O'REILLY®

Fifth Edition

CSS
The Definitive Guide
Web Layout and Presentation



Eric A. Meyer
& Estelle Weyl

O'REILLY®

CSS: The Definitive Guide

If you're a web designer or app developer interested in sophisticated page styling, improved accessibility, and less time and effort expended, this book is for you. This revised fifth edition provides a comprehensive guide to CSS implementation along with a thorough review of the latest CSS specifications.

Authors Eric Meyer and Estelle Weyl show you how to improve user experience, speed development, avoid potential bugs, and add life and depth to your applications through layout, transitions and animations, borders, backgrounds, text properties, and many other tools and techniques. They read the specs so you don't have to!

This guide covers:

- **Selectors, specificity, and the cascade, including cascade layers**
- **CSS values and units; and media, feature, and container queries**
- **Details on font technology and ways to use any available font variants**
- **Text styling, from basic decoration to changing the entire writing mode**
- **Padding, borders, outlines, and margins, including logical properties**
- **Colors, backgrounds, and gradients, including conic gradients**
- **Accessible data tables**
- **Flexible box and grid layout systems, including subgrids**
- **2D and 3D transforms, transitions, and animations**
- **Filters, blending, clipping, and masking**

"Estelle and Eric provide not only the details of CSS syntax and features, but also practical advice on how it all fits together. Whether you're new to the language, interested in a refresher, or curious about the latest developments, this book is truly the definitive guide for any developer."

—Miriam Suzanne
Cofounder of OddBird and invited expert at W3C CSS Working Group

Eric A. Meyer is an internationally recognized expert on HTML, CSS, and web standards, founder of Complex Spiral Consulting, and cofounder of the microformats movement. He's written several books on CSS and design.

Estelle Weyl is a technical writer, open web evangelist, and standardista. She teaches all things frontend, including CSS, HTML, JavaScript, web performance, and accessibility on MDN, Web.Dev, Frontend Masters, and IRL.

WEB DEVELOPMENT

US $89.99 CAN $112.99
ISBN: 978-1-098-11761-0



Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

FIFTH EDITION

CSS: The Definitive Guide

Web Layout and Presentation

Eric A. Meyer and Estelle Weyl

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

CSS: The Definitive Guide
by Eric A. Meyer and Estelle Weyl

Copyright © 2023 Eric A. Meyer and Estelle Weyl. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn
Development Editor: Rita Fernando
Production Editor: Elizabeth Faerm
Copyeditor: Sharon Wilkey
Proofreader: JM Olejarz

Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2000:	First Edition
March 2004:	Second Edition
November 2006:	Third Edition
November 2017:	Fourth Edition
June 2023:	Fifth Edition

Revision History for the Fifth Edition
2023-05-30: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098117610* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *CSS: The Definitive Guide*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-11761-0

[LSI]

Table of Contents

Preface ..... **xix**

1. CSS Fundamentals **1**
A Brief History of (Web) Style 1
Stylesheet Contents 2
Rule Structure 2
Vendor Prefixing 3
Whitespace Handling 4
CSS Comments 5
Markup 6
Elements 7
Replaced and Nonreplaced Elements 7
Element Display Roles 7
Bringing CSS and HTML Together 11
The <link> Tag 12
The <style> Element 16
The @import Directive 16
HTTP Linking 18
Inline Styles 19
Summary 19

2. Selectors **21**
Basic Style Rules 21
Type Selectors 22
Grouping 23
Grouping Selectors 23
Grouping Declarations 24
Grouping Everything 25

Class and ID Selectors 27
Class Selectors 28
Multiple Classes 30
ID Selectors 32
Deciding Between Class and ID 33
Attribute Selectors 34
Simple Attribute Selectors 34
Selection Based on Exact Attribute Value 35
Selection Based on Partial Attribute Values 37
The Case-Insensitivity Identifier 42
Using Document Structure 42
Understanding the Parent-Child Relationship 43
Defining Descendant Selectors 45
Selecting Children 47
Selecting Adjacent-Sibling Elements 48
Selecting Following Siblings 50
Summary 51

