size of the containing element. You may need to take this height into consideration when fine-tuning spacing on the page.
- A hosted font provider such as Google Fonts allows for easy web font integration.
- `@font-face` rules allow you to host your own web fonts. These allow you to specify the font name the stylesheet will use to reference the font and at what font weight and font style the font will be applied.
- Web fonts contribute a lot of size to the page; limit the number of web fonts you add to keep the size under control. Use variable fonts when available to further reduce this size.
- The `font-display` property controls how the page is initially rendered while web fonts load.
- Use `line-height` and `letter-spacing` to control the spacing of text. A line height close to 1.5 is usually best for body copy, but a smaller spacing is often better for large headings.

13

Gradients, shadows, and blend modes

This chapter covers

- Linear, radial, and conical gradients
- Color interpolation in different color spaces
- 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. Finally, you've learned how to pay attention to the details of spacing, typography, and other concerns of web design.

In the next two chapters, I'll show you techniques for achieving particular effects and adding visual interest to the page. You'll likely use some of these effects frequently—in particular, gradients and shadows—while others you may need only occasionally. It's almost certain you'll never use these effects all at once, so I'll focus primarily on isolated examples rather than a large comprehensive page.

Consider the button shown in figure 13.1. 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 13.1 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 in various ways.

13.1 Gradients

As you've been following along, you'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 is, 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 7), but it can also accept a gradient function. For example, you can define a gradient that blends from one color to another, as in figure 13.2.



Figure 13.2 A linear gradient from one color to another

Gradients are useful effects. 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 13.1 A basic linear gradient

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(to right, white, blue);
}
```

Fades left to 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 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 any color syntax, such as hex (`#0000ff`) or OKLCH (`oklch(45% 0.31 264deg)`) or the transparent keyword. Add the element from the following listing to see the gradient on the page.

Listing 13.2 Element with a background gradient

```
<!doctype html>
<html lang="en-US">
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="./style.css">
</head>
<body>
```

```
<div class="fade"></div>
</body>
</html>
```

← Element with gradient background

You can specify the angle of the gradient in several ways. In this example, you used `to right`, but you can specify other directions such as `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 13.3 Gradient using degree (`deg`) units

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(90deg, white, blue);
}
```

← 90deg value is equivalent to using “to right.”

Degrees are the most common unit, but there are a few others that you can use to indicate an 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.

NOTE These angle units are supported anywhere CSS expects an angle. This includes the hue angle in color notations. For example, `oklch(45% 0.31 0.75turn)` is a valid OKLCH color.

13.1.1 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 13.3 shows a gradient with three color stops (turquoise, white, then blue).

You can insert more color stops by adding more colors to the `linear-gradient()` function. To see this gradient on your page, update your stylesheet to match listing 13.4.

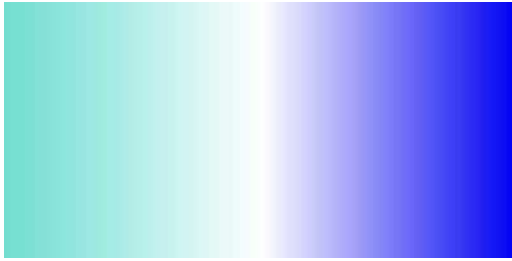


Figure 13.3 Gradient with three color stops

Listing 13.4 Linear gradient with multiple color stops

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(90deg, turquoise, 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 turquoise 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 13.4 is equivalent to the following:

```
linear-gradient(90deg, turquoise 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 13.4 shows a gradient that begins turquoise, switches immediately to white, and then switches immediately to blue. This creates the appearance of stripes.

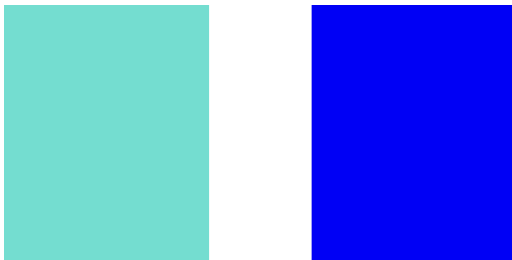


Figure 13.4 Gradient used to create stripes by placing two color stops at the same points

The code for the gradient is shown in listing 13.5. Notice that this gradient has four color stops, two of which are white.

Listing 13.5 Placing two color stops on the same points to create stripes

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(
    90deg,
    turquoise 40%,
    white 40% 60%,
    blue 60%
  );
}
```

Color stops on
the same
points

The first color stop is turquoise at 40%, so the gradient is solid turquoise from the left edge all the way to 40%. The next color is white. This is given two color stops: one at 40%, so the gradient makes a hard switch to white, then another at 60%. These make the gradient pure white from 40% to 60%. Then the final color stop, also at 60%, is blue. This makes a hard switch from white to blue, then remains blue to the right edge.

These color stops can also be defined using any valid length, so instead of 40%, you can specify 160 px or 10 em.

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 13.5).



Figure 13.5 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 in listing 13.6. Update your stylesheet to match.

Listing 13.6 Creating a diagonally striped bar

```
.fade {
  height: 1em;
  width: 400px;
  background-image: repeating-linear-gradient(
    -45deg,
    oklch(60% 0.1 257deg) 0px 10px,
    oklch(40% 0.1 257deg) 10px 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/>.

13.1.2 Color interpolation

In chapter 11, I covered the color spaces associated with the various CSS color notations. Until recently, browsers could only calculate gradients within the sRGB color space, but using other color spaces is now an option. When the browser calculates the intermediate colors in a gradient, the result can vary significantly depending on the color space it uses. Since the colors are arranged in different geometric arrangements in different color spaces, the colors between two points in one space will be different than those in another space.

Figure 13.6 shows two gradients. Both change from yellow on the left to blue on the right, but the intermediate colors are very different because they have been interpolated in different color spaces.



Figure 13.6 Gradients in different color spaces can vary significantly.

In this example, the gradient in sRGB passes through a gray color in the center of the gradient. This makes sense when you think about the RGB colors involved. `yellow` in RGB is `rgb(255 255 0)`, and `blue` is `rgb(0 0 255)`. Each parameter is linearly interpolated, so at the center of the gradient, the red, green, and blue values will all be 127.5, producing gray. This is sometimes called the “gray dead zone.” You will see similar results in sRGB with any gradient between two colors on opposite or near-opposite sides of the color wheel; the gradient must pass through the gray center of the color wheel.

On the other hand, yellow and blue in OKLCH are represented as `oklch(97% 0.21 110deg)` and `oklch(45% 0.31 264deg)`. The gradient between these values primarily involves changing the hue angle from 110 degrees to 264 degrees. The gradient passes through green, teal, and cyan. Instead of passing through the middle of the color wheel, a gradient in OKLCH will curve around it, passing through intermediary hues along the way.

WARNING The ability to specify a color space for a CSS gradient is a relatively new feature. As of mid-2024, it is not yet supported in Firefox. When you use this feature, you should also specify a fallback gradient without it.

The syntax for a gradient with a specified color space is shown in listing 13.7. I’ve included a fallback gradient without it for Firefox and older browsers. Notice the `in oklch` added as part of the first parameter to the `linear-gradient()` function. You can update the CSS on your page to see a demo of this.

Listing 13.7 Gradient in `oklch` color space

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(90deg, yellow, blue);
  background-image: linear-gradient(90deg in oklch, yellow, blue);
}
```

You can see the noticeable difference between these two by disabling and enabling the second `background-image` ruleset in your browser’s DevTools, assuming you’re using a browser that supports color space notation.

When specifying a color space, the `in` keyword is required, followed by the color space name. This can be any CSS-supported color space, including the polar color spaces `hsl`, `hwb`, `lch`, and `oklch`, and the rectangular spaces `srgb`, `srgb-linear`, `lab`, `oklab`, `xyz`, `xyz-d50`, and `xyz-d65` (note that some of these color spaces don’t have an associated CSS color notation function). If you don’t specify a color space, the default color space will be `sRGB`.

Each color space produces different results, so if you’re looking for a specific effect, it may be worth trying a few different ones to see which you like best. As a general rule, if you want a gradient between two significantly different hues, you will likely be better off using one of the polar color spaces, such as `oklch` or `hsl`. If you want a gradient from a more vivid color to a neutral color (gray, black, white, or transparent), you might achieve a better result with a rectangular color space such as `srgb`.

NOTE The color interpolation involved in gradients is identical to that used in `color-mix()`, which I covered in chapter 11. For example, `color-mix(in srgb, white 20%, blue)` is mathematically the same thing as building a gradient in `srgb` from white to blue, then selecting the color that appears 20% of the way across that gradient.

The Chrome team has put together a demo showcasing several gradients in all available color spaces. This is available at <https://codepen.io/keithjgrant/full/GRzjGqL> and is worth quickly skimming through to get a sense of how the various color spaces can vary. Note that some of these examples illustrate some browser bugs, which I’ll address shortly.

MORE OPTIONS IN POLAR COLOR SPACES

In polar color spaces, hue is a circular value. As a result, a gradient between two different hues can go in two different directions around the color wheel between those two hues. A gradient from red to yellow, for instance, will typically pass through orange. This is the shorter route and the default behavior.

However, a gradient from red to yellow could instead go the long way around the color wheel, passing through magenta, purple, blue, teal, and green. If the two colors are relatively near each other on the color wheel, then the longer gradient can produce a vibrant rainbow of color. You can even build a gradient with the same color for both color stops; the longer gradient in this case will include the entire rainbow. Figure 13.7 illustrates these two paths on a color wheel.

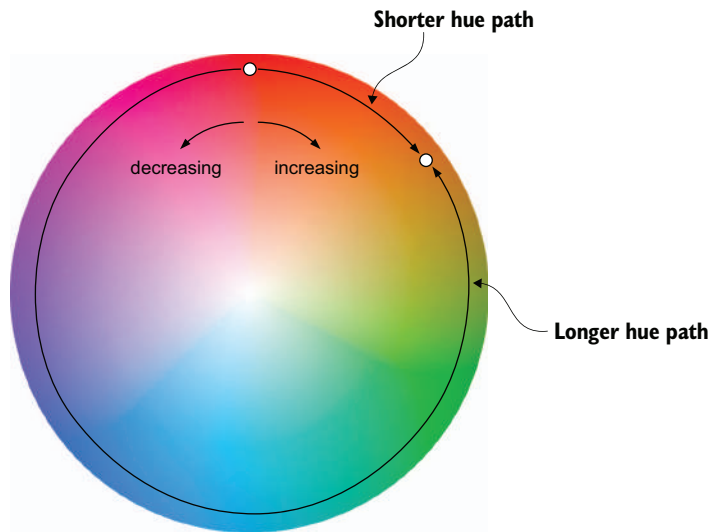


Figure 13.7 A gradient can follow either the shorter or longer path around a polar color space hue.

You can specify this in the gradient function by adding the keywords `longer hue`. Update your stylesheet to match the following listing to see an example of this.

Listing 13.8 A gradient using the longer hue

```
.fade {
  height: 200px;
  width: 400px;
  background-image: linear-gradient(90deg, yellow, blue);
  background-image: linear-gradient(
    90deg in oklch longer hue,
    yellow,
    blue
  );
}
```

← Specifies the gradient follows the longer hue path

The first parameter in the gradient function can get rather long with all these options, as it does here. When this happens, it makes sense to split the declaration across multiple lines of code, as shown previously.

The hue direction portion of the gradient supports four different values. These are only valid in conjunction with a polar color space:

- **shorter hue**—Takes the shorter route between the two hues (default behavior)
- **longer hue**—Takes the longer route between the two hues
- **increasing hue**—Takes the route that involves increasing the hue angle (clockwise)
- **decreasing hue**—Takes the route that involves decreasing the hue angle (counterclockwise)

Try these values in your gradient to see how they behave. In this case, **increasing hue** is equal to **shorter hue**, and **decreasing hue** is equal to **longer hue**. This is due to the locations of yellow and blue on the color wheel—110 degrees and 164 degrees, respectively (I found these by using browser DevTools to convert `yellow` and `blue` to OKLCH, as shown in chapter 11). With different color stops, however, the opposite will sometimes be true.

BROWSER BUGS AND COLOR QUIRKS

Many new color features are new to browsers, and they aren't without a few bugs, specifically when using OKLCH or HSL color spaces in a gradient. In certain hues (particularly blues), you might see unexpected behavior if one of the color stops is white, black, gray, or transparent.

For example, a gradient from blue to an unsaturated color like white or black produces either a green or purple in the middle of the gradient in both the OKLCH and HSL color spaces—the exact result varies based on the darkness of the unsaturated color and which browser you test in. A gradient from red to an unsaturated color includes either an orange or burgundy center. Hopefully, these will be fixed in future versions of the affected browsers. The best way to avoid these bugs is to use sRGB or another rectangular color space when your gradient involves white, black, gray, or transparent color stops.

If you do wish to use an OKLCH or HSL color space, however, you can usually work around these bugs by defining your colors in the same color space and adding a small amount of chroma or saturation in the same hue as the other color stop. For example, a gradient in OKLCH from `blue` to `oklch(100% 0.001 264deg)` instead of `blue` to `white` will behave as expected. I've written up a deeper look at these workarounds at <https://mng.bz/2Kp0>.

13.1.3 Radial gradients

Linear gradients are probably the most common type of gradient you will use, but there are other types as well. One of these is a radial gradient. Instead of starting at one end of the element and proceeding to the other end in a single direction, a radial gradient starts at a single point and proceeds outward in all directions. A basic example is shown in figure 13.8.

A radial gradient is defined using the `radial-gradient()` function. An example of this is shown in listing 13.9. Edit your stylesheet to match this code.



Figure 13.8 A radial gradient fades outward from a single point.

Listing 13.9 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 and taller for tall elements).

Radial gradients support color stops, the same as linear gradients. You can provide multiple stops and explicitly define their position within the gradient using percentages or length units. You can also specify a color space just as with linear gradients.

Radial gradients also support some additional options as part of the first function parameter:

- You can make the radial gradient a circle rather than an ellipse with the keyword `circle`. This is invalid if both a horizontal and vertical size are also specified.
- Specify the size of the gradient by providing one or two length values. One value makes the gradient a circle of the given size. Providing two values specifies a horizontal and vertical measurement. For example, `radial-gradient(50px 25px, white, blue)` produces a gradient 50 px wide and 25 px tall.
- Specify the location of the center of the gradient by giving x- and y-coordinates from the top left of the element. This is done following the `at` keyword—for example `radial-gradient(at 25px 25px, white, blue)`.

Using the `repeating-radial-gradient()` function repeats the pattern in concentric rings. This accepts the same parameters as `radial-gradient()`.

Most of these features are best explained by example, so I've included several with the corresponding code in figure 13.9. I encourage you to try these on 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> Circular radial gradient	
<code>radial-gradient(50px at 25% 75%, white, midnightblue)</code> Sized 50 px, centered 25% from the left and 75% from the top	
<code>radial-gradient(circle, midnightblue 0%, white 75%, red 100%)</code> Radial gradient with explicit color stop positions	
<code>radial-gradient(circle in oklch longer hue, yellow, blue)</code> Radial gradient with specified color space and longer hue transition	
<code>repeating-radial-gradient(circle, midnightblue 0 15px, white 15px 30px)</code> Repeating gradient with stripes	

Figure 13.9 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://mng.bz/1Gmj>.

13.1.4 Conic gradients

The final type of gradient is a conic gradient. While a radial gradient changes color out from the center of an ellipse, a conic gradient changes color around the ellipse in a circular direction. A basic conic gradient is shown in figure 13.10.



Figure 13.10 A conic gradient changes color in a circular direction.

This is defined with the `conic-gradient()` function or the repeating version, `repeating-conic-gradient()`. The gradient here is defined with the code shown in the following listing.

Listing 13.10 A basic conic gradient

```
.fade {  
  height: 200px;  
  width: 400px;  
  background-image: conic-gradient(white, blue);  
}
```

As you can see in this example, conic gradients involve an abrupt transition where the gradient ends, between the final color stop and the first one. This can produce a hard edge between the two colors. You can prevent this by ensuring the first and last color stops are the same color: `conic-gradient(white, blue, white)`.

As with linear and radial gradients, you can specify extra details of the gradient as part of the first function parameter:

- Specify the starting angle of the gradient with `from <angle>`. When omitted, the default value is `0deg`, or straight up. For example, `conic-gradient(from 90deg, white, blue, white)` starts the gradient at 90 degrees, on the right side.
- Specify the center point of the gradient with `at <x-position> <y-position>`. These x and y positions are measured from the top-left corner of the element—for example, `conic-gradient(at 30px 45px, white, blue, white)`. The positions may be specified as either a length or a percentage.
- Specify the color interpolation method with `in <color-space>`, just as with linear and radial gradients—for example: `conic-gradient(from 90deg in oklch, yellow, blue, yellow)`.

As with other gradients, you can also specify the location of the color stops. These can be given as a percentage or as an angle. I encourage you to experiment with all these settings on your test page. For more examples, see the MDN documentation at <https://mng.bz/PZGn>.

It may be helpful to use a graphical interface to construct gradients. One such resource I like is at <https://gradient.style/>. This supports OKLCH color and all three gradient types.

Most of the example gradients I've shown thus far use starkly contrasting colors. I've done this to emphasize the effect of gradients so their behavior is clear. But in real projects, you'll generally be better off using colors with much less contrast.

Instead of fading from white to black, fade from white to light gray. Or fade between two similar shades of blue. This will be much less jarring to the user. In some cases, they may not even notice the gradient, but it'll still apply a subtle depth to the page. I'll show you a real-world example of this in a moment, but first, let's look at shadows.

The gradient here is subtle. You might not notice it immediately, but it gives the button a slightly 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. By adding the keyword *inset* to the beginning of the box shadow, it is rendered inside the element rather than outside. This inset shadow is layered in front of the background image (the gradient).

The inset shadow gives the button 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 this button markup to the page:

```
<button class="button">Sign up now</button>
```

Next, add the styles from the next 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 13.11 Button styles with a gradient and shadow

```
.button {
  padding: 1em;
  border: 0;
  font-family: Helvetica, Arial, sans-serif;
  color: white;
  border-radius: 0.5em;
  background-image: linear-gradient(
    to bottom,
    oklch(57% 0.11 263deg),
    oklch(40% 0.13 263deg)
  );
  box-shadow: 0.1em 0.1em 0.5em oklch(26% 0.07 263deg);
}

.button:active {
  box-shadow: inset 0 0 0.5em oklch(26% 0.07 263deg),
             inset 0 0.5em 1em rgb(0 0 0 / 0.4);
}
```

Gradient from medium blue to a slightly darker 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.

Notice that in the active button state, I've added more than one shadow definition, separating them with a comma. Multiple shadows can be added to an element this way.

The first inset shadow (`inset 0 0 0.5em oklch(26% 0.07 263deg)`) 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 rgb(0 0 0 / 0.4)`) has a bit of vertical offset, making the shadow more apparent along the top of the button. The RGB color definition defines a

semitransparent black. I encourage you to experiment with these values to see how they affect the final rendering.

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. This sort of design is not very common online anymore, but it is an excellent use case for seeing how gradients and shadows can work together. Let's examine some other ways you could design a button.

13.2.2 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, gradients, and rounded corners had to be created using images.

Flat design doesn't necessarily mean none of these effects are used, however. Rather, 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.

Next, you'll convert your button to one with a flat design. This new button is shown in figure 13.14. It doesn't look like something from the physical world, though it does have a slight shadow beneath it.

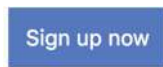


Figure 13.14 Button with a flat appearance

The styles for this are shown in listing 13.12, along with hover and active states. Update your CSS to match the changes marked in bold. Note also that the border-radius has been removed from the previous listing.

Listing 13.12 A flat button with hover and active states

```
.button {
  padding: 0.8em;
  border: 0;
  font-family: Helvetica, Arial, sans-serif;
  color: white;
  background-color: oklch(57% 0.11 263deg);
  box-shadow: 0 0.2em 0.2em rgb(0 0 0 / 0.15);
}
```

Solid background
color (no gradient)

Very subtle
shadow

```

.button:hover {
  background-color: oklch(53% 0.13 263deg);
}
.button:active {
  background-color: oklch(40% 0.13 263deg);
}

```

← Slightly darker hover and active states

The box shadow here has a few changes. It has only a vertical offset, so the shadow is straight down rather than at a more natural-looking angle. I also used an RGB 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.

13.2.3 Creating buttons with a hybrid look

Flat design is still popular, but it's constantly evolving. One common approach is to come back toward the middle between flat design and skeuomorphism. You'll remake your button one last time to match figure 13.15. This design will use elements of both design styles.



Figure 13.15 A different type of flat button

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 3D 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 that contour 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 shown in the following listing.

Listing 13.13 Modern button styles

```

.button {
  padding: 0.8em;
  border: 0;
  border-radius: 0.5em;
  font-family: Helvetica, Arial, sans-serif;
  color: white;
  background-color: oklch(57% 0.11 263deg);
  box-shadow: 0 0.4em oklch(40% 0.13 263deg);
  text-shadow: 1px 1px oklch(40% 0.13 263deg);
}
.button:active {
  background-color: oklch(53% 0.13 263deg);
  transform: translateY(0.1em);
  box-shadow: 0 0.3em oklch(40% 0.13 263deg);
}

```

← 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. Unlike `box-shadow`, `text-shadow` does not 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 further when we take an in-depth look at transforms in chapter 16. 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.

13.3 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 unless the image happens to be partially transparent. You can also define multiple gradient background images, and as long as the first gradients listed include transparent or partially transparent colors, then the gradients will layer together accordingly.

However, sometimes you may want to layer together two backgrounds that are fully opaque. 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 13.16 shows two backgrounds combined with the `multiply` blend mode. Both backgrounds use the same image, positioned differently.



Figure 13.16 Two backgrounds combined with the multiply blend mode

This produces an interesting effect, where both copies of the image are still clearly visible, even where they overlap, yet it doesn't wash out or fade the color like simple transparency does.

I'll show you how to create an element with two backgrounds and blend them to match in figure 13.16. On a new page, add a div with the class `blend`. You'll reuse this markup for the next several examples:

```
<div class="blend"></div>
```

You'll use an empty element for now to illustrate the effect. Next, add the code from the following listing to a stylesheet and link it to the page.

Listing 13.14 Blending two background images

```
.blend {
  min-height: 70vmin;
  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, and 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 the edges of the image being clipped. Using contain ensures the entire image is visible, though this may result in part 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 effects they can have. This can be amusing to play with, but you might be wondering what the practical applications are. The following 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.

13.3.1 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 13.17.

A background-blend-mode merges not only multiple background images but also the background-color. The blend mode combines these stacked layers, 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 the following listing.

Listing 13.15 Blending a background image with the background color's hue

```
.blend {
  min-height: 70vmin;
  background-image: url("images/bear.jpg");
  background-color: #148;
  background-size: cover;
  background-repeat: no-repeat;
  background-position: center;
  background-blend-mode: luminosity;
}
```

Blue background color

Luminosity blend mode



Figure 13.17 Image colored with a uniform blue hue

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) behaves 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.

13.3.2 Types of blend modes

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 13.1. They each fall into one of five types: darkening, lightening, contrasting, compositing, or comparing.

Table 13.1 Blend modes in five basic categories

Type of effect	Blend mode	Description
Darken	multiply	The lighter the front color, the more the base color will show through.
	darken	Selects the 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 two colors.
	color-dodge	Lightens the base color, decreasing contrast.
Contrast	overlay	Increases contrast by applying multiply to dark colors and screen to light colors, at half strength.
	hard-light	Greatly increases contrast. Like overlay but applies multiply or screen at full strength.
	soft-light	Similar to hard-light but uses burn/dodge instead of multiply/screen.
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.

Some blend modes are more practically useful than others. Choosing the right one often involves a bit of trial and error. You can preview many of these in an interactive blend mode tool at <https://garden.bradwoods.io/notes/css/blend-modes>.

13.3.3 Adding texture to an image

Another application of blend modes is to add texture to an image. You may have a crisp image, but sometimes you want it to appear differently for stylistic reasons. You can blend in a second image to artificially add film grain or some other texture to the image.

Consider the image shown in figure 13.18. 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.

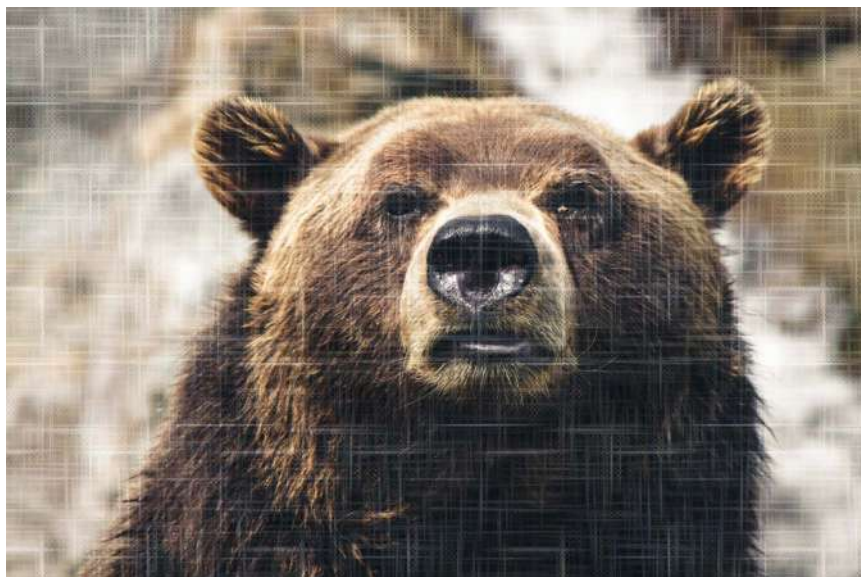


Figure 13.18 Image with texture blended in

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 the following listing to see this effect in your browser.

Listing 13.16 Using `hard-light` to add texture to the image

```
.blend {
  min-height: 70vmin;
  background-image: url("images/scratches.png"),
    url("images/bear.jpg");
  background-size: 200px, cover;
  background-repeat: repeat, no-repeat;
  background-position: 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`

The background size of the texture image (figure 13.19) 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.

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 the darkness of the base image.

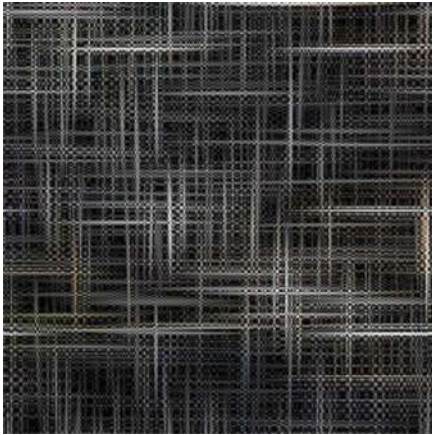


Figure 13.19 Grayscale canvas texture image

13.3.4 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 13.20.



Figure 13.20 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, you'll need to add the heading to the markup as a child of the container. Update your HTML to match the following:

```
<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 the following listing to see this effect.

Listing 13.17 Using mix blend mode to blend multiple elements

```
.blend {
  background-image: url("images/bear.jpg");
  background-size: cover;
  background-position: center;
  padding: 50vmin 0 1em;
}

.blend > h1 {
  margin: 0;
  font-family: Helvetica, Arial, sans-serif;
  font-size: 6rem;
  text-align: center;
  mix-blend-mode: hard-light;
  background-color: oklch(25% 0.2 25deg);
  color: #808080;
  border: 0.1em solid #ccc;
  border-width: 0.1em 0;
}
```

Uses hard-light blend mode

Dark red background color

Medium-gray text color will become transparent.

It can be difficult when using this effect to achieve high enough contrast for readability, 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.

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

- Color gradients may be defined as a background image on an element. These may be linear, radial, or conical gradients.

- Specify the color space of a gradient to control how colors of the gradient are interpolated.
- Combine gradients with shadows to add the appearance of depth on the page. Even “flat” designs may benefit from subtle shadows or gradients.
- Placing color stops at the same point in a gradient can produce stripes.
- Blend modes merge two or more backgrounds. Use this to colorize or texture an image.

14

Masks, shapes, and clipping

This chapter covers

- Applying filters to manipulate the appearance of elements or their backgrounds
- Masking out portions of an element with mask images
- Shaping elements with clip paths
- Floating elements to the left or right
- Aligning text to shapes

In the last chapter, I showed you some techniques for adding visual interest and creativity to the page. I'll continue down that path in this chapter, beginning with visual effects applied using several functions such as blurring elements or desaturating color. Then I'll show you how to use masks and clip paths, which provide two ways you can selectively hide portions of an element to create interesting shapes. Finally, you'll learn about floats and the `shape-outside` property, which allow you to wrap text around elements and align it to a shape.

These are effects you will need only in certain circumstances; applying them all at once will likely be a bit chaotic, so I will use smaller, isolated examples to illustrate each of these.

14.1 Filters

The `filter` property allows you to apply an effect such as a blur, color shift, or desaturation to an element. Several of these effects are available, but I'll begin by using blur as an example to illustrate how the property works.

This property applies a filter to the entire element, including its text, borders, and background. Figure 14.1 shows a tile similar to the one you built in chapter 12. On the left, no filter is applied. On the right, I have applied `filter: blur(3px)`, which applies a 3 px blur to the element.

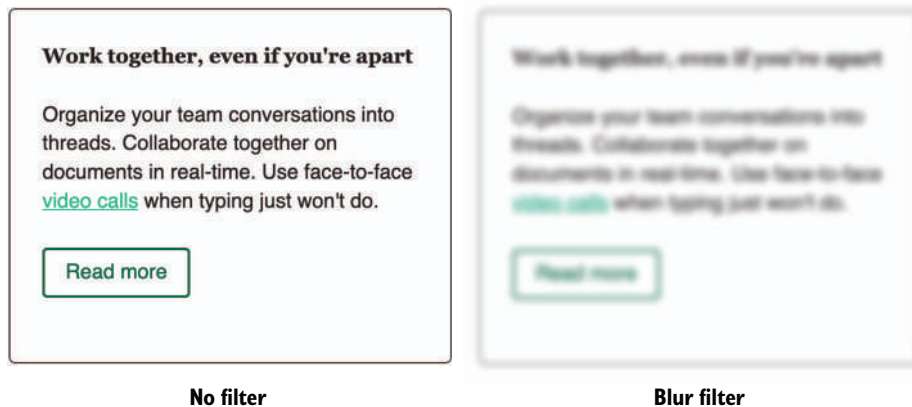


Figure 14.1 A filter applies to an entire element, including its border, children elements, and background.

This illustrates the principle, but obviously this is not a particularly useful application of a filter, as it makes the text nearly impossible to read. Instead, I'll show you how to apply a filter to an image.

Create a new HTML document and stylesheet. You will use these to place the various examples throughout this chapter. Add the markup shown in listing 14.1. As usual, the linked bird image is available in the code repository at <https://github.com/cssindepth/css-in-depth-2>.

Listing 14.1 Markup with bird image

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="style.css">
```

```
</head>
<body>
  
</body>
</html>
```

With a blur filter applied, the image will appear as shown in figure 14.2.



Figure 14.2 Image with a blur filter applied

The `blur()` function takes a single parameter, a length used to indicate the strength of the blur. This indicates how many pixels on the screen will blend together (though `em`, `rem`, and other length units may be used as well as `px`). A length of `0 px` indicates no blur; values above `0` will provide a blur. The larger this value, the stronger the blur becomes.

In your stylesheet, add the `filter` declaration as shown in the following listing to add a `5 px` blur to the image.

Listing 14.2 Applying a blur filter

```
img {
  filter: blur(5px);
}
```

Use your browser's DevTools to modify the `5 px` value up or down to see the effects a stronger or weaker blur can have on the image. With a low value, the image is still somewhat comprehensible, but the higher you go, the more it becomes a vague smudge of color. This can be useful when you need to give the general impression of an image without drawing too much attention. I'll show you an example of this shortly, but first, let's look at the other filter types available.

14.1.1 Types of filters

There are 10 different filter functions available. Most of these involve manipulating color in some way, either by adjusting contrast or color saturation, but there are also options for adding a drop shadow or changing opacity.

All the available filter functions are described in table 14.1. Like `blur()`, these all accept parameters that tune their behavior. Many of these values are percentages; these can be expressed as a percent (50%) or as a decimal (0.5). All percentage parameters are optional, defaulting to 100% if omitted. I encourage you to try each of these filters on your page to see the kinds of effects you can achieve.

Table 14.1 Filter functions

Function example	Description	Parameter(s)
<code>blur(10px)</code>	Applies a Gaussian blur	The higher the length, the stronger the blur effect.
<code>brightness(150%)</code>	Increases or decreases the brightness	A value lower than 100% reduces the brightness; a value greater than 100% increases it.
<code>contrast(150%)</code>	Increases or decreases the contrast	A value lower than 100% reduces contrast (fades the image); a value greater than 100% increases it.
<code>drop-shadow(10px 10px 15px black)</code>	Adds a drop shadow, similar to the <code>drop-shadow</code> property	The first two parameters are lengths indicating the x- and y-offsets, respectively. The third is a blur amount (optional). The fourth is a color (optional). Unlike the <code>drop-shadow</code> property, the spread radius value and <code>inset</code> keyword are not supported.
<code>grayscale(50%)</code>	Desaturates the color	A value between 0% and 100%, which produces a fully grayscale image.
<code>hue-rotate(30deg)</code>	Shifts the hue value of every pixel	Any angle indicating how far around the color wheel to shift colors.
<code>invert(100%)</code>	Inverts the colors	A value of 100% completely inverts the colors. A value between 0% and 100% applies the effect at the indicated strength.
<code>opacity(50%)</code>	Makes the element transparent, similar to the <code>opacity</code> property	A value of 0% makes the element completely transparent; 100% is fully opaque.
<code>saturate(150%)</code>	Increases or decreases the color saturation	Values higher than 100% increase the image color saturation; values lower than 100% desaturate the image. <code>saturate(25%)</code> is equivalent to <code>grayscale(75%)</code> .
<code>sepia(100%)</code>	Replaces color with a sepia-tone effect	A value between 0% and 100%, controlling the strength of the effect.

You can also use multiple filter functions at the same time by separating them with a space—for example: `filter: blur(5px) sepia(20%)`. When you do this, the filter effects are applied from left to right.

The `filter` property is most practical when used selectively or in a more subtle way. For instance, when the user hovers over an image, you can apply a slight blur or desaturation to all the other images in the gallery. Listing 14.3 shows how you might accomplish this using some `:hover` effects. I leave it as an exercise for you to write some HTML that will work with these styles if you wish.

Listing 14.3 Applying a slight filter to gallery images

```
.gallery:hover img {  
  filter: blur(2px) grayscale(50%);  
}  
  
.gallery img:hover {  
  filter: none;  
}  
  
.gallery img {  
  transition: filter 1s;  
}
```

← Applies filters to images when the mouse hovers over the gallery

← Removes the filter from the currently hovered image

← Gradually transitions the filter effect on and off over 1 second

Any time the mouse hovers over the gallery, the filter is applied to all the images within, except the one the user is currently hovering over. These filters illustrate the effect, but in a production application, I would probably make them a little more subtle: a blur of only 1 px and a grayscale filter with a lower percentage.

TIP A crafty way to achieve this same result is to use the `:has()` selector to target all nonhovered images when one image is moused over: `.gallery z:has(img:hover) img:not(:hover)`. Firefox only added `:has()` support in late 2023, however, so I might not use this approach without first checking the latest browser usage statistics.

I also used the `transition` property in this example. This causes the filters to turn on and off gradually, rather than in an instant. I found that, without the transition, the sudden change drew my eye away from the image being focused on, which is the opposite of the effect I was going for. I haven't covered transitions yet; we will take a close look at them in the next chapter.

14.1.2 Backdrop filter

Sometimes you might want to apply a filter behind some content, rather than applying it to the content directly. This can be done with a different property, `backdrop-filter`. This property accepts the same filter functions I've already covered.

A backdrop filter is useful when you want to place text in front of a background image. Using the right filter will allow the text to remain legible without entirely obscuring the image behind it. Figure 14.3 shows an example of this.



Figure 14.3 A backdrop filter can be used to obscure content behind a semitransparent element.

In this example, a background image is applied to an element. Within that element is another div that has a `blur()` backdrop filter and a semitransparent white background color. Unlike the `filter` property, the `blur` in this example does not apply to the text, nor would it apply to the border if the div had one. Instead, it only affects the content behind it, where the image shows through the div.

The markup for this page is shown in the following listing. Replace the contents of your demo page with this HTML.

Listing 14.4 Markup for backdrop filter

```
<div class="box">
  <div class="box__content">
    <h1>Common Kingfisher</h1>
    <p>
      The Common Kingfisher is a bird native found in Europe, Asia, and
      north Africa. It has bright blue plumage with an orange belly.
    </p>
  </div>
</div>
```

You'll apply the background image to the `box` and the `backdrop-filter` to `box__content`, along with some padding and margin to arrange the element. The styles for this are shown next. Add these styles to your stylesheet.

Listing 14.5 Backdrop blur filter

```
.box {
  padding: 30px;
  background-image: url(images/bird.jpg);
  background-size: cover;
  background-position: center 30%;
}
```

```

.box__content {
  max-inline-size: 500px;
  margin-inline: auto;
  padding: 50px 30px 70px;
  border-radius: 10px;
  background-color: oklch(100% none none / 0.3);
  -webkit-backdrop-filter: blur(10px);
  backdrop-filter: blur(10px);
}

```

Note that I’ve included the backdrop filter twice: first with the *-webkit vendor prefix*, then again without it. Safari only supports this property with the prefix, so it is necessary to include both.

Vendor prefixes

Vendor prefixes are an old technique browsers used to use when implementing new or experimental CSS features. Browsers no longer add features using vendor prefixes, instead relying on user-enabled Experimental Web Platform Features flags (see chapter 1). Using this flag prevents developers from relying on unstable features that are subject to breaking changes.

Vendor prefixes are rarely needed any longer. However, some browsers haven’t yet updated their support for a nonprefixed version of a small number of properties. There are a few such properties in this chapter: `backdrop-filter` and some `mask-*` properties. These each require the *-webkit* prefix to work in either Safari or Chrome. Other prefixes you may encounter are *-moz* (Firefox), *-o* (older versions of Opera), and *-ms* (Internet Explorer and older versions of Edge).

You can use automated tools to add prefixed properties when necessary. Autoprefixer is one such tool, which can be found at <https://autoprefixer.github.io/>. The preprocessor Lightning CSS (<https://lightningcss.dev/>) also offers this functionality. I don’t always rely on these any longer, because so few properties need prefixes now, but they can be helpful if you don’t want to mentally keep track of which properties require them.

A backdrop filter can be useful for small interface elements such as notification messages or application control buttons.

The previous example uses static positioning, where the background of the parent element shows through. The same effect can be applied to elements stacked in front of one another using fixed or absolute positioning as well.

14.2 Masks

Every visible element on the page is rendered as a rectangle. This can sometimes be limiting from a design perspective. A border radius can round off the corners of an element or even make it an oval by applying `border-radius: 50%`, but that only goes so far.

Fortunately, CSS provides a few techniques for manipulating the visual shape of an element. One of these techniques is called *masking*. Masking, or *applying a mask*, allows you to use an image as a reference for the browser to selectively hide or show portions of an element.

Figure 14.4 shows an example of a masked element. The element has the bird image from earlier set as its background image, and a star-shaped mask is applied. The mask image itself is shown in figure 14.5.



Figure 14.4 An element with a mask applied



Figure 14.5 Star-shaped mask image

The mask image is usually very simple, as in this example. The star is fully opaque, and everything surrounding the star is fully transparent. The browser essentially overlays it with the element, allowing only portions of the element to show where it aligns with visible portions of the mask. The exact color of the mask is irrelevant, but using black is helpful when viewing the mask image so its shape is easily visible.

WARNING For browser security reasons, a mask image will only load from a URL on the same domain unless you set up CORS (Cross-Origin Resource Sharing) headers. It will also not load using the `file://` protocol from your local disc. You can read more about CORS at <https://mng.bz/JZaV>.

To see this masked element in your page, replace the body of the HTML with a simple div:

```
<div class="mask"></div>
```

Next, add the styles shown in listing 14.6 to your stylesheet. This applies some sizing to the element, sets up the background image, and defines the mask.

The `mask-image` property (and the vendor-prefixed `-webkit-mask-image` fallback) is conceptually similar to the `background-image` property. Likewise, `mask-size` is similar to `background-size`; it also accepts the keyword values `cover` and `contain`.

Listing 14.6 Applying an image mask

```
.mask {
  width: 300px;
  min-height: 300px;
  background-image: url(images/bird.jpg);
  background-size: cover;
  background-position: 100%;
  -webkit-mask-image: url(images/star-mask.png);
  -webkit-mask-size: 100% 100%;
  mask-image: url(images/star-mask.png);
  mask-size: 100% 100%;
}
```

Vendor-prefixed fallback values (for Chrome)

Specifies the mask image URL

Sizes the mask image

To preview this on your page, you will not be able to open the HTML file directly in your browser; the mask will not be processed over the `file://` protocol due to browser security restrictions. You will need to serve it via an HTTP server.

One quick way to do this is to run the `http-server` Node.js package to statically serve files from your file directory. Install Node.js if you don't have it installed (see <https://nodejs.org> for instructions on your operating system).

With Node installed, run `npx http-server .` in the same directory as your HTML and CSS files. You should then be able to open your file at `http://localhost:8080/index.html` (changing the file path as needed to match yours). You can also use any other HTTP server if one is more familiar to you.

When defining masks, be sure to provide `-webkit-`prefixed fallback declarations for all the mask properties. Chrome and other browsers based on the Blink engine (Edge and Opera) have supported these properties only with the prefix until recently. Support for the unprefixed version of these was added at the end of 2023.

14.2.1 Masking with a gradient

The `mask-image` property can also be defined using a gradient, just as `background-image` can. The resulting gradient is treated as the mask image, so any transparent portions will mask out that portion of the element. (Unlike a `url()` path, this will work over the `file://` protocol while you're testing.)

Figure 14.6 shows the same element as previously but with a gradient applied as the mask image. The gradient fades from transparent at the top to black at the bottom.

The CSS for this mask is shown in listing 14.7. Update your stylesheet to include the changes shown in bold. Note that the `mask-size` property



Figure 14.6 A gradient applied as a mask

is no longer necessary in this case since the gradient automatically fills the element exactly.

Listing 14.7 Applying a gradient as a mask

```
.mask {  
  width: 300px;  
  min-height: 300px;  
  background-image: url(images/bird.jpg);  
  background-size: cover;  
  background-position: 100%;  
  -webkit-mask-image: linear-gradient(to bottom, transparent, black);  
  mask-image: linear-gradient(to bottom, transparent, black);  
}
```

You can experiment with different gradients to achieve various effects. Try something like `radial-gradient(black 40%, transparent 70%)` to apply a vignette effect. Or use a repeating gradient to apply stripes, such as `repeating-linear-gradient(45deg, transparent 0px 30px, black 30px 60px)`.

Working with masks can be a little difficult since they're essentially invisible. If, after adding a mask, your element is entirely visible or entirely hidden, it can be tricky to diagnose how the mask is being applied. If it isn't working as I expect, I sometimes find it helpful to temporarily change the `mask-image` to a `background-image` (and `mask-size` to `background-size`), since the syntaxes are the same. Then I can visibly see the mask as it's applied to the element and correct the declarations.

14.2.2 Masking using luminance

By default, the mask image is applied using its alpha channel; transparent or semi-transparent portions of it affect the opacity of the element to which it's applied. You can instead apply it based on its luminance—black and white rather than transparent and opaque. This may be helpful in situations where you don't have a transparent mask image available.

To change a mask to use this behavior, use the `mask-mode` property. Note that this property has only just been added to Chrome at the end of 2023; a `-webkit-`prefixed version of this property has not been supported. Check <https://mng.bz/wxVP> for the latest browser support statistics before relying on this for essential parts of your page.

The initial value of `mask-mode` is `alpha`. Assigning the value `luminance` changes the mask behavior so that black portions of the mask image produce transparent pixels and white portions produce opaque pixels. Shades between white and black produce partially transparent masking accordingly.

Listing 14.8 shows CSS using this approach. This produces the exact same result as the previous listing using the gradient's luminance rather than its opacity for the mask.

Listing 14.8 Masking via image luminance

```

.mask {
  width: 300px;
  min-height: 300px;
  background-image: url(images/bird.jpg);
  background-size: cover;
  background-position: 100%;
  -webkit-mask-image: linear-gradient(to bottom, black, white);
  mask-image: linear-gradient(to bottom, black, white);
  mask-mode: luminance;
}

```

Defines the mask in shades of gray

← Sets luminance mask mode

Note that in the previous example (listing 14.7) you used black as the opaque color for the visible portion of the mask—the exact color was irrelevant as long as it was fully opaque—but black behaves differently here. When using luminance as the mask mode, black is transparent and white is visible.

14.2.3 Other mask properties

There are multiple mask properties to adjust the behavior, size, and position of the mask. Many of these are conceptually similar to some of the `background-*` properties:

- **mask**—A shorthand property for all of the following properties.
- **mask-image**—Specifies the image to use for the mask. This may be an image file referenced via `url()`, a gradient function, or the ID selector of an SVG `<mask>` element defined on the page (for example, `url(#my-svg-mask)`). Multiple masks can be provided, each separated by a comma.
- **mask-position**—Specifies `x` and `y` values of the mask position relative to the top-left corner `mask-origin`. The keyword values `top`, `right`, `bottom`, `left`, and `center` are also supported.
- **mask-size**—Sets the size of the image mask. This can be `x` and `y` values as a length or percentage or one of the keywords `cover`, `contain`, or `auto`. If you use the **mask** shorthand property, this must follow the **mask-position**, separated by a slash, such as `mask: url(/mask.png) 10px 10px / 90%`.
- **mask-repeat**—Specifies whether the mask image should tile for a repeating effect. The initial value is `repeat`. Other values are `no-repeat`, `repeat-x`, `repeat-y`, `space`, and `round`. `space` and `round` will repeat the mask as much as the element will allow without clipping a portion of it off; `space` adds a gap between each copy of the mask, while `round` allows the repeated mask to stretch to fill the element.
- **mask-origin**—Specifies the origin used for positioning the mask. The initial value is `content-box`, but you can change this to `padding-box` or `border-box`.
- **mask-clip**—Specifies the area of the element that will be affected by the mask. The initial value is `border-box`, but you can change this to `padding-box` or `content-box`.

- `mask-mode`—Specifies whether to use the alpha channel or the luminance of the mask image to define the mask. This can be set to `alpha`, `luminance`, or `match-source`. The initial value is `match-source`, which uses an alpha channel for image masks but a luminance channel for SVG sources referenced via an ID selector (see `mask-image`).
- `mask-composite`—If multiple mask images are defined, this specifies how they are to be combined. Supported values are `add` (the initial value), `subtract`, `intersect`, and `exclude`. See <https://mng.bz/qOWr> for a closer look at this.

Chrome released a large update of its mask support in December 2023. Prior to this, the properties `mask-clip`, `mask-mode`, and `mask-composite` were not fully supported, and the other properties all required the `-webkit` vendor prefix.

14.3 *Clipping paths*

A clipping path is another method for selectively hiding a portion of an element. It is conceptually similar to masks, but instead of using an image to mask portions of the element, it uses a mathematically defined shape. This is specified with the `clip-path` property.

I'll show you how to build a simple example first and then work up to something more complex. Figure 14.7 shows the result you will achieve in the first example. This clipping shape is defined as a circle. All portions of the element outside the clipping shape are hidden. In this case, the circle is larger than the height of the image, so the result has straight top and bottom edges and curved sides.



Figure 14.7 A rectangular image with a circular clipping path applied

First, add a new image to your page to work with, as shown in listing 14.9. You can either replace the previous markup or append this to the end of your page.

Listing 14.9 Image markup for the clipping path

```

```

You'll use the `clipped` class name to target this element with your CSS selector, then use the `clip-path` property to set a clipping path. One simple way to use this property is with the `circle()` function. This function accepts a length or percentage specifying the radius of the circle. Next, add the styles shown in the following listing to your stylesheet.