3. Pseudo-Class and -Element Selectors. 53
Pseudo-Class Selectors 53
Combining Pseudo-Classes 54
Structural Pseudo-Classes 54
Location Pseudo-Classes 68
User Action Pseudo-Classes 74
UI-State Pseudo-Classes 77
The :lang() and :dir() Pseudo-Classes 83
Logical Pseudo-Classes 84
The :has() Pseudo-Class 89
Other Pseudo-Classes 94
Pseudo-Element Selectors 95
Styling the First Letter 95
Styling the First Line 96
Restrictions on ::first-letter and ::first-line 97
The Placeholder Text Pseudo-Element 97
The Form Button Pseudo-Element 98
Generating Content Before and After Elements 98
Highlight Pseudo-Elements 99
The Backdrop Pseudo-Element 101
The Video-Cue Pseudo-Element 102
Shadow Pseudo-Classes and -Elements 103
Shadow Pseudo-Classes 103
Shadow Pseudo-Elements 104

Summary 105

4. Specificity, Inheritance, and the Cascade. 107
Specificity 107
Declarations and Specificity 109
Resolving Multiple Matches 111
Zeroed Selector Specificity 111
ID and Attribute Selector Specificity 112
Importance 112
Inheritance 113
The Cascade 116
Sorting by Importance and Origin 118
Sorting by Element Attachment 119
Sorting by Cascade Layer 119
Sorting by Specificity 123
Sorting by Order 123
Working with Non-CSS Presentational Hints 126
Summary 126

5. Values and Units. 127
Keywords, Strings, and Other Text Values 127
Keywords 127
The all Property 130
Strings 131
Identifiers 132
URLs 132
Images 134
Numbers and Percentages 134
Integers 135
Numbers 135
Percentages 135
Fractions 136
Distances 136
Absolute Length Units 136
Resolution Units 139
Relative Length Units 139
Root-Relative Length Units 142
Viewport-Relative Units 144
Function Values 146
Calculation Values 148
Maximum Values 149
Minimum Values 150

Clamping Values 150
Attribute Values 151
Color 151
Named Colors 152
Color Keywords 153
Colors by RGB and RGBa 153
HSL and HSLa Colors 158
Colors with HWB 159
Lab Colors 160
LCH Colors 161
Oklab and Oklch 162
Using color() 162
Applying Color 163
Affecting Form Elements 165
Inheriting Color 165
Angles 166
Time and Frequency 167
Ratios 167
Position 168
Custom Properties 168
Custom Property Fallbacks 171
Summary 172

6. Basic Visual Formatting. 173
Basic Boxes 173
A Quick Primer 174
The Containing Block 175
Altering Element Display 176
Changing Roles 177
Handling Block Boxes 179
Logical Element Sizing 181
Content-Based Sizing Values 183
Minimum and Maximum Logical Sizing 185
Height and Width 186
Altering Box Sizing 189
Block-Axis Properties 190
Auto Block Sizing 191
Percentage Heights 192
Handling Content Overflow 193
Negative Margins and Collapsing 195
Collapsing Block-Axis Margins 196
Inline-Axis Formatting 200

Inline-Axis Properties 201
Using auto 202
Negative Margins 205
Percentages 207
Replaced Elements 208
List Items 209
Box Sizing with Aspect Ratios 209
Inline Formatting 210
Line Layout 211
Basic Terms and Concepts 212
Line Heights 215
Inline Nonreplaced Elements 215
Building the Boxes 216
Setting Vertical Alignment 218
Managing the Line Height 220
Adding Box Properties to Nonreplaced Elements 223
Changing Breaking Behavior 224
Glyphs Versus Content Area 225
Inline Replaced Elements 226
Adding Box Properties to Replaced Elements 227
Replaced Elements and the Baseline 228
Inline-Block Elements 229
Flow Display 231
Content Display 232
Other Display Values 233
Element Visibility 233
Animating Visibility 234
Summary 235

7. Padding, Borders, Outlines, and Margins. 237
Basic Element Boxes 237
Padding 238
Replicating Values 240
Single-Side Padding 242
Logical Padding 243
Percentage Values and Padding 245
Padding and Inline Elements 247
Padding and Replaced Elements 249
Borders 250
Borders with Style 251
Border Widths 256
Border Colors 260

Single-Side Shorthand Border Properties 263
Global Borders 265
Borders and Inline Elements 266
Rounding Border Corners 267
Image Borders 276
Outlines 293
Outline Styles 294
Outline Width 295
Outline Color 296
How They Are Different 297
Margins 299
Length Values and Margins 300
Percentages and Margins 301
Single-Side Margin Properties 302
Margin Collapsing 303
Negative Margins 305
Margins and Inline Elements 306
Summary 308