Listing 14.10 Applying a circular clipping path

```
.clipped {  
  clip-path: circle(398px);  
}
```

A diagram consisting of a horizontal line with a vertical tick mark at its right end. A curved arrow points from the text "Clips a circle with a radius of 398 px" to the tick mark.

Clips a circle with a radius of 398 px

You defined the image with a width of 796 pixels, so this radius of 398 is exactly half of that. You can also achieve this same result with a special keyword: `circle(farthest-side)`. Similarly, specifying `circle(closest-side)` will size the circle so it reaches the top and bottom edges of the element, as those are the closest sides to the center.

NOTE Setting `clip-path` to any value other than the initial value `none` creates a new stacking context (see chapter 6).

This example has the circle centered on the middle of the element, but you can reposition the clipping path elsewhere using the `at` keyword and x- and y-coordinates. The center of the circle will be placed where these coordinates indicate. Update the clipping path in your stylesheet to the following snippet to see an example:

```
.clipped {  
  clip-path: circle(229px at 337px 293px);  
}
```

This centers the clipping shape on the eagle's head and sizes it so most of the image is hidden, apart from the eagle's face and beak. These x- and y-coordinates are measured from the top left of the element's border box. The size and position of the circle may also be specified as a percent.

You can also define an oval clip path using the `ellipse()` function. This is similar to the `circle()` function, except it takes two size parameters, defining both a vertical and horizontal radius. Figure 14.8 shows an image clipped to an ellipse.

The code for this clip path is shown in listing 14.11. Update your stylesheet to match this ruleset.



Figure 14.8 An ellipse clip path

Listing 14.11 An ellipse clip path

```
.clipped {
  clip-path: ellipse(300px 170px at 360px 290px);
}
```



Oval with a 300 px
horizontal radius and a
170 px vertical radius

Keep in mind that, as with masks, none of these clipping paths actually change the size of the element being clipped; they simply hide portions of it. The element still takes up the normal amount of space on the page, leaving empty space where it is clipped.

For most practical applications, you'll want to ensure your clipping shape is sized close to the full width and height of the element being clipped. When clipping images, it often makes sense to crop the image down to size in an image editor and then apply a clipping path to shape it how you like.

14.3.1 Polygon clipping paths

You can define more complex shapes using the `polygon()` function. This function accepts any number of x- and y-coordinate pairs, with a comma between each pair. Each of these coordinates sets the position of a vertex of the polygon. For example, `polygon(50% 0%, 100% 100%, 0% 100%)` defines a triangle shape from the top center of the element (50% 0%) to the bottom right (100% 100%) and the bottom left (0% 100%).

You can use this function to define a shape with an arbitrary number of points, making it as simple or complex as you wish. Another example is shown in the following listing. Update your stylesheet to use this clip path.

Listing 14.12 A polygonal clipping shape

```
.clipped {
  clip-path: polygon(380px 50px, 650px 210px, 520px 500px, 20px 360px);
}
```



Polygon with four vertices

This clip path produces the result shown in figure 14.9. This is a sort of kite shape, turned on its side.



Figure 14.9 Polygonal clipping

With polygons, you can either shape the visible space around a key portion of an image, as I have done here, or you can carve out an interesting border near the edges of the element. The article by Temani Afif at <https://mng.bz/7d0v> shows several interesting borders created using gradient masks and clip paths.

TIP See <https://bennettfeely.com/clippy/> for a very helpful clip-path maker tool, with multiple useful preset shapes to build from.

14.3.2 Firefox clip-path tools

Writing a clipping path, especially a polygonal one, can be tedious and requires a lot of fiddling with values to get it looking how you want. Firefox DevTools has a visual editing tool that I find invaluable for building or adjusting clip paths. Unfortunately, none of the other major browsers yet have this tool.

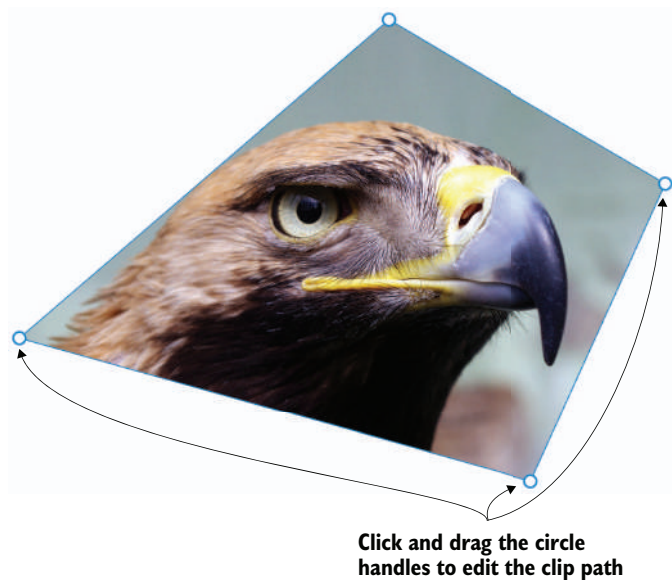
In Firefox, inspect the image on your page that has a clip path by right-clicking and selecting Inspect. In the Rules pane, you will see the `clip-path` property applied to the element. Alongside the `clip-path` function is a small polygon icon (figure 14.10). This will be present for any clip path defined using `circle()`, `ellipse()`, `polygon()`, or `inset()` (which I will cover shortly).

Clicking this polygon icon will enable interactive editing of the `clip-path` in the main browser window. The shape of the path is outlined in a thin blue line, and a small circle appears at each point that can be edited, as shown in figure 14.11. Click and drag one of the circles to edit the shape. You should see your changes appear in the DevTools Rules pane.



Click to enable interactive clip-path editing

Figure 14.10 Clicking the polygon icon to edit the clip path



Click and drag the circle handles to edit the clip path

Figure 14.11 Interactive clip-path editing in Firefox

When editing a polygon, there is a control handle for each vertex; double-click on an edge of the polygon to add a new point to the shape. When editing a circle, you will see two control handles: one for the center to move the center point and one on the right to adjust the radius. An ellipse is similar, with an additional control point on the bottom to control the vertical radius.

14.3.3 Other clip-path types

There are a few other types of clip paths available. I'll leave it to you to experiment with these:

- `inset()`—Clips to a rectangle within the element, inset from its edges by the specified amount. Providing one value, such as `inset(15px)`, places the clip path that far in on each side of the element. Providing two, three, or four values allows you to control the various sides independently, similar to the way the padding property can accept different values for each side.
- `path()`—Clips to a given SVG path string—for example: `path("M68,312C17,239 117,63 223,62C328,61 409,276 370,319C330,365 118,384 68,312Z")`. SVG paths are generally too cryptic to edit by hand; you will need to export them from vector-editing software.
- `margin-box`—Clips to the element's margin box.
- `border-box`—Clips to the element's border box.
- `padding-box`—Clips to the element's padding box.
- `content-box`—Clips to the element's content box.

The `path()` function is the most versatile of these, as it provides a lot of freedom to define complex shapes and curves beyond what `polygon()` is capable of. However, it requires familiarity with and access to vector-editing software. For a quick primer on the syntax of an SVG path, see Chris Coyier's write-up at <https://mng.bz/ma0a>.

14.4 Floats and shapes

Using masks or clip paths to shape elements can be visually interesting, but the element remains a simple rectangular box in the layout of the page. The `shape-outside` property allows you to define a more complex shape for this underlying layout. This causes inline content, such as text, to wrap around the element in a different pattern other than a straight, vertical edge.

This is often useful in conjunction with a mask, clip path, or border radius that follows the same contours. This way the element appears to have a shape, and the text beside it flows around the same shape. Figure 14.12 shows an example of this: the eagle image is clipped to a circle and the text beside it wraps around that same circle, which is defined using `shape-outside`.

To use `shape-outside`, the element must also be *float*ed. Floating an element pulls it to one side, allowing inline content to flow around its margin box. Floating can be a complex topic in itself, so I'll take some time to explain how that works on its own before adding in shapes.



Hope is the thing with feathers

Emily Dickinson

“Hope” is the thing with feathers
That perches in the soul,
And sings the tune without the words,
And never stops at all,

And sweetest in the Gale is heard;
And sore must be the storm
That could abash the little bird
That kept so many warm.

I’ve heard it in the chilliest land,
And on the strangest sea;
Yet, never, in extremity,
It asked a crumb of me.

Figure 14.12 Defining a shape causes inline content to flow around the contours of the shape.

14.4.1 Floating

Floating pulls an element (often an image) to either the left or right side of its container, allowing the document flow to wrap around it. Figure 14.13 illustrates the behavior of a floated element in the middle of a paragraph of text. This layout is common in newspapers and magazines, so floats were added to CSS to achieve this effect.

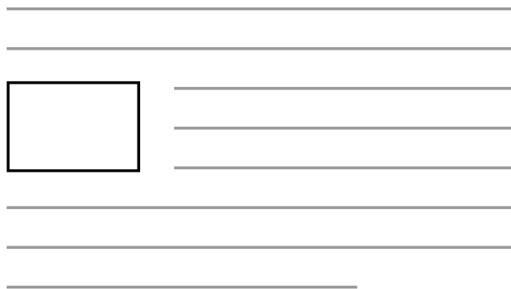


Figure 14.13 Lines of text wrap around a floated element.

A floated element is removed from the normal document flow and brought to the edge of the container. The document flow then resumes, but inline content wraps around the space where the floated element now resides. The floated element is not moved up or down from its initial location in the document flow; this means its final position is dependent upon where it appears amid the text in the HTML.

You float an element by applying `float: left` or `float: right` to float it in the respective direction. If you float multiple elements in the same direction, they’ll stack alongside one another, as shown in figure 14.14.

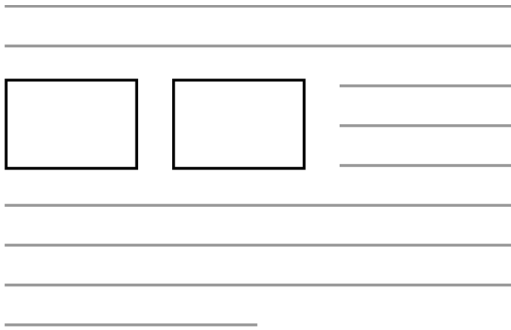


Figure 14.14 Two floated elements stacked alongside one another

I'll have you add some content to your page and float an image beside it. After that's in place, you will add some clipping and define a shape for the text. Add the markup shown in the next listing to your HTML. You can either delete the existing content in the `<body>` or add this beneath it.

Listing 14.13 Page content with an image to float

```
<div>
  <div class="poem-image">      ← Element to be floated
    
  </div>
  <h1>Hope is the thing with feathers</h1>
  <p><cite>Emily Dickinson</cite></p>
  <p>
    "Hope" is the thing with feathers<br>
    That perches in the soul,<br>
    And sings the tune without the words,<br>
    And never stops at all,
  </p>
  <p>
    And sweetest in the Gale is heard;<br>
    And sore must be the storm<br>
    That could abash the little bird<br>
    That kept so many warm.
  </p>
  <p>
    I've heard it in the chillest land,<br>
    And on the strangest sea;<br>
    Yet, never, in extremity,<br>
    It asked a crumb of me.
  </p>
</div>
```

TIP I've put the `<img>` in this example inside an extra `div`. This isn't strictly necessary, but I have found in certain cases it is easier to work with shapes on a "normal" element rather than an image. An image is technically a *replaced element* in HTML, and in a few cases, its size or position can behave a little

differently than a normal element. Specifically, using `object-position` surfaces a bug in Chrome that can affect the position of text that wraps the element.

Next, you'll apply a little CSS to size the image appropriately and float it to the left. After this is added, you'll end up with the result shown in figure 14.15.

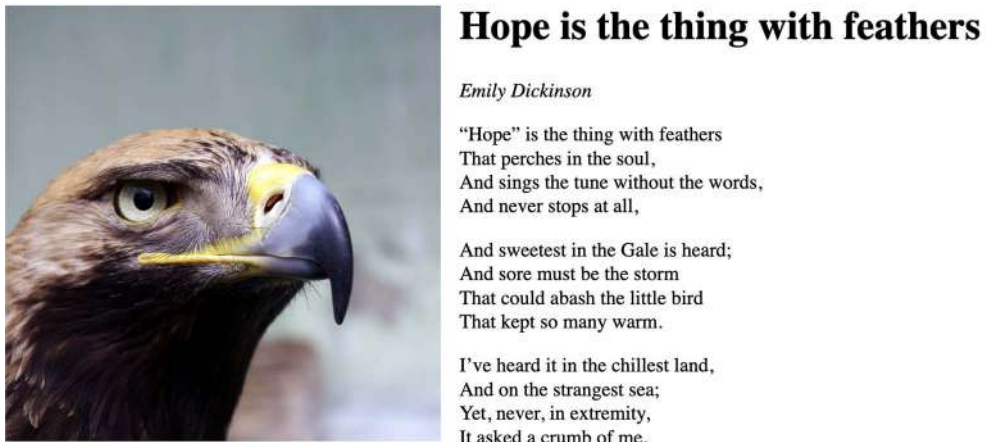


Figure 14.15 Image floated left of text

Add the CSS in listing 14.14 to your stylesheet to see this in your browser. I've included `display: block` on the image so the line height doesn't add extra height to its parent div—though I typically already have this rule applied in my reset styles (see chapter 8).

Listing 14.14 Floating the image to the left

```
.poem-image {  
  float: left;  
  margin-right: 15px;  
}  
  
.poem-image > img {  
  display: block;  
  height: 350px;  
  width: 350px;  
  object-fit: cover;  
}
```

In the HTML, the location of the floated element amid the other inline content is important for determining which lines of text will wrap around it. Content on lines above the image will appear above it, but any text that would render on the same line

or lines beneath that will wrap around it. In this example, you want the entire poem to wrap to the right of the image, so you have placed the image first in the HTML.

If you are new to floats, I encourage you to move the element around to get a feel for how this behaves. You can place it between paragraphs to see where it ends up. If you are floating an inline element, you can even place it in the middle of a sentence.

One interesting—and sometimes unexpected—thing about floats is they do not add any height to their parent element. If a floated element is the only child of a `<div>`, the div will have a height of 0. This is done so you can float an image inside one paragraph (`<p>`), and the content of successive paragraphs will continue to flow around the image, as illustrated in figure 14.16.

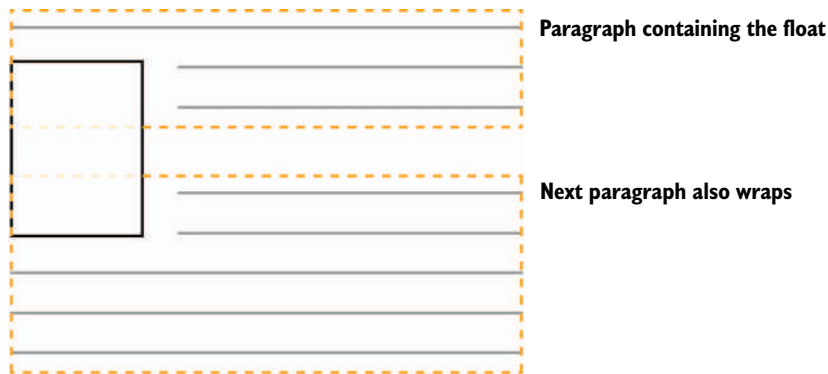


Figure 14.16 The float in one container extends into the next container, allowing text in both containers to wrap around the floated element.

This behavior makes sense when both of the containers are paragraphs, but it also applies if they are divs or any other block-level element. When you want to force an element to appear beneath a floated element, you can apply `clear: both` to it; you may also clear left or right to clear only elements floated in the indicated direction.

You can force the container itself to contain its floated child by applying `clear` to its `::after` pseudo-element. Then the height of the element will extend to the bottom of the floated child:

```
.contain-floats::after {  
  display: block;  
  content: "";  
  clear: both;  
}
```

Before CSS had flexbox or grid layout, floats were the only option to build complex page layouts. Because this was not their original purpose, it was often a difficult and hacky process to make this work. With modern layout techniques available to us,

however, we can avoid that, only using floats when needed for their original purpose: wrapping text around an element.

14.4.2 Defining a shape

Now that the image on your page is floated, you can define a shape for the text beside it using `shape-outside`. This property accepts most of the same values that `clip-path` does: `margin-box`, `border-box`, `padding-box`, `content-box`, `inset()`, `circle()`, `ellipse()`, and `polygon()`. The notable exception is the `path()` function for SVG paths—this is not supported in any browsers, though it is included in the CSS specification, so it may be available in the future.

This often means you can repeat the exact same value for the element's `clip-path` and its `shape`. It may be helpful to set this value to a custom property and then reference it in both of these properties, especially if you define a more complex polygon. Update your CSS to match the following listing to set both of these values to a circle.

Listing 14.15 A circle clip path and shape

```
.poem-image {  
  float: left;  
  margin-right: 15px;  
  --shape: circle(50%);  
  clip-path: var(--shape);  
  shape-outside: var(--shape);  
}  
  
.poem-image > img {  
  display: block;  
  height: 350px;  
  width: 350px;  
  object-fit: cover;  
}
```

Now the eagle image is clipped to a circle, and the text beside it wraps around it neatly. The result is shown in figure 14.17 (this is identical to figure 14.12 shown earlier).

You can modify the `--shape` custom property to another shape using `polygon()`, `ellipse()` or another value to see the text wrap around that shape. The values `margin-box`, `border-box`, `padding-box`, and `content-box` are useful when the floated element has a `border-radius` that you want to wrap text around.

In fact, since a circle shape is achievable by using a border radius of 50%, you can produce the same result by using a border radius rather than a clipping path, as shown in listing 14.16. Note that this example requires the border radius on both the image (to clip the image) and on its div (to influence the `shape-outside`).



Hope is the thing with feathers

Emily Dickinson

"Hope" is the thing with feathers
That perches in the soul,
And sings the tune without the words,
And never stops at all,

And sweetest in the Gale is heard;
And sore must be the storm
That could abash the little bird
That kept so many warm.

I've heard it in the chillest land,
And on the strangest sea;
Yet, never, in extremity,
It asked a crumb of me.

Figure 14.17 Text wrapped around a circle shape

Listing 14.16 A shape around a border radius

```
.poem-image {
  float: left;
  margin-right: 15px;
  shape-outside: margin-box;
  border-radius: 50%;
}
```

← The shape will
contour around
the border radius.

```
.poem-image > img {
  display: block;
  height: 350px;
  width: 350px;
  object-fit: cover;
  border-radius: 50%;
}
```

This example uses a right margin to add a small gap between the image and the wrapping text. There is also a `shape-margin` property you can use instead. For example, `shape-margin: 15px` will add a 15 px margin in all directions along the contours of the `shape-outside`.

A `shape-margin`, however, will not extend outside the floated element's margin box. In cases like this one, where the clip path comes right to the edge of the element, I find this does not give the result I want—the text near the top and bottom will follow the contours of the circle, but text in the middle will follow the straight edge of the margin box. I find it best to use `shape-margin` only when I am sure there is adequate space between the edges of the shape and the edge of the element. I'll show an example of this property in a moment.

MATCHING A SHAPE TO A GRADIENT MASK

The `shape-outside` property can also be set to an image, just as a mask can. This is particularly useful with gradients. These allow you to cut a diagonal edge off the element and align text to it, as shown in figure 14.18.

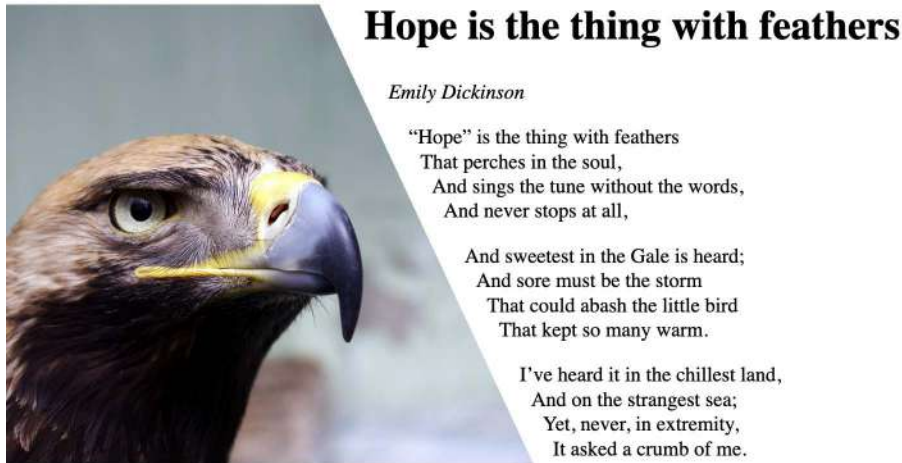


Figure 14.18 Use a linear gradient to define a diagonal shape.

In this example, I used the same linear gradient, set at a 65-degree angle, to define both the image mask and the shape. Edit the CSS on your page to use this gradient as shown in the following listing. Note that this listing no longer restricts the width of the image, allowing more space for the diagonal-shape edge.

Listing 14.17 A shape defined by a gradient

```
.poem-image {
  float: left;
  --gradient: linear-gradient(
    65deg,
    black 0 375px,
    transparent 375px);
  mask: var(--gradient);
  shape-outside: var(--gradient);
  shape-margin: 15px;
}

.poem-image > img {
  display: block;
  height: 350px;
  object-position: -100px 0;
}
```

Diagonal
gradient

Adjusts image
to keep the
eagle in frame

By setting the color stops at 375 px, this gradient has a hard cut from black to transparent. I arrived at this value, and the 65-degree angle, by adjusting values in my browser DevTools. I then added the `object-position` to bring the eagle back into frame, as it was partially cut off on the right side of the shape. I encourage you to adjust these values in your browser as well, to see how different slopes look.

The `shape-margin` property works here because the floated element extends to the right beyond the edge of the shape. A radial gradient can be used as well. For example, setting the `--gradient` property to `radial-gradient(circle at 175px, black 175px, transparent 175px)` produces a circular shape nearly identical to the `circle(50%)` clip-path example (listing 14.15).

In the previous linear gradient example, you defined a hard color stop from black to transparent, but this is not required. If you define a smooth gradient from black to transparent, the shape will be defined along the edge where this gradient reaches 0% opacity.

The `shape-image-threshold` property can modify the edge where this takes effect. For example, setting `shape-image-threshold: 0.3` will wrap text around the shape where the gradient image reaches 30% opacity. You can use this to make the text overlap the image where it is still partially opaque, as shown in figure 14.19.

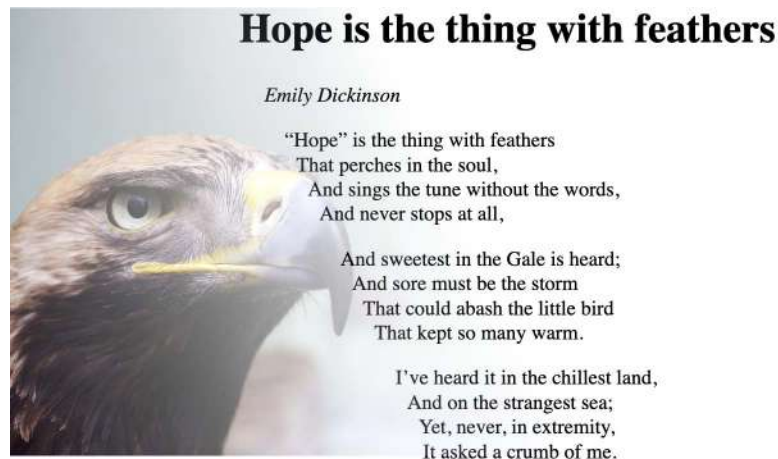


Figure 14.19 Setting a shape image threshold defines the opacity level where the shape is drawn.

The effects in this chapter can set your page apart from many more common designs on the web. These features more closely match things you are likely to see in magazines or other types of print design. The use of floats on the web nearly disappeared with the advent of flexbox and grid layout, but it still has its place, especially if you want to do something unique that stands out.

Summary

- A filter applies color manipulation or a blur effect to an entire element.
- A backdrop filter applies a filter effect only to images or content that show through from behind the affected element.
- A mask selectively hides a portion of an element based on the opacity of the mask image or mask gradient.
- A clipping path selectively hides a portion of an element based on the shape defined by the clipping path. This path is typically a circle, ellipse, or custom-defined polygon.
- Floating an element pulls it to the left or right side of its container and causes inline content to flow around its shape.
- The shape of a floated element can be manipulated with the `shape-outside` property. This can be defined using the clip path or mask image of the element so that wrapped text follows the contours of the visible portion of the element.

Part 5

Adding motion

Motion and dynamic change are the final piece of polish you can add to a page. Elements can fade out; menus can slide in; colors can shift. At its most basic, this may make the page more interesting to interact with, but done well, it goes beyond this. The right change at the right time effectively communicates something to the user or draws their attention to the appropriate part of the page. In these final chapters, we'll look at the parts of CSS that bring motion to the page.

15

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. You can add motion and change to the page, and the easiest way to do any of this 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 have 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. Some use cases can get a bit more complicated, so we'll take a look at a number of various scenarios.

15.1 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 smoothly instead of changing immediately to the new value.

You'll build a basic example using a button, then we'll 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 15.1, along with the in-transition intermediate state.

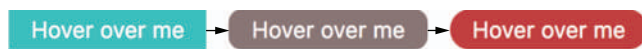


Figure 15.1 The element before, during, and after transition

Add a button to a new page and link it to a stylesheet. The markup for the button is shown in the following listing.

Listing 15.1 Adding a simple button to the page

```
<button type="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 15.2 Button styles with a transition

```
button {
  padding: 0.3em 1em;
  border: 0;
  font-size: 1rem;
  color: white;
  background-color: oklch(74% 0.11 195deg);
  transition-property: all;
  transition-duration: 0.5s;
}

button:hover {
  border-radius: 1em;
  background-color: oklch(55% 0.16 24deg);
}
```

Teal button

Transitions all property changes

Transitions for 0.5 seconds

Hover state red button 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. The transitions also apply in reverse when you move the mouse cursor off of the element. 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.

TIP You can use the JavaScript `transitionend` event to perform an action after a transition completes.

You can also use the shorthand property, `transition`. The syntax for this is shown in figure 15.2. The shorthand accepts up to four values for the four transition properties: `transition-property`, `transition-duration`, `transition-timing-function`, and `transition-delay`.

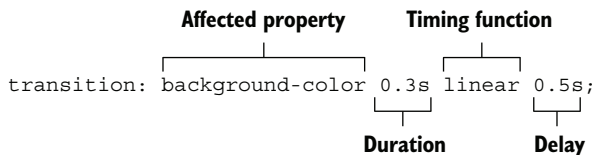


Figure 15.2 The transition shorthand property syntax

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, time values don't accept a unitless `0`. 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 `0.5s` transition delay, you will 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;
```

Alternatively, use the longhand properties. The following is equivalent:

```
transition-property: border-radius, background-color;  
transition-duration: 0.3s, 0.6s;  
transition-timing-function: linear, ease;
```

We'll look at an example using multiple transitions later in this chapter.

Accessibility: Reduced motion

It's important to consider how motion on the page, especially large amounts of it, can affect your users. Some users may have vestibular motion disorders; this is often a problem with the inner ear that can cause varying degrees of discomfort when certain motion occurs onscreen. Operating systems allow these users to request reduced motion onscreen.

You should check for this option using the `prefers-reduced-motion` media query and adjust your CSS accordingly. In chapter 8, I suggested the following ruleset as part of your reset styles. By including this in your stylesheet, all your transitions and animations will be removed so you won't need to do it repeatedly:

```
@layer reset {  
  @media (prefers-reduced-motion: reduce) {  
    *,  
    *::before,  
    *::after {  
      animation-duration: 0.01ms !important;  
      animation-iteration-count: 1 !important;  
      transition-duration: 0.01ms !important;  
      scroll-behavior: auto !important;  
    }  
  }  
}
```


This makes use of the `!important` annotation to ensure transitions and animations are essentially removed in all cases. Note that this doesn't set `transition: none` but rather specifies a duration so short the motion will not be noticeable. This allows aspects of the page that rely on the transition or animation—such as the JavaScript `animationend` event—to continue working normally.

15.2 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. In contrast, `ease-out` decelerates, starting with a rapid change and ending slowly. Figure 15.3 illustrates how a box would move from left to right with these timing functions.



Figure 15.3 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 I'll have you build an example to see it live in the browser. Create a new HTML page and add the following code.

Listing 15.3 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 the following listing into it.

Listing 15.4 Transitioning the box from left to right

```

.container {
  position: relative;
  height: 50px;
}

.box {
  position: absolute;
  left: 0;
  height: 50px;
  width: 50px;
  background-color: oklch(70% 0.18 145deg);
  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 topics 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.

15.2.1 Custom 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 15.4, 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.

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.

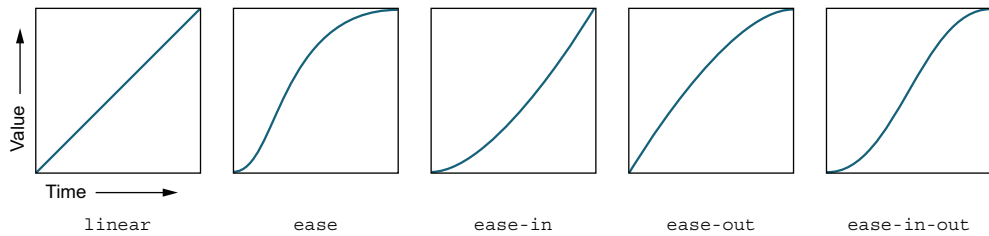


Figure 15.4 The Bézier curves of timing functions illustrate how the value changes over time.

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 15.5).

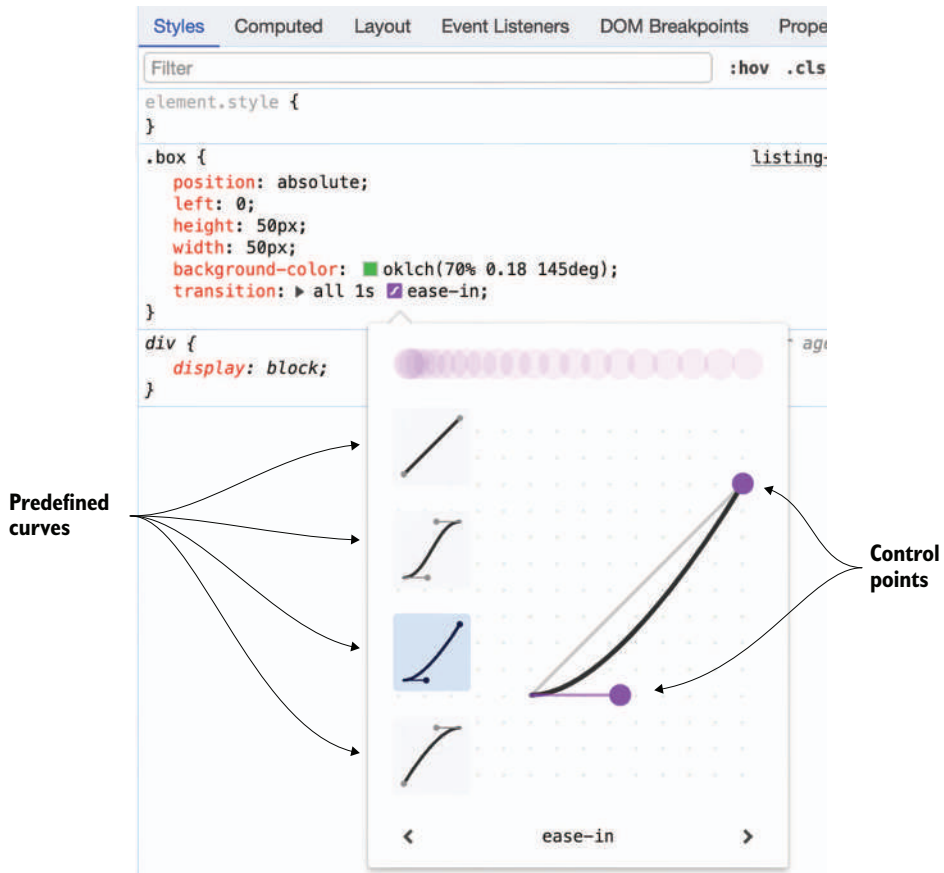


Figure 15.5 Editing a Bézier curve in the Chrome DevTools

On the left of the popup, this interface provides a series of premade curves to choose from. Firefox displays more than Chrome, though Chrome allows you to cycle through additional options with the small left and right arrows near the bottom. You can click one of the curves to select it. On the right, the selected Bézier curve is shown at a larger scale.

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 handles 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, an accelerating curve, as well as the `linear` keyword. It’s best to reuse 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 16 for an example of a bounce).

Let’s take a closer look at how `cubic-bezier()` works. Another example curve is shown in figure 15.6, which 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).

Because the two end points are fixed, the positions of the two handles are all you need to define the curve. In CSS, this curve can be defined as `cubic-bezier(0.45,`

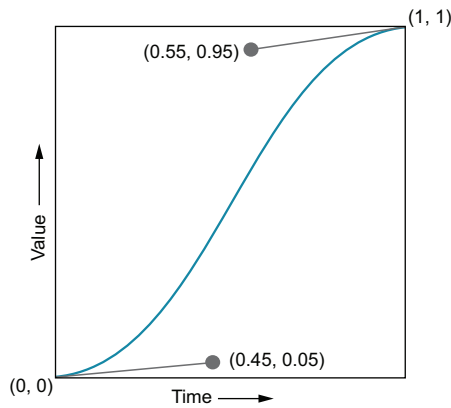


Figure 15.6 A Bézier curve representing a timing function

0.05, 0.55, 0.95). The four values represent the x- and y-coordinates of the two handles' control points.

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 using the DevTools for this, but you can also use an online resource such as <https://cubic-bezier.com>.

15.2.2 Steps

One final 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 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 15.7.

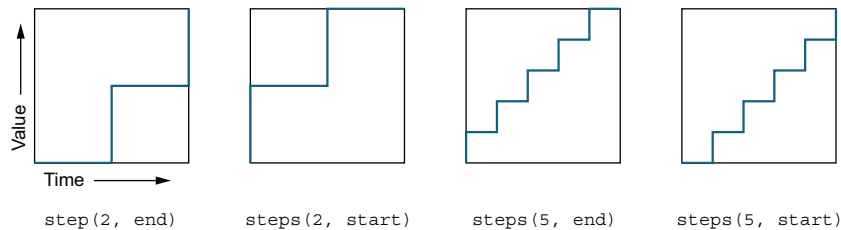


Figure 15.7 The `step()` function changes the value incrementally.

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 the following listing.

Listing 15.5 Using `steps()` to increment the value

```

.box {
  position: absolute;
  left: 0;
  height: 50px;
  width: 50px;
  background-color: oklch(70% 0.18 145deg);
  transition: all 1s steps(3);
}

```

Transitions in three discrete steps

Now, instead of moving fluidly from left to right for 1 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/>.

15.3 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 15.8. 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 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 in listing 15.6. 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.

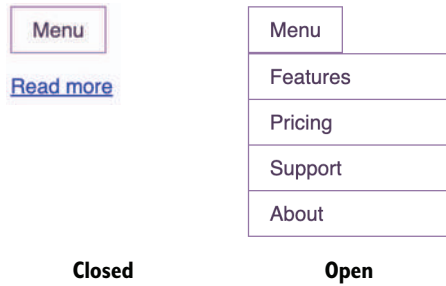


Figure 15.8 The dropdown menu in its closed (left) and open (right) states

Listing 15.6 Transitional dropdown menu

```
<div class="dropdown" aria-haspopup="true">
  <button class="dropdown__toggle" type="button">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>

<script type="module">
  var toggle = document.getElementsByClassName('dropdown__toggle')[0];
  var dropdown = toggle.parentElement;
  toggle.addEventListener('click', function () {
    dropdown.classList.toggle('is-open');
  });
</script>
```

The drawer that will appear and disappear to reveal the menu

A link that will appear below the dropdown

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 listing 15.7. 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 15.7 Transitional dropdown menu styles

```
@layer global, modules;

@layer global {
```

```

body {
  font-family: Helvetica, Arial, sans-serif;
}

@layer modules {
  .dropdown {
    --border-color: oklch(61% 0.08 314deg);
    --text-color: oklch(39% 0.06 314deg);
    --text-color-focused: oklch(39% 0.2 314deg);
    --background-color: white;
    --highlight-color: oklch(95% 0.01 314deg);
  }

  .dropdown__toggle {
    display: block;
    padding: 0.5em 1em;
    border: 1px solid var(--border-color);
    color: var(--text-color);
    background-color: var(--background-color);
    font: inherit;
    text-decoration: none;
    transition: background-color 0.2s linear;
  }

  .dropdown__toggle:hover {
    background-color: var(--highlight-color);
  }

  .dropdown__drawer {
    position: absolute;
    display: none;
    background-color: var(--background-color);
    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: var(--text-color);
    background-color: var(--background-color);
    text-decoration: none;
    transition: all 0.2s linear;
    border: 1px solid var(--border-color);
  }

  .menu > li > a:hover {

```

Transitions the background color when it changes

Changes the background color on hover

Transitions background and text colors


```
background-color: var(--highlight-color);
color: var(--text-color-focused);
}
}
```

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 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 16), use slightly longer transitions between 300 and 500 ms.

When I'm working on a transition, I sometimes slow it down to 2 or 3 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.

15.3.1 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/>) 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://mng.bz/1Gdj> are shown in figure 15.9.

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 17. 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 Formal Definition table.

TIP 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.

Formal definition

Initial value	transparent
Applies to	all elements. It also applies to <code>::first-letter</code> and <code>::first-line</code> .
Inherited	no
Computed value	computed color
Animation type	a color

Figure 15.9 MDN documentation provides a technical summary box for each property.

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.

NOTE The specification of the `display` property has recently changed so it can be transitioned similar to the way `visibility` works. This will require the addition of a `transition-behavior: allow-discrete` property. This does not yet have broad browser support, but you can read more about it at <https://mng.bz/PZqn>.

15.3.2 Fading in and out

Next, you’ll use an `opacity` transition to give the menu a fade-in and fade-out effect as it opens and closes. The result will be like that shown in figure 15.10.



Figure 15.10 Menu fading in

The `opacity` property can be any value between 0 (fully invisible) and 1 (fully opaque). Listing 15.8 shows the basic idea. This alone won’t work, however, for reasons you’ll soon see. Go ahead and edit your stylesheet to match the following listing.

Listing 15.8 Adding opacity and transition rules

```

.dropdown__drawer {
  position: absolute;
  background-color: var(--background-color);
  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. 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`, whereas 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 the `visibility` property's ability to animate with a little trick. Change your CSS to match the code in the next listing; then I'll walk you through how it works.

Listing 15.9 Using a transition delay to manipulate when visibility changes

```

.dropdown__drawer {
  position: absolute;
  background-color: var(--background-color);
  width: 10em;
  visibility: hidden;
  opacity: 0;
  transition:
    opacity 0.2s linear,
    visibility 0s linear 0.2s;
}
.dropdown.is-open .dropdown__drawer {
  visibility: visible;
  opacity: 1;
  transition-delay: 0s;
}

```

Hidden and transparent when closed

Delays the visibility transition 0.2 seconds when closing

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 (instantly) 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.

The fading effect in this example 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 the desired effect (which you’ll see in a moment). When a transition or animation can be accomplished through CSS alone, that’s almost always preferable.

15.4 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 15.11.

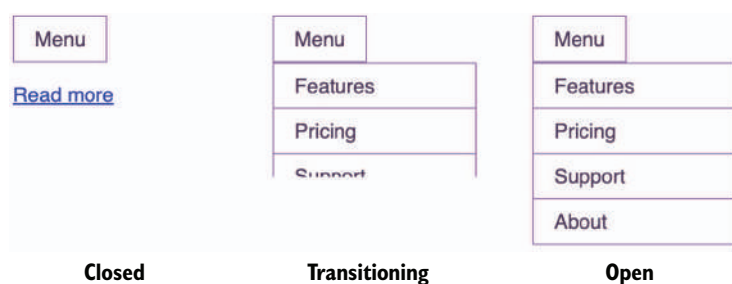


Figure 15.11 Sliding open the element by transitioning its height

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 listing 15.10. 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 15.10 Transitioning the height value

```
.dropdown_drawer {
  position: absolute;
  background-color: var(--background-color);
  width: 10em;
```

```

height: 0;
overflow: hidden;
transition: height 0.2s ease-out;
}
.dropdown.is-open .dropdown_drawer {
  height: auto;
}

```

The closed drawer has no height and overflow is hidden.

Transitions the height

The 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. The dropdown will still open and close, but it won't animate.

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 set the element's height to this value when the dropdown opens. Edit the script on your page to match the following listing.

Listing 15.11 Setting the height explicitly so the transition works

```

<script type="module">
  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 () {
    dropdown.classList.toggle("is-open");
    if (dropdown.classList.contains("is-open")) {
      drawer.style.setProperty("height", height + "px");
    } else {
      drawer.style.setProperty("height", "0");
    }
  });
</script>

```

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`, and 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.

15.5 Transitioning custom properties

In earlier examples, you've transitioned properties that make use of custom properties. For example, in listing 15.7, you transitioned `background-color: var(--background-color)` to `background-color: var(--highlight-color)` on a dropdown toggle button. However, in some cases, you may find you wish to transition the custom property itself, rather than the property where it's used. For instance, say you wanted to transition `--background-color` from red to blue. By default, this sort of transition does not work.

For the browser to accomplish this, it will need a little more information. Specifically, you'll need to declare what the data type of the property is. This way, the browser knows how to properly interpolate between two colors, or lengths, or some other specific data type. This is done with the `@property` at-rule.

WARNING As of mid-2024, Firefox does not yet support `@property`, but it is expected to add support soon. Consult https://caniuse.com/mdn-css_at-rules_property for the latest support statistics before relying on this feature for essential parts of your page.

A `@property` rule must describe three aspects of your custom property: its data type (or syntax), whether or not the property should inherit from parent elements by default, and its initial value. An example `@property` rule looks like the following:

```
@property --hue {  
  syntax: "<angle>";  
  inherits: false;  
  initial-value: 0deg;  
}
```

This example defines a `--hue` custom property that must be an angle (such as `15deg`). It does not inherit as normal, and it has an initial value of `0deg`. With this definition provided, the browser will know that any other data type is invalid. For instance, if given a percentage like `--hue: 15%`, it will ignore the declaration and revert to the initial value of 0 degrees. It will also be equal to the initial value if you don't declare a value.

Listing 15.12 shows an example of a defined custom property, which is used to transition the background color of a button on hover. By changing the hue angle of the OKLCH color, the button will transition from purple to blue on hover. Without the `@property` rule, the transition effect will not work.

Listing 15.12 Transitioning a custom property

```

@property --hue {
  syntax: "<angle>";
  inherits: false;
  initial-value: 0deg;
}

button {
  padding: 0.5em 1em;
  border: none;
  color: white;
  --hue: 314deg;
  background-color: oklch(39% 0.1 var(--hue));
  transition: --hue 0.5s linear;
}

button:hover {
  --hue: 250deg;
}

```

Specifies the `--hue` property definition

Uses the property

Causes the property to transition

This example defines an `<angle>` data type. The syntax of a defined property may be any of the following data types: `<length>`, `<number>`, `<percentage>`, `<length-percentage>` (for values that may be either a length or percentage), `<color>`, `<image>`, `<url>` (defined using the `url()` function), `<integer>`, `<angle>`, `<time>`, `<resolution>` (such as `300dpi`), `<transform-function>` (such as `scale()` or `rotate()`; see chapter 16), or `<custom-ident>` (a user-defined string). You can also specify keyword values, such as `auto`. Note that the given syntax must be enclosed in quotes.

Providing definitions for your custom properties isn't only useful for transitions and animations but also as means of implementing *type safety*. This means you can ensure the browser ignores any invalid custom property values if you accidentally author one in a way that doesn't match the syntax definition.

You can also define more complex syntaxes. However, in some cases, the browser will not be able to interpolate more complex values for a transition or animation when you do.

Use a `|` symbol as an “or” to indicate that the value can be one of two or more values. For example, `syntax: "true | false"` defines a Boolean property, or `syntax: "<color> | <hue> | auto"` accepts either a color, hue, or the keyword `auto`.

Use the symbol `+` to indicate a space-separated list of values. For example, `syntax: "<integer>+"` accepts properties with one or more integer values. Similarly, the symbol `#` defines a comma-separated list.

Finally, you can also define a universal syntax with an asterisk. Thus `syntax: "*"` will allow the property to be set to any value. When you do this, defining an initial value is not required.

You aren't done with transitions just yet. They'll be useful in conjunction with transforms in the next chapter.

Summary

- A transition causes one or more properties of an element to interpolate and animate smoothly between values when the properties change dynamically.
- A cubic Bézier curve defines the timing function used when transitioning property values. This can be linear, accelerating, or decelerating.
- Accelerating transitions are best for attracting the user's attention.
- Decelerating transitions are best for indicating to the user that their action has taken effect.
- Some properties have discrete values that cannot be changed smoothly with a transition.
- JavaScript is sometimes necessary to coordinate transitions using class name changes when CSS alone cannot do what you need.
- The `@property` at-rule defines specific characteristics of a custom property. This enables transitions of the property and provides type safety for your code.

16

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 and performance considerations
- 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 will give you a grasp of how they work in isolation before you 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 you'll use 3D transforms in conjunction with animation.

16.1 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 16.1):

- *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

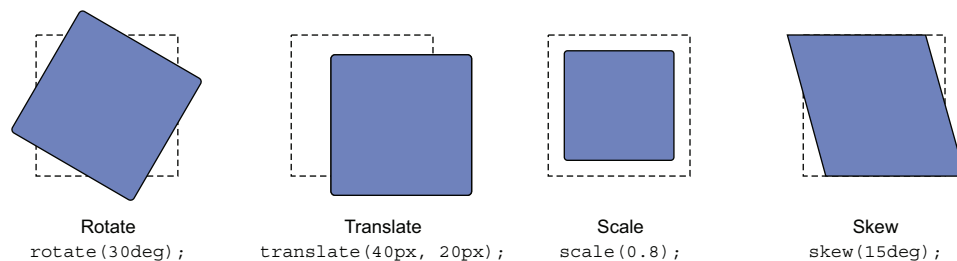


Figure 16.1 The four basic types of transform. The dashed lines represent each element's initial position and size.

Each transform is applied using the corresponding function as a value of the `transform` property. You'll create a simple example to try these out in your browser. This will be a card with an image and text (figure 16.2), which you can apply transforms to. Create a new page and stylesheet and link them. Add the HTML shown in the following listing to the page.

Listing 16.1 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 Austr White is our most prolific producer.</p>
</div>
```



Figure 16.2 A basic card with a rotate transform applied

Next, in the stylesheet, add the CSS in the following listing. This includes a few page styles, colors, and a card with a rotation transform applied.

Listing 16.2 Styling a card and applying a transform

```
body {
  background-color: hsl(210, 80%, 20%);
  font-family: Helvetica, Arial, sans-serif;
}

img {
  max-inline-size: 100%;
}

.card {
  max-inline-size: 300px;
  padding: 0.5em;
  margin-inline: auto;
  background-color: white;
  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 percentage or a number between 0 and 1. Values less than 1.0 (or 100%) shrink the element; values greater than 1.0 or (100%) 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 16.3).

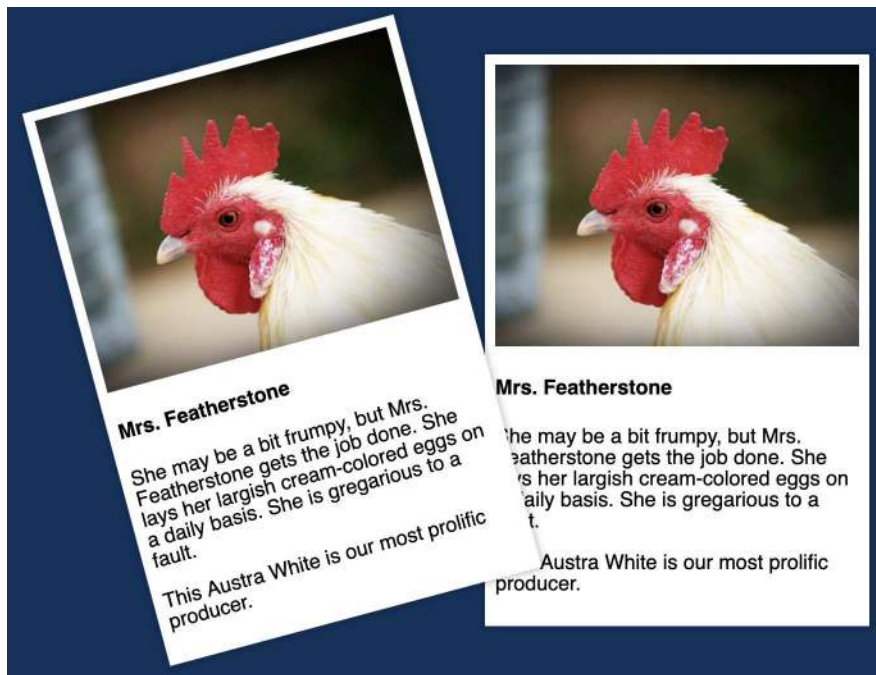


Figure 16.3 Transforming one element doesn't cause other elements to move, so they might overlap.

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 `<span>` 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).

16.1.1 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 16.4 shows some elements transformed around different points of origin.

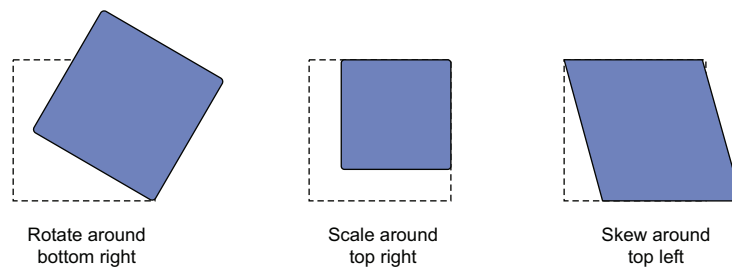


Figure 16.4 Rotate, scale, and skew made with the transform origin at various corners of the element

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%;
```

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.

16.1.2 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: translate(50px, 0) rotate(15deg)`, the element is first rotated 15 degrees clockwise, then translated 50 px to the right. Edit your stylesheet with the changes shown in the following listing.

Listing 16.3 Applying multiple transforms

```
.card {  
  max-inline-size: 300px;  
  padding: 0.5em;  
  margin-inline: auto;  
  background-color: white;  
  transform: translate(50px, 0) rotate(15deg);  
}
```



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.

Putting transforms in the “wrong” order can make them tricky to work with. I generally find it's easier to do `translate()` manipulations last chronologically (first in source order for `transform`), as in this example. This way, I can work with the final rotated element in normal left/right, up/down coordinates for the translation.

To understand this, reverse the order to `transform: rotate(15deg) translate(50px, 0)`. Notice that changing the values for `translate()` in your DevTools 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. Similarly, you will usually want to apply `scale()` transforms before `translate()` as well.

16.1.3 Individual transform properties

There are standalone CSS properties for the three most common transforms: `translate`, `rotate`, and `scale`. For a simple transform, these are often more succinct and easier to work with. There are no individual properties for skew transforms or some of the more advanced 3D transforms I'll cover later in this chapter.

You can edit your stylesheet to match next listing. This example uses the individual `translate` and `rotate` properties to achieve the exact same result as the previous listing.

Listing 16.4 Applying individual transform properties

```
.card {  
  max-inline-size: 300px;  
  padding: 0.5em;  
  margin-inline: auto;  
  background-color: white;  
  translate: 50px 0;  
  rotate: 15deg;  
}
```

When using multiple individual transform properties, `scale` is applied first, then `rotate`, then `translate`; this is generally the order that is most intuitive to work with. All three are applied after any additional transform specified using the `transform` property.

I often favor the use of these three properties either when I'm doing a simple transform or when I want to transition/animate one of them individually. I'll show you some examples of this as you continue building your page. On the other hand, I use the `transform` property when I want to change the order the transforms apply or when I want to transition/animate them all together.

16.2 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.

You'll build a page that makes use of this concept. A screenshot of the page you'll make is shown in figure 16.5. You'll be adding a lot of motion to this page.

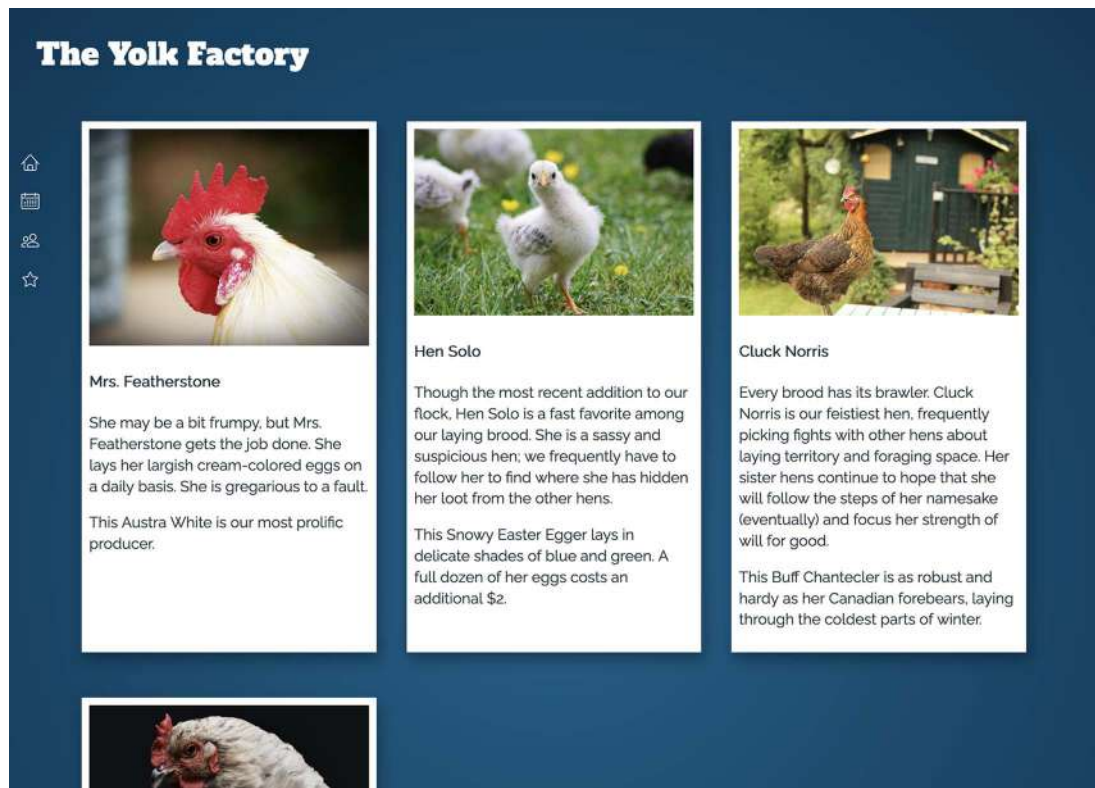


Figure 16.5 The menu icons on the left will feature several transforms and transitions.

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 of transforms. You'll 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 markup shown in the following listing. This 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.

Listing 16.5 Page markup for transforms in motion

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>The Yolk Factory</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Alfa+Slab+One
      ➤ &family=Raleway&display=fallback"
      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="/">
            
            <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>
      </ul>
    </nav>
  </body>
</html>
```

← Adds Alfa Slab One and Raleway fonts to the page

Nav links each contain an image and a label.


```

<li>
  <a href="/about">
    
    <span class="nav-links__label">About</span>
  </a>
</li>
</ul>
</nav>
</body>
</html>

```

Nav links each contain an image and a label.

The `nav` element contains the largest part of this markup. It includes an unordered list (`<ul>`) of links. Each link consists of an icon image and a text label. Notice that the icon images here are in SVG format. This will become important later. You'll add more content to the page when you're ready to style it in the next chapter.

DEFINITION SVG—short for Scalable Vector Graphics—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 without a loss of resolution.

Next, you'll add some initial styles, including a background gradient and padding around the main heading. You'll also apply the web fonts to the page. Add listing 16.6 to your stylesheet. These are the global styles and page header; you'll work on laying out the menu afterward.

Listing 16.6 Global styles and heading

```

@layer reset, theme, global, modules;

@layer reset {
  *,
  *::before,
  *::after {
    box-sizing: border-box;
  }

  body {
    margin: unset;
  }

  img {
    max-inline-size: 100%;
  }

  @media (prefers-reduced-motion: reduce) {
    *,
    *::before,
    *::after {
      animation-duration: 0.01ms !important;
      animation-iteration-count: 1 !important;
      transition-duration: 0.01ms !important;
      scroll-behavior: auto !important;
    }
  }
}

```

```

    }
  }

  @layer theme {
    :root {
      --bg-color-1: oklch(47% 0.1 238deg);
      --bg-color-2: oklch(32% 0.08 238deg);
      --font-color: white;
      --accent-yellow: oklch(87% 0.13 83deg);
    }
  }

  @layer global {
    body {
      min-block-size: 100dvh;
      background-color: var(--bg-color-1);
      background-image: radial-gradient(
        var(--bg-color-1),
        var(--bg-color-2)
      );
      color: var(--font-color);
      font-family: Raleway, Helvetica, Arial, sans-serif;
      line-height: 1.4;
    }

    h1,
    h2,
    h3 {
      font-family: "Alfa Slab One", serif;
      font-weight: 400;
    }
  }

  @layer modules {
    .page-header {
      margin-block: unset;
      padding: 1rem;
    }

    @media (min-width: 480px) {
      .page-header {
        padding: 2rem 2rem 3rem;
      }
    }
  }

```

Ensures that the body fills the viewport so the gradient fills the screen

Deep blue background gradient

Smaller padding for the header on mobile viewports

Larger padding for the header on larger screens

This example uses several 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 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.

You'll style the menu in several stages. First, you'll get the menu laid out, and then provide some responsive behavior. You'll do this with a mobile-first approach (chapter 7), so it's best to start with the small viewport. The heading and menu should look like figure 16.6.



Figure 16.6 Mobile design for the nav menu

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 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 the modules layer of your stylesheet.

Listing 16.7 Mobile styles for the nav menu links

```
@layer modules {
  .nav-links {
    display: flex;
    justify-content: space-between;
    gap: 0.8em;
    margin-block: 0 1rem;
    padding-inline: 1rem;
    list-style: none;
  }
  .nav-links > li > a {
    display: block;
    padding-block: 0.8em;
    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: var(--accent-yellow);
  }
}
```

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 will look like figure 16.7.

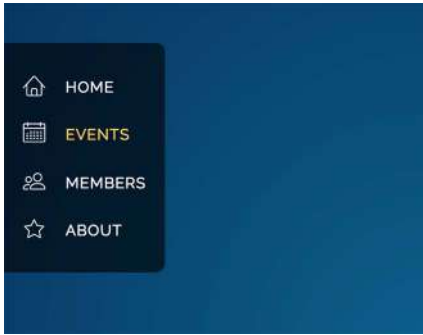


Figure 16.7 The nav menu docked on the left side of the screen for large viewports

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 a dark background. You'll put this into place now.

Update your stylesheet with the changes marked in bold in listing 16.8. Make sure the second media query and its contents are placed after the existing `nav-links` styles so they can override the mobile styles where necessary.

Listing 16.8 Positioning the menu for large viewports

```
@layer modules {
  @media (min-width: 480px) {
    .main-nav {
      position: fixed;
      top: 8.25rem;
      left: 0;
      z-index: 10;
      background-color: transparent;
      transition: background-color 0.5s linear;
      border-top-right-radius: 0.5em;
      border-bottom-right-radius: 0.5em;
    }
    .main-nav:hover,
    .main-nav:focus-within {
      background-color: rgb(0 0 0 / 0.6);
    }
  }

  .nav-links {
    display: flex;
    justify-content: space-between;
    gap: 0.8em;
    margin-block: 0 1rem;
    padding-inline: 1rem;
    list-style: none;
  }

  .nav-links > li > a {
    display: block;
    padding-block: 0.8em;
    color: white;
  }
}
```

← 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, semitransparent background on hover

```

    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,
  .nav-links > li > a:focus {
    color: var(--accent-yellow);
  }
}

@media (min-width: 480px) {
  .nav-links {
    display: block;
    padding: 1em;
    margin-block-end: 0;
  }
  .nav-links__label {
    margin-left: 1em;
  }
}
}

```

← 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 you can 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.

We'll take on each of these in turn.

16.2.1 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 `<ul>`, is a 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 to achieve some of these effects.

First, you'll 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 16.8, the Events menu item is moused over, and its calendar icon is scaled up slightly.

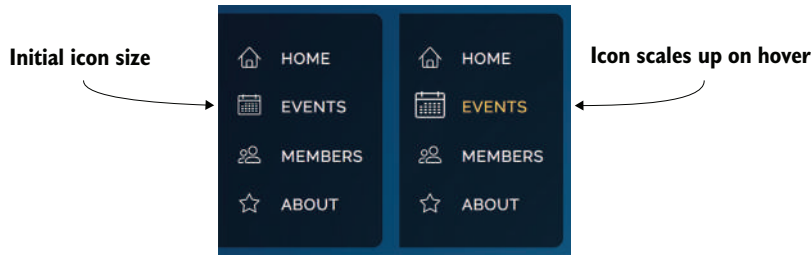


Figure 16.8 Hovering over a link causes its icon to scale up slightly.

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 match the following listing. This adds the effect when the element is either hovered (via mouse) or focused (via keyboard).

Listing 16.9 Scaling up the icon when its link is hovered or focused

```
@media (min-width: 30em) {
  .nav-links {
    display: block;
    padding: 1em;
    margin-block-end: 0;
  }
  .nav-links__label {
    margin-left: 1em;
  }

  .nav-links__icon {
    transition: scale 0.2s ease-out;
  }

  .nav-links a:hover > .nav-links__icon,
  .nav-links a:focus > .nav-links__icon {
    scale: 1.3;
  }
}
```

Transitions the scale 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. Early on in the web, 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, an icon is rendered by showing a single character. Some services like Font Awesome provide hundreds of general-use icons to make this easy, but there are some accessibility concerns when using this approach.

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 `<img>` 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 12.165l-8.829-9.758c-0.274-0.303-
0.644-0.47-1.042-0.47-0 0 0 0 0 0-0.397 0-0.767 0.167-1.042 0.47l-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 0.371-0.165l2.129-2.353v8.018c0 0.827 0.673 1.5
1.5 1.5h11c0.827 0 1.5-0.673 1.5-1.5v-8.018l2.129 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-9.123l5.7-6.3c0.082-0.091 0.189-0.141 0.3-
0.141s0.218 0.050 0.3 0.141l5.7 6.3v9.123z"></path>
      <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, you can turn your attention to the labels beside them.

16.2.2 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 with their keyboard, you can fade in the labels. This way, when the user moves their mouse near the icons, the entire menu appears, using several effects at once: the background and the labels will fade in, with the labels starting a little to the left of their final position (figure 16.9).

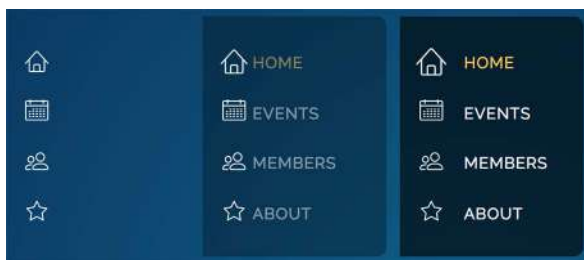


Figure 16.9 Upon hover, the menu fades in, while the labels fade in and slide from the left.

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 bold in the following listing.

Listing 16.10 Transitioning in the nav item labels

```
@media (min-width: 480px) {
  .nav-links {
    display: block;
    padding: 1em;
    margin-block-end: 0;
  }
  .nav-links__label {
    display: inline-block;
    margin-left: 1em;
    padding-right: 1em;
    opacity: 0;
    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:focus-within .nav-links__label {
    opacity: 1;
    translate: 0;
  }
}
```

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


```

.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);
}
}

```

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` and `:focus-within` pseudo-classes are on the top-level `nav-links` element. This way, the labels all appear as soon as the menu is moused over or the user focuses on one of the menu items 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 16.10.

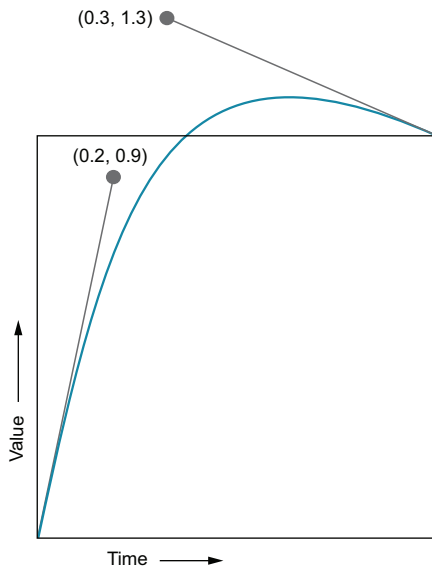


Figure 16.10 A Bézier curve with a bounce at the end

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.

16.2.3 *Staggering the transitions*

The menu looks pretty good at this point. I'll show you one last tweak to really make it feel polished. You'll use the `transition-delay` property to set a slightly different delay for each menu item. This will stagger the animations so they fly in a rolling "wave" rather than all at once (figure 16.11).

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, as indicated in the following listing.

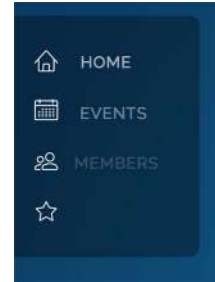


Figure 16.11 The menu items will fly in consecutively from first to last.

Listing 16.11 Adding a staggered transition delay to the menu items

```
.nav-links:hover .nav-links__label,
.nav-links:focus-within .nav-links__label {
  opacity: 1;
  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

Repeats 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 are 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 there's a chance the menu could exceed this count at some point, and 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 are a couple more things to know about transforms.

16.3 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 15) 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.

16.3.1 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 (see figure 16.12).



Figure 16.12 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, 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 repainted. But, because the background color has no effect on the position or sizing of any elements on the page, layout doesn't need to be recalculated to account for this change. Therefore, 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 graphics processing unit (GPU) 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 will be displayed on screen. These are drawn in a certain order so that the correct layers appear in front of other layers in cases where they overlap.

Certain properties—specifically, `opacity` and the transform properties—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 repainting.

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 on screen. 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 `opacity` or the transform properties on an element, modern browsers, 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.

However, don't apply this blindly to the page until you're seeing performance problems as it tends 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 by Sara Soueidan at <https://mng.bz/JZqV>.

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; 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 `opacity` and the transform properties. 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/>.

16.4 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 three declarations each produce the same result:

```
transform: translate(15px, 50px);
transform: translateX(15px) translateY(50px);
translate: 15px 50px;
```

You can also translate on the z dimension using `translateZ()` or by providing a third value to the `translate` property. This z value 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 16.13 for an illustration of these functions.

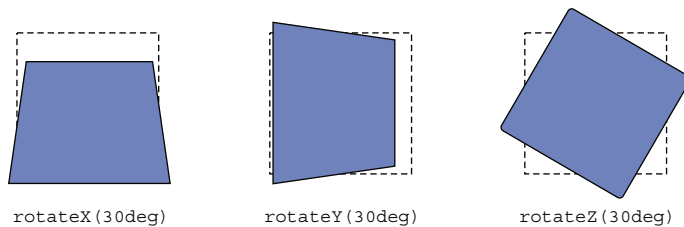


Figure 16.13 Rotation on each of the three axes with a 300 px perspective applied (a dashed line represents the initial element position)

When using the `rotate` property to transform, you can specify the x-, y-, or z-axis name along with the angle, such as `rotate: z 20deg` or `rotate: x 30deg`.

16.4.1 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 16.14.

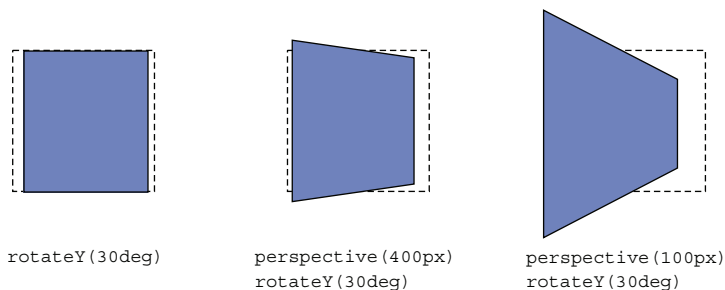


Figure 16.14 The same rotation applied at different perspectives

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. I'll have you 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 16.15). Each element is rotated the same and has the same `perspective()` transform applied; thus, all four elements appear the same.



Figure 16.15 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 in the following listing.

Listing 16.12 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 the following listing.

Listing 16.13 Applying 3D transforms to the boxes

```
.row {
  display: flex;
  gap: 4em;
```

```

    justify-content: center;
  }

  .box {
    box-sizing: border-box;
    width: 150px;
    padding: 60px 0;
    text-align: center;
    background-color: oklch(60% 0.12 158deg);
    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 the four elements, each from the same relative position.

WARNING The `perspective()` transform must be applied after other transforms you want it to affect. This means it will not work well in conjunction with the individual transform properties, such as `rotate`, because these apply after the `transform` property. When using `perspective()`, you should list it first, followed by other transform functions in the same `transform` declaration.

Sometimes you’ll want multiple elements to share a common perspective, as if they all exist within the same 3D space. Figure 16.16 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 16.16 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 in the following listing.

Listing 16.14 Establishing a common perspective

```

.row {
  display: flex;
  gap: 4em;
  justify-content: center;
  perspective: 200px;
}

```

Adds the perspective to the container


```
.box {
  box-sizing: border-box;
  width: 150px;
  padding: 60px 0;
  text-align: center;
  background-color: oklch(60% 0.12 158deg);
  transform: rotateX(30deg);
}
```

Does 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 a distance.

16.4.2 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 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 16.17 shows the previous example but with the camera shifted to the bottom left.



Figure 16.17 Moving the perspective origin increases the perspective distortion of elements toward the far edges.

To see this in your sample page, add the declaration indicated in the following listing.

Listing 16.15 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).

The effect in this example is very dramatic, due to the relatively close perspective of 200 px. You can increase this to a larger perspective to lessen the effect.

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 16.18 has been transformed with `rotateY(180deg)`. It looks like a mirror image of the original.



Figure 16.18 Rotating an element to see its back side

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 declaration applied, the element will only be visible if it's facing toward the viewer and will be 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://3dtransforms.desandro.com/card-flip>.

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 and 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 image of a 3D scene. The perspective will be all wrong, and the illusion of depth in the scene is shattered. See figure 16.19 for an example illustrating this.

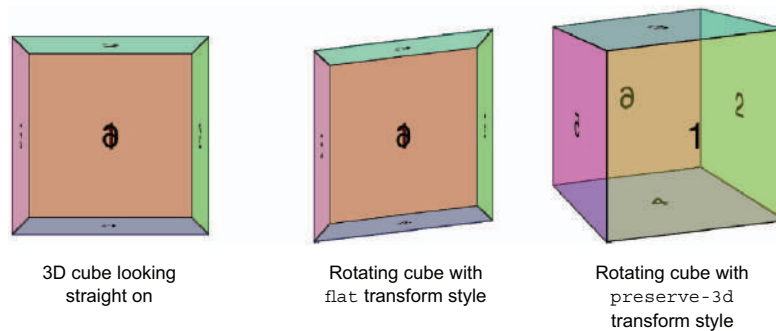


Figure 16.19 If you perform a 3D transform on the parent of other 3D-transformed elements, you'll probably want to apply `preserve-3d` (right).

The 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 the entire cube together (that is, the parent element). To correct this, you should apply `transform-style: preserve-3d` to the parent element (right).

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

- Transforms can scale, rotate, translate, and skew elements in two and three dimensions.
- Multiple transforms can be applied using the `transform` property. These are applied from right to left.

- The individual `scale`, `rotate`, and `translate` properties perform simpler individual transformations.
- Transforms and the `opacity` property are the most performant properties to change in transitions and animations. They do not require the browser to recalculate layout or repaint the contents of the page.
- A `perspective()` transform applied after other transforms make those transforms appear three dimensional. You can also apply the `transform` property to a container element to make the children appear to share the same three-dimensional space.
- A custom timing function curve can be used to add a bounce effect to transitions.

17

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 17.1). These interpolated frames will have values in between those of the keyframes, so the value changes smoothly from one keyframe to the next.



Figure 17.1 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 are not only something you add to a page to jazz it up; they can also convey meaningful feedback to the user.

17.1 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.

You'll build a basic animation to get familiar with the syntax. This animation will have three keyframes, shown in figure 17.2. 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.

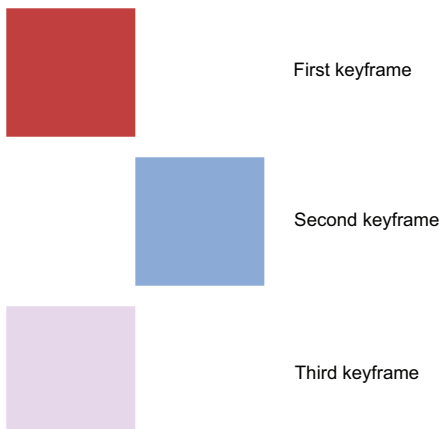


Figure 17.2 Three keyframes, animating the element's color and position

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 17.1 Defining a keyframe at-rule

```
@keyframes over-and-back {
  0% {
    background-color: hsl(0, 50%, 50%);
    transform: translate(0);
  }
  50% {
    transform: translate(50px);
  }
  100% {
    background-color: hsl(270, 50%, 90%);
    transform: translate(0);
  }
}
```

← Names the animation

First keyframe declarations

← The 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 and 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.

Add this to a page to see it working. Create a new HTML document and add the markup shown in the following listing.

Listing 17.2 Page with a single box element for animation

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
    <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 17.3 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 and 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 of 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.

TIP Instead of 0% and 100% for the beginning and ending keyframe, you can also use the keywords `from` and `to`. I will show an example of this later in the chapter (listing 17.12).

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.

17.2 Animating 3D transforms

Next, you'll add an animation to the page you began in the previous chapter. After listing 17.11, 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.

17.2.1 Building the layout without animations

In this demo, you'll add some cards in the main area of the page (figure 17.3). Then, you'll add an animation to make them fly in using 3D transforms.

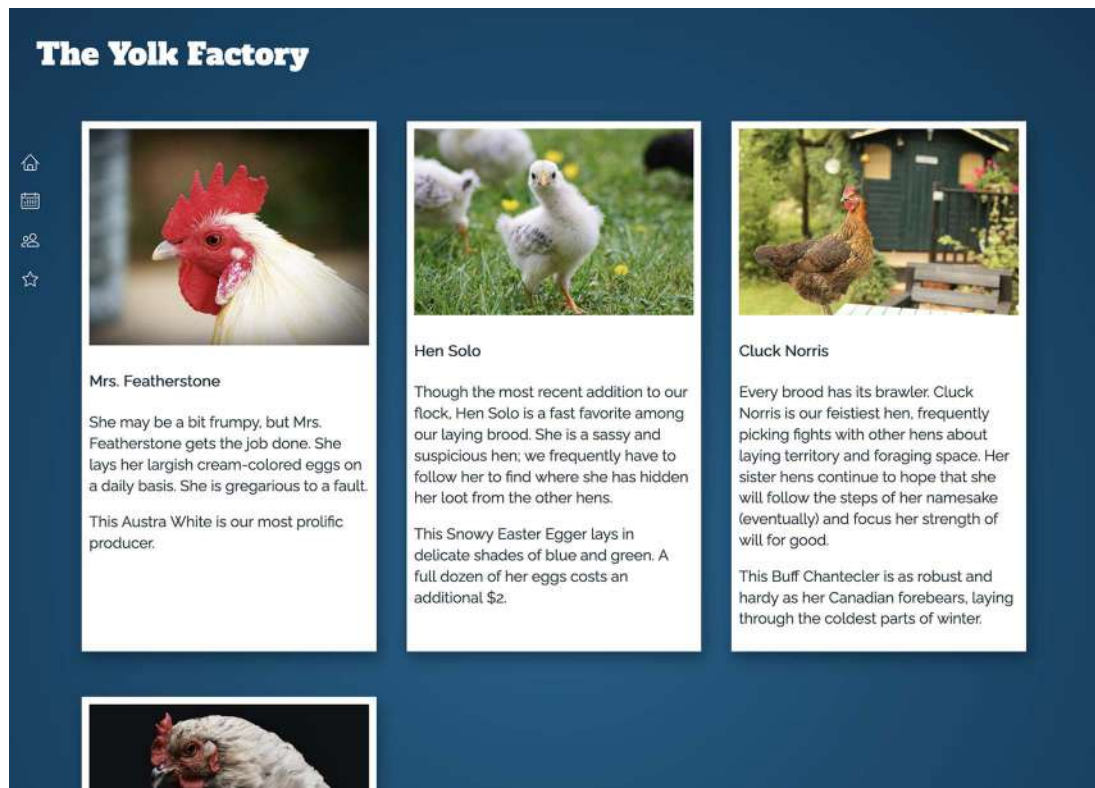


Figure 17.3 Additional cards for the main region of the page

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 17.3 more closely.)

Listing 17.4 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'll do the mobile-first layout, then layer on grid-based styles for larger viewports.

The mobile layout is shown in figure 17.4. On small screens, the cards will fill the width of the screen, with a little margin added to the left and right sides.

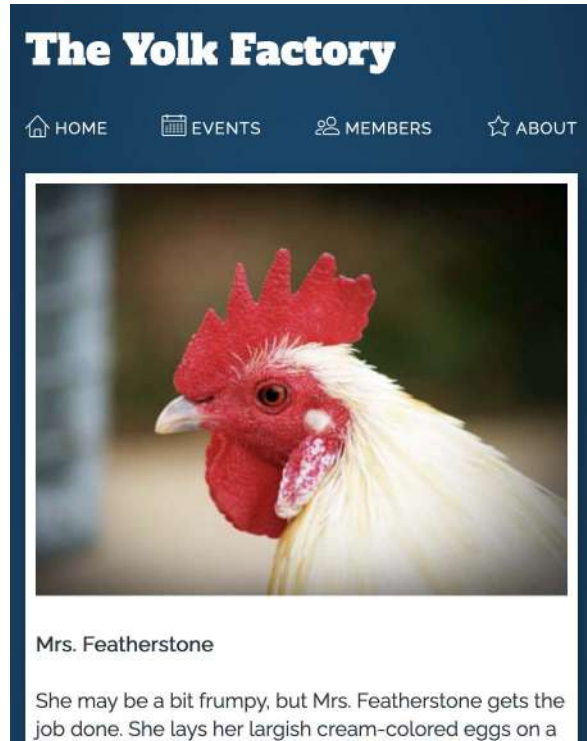


Figure 17.4 In the mobile layout, cards will fill the screen width, stacked beneath the menu.

Add mobile styles in the following listing to your stylesheet inside the modules layer.

Listing 17.5 Mobile styles for the cards

```
@layer modules {
  .flyin-grid {
    margin-inline: 1rem;
  }

  .card {
    margin-block-end: 1em;
    padding: 0.5em;
    background-color: white;
    color: oklch(32% 0.02 248deg);
    box-shadow: 3px 8px 15px rgb(0 0 0 / 0.3);
  }
}
```

← Adds a small left and right margin around the container

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 add in the grid layout, applied only to larger breakpoints. This will get you close to the final design (figure 17.3). Update your stylesheet to match the following listing.

Listing 17.6 Applying a grid layout

```
.flyin-grid {
  margin-inline: 1rem;
}

@media (min-width: 30em) {
  .flyin-grid {
    margin-inline: 5rem;
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
    gap: 2em;
  }
}
```

Defines column widths ←

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. Small viewports will still display the simpler mobile layout.

17.2.2 Adding animation to the layout

The page has the design and layout in place, so you can now work in some animation. When the page loads, you'll fly in the cards, as illustrated in figure 17.5. 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. The figure 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 17.7. 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.



Figure 17.5 Use 3D transforms to fly in the cards from off in the distance.

Listing 17.7 Adding the fly-in animation

```
.flyin-grid {
  margin-inline: 1rem;
  perspective: 500px;
}
```



Sets a shared perspective on the container

```

.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);
  }
}

```

← Applies animation to each item

← Starts in the distance, rotated

← Much closer, but still mostly rotated

← Finishes in normal position

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.

17.3 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 delaying each item's animation for slightly different amounts of time, you can make them fly in one after the other, as shown in figure 17.6.

The next code applies these delays to the four grid items. But it won't quite work like you want it to. Add the following listing to your stylesheet; then we'll take a look at the problem and how to address it.

Listing 17.8 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;
}

```

← Staggers the start of each item's animation a little longer than the previous item

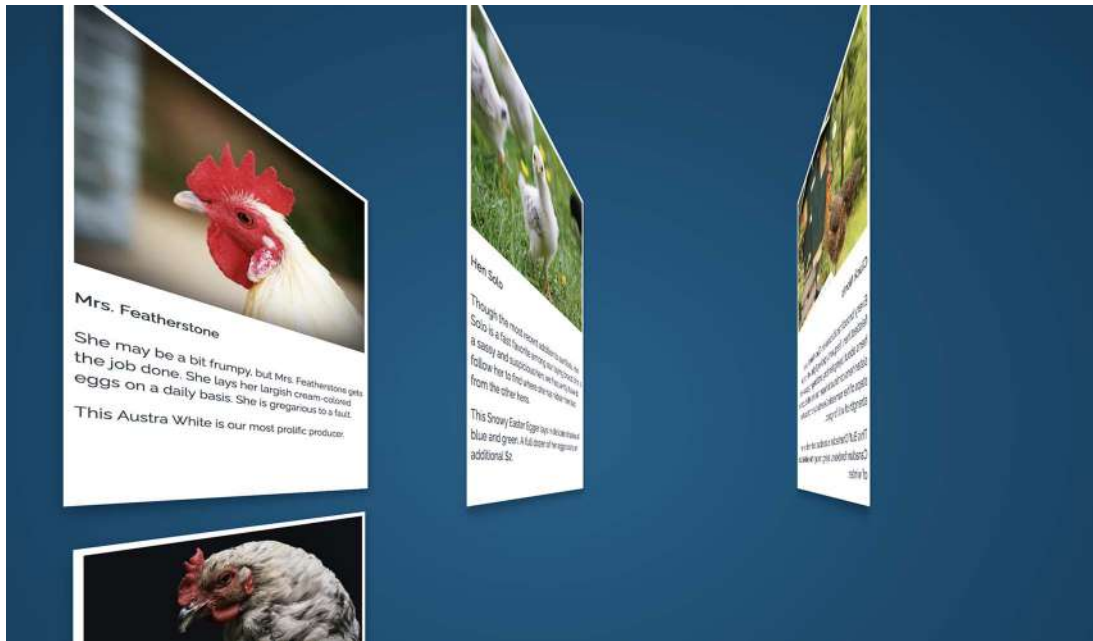


Figure 17.6 Elements flying in with staggered animation

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 17.7). This is a bit

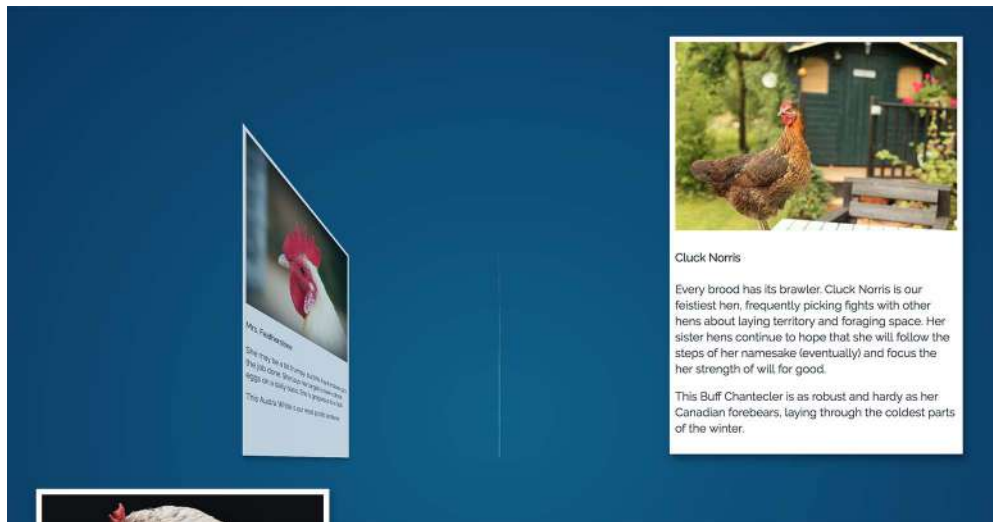


Figure 17.7 The later items appear in their final position before their animations play.

jarring and doesn't look like the effect you're going for. Instead, you want the elements to all be invisible initially and appear only during the course of their respective animations.

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 17.8).

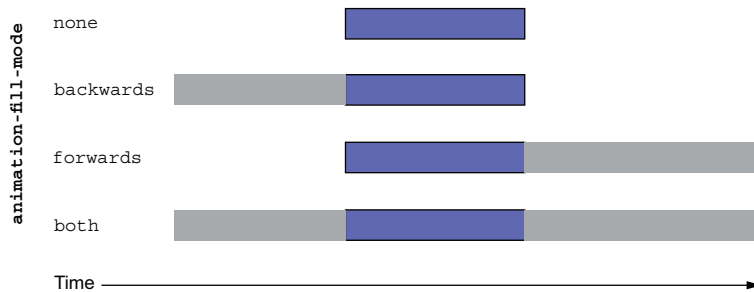


Figure 17.8 Use `animation-fill-mode` to apply animation styles before or after the animation plays.

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 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 the following listing.

Listing 17.9 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.

Take a moment to enjoy the page you've built. You've brought together techniques from the entire book into this page: flexbox and grid layout, positioning, responsive design, custom fonts, color selection, subtle shadows and gradients, and a combination of transitions and animations. The code itself is organized meaningfully into layers and small, understandable modules.

17.4 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.

17.4.1 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, you'll build a small form with a Save 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 17.9. 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. It features a label "Tell us about your first trip to the zoo:" followed by a large, empty text input area. Below the text area is a blue button with the word "Save" in white text.

Figure 17.9 A simple form with a Save button

Create a new page and a blank stylesheet for this form. Add the HTML in the following listing.

Listing 17.10 Form with Save button

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <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 the following listing to your stylesheet.

Listing 17.11 Laying out and styling the form

```

body {
  font-family: Helvetica, Arial, sans-serif;
}

form {
  max-width: 500px;
}

label,
textarea {
  display: block;
  margin-bottom: 1em;
}

textarea {
  width: 100%;
  font-size: inherit;
}

button {
  padding: 0.6em 1em;
  border: 0;
  font: inherit;
  color: white;
  background-color: oklch(54% 0.14 260deg);
  transition: background-color 0.3s linear;
}

button:hover {
  background-color: oklch(46% 0.11 260deg);
}

```

Limits the width of the form

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 17.10). When the user submits the form, you’ll use JavaScript to add the `is-loading` class to the button, enacting the animation.

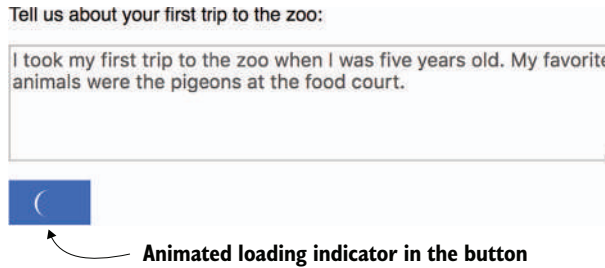


Figure 17.10 When the user clicks Save, a loading spinner appears in the button.

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 and then applying an animation to make it spin. It’ll also need a little JavaScript to add the `is-loading` class 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 the following listing to your stylesheet.

Listing 17.12 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 {
  from {
    transform: rotate(0deg);
  }
}
```

← Hides the button text

Positions pseudo-element in the center of the button

← Loops spin animation repeatedly

```

    to {
      transform: rotate(360deg);
    }
  }
}

```

← 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 position the pseudo-element vertically and horizontally in the center of 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 degrees to 360 degrees. 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 listing 17.13 to your page. This provides JavaScript functionality to add the `is-loading` class when the button is clicked. Place this at the beginning of the `<body>` tag.

Listing 17.13 Adding an `is-loading` class to the button when clicked

```

<script type="module">
  var input = document.getElementById('trip');
  var button = document.getElementById('submit-button');

  button.addEventListener('click', (event) => {
    event.preventDefault();
    button.classList.add('is-loading');
    button.disabled = true;
    input.disabled = true;

    /* submit form data here */
  });
</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 reenable 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.

17.4.2 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 17.11).

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 the following listing to your stylesheet.



Figure 17.11
Moving the button
left and right rapidly
to shake it

Listing 17.14 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. One way to do this is to use a `keyup` event listener and a timeout function: when the user types a character into the text area, you'll set a 1-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. The script tag in the next listing defines this logic. Update the `script` tag in your page to include the changes indicated in this listing.

Listing 17.15 Adding the `shake` class after a 1-second delay

```
<script type="module">
  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 1 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://mng.bz/wxaP>.

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 predefined keyframes you can use, visit <https://animista.net/>. This has a large library of animations to choose from, including bouncing in, rolling out, and wobbling like jelly.

17.5 Scroll-based timelines

Animations typically play from beginning to end in a linear fashion, but they can also be tied to scrolling. As a user scrolls down the page, the animation will play in time with their scrolling and will play in reverse if they scroll back up. This effect has typically required JavaScript to achieve, but a new CSS property adds the ability to do this without any JavaScript.

You can do this by setting `animation-timeline: scroll()` on the animated element. However, as of mid-2024, this is not yet supported in Firefox or Safari. It's not yet suitable for production use except as a purely decorative enhancement on top of an already functional page. See <https://mng.bz/qOvr> for the latest browser support information.

I'll show you a simple example that you can test out in Chrome or Edge. In a new stylesheet, add the code shown in listing 17.16. This positions a blue indicator square in the upper-right corner of the page and defines a basic spin animation.

Listing 17.16 Using a scroll-driven animation timeline

```

body {
  margin: unset;
  height: 200lvh;
}

.indicator {
  position: fixed;
  top: 15px;
  right: 15px;
  height: 50px;
  width: 50px;
  background-color: oklch(74% 0.08 260deg);
  animation-name: spin;
  animation-timeline: scroll();
}

@keyframes spin {
  from {
    rotate: 0deg;
  }

  to {
    rotate: 360deg;
  }
}

```

← Forces page scrolling

← Ties the animation to the scroll position

This indicator square will now spin as the user scrolls, with its position in the animation tied to where the user is scrolled; if the user scrolls 25% of the way down the page and stops, the animation will be stopped 25% of the way from the first frame toward the last.

This is best seen interactively. Add this stylesheet to a page with a `<div class="indicator">` element, like the one shown in the following listing, so you can see it in your browser.

Listing 17.17 Markup for scroll animation demo

```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link href="styles.css" rel="stylesheet">
  </head>
  <body>
    <div class="indicator"></div>
  </body>
</html>

```

← Spinning indicator element

By default, the `scroll()` function will bind the element's animation to the nearest ancestor element in the DOM tree that has a scrollbar. You can provide arguments to this function to specify different behavior:

- `scroll(nearest)`—Default behavior; animates relative to the scroll position of the nearest ancestor
- `scroll(root)`—Animates relative to the scroll position of the root element of the document
- `scroll(self)`—Animates relative to the scroll position of the current element

Furthermore, you can specify which scroll direction the animation should be tied to. This can be the `block` direction (the default behavior), `inline`, `y`, or `x`. For example, specifying `scroll(inline root)` will animate relative to any inline scrolling of the root element, and `scroll(y nearest)` will animate relative to vertical scrolling of the nearest element that has scrollbars.

The previous example is rather minimal, but with some creative design and more complex animations, you can achieve an unlimited number of effects. You can have decorative elements on the page animate in from out of frame as the user scrolls to them, or you can design a *parallax* effect, where different parts of the page move at different speeds as the user scrolls.

For several interesting examples of scroll-driven animations, see <https://mng.bz/7dqv>. Effects like this can make a page feel much more alive and interactive, even if it consists of just read-only content.

17.6 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 on 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 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 impresses you, open your browser's DevTools and try to figure out how it works. New CSS features emerge frequently, especially in the last several years. Follow developers and designers online that provide creative demos or offer interesting tutorials. Try new things, and keep learning.

Summary

- `@keyframe` animations define key points in an animation. Use the `animation-*` properties to apply a keyframe animation to an element.
- The `animation` property is shorthand for multiple animation properties, including `animation-name`, `animation-duration`, `animation-timing-function`, and `animation-iteration-count` properties.

- The `animation-fill-mode` property can apply the first frame of animation to an element before animation begins and/or apply the last frame of animation after the animation completes. Use this to make an animation begin or end seamlessly.
- A well-placed animation can communicate meaningfully to the user when they need to know something or need to interact with something.
- JavaScript can trigger animations to play at the appropriate time.
- Animations add 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. It has a specificity of 0,0,1. Examples: `p`; `h1`; `strong`.
- **.class**—Class selector. This targets elements that have the specified class name as part of their class attribute. It has a specificity of 0,1,0. Examples: `.media`; `.nav-menu`.
- **#id**—ID selector. This targets the element with the specified ID attribute. It has a specificity of 1,0,0. Example: `#sidebar`.
- *****—Universal selector. This targets all elements. It 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 `<li>` 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` targets an `<h2>` that immediately follows a `<p>`.

- *General sibling combinator* (~)—Targets all sibling elements following a specified element. Note that this doesn't target siblings that appear before the indicated element. Example: `li.active ~ li`.

Note that the two sibling combinators only target elements that follow a specified sibling; they cannot select elements that precede it. To select preceding siblings, use the `:has()` selector instead. For example, `li:has(~ li.featured)` will target list items before `<li class="featured">`, or `ul:has(> li.featured) > li` will target all list items in a list that contains an `<li class="featured">`.

A.3 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.4 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), except `:is()` and `:where()`, which have special behavior regarding their specificity, as noted later.

There are quite a lot of pseudo-class selectors available. I have grouped these into three categories: general purpose, those that select based on position among siblings, and those that pertain to forms.

These can be used on their own or as part of a compound selector. For example, `:focus` can target any element that has focus, whereas `input:focus` targets only `<input>` elements that have focus. Some pseudo-classes must be immediately followed by more information in parentheses to further specify what they should target; parentheses are included when necessary in the definitions provided.

This list of pseudo-classes is not exhaustive. See the MDN documentation at <https://mng.bz/ma5a> for a complete list.

A.4.1 General-purpose pseudo-classes

- `:active`—Targets an element that is currently being activated by the user by mouse or keyboard interaction. This usually pertains to links or buttons.
- `:any-link`—Targets elements that match either `:link` or `:visited`.
- `: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.
- `:focus`—Targets elements that have received focus, whether from a mouse click, screen tap, or Tab key navigation.

- `:focus-visible`—Targets elements that have received focus when the browser has determined the focus should be made visually evident. Typically, this means an element that has been navigated to via keyboard but not a mouse click. This is useful because you may not want a “focus ring” (usually an outline) to appear when a user clicks a button or input. But for accessibility, the focus ring should still appear for users using keyboard navigation, so it is clear where they are on the page.
- `:focus-within`—Targets elements that either match `:focus` or contain descendant elements that do.
- `:has(<selector list>)`—Sometimes called a “parent selector,” this targets an element if the selector list targets at least one element relative to that element. For example, `article:has(h2)` will target an `<article>` if it contains an `<h2>` anywhere among its descendant elements. The relative selector may begin with a combinator, so `h1:has(+ h2)` will target an `<h1>` that is immediately followed by an `<h2>`. Support for this selector is still relatively new; see <https://caniuse.com/css-has> for the latest support information.
- `:hover`—Targets elements while the mouse cursor hovers over them.
- `:is(<selector list>)`—Targets any elements matched by the selector list. Specificity is equal to the highest specificity selector in the list. Example: `:is(h1, h2, h3)`. See chapter 8 for more details.
- `:lang(<language>)`—Targets elements based on their language. Language can be set either via the `lang` HTML attribute or via a `<meta>` tag. Example: `article:lang(en-US)`.
- `:link`—Targets a link (`<a>`) that has not yet been visited by the user.
- `:modal`—Targets a `<dialog>` element that has been opened with JavaScript’s `showModal()` method. Also see the `::backdrop` pseudo-element.
- `: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.
- `: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.
- `:scope`—Targets the scope root of the current scope (see chapter 9).
- `:visited`—Targets a link (`<a>`) that has already been visited by the user.
- `:where(<selector list>)`—Behaves the same as `:is()`, but selector specificity is always 0,0,0. Example: `:where(:hover, .is-selected)`. See chapter 8 for more details.

A.4.2 Selecting based on position among sibling elements

Several pseudo-class selectors involve targeting elements based on where they reside among sibling elements. Some of these are straightforward, targeting the first or last element. Others use the formula $an+b$ to specify a specific subset of elements to target.

In this formula, a and b are integers. To know exactly how a formula works, imagine solving it for all integer values of n , beginning with zero (the keywords *even* or *odd* can also be substituted for the equation). The results of this equation indicate which elements are targeted. Figure A.1 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

Figure A.1 Examples of the $an+b$ pseudo-class formula

The following pseudo-class selectors all relate to an element's position among its sibling elements:

- `:first-child`—Targets elements that are the first element within their parent element.
- `:first-of-type`—Similar in nature to `:first-child`, except instead of considering the position among all children, it considers an element's numeric position only among other children with the same tag name.
- `:last-child`—Targets elements that are the last element within their parent element.
- `:last-of-type`—Targets the last child element of its type.
- `:nth-child( $an+b$ )`—Targets elements based upon their position among their siblings. For example, `.card:nth-child(2n)` targets every even-numbered element that also has the class `card`.
- `:nth-child( $an+b$  of <selector>)`—By adding the `of` keyword and a selector, this will target the n th element among those that match the given selector. For example, `li:nth-child(3 of .featured)` will target the third `<li>` with the `featured` class.
- `: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()`, including support for the optional `of <selector>` syntax.

- `: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`.
- `:nth-of-type(an+b)`—Targets elements of their type based on their numerical order and the specified formula, similar to `:nth-child`.
- `:only-child`—Targets elements that are the only element within their parent element (no siblings).
- `:only-of-type`—Targets elements that are the only child of their type.

A.4.3 Form field pseudo-classes

The following pseudo-class selectors all relate to forms and form fields:

- `:checked`—Targets selected checkboxes, radio buttons, or select box options.
- `:disabled`—Targets disabled elements, including inputs, selects, and buttons.
- `:enabled`—Targets enabled elements, meaning they can be activated or accept focus.
- `: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.
- `:optional`—Targets elements that do not have a required attribute set.
- `:placeholder-shown`—Targets an `<input>` or `<textarea>` that's currently displaying placeholder text.
- `:required`—Targets elements with a required attribute set.
- `:user-invalid`—Targets form fields with invalid input but only after the user has interacted with them. This has limited browser support at the time of writing.
- `:user-valid`—Targets form fields with valid input but only after the user has interacted with them. This has limited browser support at the time of writing.
- `:valid`—Targets form fields with valid values.

A.5 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 DOM. 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 some pseudo-elements for backward compatibility reasons. Pseudo-elements have the same specificity as a type selector (0,0,1):

- `::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`.

- `::backdrop`—Targets a box the size of the viewport, which is rendered immediately behind the corresponding element. The corresponding element must either be a `<dialog>` or an element made full screen via the JavaScript `requestFullscreen()` method (such as a video). Example: `dialog::backdrop`.
- `::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`.
- `::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.
- `::marker`—Selects the marker of a list item (or any element with `display: list-item`). This is typically a bullet or number, depending on list type. This allows you to style its color or size, for example. You can also use the `content` property to replace the marker with new content, such as a different unicode character.
- `::placeholder`—Allows you to style the placeholder text shown in an `<input>` or `<textarea>`.
- `::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.6 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 (-). This is 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"]`.

These use square brackets in their syntax, which can seem odd when you first encounter them. These are still normal selectors, like any listed previously. The following ruleset is an example built using an attribute selector. This selects any `<input type="password">` element and gives it a blue border:

```
input[type="password"] {  
    border: 2px solid blue;  
}
```

All of the previous attribute selectors are case sensitive. You can make any of them case insensitive by adding an `i` before the closing bracket. Example: `input[value="search" i]`.

appendix B

Preprocessors

Using a preprocessor has long been 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.

Many CSS features, such as variables and nesting, first began in preprocessors, before they existed in the language itself. In many ways, preprocessors acted as a proving ground, where developers proved the utility of these features, and the W3C eventually added some of these features to the language itself. Because many of the features now live in CSS, preprocessors are less necessary than they have been in the past. They are still very popular, however, and it is worth considering whether you want to use one in your projects.

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 the following:

```
$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 reuse 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 are several others as well. Sass is the most popular, so I'll focus mostly on that in this appendix, but Less is similar, with mostly minor syntactic differences distinguishing the two. 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

There are a number of ways to install Sass, so you have options depending on what language or packaging tools you are familiar with. If you're comfortable with JavaScript and the Node environment, you can get Sass by installing the 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 to 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/about-npm>.

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
- `npm install --save-dev sass`—Installs the Sass package and adds it to `package.json` as a development dependency

You'll then need to decide 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, you can 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 the following 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 `sass-loader` are available for common task runners like Webpack or Vite. If you want to use a plugin, look for one that integrates Sass or Less into the workflow you're most familiar with.

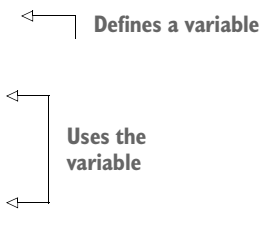
B.1.3 *Understanding important Sass features*

I've shown you one example of a Sass variable (`$brand-blue`). Add the code in the following listing to your `index.scss` file to see Sass compile it for you.

Listing B.2 A Sass variable

```
$brand-blue: #0086b3;

a:link {
  color: $brand-blue;
}
.page-heading {
  font-size: 1.6rem;
  color: $brand-blue;
}
```



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 hex values, so now the browser can understand them. Sass also produced a source map file and added a comment to the end of the stylesheet, giving a path to the *source map*.

DEFINITION A *source map* is 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.

INLINE COMPUTATION

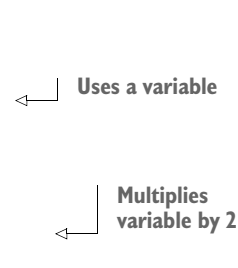
Sass also supports inline arithmetic using `+`, `-`, `*`, `/`, and `%` (for modular division). This lets you derive multiple values from one source value.

Listing B.3 Using inline computations

```
$padding-left: 3em;

.note-author {
  left-padding: $padding-left;
  font-weight: bold;
}

.note-body {
  left-padding: $padding-left * 2;
}
```



Use `npm run sass` to compile this, which produces the following 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`. This feature inspired the addition of the `calc()` function in CSS.

NESTED SELECTORS

Sass allows you to nest selectors inside other declaration blocks. You can use nesting to group related code in the same block.

Listing B.4 Nesting selectors

```
.site-nav {
  display: flex;

  > li {
    margin-top: 0;

    &.is-active {
      display: block;
    }
  }
}
```

← **Nested selector**

← **The 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.

This is very similar to nesting in CSS, but it does have a few minor differences, particularly in the way it affects selector specificity. See chapter 8 for more details on this. At this time, there is no way to use plain CSS nesting when preprocessing using Sass. Sass will always process nested styles into a flat structure.

You can also nest media queries inside a declaration block. This can be used to avoid repeating the same selector.

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 (@USE)

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.

Sass partial files are named with a leading underscore. Create a new file in your project as `sass/_button.scss`. Add the styles shown in the following listing 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 `@use` at-rule as shown in listing B.7. This must be added at the very beginning of the stylesheet; `@use` can only be used before any other style rules.

Listing B.7 Importing a partial

```
@use "button";
```

← Path to the partial file, without its leading underscore

When you run Sass, the partial file will be compiled and inserted where you indicated with the `@use` 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.

The `@use` rule has more complex features beyond this, including namespacing and the ability to import only certain parts of a partial file. See the documentation at <https://sass-lang.com/guide/#modules> for more on this.

MIXINS

A *mixin* is a small chunk of CSS that you can reuse 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.

A mixin is defined using an `@mixin` at-rule and used with an `@include` at-rule. The following listing is an example of a mixin.

Listing B.8 A Saas mixin

```
@mixin box {
  border-radius: 5px;
  box-shadow: 5px 5px 10px rgb(0 0 0 / 0.1);
}

.main-tile {
  @include box;
  color: #333;
}
```

← Defines a mixin named **box**

← Applies the mixin

The preprocessor takes the code from the mixin and copies it in place of the `@include` rule. The resulting code looks like the following:

```
.main-tile {
  border-radius: 5px;
  box-shadow: 5px 5px 10px rgba(0, 0, 0, 0.1);
  color: #333;
}
```

Notice there’s no mention of the `box` mixin 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. Listing B.9 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. Listing B.9 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 reuse 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.

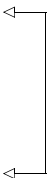
EXTEND

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 following 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-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.

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 whether mixins or extends make more sense to use on a case-by-case basis. In general, you should probably favor mixins.

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 following listing to produce the colors you need.

Listing B.11 Sass color functions

```

$green: #63a35c;

$green-dark: darken($green, 10%);
$green-light: lighten($green, 10%);

$green-vivid: saturate($green, 20%);
$green-dull: desaturate($green, 20%);

$purple: adjust-hue($green, 180deg);
$yellow: adjust-hue($green, -70deg);

```

Darkens by 10%

Lightens by 10%

Adjusts color saturation

Rotates hue around the color wheel

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 are several more color functions. See <http://jackiebalzer.com/color> for a comprehensive reference. Note that CSS color manipulation functions are generally more powerful, and they are gaining support in browsers (chapter 11), but using the Sass version may be helpful in the meantime to avoid browser support issues.

Loops

Use loops to iterate over a value, producing slight variations. In chapter 16, you used several `:nth-child()` selectors to target consecutive menu items, giving each a slightly different `transition-delay` (listing 16.11). This sort of code can be done more succinctly with a Sass loop, which uses a `@for` at-rule.

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 4

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 that I used the variable in the selector, escaping it with the `#{}` notation. The resulting CSS looks like the following:

```

.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 to use Sass. But Sass (or Less) can remove much of the busywork when working with CSS.

B.2 *PostCSS*

PostCSS (<https://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.

B.2.1 *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 14.

If your source code looks like this,

```
.example {  
    backdrop-filter: blur(5px);  
    mask-image: url(star-mask.png);  
}
```

Autoprefixer adds additional declarations, providing vendor-prefixed fallbacks to older browsers, and then outputs

```
.example {  
  -webkit-backdrop-filter: blur(5px);  
  backdrop-filter: blur(5px);  
  -webkit-mask-image: url(images/star-mask.png);  
  mask-image: url(images/star-mask.png);  
}
```

It's not always easy to keep track of when vendor prefixes are necessary, and doing it by hand can be 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 `["last 3"]` ensures that your code is compatible (when possible) with the latest three versions of all major browsers. Autoprefixer uses the latest data from the caniuse.com database to determine when prefixes are needed.

B.2.2 cssnano

Cssnano (<https://cssnano.github.io/cssnano/>) 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.3 Lightning CSS

Lightning CSS (<https://lightningcss.dev/>) is a relatively new tool that includes several features listed previously, all in one tool: preprocessor conveniences, automatically adding vendor prefixes, and minification. Lightning CSS provides preprocessing support for a lot of newer CSS features that aren't yet universally supported in browsers, such as nesting, wide gamut color functions, and newer media query syntaxes using less-than (<) and greater-than (>) symbols.

Most importantly, it is highly configurable, so you can disable these features when you feel native browser support is sufficient enough for you to rely on them without transpiling them away. I encourage you to look through the Lightning CSS documentation for a complete list of supported features.

Lightning CSS also minifies your CSS for you and adds vendor prefixes based on how you configure it. It even *removes* unnecessary or outdated vendor prefixes if your

source code has them. Lightning CSS also supports partials, much like Sass, using the `@import` rule.

CSS has come a long way in the past several years, and it is possible these tools will be considered obsolete at some point in the future, but I don't think that day has quite arrived. When you start a project, it's important to consider your toolset. The preprocessors listed in this appendix all have a proven reputation. It's worth having a general feeling for what each can offer so you can choose which, if any, is right for your needs.

Symbols

@container at-rule 236, 274–275, 283
@extend Sass rule 495
@font-face at-rule 335, 341–344, 346, 348
@for Sass rule 497
@import at-rule 4, 215, 219
@include Sass rule 494–495
@keyframes at-rule 460, 462, 476–477
@layer at-rule 213–215, 236
@media at-rule 47, 180, 193
 See also media queries
@mixin Sass rule 494
@property at-rule 428, 430
@scope at-rule 236, 239, 256–257, 263–264
@supports at-rule 31–32, 236
@use Sass rule 493–494

Numerics

3D transforms 451–457
 advanced 455–457
 backface-visibility property 456
 perspective-origin property 455
 transform-style (preserve-3D) property 457
animating 463–468
 adding animation to layout 466
 building layout without animations 463–466
controlling perspective 452–455

A

absolute positioning 160–162
 Close button 160
 pseudo-element 161
absolute units 35–37
accent-color property 223
:active pseudo-class 18, 186, 370–371, 482
adjacent sibling combinator 481
Afif, Temani 397
::after pseudo-element 166, 190, 232, 403
align-content property 112, 115, 151
align-items property 112, 115, 118, 151
alignment properties 150
align-self property 113, 116, 118, 151
all property 220
alpha channel 309
Andrew, Rachel 129, 152
animation-delay property 468–471
animation-duration property 462
animationend event 415, 477
animation-fill-mode property 468, 470–471
animation-iteration-count property 462
animation-name property 462
animation property 460, 462
animations 459
 3D transforms 463–468
 conveying meaning through 471–477
 drawing user attention 475–477
 responding to user interaction 471–474
 delaying 468–471
 dropdown menus, transitioning to auto
 height 426
 fill mode 468–471
 keyframes 460–463
 performance, rendering pipeline 449–451
 scroll-based timelines 477–479
animation-timing-function property 462
:any-link pseudo-class 18, 212–213, 482

`<aside>` element 63, 284
 at keyword 395
 at-rules 4
 nesting 235–237
 attributes 4
 attribute selectors 13, 486
 author styles 8–9, 24, 33, 217, 219, 462
 auto-fill keyword 139, 202
 auto-fit keyword 140, 202
 auto height 426
 Autoprefixer 498
 auto value 76, 102

B

backdrop filter 387, 389
 backface-visibility property 456
 background-attachment property 358
 background-blend-mode property 376, 380
 background-clip property 358
 background-color property 358, 423, 461
 background-image property 358–359, 390–391
 background-origin property 358
 background-position property 28, 358
 background property 358
 background-repeat property 358, 376
 background-size property 358, 376
 BEM (Block-Element-Modifier) 237, 242–243, 256, 263, 298
 Bézier curves, custom 416–419
 bleed 79
 blend modes 357, 374–381
 adding texture to images 378
 mix 380
 tinting images 376
 types of 377
 block flow direction 67
 block-level elements 65, 78–79, 200, 203, 277, 403
 blocks 242
 block-size property 67, 69, 73
 blur() function 385, 388
`<body>` element 19, 66, 330, 340
 body copy, spacing 351
 border area 71
 border-box 399
 border-box value 73, 404
 border-collapse property 20
 border-color property 168
 border-inline-start property 68
 border property 373
 border-radius property 404, 413

border-spacing property 20
 border triangle 166–168
 bottom property 154
 boxes, flexbox and 94–100
 adding padding and spacing 98–100
 building menus 97
 box model 4, 61, 70–75, 330
 adjusting 73
 magic numbers 72
 universal border-box sizing 74
 box-shadow property 28, 370
 box-sizing property 73–74
 breaking changes 267–268
 breakpoints 182, 194
 brightness() function 386
 browsers, evergreen 29
 button module, variants 244
 buttons, creating with hybrid look 373

C

calc() function 54, 323, 423, 492
 Cartesian coordinates 28, 171, 187, 307, 418
 cascade 5–8, 212, 256, 258
 cascaded values 18
 cascade layers 212–228
 additional layers 228
 defining 213–215
 global layer 224
 layer order and priority 216–219
 layout layer 225
 modules layer 226
 reset layer 221
 revert-layer keyword 219
 theme layer 223
 utilities layer 227
 centered class 118
 centering content, horizontally 65
 child combinator 86, 481
 ch unit 347
 circle() function 395, 404
 clamp() function 55, 283
 classes
 form field pseudo-classes 485
 general purpose pseudo-classes 482
 pseudo-class selectors 482–485
 selecting based on position among sibling elements 484
 class selector 6, 11–13, 225, 230–231, 281, 481
 clear property 403
 clip-path property 394–395, 397, 404
 clipping, overview of 383

- clipping paths 394–399
 - Firefox clip-path tools 397
 - other clip path types 399
 - polygon clipping paths 396
- clip value 75
- Close button, positioning 160
- cm unit 36
- collapsed margins 79–83
- collapsing
 - between text 80
 - multiple margins 81
 - outside container 81
- color
 - conversion 314
 - gamuts and color spaces 306–308
 - OKLCH. *See* OKLCH color; OKLCH color notation
 - overview of 299
- color() function 312
- color and contrast 293
- color fonts 350
- color interpolation 363–366
 - browser bugs and color quirks 366
 - polar color spaces 364
- color-mix() function 320, 364
- color notations 308–312
 - color() function 312
 - hex color 309
 - HSL color 309
 - HWB color 310
 - LAB and OKLAB color 311
 - LCH and OKLCH color 311
 - next-generation color spaces 310–311
 - RGB color 309
 - See also* specific color notations
- color pickers 320
- color property 19
- colors, contrast for 323–325
- color-scheme property 223, 287
- color space 307
- color stops 360
 - repeating gradients 362
 - stripes 361
- column class 185
- columns, adding responsive 200–202
- combinators 481
- communicating with contrast 295–299
 - establishing patterns 296
 - implementing design 297
- complement, defined 318
- components. *See* modules
- composite, rendering pipeline 450
- compound selectors 482
- computed value 38
- conflicting declarations 11
- conic-gradient() function 368–369
- container element 271
- container-name property 275
- container queries 202, 270
 - container-relative units 280–283
 - container size queries 271, 274–276
 - container style queries 271, 283–289
 - example of 271–276
- containers 276–280
 - decoupling modules from 284
 - flexbox and 94–100
 - fluid layouts 202–206
 - modular CSS and 279
 - names of 278
 - reducing code duplication 285–289
 - spacing elements within 83–87
 - types of 277–278
- container shorthand property 279
- container-type property 275, 277, 279
- containment, defined 277
- contain property 144, 277
- contain value 376
- content area 71
- content attribute 192
- content-box 399
- content-box value 73, 404
- content property 161–162
- context-dependent selectors, avoiding 245
- contrast 295–299
 - establishing patterns 296
 - for font colors 323–325
 - implementing design 297
- contrast() function 386
- conveying meaning through animation
 - 471–477
 - drawing user attention 475–477
 - responding to user interaction 471–474
- CORS (Cross-Origin Resource Sharing) 390
- cover property 144
- cover value 376
- cross axis 94
- cross end 94
- cross start 94
- CSS3 49
- CSS (Cascading Style Sheets) 3–5, 49
 - color notations. *See* color notations
 - combinators 481
 - creating triangles 166–168
 - custom properties 57–61

CSS (Cascading Style Sheets) (*continued*)

- filters 384–389
- inheritance 19
- layers and nesting 211
- Lightning CSS 499
- masks 389–394
- modules 239–240
- nesting 230–237
- pattern libraries 265–268
- progressive enhancement 29–33
- scope 256–265
- shorthand properties 25–28
- special values 21–25
- CSS First workflow 266
- CSS in JS 256
- cssnano 499
- CTA (Call to Action) 304
- cubic-bezier() function 447
- currentColor keyword 322
- cursor property 110
- custom axes 350
- custom Bézier curves 416–419
- custom properties 57–61
 - changing dynamically 59–61
 - transitions 428–429
- custom properties, naming 216–217

D

- dark mode 196, 286–289, 317
- data-scope attribute 264
- data-theme attribute 286, 288
- declaration block 4
- declarations
 - conflicting 11
 - nonnested 236
- decreasing hue 366
- deg units 309, 315, 360
- deleting code 267–268
- dense keyword 141
- depth
 - adding with gradients and shadows 370–374
 - creating buttons with hybrid look 373
 - creating elements with flat design 372
- descendant combinator 232, 481
- direct 46
- design, implementing 297
- desugars 233
- device-width 192
- DevTools 20–21, 69, 146, 192, 276, 313–314, 316, 319, 346, 364, 397, 407, 417, 491

- direction, flexbox 106–110
 - changing 108
 - styling login form 109–110
- display
 - block declaration 158
 - block rule 443
 - flex 443
- Display p3 gamut 306, 311
- display property 423–425
 - flexbox and 94–100
- documentation 266
- document flow 62–65, 75, 93, 118, 154, 175, 201, 277, 400, 444
- DOM (Document Object Model) 19, 43, 121, 146
- donut scoping 261
- double-container pattern 66, 70, 79, 97, 303
- dropdown menus
 - creating 163–166
 - transitioning to auto height 426
- dropdown module 250–251, 262, 264
- drop-shadow() function 386
- dynamic viewport units 53

E

- Eckles, Stephanie 221
- elements 242
 - creating with flat design 372
 - floats 399–407
 - height property 75–78
 - modules with multiple elements 246
 - shapes 399–407
 - small, spacing 353–356
- ellipse() function 395, 404
- em units 37–44
 - using for font-size 39–44
 - vs. pixels 193, 327–329
- encapsulation 239
- end property 68
- experimental web platform features 33, 256, 389
- ex unit 347

F

- featured class 16, 140
- feature queries 31–33
- fill mode 468–471
- filter property 174, 385, 387–388
- filters 384–389
 - backdrop filter 387–389
 - types of 386–387

- Firefox, clip-path tools 397
- Firefox, grid debug tools 146
- fixed positioning 154–160
 - controlling size of positioned elements 159
 - creating modal dialog with 154–158
 - preventing screen from scrolling while modal dialog is open 158
- flat design, creating elements with 372
- flex-basis property 102, 108, 113, 202, 277
- flexbox 93
 - alignment, spacing, and other details 110–118
 - direction 106–110
 - items 100–105
 - principles of 94–100
 - sizing items 100–105
 - working with grid layout 129
- flex-direction property 106–108, 111
- flex-flow property 111, 114
- flex-grow property 102–103, 108, 113, 144, 203, 277
- flex property 100–101, 106, 113
- flex-shrink property 104–105, 108, 113, 203
- flex-wrap property 111, 114, 129, 202
- floats 399–407
 - defining 404–407
 - floating 400–404
- fluid layouts 202–206
 - adding styles for large viewports 203
 - tables 204
- fly-in labels 446
- focus 166, 319, 444
- :focus-visible pseudo-class 483
- :focus-within pseudo-class 447, 483
- FOIT (Flash of Invisible Text) 345
- font axes 348–350
- font-display property 346–347
- font-family property 9, 19–20, 340, 342
- font-optical-sizing property 349
- font property 19, 158
- font-relative units 347
- fonts 335–341
 - Google Fonts 337–341
 - icon fonts 445
 - using @font-face 341–344
 - variable fonts, font variation axes 348–350
 - web fonts 335–337
- font size, viewport-relative units for
 - calc() function 54
 - clamp() function 55
- font-size property 19
 - using ems for 39–44
 - using rems for 39–44

- font-style property 19
- font-variant property 19
- font-variation-setting property 348–349
- font-weight property 19, 55, 348
- form fields, pseudo-class selectors 485
- FOUT (Flash of Unstyled Text) 344
- Frost, Brad 201
- fr unit 121

G

- gamuts and color spaces 306
- gap property 99, 111, 121
- general sibling combinator 482
- global layer 224
- Google Fonts 337–341
- gradient masks 406
- gradients 357–366
 - adding depth with 370–374
 - color interpolation 363–366
 - conic 368
 - creating buttons with hybrid look 373
 - creating elements with flat design 372
 - masking with 391
 - multiple color stops 360
 - radial 366
 - shadows and 370–374
- grad units 360
- grayscale() function 386
- grid 119
 - alignment properties 150
 - alternate syntaxes 132–135
 - anatomy of 122
 - building basic 120–122
 - debugging with dev tools 146–147
 - dense placement 141–142
 - explicit and implicit 136–146
 - item placement algorithm 136, 141
 - numbering grid lines 127
 - subgrid 146–150
 - working with flexbox 129
- grid-area property 134–135
- grid areas 122
- grid areas, naming 134–135
- grid-auto-columns property 136
- grid-auto-flow property 141
- grid-auto-rows property 136
- grid cells 122
- grid-column-end property 128
- grid-column property 127–128
- grid-column-start property 128

grid lines 122, 134
 naming 132–134
 negative 127, 136
 numbering 127–128
 grid-row-end property 128
 grid-row property 127–128, 150
 grid-row-start property 128
 grid-template-areas property 134–135
 grid-template-columns property 126, 149–150
 grid-template-rows property 126, 149–150
 grid tracks 122

H

hamburger icon 181
 hard-light blend mode 378, 381
 hardware acceleration 450
 :has() pseudo-class 159, 387, 483
 has-error class 252
 headings, spacing 353–356
 height property 73, 75–78, 100
 max-height property 77
 min-height property 77
 overflow property 75
 percentage-based heights 76
 hex color 29, 306, 309, 314
 hidden value 75, 425
 horizontal overflow 76
 :hover pseudo-class 18, 166, 232
 hsl() function 309–310
 hsla() function 310
 hsl color space 364
 HSL (hue, saturation, lightness) color 309
 <html> tag 286, 288, 313
 hue-rotate() function 386
 hwb() function 308
 HWB (hue, whiteness, blackness) color 310
 hybrid look, creating buttons with 373

I

icon fonts 445
 icons, scaling up 443
 ID selector 7, 481
 if 53
 IKEA 239
 images
 adding texture to 378
 responsive design 206–208
 tinting 376
 images, object-fit 144–145
 img selector 257

implementing design 297
 implicit grids 136–146
 adding variety 140–142
 adjusting grid items to fill grid track 142–146
 implicit scope 264
 !important declaration 7, 10, 14, 19, 197, 217–218, 227, 415
 increasing hue 366
 individual transform properties 436
 infinite keyword 474
 inheritance 19–20
 inherit keyword 21–23
 initial containing block 161
 initial keyword 23
 initial-scale 192
 in keyword 364
 inline base direction 67
 inline elements 65
 spacing 333–334
 inline-size container type 277
 inline-size property 73, 110
 inline styles 7, 11
 inner bound. *See* scope, scoping limit
 inset() function 399, 404
 inset property 154, 371
 interpolation method 321–322, 363–366
 in unit 36
 invert() function 386
 :is() pseudo-class 228–230, 234
 is-expanded class 252
 is-loading class 252, 473–475
 is-open class 156, 158, 190, 251, 427
 'ital' axes 349

J

JavaScript, alternate approaches in 256
 justify-content property 111, 114, 151, 303
 justify-items property 151
 justify-self property 151

K

keyframes 460–463
 keyup event listener 476

L

lab() function 308
 LAB color 311
 labels, fly-in 446
 landmarks 188

- large viewport units 53
- layers 7, 212–220, 265
 - and nesting 211
 - cascade layers 221–228
 - defining 213–215
 - layer order and priority 216–219
 - revert-layer keyword 219
- layout, rendering pipeline 450
- layout layer 225
- lch() function 308
- LCH color 311
- leading 331, 353
- leading-trim property 331
- left property 154, 200, 474
- length values 35
- Less preprocessor 57, 489
- letter-spacing property 19, 350, 353
- lh unit 347
- light and dark themes 196
- Lightning CSS 499
- linear-gradient() function 359–360, 362, 364
- line height, spacing 329–332
- line-height property 19, 55, 57, 162, 350–351, 353
 - unitless numbers 55–57
- <link> tag 195, 215, 339
- link styles 17
- list-style-image property 20
- list-style-position property 20
- list-style property 20
- list-style-type property 20
- lobotomized owl selector 86
- local() function 343
- localStorage API 289
- logical properties 53, 67–69
 - shorthand 69
- longer hue 365–366
- LoVe/HAtE 18
- luminance, masking using 392

M

- magic numbers 72–73
- <main> element 63, 101
- main axis 94
- main end 94
- main start 94
- Marcotte, Ethan 37, 180
- margin area 71
- margin-box 399, 404
- margin-box value 404

- margin property 26
- margins
 - collapsed 79–83
 - negative 78
- mask-clip property 393
- mask-composite property 394
- mask-image property 390–391, 393
- mask-mode property 392, 394
- mask-origin property 393
- mask-position property 393
- mask-repeat property 393
- masks 389–394
 - masking using luminance 392
 - masking with gradients 391
 - other mask properties 393
- mask-size property 390, 392–393
- masonry layout 150
- Matuzović, Manuel 150
- max-height property 77–78
- max-inline-size property 70, 248
- max-width media feature 194
- max-width property 67
- media features 194
- Media module 242, 246–249, 254, 257, 259, 272–274
- media queries 192–202
 - adding breakpoints to page 197–200
 - adding responsive columns 200–202
 - media types 196–197
 - nesting 235–237
 - prefers-color-scheme 196
 - range syntax 195
 - resolution based 194
 - types of 194–197
- .media selector 257
- media types 196
- menus
 - flexbox and 97–100
 - fly-in labels 446
 - mobile menus 188
- min-height property 77, 156, 376
- minifier 499
- minmax() function 139
- min-width media feature 194
- mix-blend-mode property 380
- mix blend modes 380
- mm unit 36
- mobile browser emulation 192
- mobile design, viewport meta tag 191
- mobile first 180
- mobile menus 188
- modal-backdrop 158, 171

- modal dialog box 155
 - creating with fixed positioning 154–158
 - preventing screen from scrolling while open 158
- modifiers 242
- modular CSS, containers and 279
- modules
 - composing into larger structures 249–255
 - decoupling from containers 284
 - defined 239
 - naming modules 254–256
 - overview of 240
 - simple module 241
 - variations of 242, 244–245
 - with multiple elements 246, 248–249
- modules layer 226
- most 67
- motion, transforms in 437

N

- naming 254–255
- nav element 185, 439
- navigation menu 181, 188, 295, 355
- nav-links element 442, 447
- .nav-menu class 199
- nav-menu class 185, 191, 245
- negative margins 78
- nesting 230–237
 - media queries and other at-rules 235–237
 - nesting rulesets 235–237
 - nuances of nesting selector 233–234
 - using nesting selector 231–235
- nesting selector (&) 231–234
- non-animatable properties 420–426
 - fading in and out 424–426
 - properties that cannot be animated 423
- none keyword 312
- nonnested declarations 236
- normal document flow 63, 65–70
 - centering content horizontally 65
 - logical properties 67–69
 - shorthand logical properties 69
- normalize. *See* reset styles
- normal keyword 351
- no-scroll class 159
- not keyword 276
- npm run sass command 490
- :nth-child() pseudo-class 283, 448
- numbers, unitless 55–57

O

- object-fit property 144
- object-position property 402, 407
- oklab() function 308
- OKLAB color 311
- oklch() function 308–309
- OKLCH color 311–312
- OKLCH color notation 313–323
 - deriving colors from others on page 318–323
 - naming color variables 316
 - selecting new colors for palette 318
 - switching stylesheet to 315
- oklch color space 364
- opacity() function 386
- opacity property 174, 424, 450, 470
- 'opsz' axis 349
- order of appearance 16
- order property 113, 116
- origin, changing transform 435
- origin of stylesheets 8
 - important declarations 10
 - user-agent styles 8
- outline 72
- outline-color property 72
- outline-offset property 72
- outline-style property 72
- outline-width property 72
- overflow property 75–76
- overflow-x property 76
- overflow-y property 76
- overlay blend mode 378

P

- padding, flexbox and 98–100
- padding area 71
- padding-box 399
- padding-box value 404
- padding property 26, 399
- paint, rendering pipeline 450
- palettes, gamuts and color spaces 306
- panel-body class 46
- parallax effect 479
- path() function 399, 404
- paths, clipping 394–399
 - Firefox clip-path tools 397
 - other clip path types 399
 - polygon clipping paths 396
- pattern libraries 265–268
 - breaking changes 267–268
 - CSS First workflow 266
 - refactoring and breaking changes 267–268

- patterns, establishing 296
- pc (pica) unit 36
- percentage-based heights 76
- performance
 - animation 449–451
 - web fonts 344–350
- perspective 452–455
- perspective() function 453–454
- perspective-origin property 455
- perspective property 453–454
- Pickering, Heydon 86
- pixels 44–50
 - ems vs. 327–329
 - making panels responsive 47
 - pc (pica) vs. 36–37
 - resizing single components 48–50
 - setting default font size 45
- place-content property 152
- place-items property 112, 116, 152
- place-self property 113, 116, 152
- pointer value 110
- polar color spaces 364
- polygon() function 396, 404
- polygons, clipping paths 396
- Portis, Eric 311
- position
 - fixed declaration 443
 - fixed property 154
- positioning 153
 - absolute 160–162
 - fixed 154–160
 - in modules 252
 - relative 163–168
 - sticky 175–177
- positioning elements
 - stacking contexts 169–175
 - z-index property 169–175
- position property 78, 153–154, 172
- PostCSS 498
 - Autoprefixer 498
 - cssnano 499
- preconnect 340, 438
- prefers-color-scheme media query 286
- prefers-reduced-motion media query 218, 414
- preprocessors 488
 - PostCSS 498–499
 - Sass, features of 490
- print styles 196–197
- progressive enhancement 29–33
 - enhancing selectors 30
 - feature queries using @supports() 31–33
 - using cascade for 29

- property, defined 4
- proximity 258–261
- pseudo-classes
 - :is() 228–230
 - :where() 228–230
- pseudo-class selectors 13, 482–485
 - form field pseudo-classes 485
 - general purpose pseudo-classes 482
 - selecting based on position among sibling elements 484
- pseudo-elements
 - positioning 161
 - selectors 485
- pt (point) unit 37
- pull-quote class 245
- px (pixel) unit 37, 193

Q

- q (quarter-millimeter) unit 36–37

R

- race-detail module 281, 284, 288
- radial-gradient() function 366
- radial gradients 366
- rad unit 360
- range syntax 195
- readability, adjusting spacing for 350–356
 - body copy spacing 351
 - headings, small elements, and spacing 353–356
- Rec2020 gamut 306
- refactoring 267–268
- registered axes 349
- relative color 323
- relative positioning 163–168
 - creating CSS triangles 166–168
 - creating dropdown menus 163–166
- relative units 35
 - pixels 44–50
 - responsive design 36
- remainder, flexbox sizing 102–103
- rem units 37–44
 - using for font-size 39–44
- rendering pipeline 449–451
- render tree 169
- repeat() function 126–127, 134, 139
- repeating-conic-gradient() function 369
- repeating-linear-gradient() function 362
- repeating-radial-gradient() function 367
- replaced elements 402
- reset layer 221

- reset styles 221–223
- resilience 33
- resizing a module 48
- responsive design 37, 179
 - fluid layouts 202–206
 - images 206–208
 - media queries 192–202
 - mobile first 180
 - overview of 36
- revert keyword 24, 219
- revert-layer keyword 219
 - all property 220
- rgb() function 309–310
- rgba() function 310
- RGB (red, green, blue) color 309
- right property 154, 190, 200
- rlh unit 347
- :root pseudo-class 43, 57, 61, 317, 483
- rotate() function 432–433
- rotate property 452
- rotateX() function 452–453
- rotateY() function 452, 466
- rotateZ() function 452
- rulesets 4
 - nesting 235–237

S

- sans-serif typeface 337
- sass command 490
- Sass preprocessor 57, 489
 - features of 490
 - installing 489
 - running 490
- saturate() function 386
- scale() function 434
- scaling up icons 443
- scope 7, 238, 256–265
 - implicit 264
 - layers 265
 - proximity 7, 258–261
 - scoping limit 261–264
- scoping limit 261–264
- scripts entry 490
- script tag 476
- scroll() function 478
- scroll-based timelines 477–479
- scrollHeight property 427
- scroll value 76
- selectors
 - compound 482
 - :is() pseudo-class 228–230
 - progressive enhancement 30
 - pseudo-class selectors 482–485
 - pseudo-element selectors 485
 - specificity 7, 11, 13–14
 - where() pseudo-class 228–230
- selector specificity. *See* specificity
- semver 267
- sepia() function 386
- serif typeface 337
- shadow DOM 263–264
- shadows 357, 370–374
 - adding depth with gradients and 370–372
 - creating buttons with hybrid look 373
 - creating elements with flat design 372
- shake animation 475–477
- shape-image-threshold property 407
- shape-margin property 405, 407
- shape-outside property 383, 399, 404, 406, 408
- shapes 383, 399–407
 - defining 404–407
 - floating 400–404
 - overview of 383
- shorter hue 366
- shorthand logical properties 69
- shorthand properties 25–28
 - order of values 26–28
 - overriding other styles 25
- sibling combinators 482
- size attribute 110
- size container type 277
- skeuomorphism 372–373
- 'slnt' axes 349
- slot class 260
- small viewport units 53
- soft-light blend mode 378
- source map 491
- source order 7, 16
 - cascaded values 18
 - link styles and 17
- spacing 327–334
 - adjusting for readability 350–356
 - ems vs. pixels 327–329
 - flexbox and 98–100
 - inline elements 333–334
 - line height 329–332
- spacing elements within containers 83–87
 - changing content 85
 - creating general solution 86
- span keyword 128, 141
- SPAs (single page applications) 256

- special values 21–25
 - inherit keyword 21
 - initial keyword 23
 - revert keyword 24
 - unset keyword 23
- specificity 11
 - considerations 14
 - notation for 13
 - of nested selectors 234
 - source order 16–18
- spin animation 473–474
- sponsor-link class 85
- sprite sheet 445
- src property 342
- srcset attribute 207
- sRGB (Standard RGB) 306
- stack class 86–87
- stacking contexts 169–175
 - manipulating stacking order with z-index 171
 - overview of 172–175
 - rendering process and stacking order 169
- stacking order 173–174
- state classes 252
- static value 154
- steps 419
- steps() function 419
- sticky positioning 175–177
- stripes 361
- style() function 283
- style attribute 11
- style queries
 - benefits of 285
 - container style queries 283–289
- styles, conflicting declarations, overview 11
- stylesheets, origin of 7–10
- subelements, using with variants 248
- subgrid 146, 149–150
- submenu class 166
- Sullivan, Nicole 246
- SVG (Scalable Vector Graphics) 439
- syntactic sugar 233

T

- table-caption display 83
- table-cell display 83
- table display 83
- table-inline display 83
- table-row display 83
- tables, fluid layouts 204
- tag names, avoiding generic in module
 - selectors 249

- tagname selector 481
- text-align property 19
- text-indent property 19, 162
- text-shadow property 28, 187, 370
- text-transform property 19, 356
- theme layer 223, 300
- three column layout 105, 120–121, 183, 301, 304–305
- tile class 101, 124
- tile module 304
- tile-row module 304
- timelines, scroll-based 477–479
- timing functions 415–420
 - custom Bézier curves 416–419
 - steps 419
- to keyword 261
- top-nav module 302
- top property 154, 200, 474
- to right angle 359
- tracking 353
- transform declaration 454
- transform-origin property 435
- transform property 174, 374, 431–432, 437, 454, 461, 470
- transforms 431–437
 - 3D 451–457
 - applying multiple 436
 - changing transform origin 435
 - individual transform properties 436
 - in motion 437
- transform-style (preserve-3D) property 457
- transition-* properties 412–413
- transition-behavior: allow-discrete property 424
- transition-delay property 413, 425, 448, 468
- transition-duration property 412–413
- transitionend event 413
- transition property 387
- transition-property property 412–413
- transitions 411–415
 - custom properties 428–429
 - non-animatable properties 420–426
 - reduced motion 414
 - staggering 448
 - timing functions 415–420
- transition shorthand property 413
- transition-timing-function property 413
- translate() function 451
- translate function 447
- translate property 451
- translateX() function 451
- translateY() function 374, 451
- translateZ() function 451, 466

transparent keyword 359
 triangle, CSS only 166–168
 TRouBLe mnemonic 26
 turn unit 360
 two column layout 101–103, 122, 177, 271
 typeface 338
 type safety 429
 type selector 12, 43, 481
 typography 326

U

ul element 439
 unicode-range property 342
 unitless numbers 55–57
 units
 overview of relative 36–37
 responsive design 36
 viewport-relative 50–55
 universal selector (*) 13, 74, 481
 unset keyword 23
 url() function 429
 user-agent styles 8–10, 24, 65, 69, 98, 217, 221
 user attention 475–477
 user interaction 471–474
 user-scalable=no 192
 user styles 8
 utilities layer 227
 utility classes 227

V

values
 cascaded 18
 defined 4
 var() function 58–59
 variable fonts, variation axes 348–350
 variations of modules 242
 avoiding context-dependent selectors 245
 button module variants 244

vendor prefixes 389, 498–499
 vertical-align property 151
 viewport 50
 viewport meta tag 191
 viewport-relative units 50–55
 for font size 54–55
 selecting from newer units 52–54
 viewport sizes, using multiple images for 206
 visibility property 424–425
 visible value 75, 425

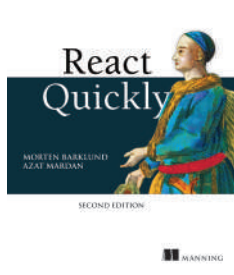
W

WCAG (Web Content Accessibility Guidelines) 323
 'width' axes 349
 web fonts 335–337
 performance considerations 344–350
 -webkit vendor prefix 389
 web-safe fonts 335
 Westfall, Brad 66
 'wght' axes 349
 :where() pseudo-class 228–230
 whitespace 295
 white-space property 19
 wide gamut color 310, 313, 499
 width property 73, 100, 110
 will-change property 451
 WOFF (Web Open Font Format) 343
 word-spacing property 19
 writing mode 67–68

Z

z-index property 55, 169, 171–175
 manipulating stacking order with 171
 rendering process and stacking order 169
 stacking contexts 172–175

RELATED MANNING TITLES

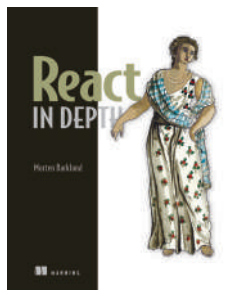


React Quickly, Second Edition
by Morten Barklund and Azat Mardan

ISBN 9781633439290

456 pages, \$59.99

July 2023

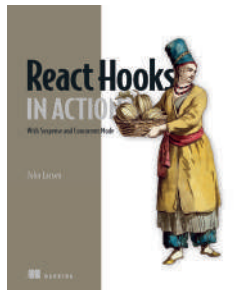


React in Depth
by Morten Barklund

ISBN 9781633437340

400 pages (estimated), \$59.99

Summer 2024 (estimated)

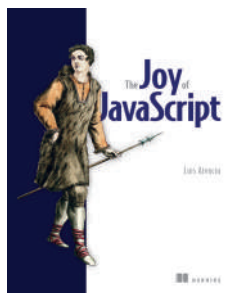


React Hooks in Action
by John Larsen

ISBN 9781617297632

376 pages, \$49.99

March 2021



The Joy of Javascript
by Luis Atencio

ISBN 9781617295867

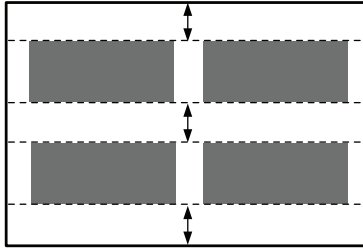
360 pages, \$39.99

February 2021

For ordering information, go to www.manning.com

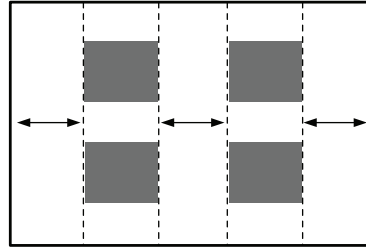
Alignment properties

`align-content`



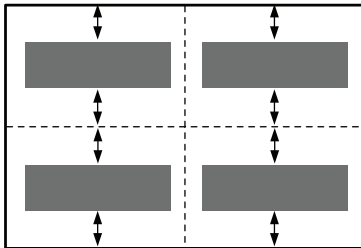
Controls the spacing of flex/grid rows within their container

`justify-content`



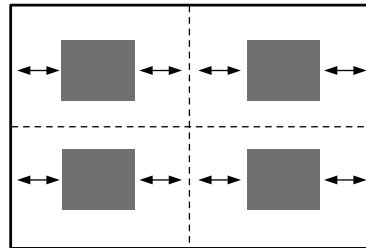
Controls the spacing of flex items or grid columns within their container

`align-items`



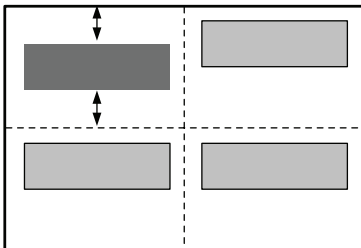
Controls the placement of flex/grid items within their row

`justify-items`



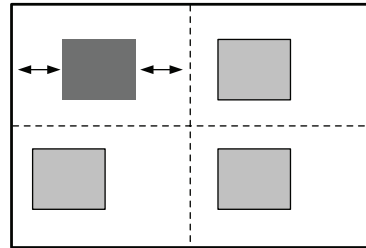
Controls the placement of grid items within their column (has no effect on flex items)

`align-self`



Controls the placement of a single flex/grid item within its row

`justify-self`



Controls the placement of a single grid item within its column (has no effect on flex items)

CSS alignment properties control the vertical and horizontal placement of elements in flexbox and grid layouts. It can be easy to confuse the properties involved in alignment; use this table as a quick reference to keep them straight.

CSS IN DEPTH Second Edition

Keith J. Grant • Foreword by Chris Coyier

To deliver truly beautiful frontends, you need to know CSS inside and out. And as CSS grows and matures, even experienced CSS developers will find a whole new set of skills to catch up on! This guide will help you discover everything that CSS is capable of—from the must-knows and brand-new features to the hidden insights you won't find anywhere else!

CSS in Depth, Second Edition, deep dives into the CSS techniques used by the world's most skilled web developers. Inside, you'll learn how to build and organize a modular CSS code-base that's easy to understand, maintain, and reuse. This new edition is fully revised to cover the CSS standard's latest additions, including cascade layers, logical properties, new color functions, subgrid, and container queries.

What's Inside

- Vertical centering, margin collapsing, and positioning
- Dropdown menus and modal dialog boxes
- Animations to guide user focus
- Avoiding common CSS pitfalls

Written for web developers who know the basics of CSS and HTML.

Keith J. Grant is a Principal Software Engineer at Red Hat, where he works on front-end code for the Ansible Controller web application using JavaScript and CSS.

The technical editor on this book was Chris Chinchilla.

For print book owners, all ebook formats are free:
<https://www.manning.com/freebook>

“A systemic approach to CSS, clearly describing what makes the language unique and how all the pieces fit together.”

—Miriam Suzanne, Invited Expert on the W3C CSS Working Group

“Clear explanations and practical examples demystify complex concepts. An invaluable resource for both new and experienced developers.”

—Adam Argyle, Chrome CSS Developer Advocate, Google

“An excellent in-depth book diving into the gritty details. Offers a great code-along experience with its examples and explanations.”

—Simon Verhoeven
Software Engineer, Itenium



ISBN-13: 978-1-63343-755-5