8. Backgrounds. 309
Setting Background Colors 309
Explicitly Setting a Transparent Background 310
Background and Color Combinations 311
Clipping the Background 313
Working with Background Images 315
Using an Image 316
Understanding Why Backgrounds Aren't Inherited 317
Following Good Background Practices 318
Positioning Background Images 319
Background Repeating (or Lack Thereof) 330
Getting Attached 339
Sizing Background Images 343
Bringing It All Together 350
Working with Multiple Backgrounds 352
Using the Background Shorthand 356
Creating Box Shadows 357
Summary 360

9. Gradients. 361
Linear Gradients 362
Setting Gradient Colors 363
Positioning Color Stops 364

Setting Color Hints 369
Understanding Gradient Lines: The Gory Details 371
Repeating Linear Gradients 376
Radial Gradients 379
Setting Shape and Size 380
Positioning Radial Gradients 382
Using Radial Color Stops and the Gradient Ray 383
Handling Degenerate Cases 388
Repeating Radial Gradients 391
Conic Gradients 392
Creating Conic Color Stops 395
Repeating Conic Gradients 397
Manipulating Gradient Images 399
Creating Special Effects 400
Triggering Average Gradient Colors 401
Summary 402

10. Floating and Positioning. 403
Floating 403
Floated Elements 404
Floating: The Details 406
Applied Behavior 413
Floats, Content, and Overlapping 417
Clearing 418
Positioning 422
Types of Positioning 422
The Containing Block 424
Offset Properties 424
Inset Shorthands 427
Setting Width and Height 428
Limiting Width and Height 430
Absolute Positioning 432
Containing Blocks and Absolutely Positioned Elements 432
Placement and Sizing of Absolutely Positioned Elements 435
Auto-edges 436
Placing and Sizing Nonreplaced Elements 437
Placing and Sizing Replaced Elements 441
Placement on the Z-Axis 444
Fixed Positioning 449
Relative Positioning 450
Sticky Positioning 452
Summary 456

11. Flexible Box Layout. 457
Flexbox Fundamentals 457
A Simple Example 459
Flex Containers 462
Using the flex-direction Property 462
Working with Other Writing Directions 467
Wrapping Flex Lines 468
Defining Flexible Flows 470
Understanding Axes 471
Arrangement of Flex Items 474
Flex Item Alignment 475
Justifying Content 475
Aligning Items 482
Aligning Flex Lines 489
Using the place-content Property 493
Opening Gaps Between Flex Items 494
Flex Items 498
What Are Flex Items? 498
Flex Item Features 500
Absolute Positioning 501
Minimum Widths 502
Flex-Item-Specific Properties 503
The flex Property 503
The flex-grow Property 505
Growth Factors and the flex Property 508
The flex-shrink Property 512
The flex-basis Property 522
The flex Shorthand 535
The order Property 541
Tabbed Navigation Revisited 544
Summary 546

12. Grid Layout. 547
Creating a Grid Container 547
Understanding Basic Grid Terminology 549
Creating Grid Lines 552
Using Fixed-Width Grid Tracks 554
Using Flexible Grid Tracks 558
Fitting Track Contents 565
Repeating Grid Tracks 567
Defining Grid Areas 571
Placing Elements in the Grid 577

Using Column and Row Lines 577
Using Row and Column Shorthands 582
Working with Implicit Grid 584
Handling Errors 587
Using Areas 588
Understanding Grid-Item Overlap 590
Specifying Grid Flow 591
Defining Automatic Grid Tracks 597
Using the grid Shorthand 599
Using Subgrids 602
Defining Explicit Tracks 605
Dealing with Offsets 605
Naming Subgridded Lines 610
Giving Subgrids Their Own Gaps 612
Grid Items and the Box Model 613
Setting Alignment in Grids 618
Aligning and Justifying Individual Items 619
Aligning and Justifying All Items 621
Distributing Grid Items and Tracks 624
Layering and Ordering 626
Summary 628

13. Table Layout in CSS. 629
Table Formatting 629
Visually Arranging a Table 629
Table Arrangement Rules 630
Setting Table Display Values 631
Inserting Anonymous Table Objects 635
Working with Table Layers 639
Using Captions 641
Table Cell Borders 642
Separated Cell Borders 643
Collapsed Cell Borders 646
Table Sizing 650
Width 650
Height 656
Alignment 657
Summary 660

14. Fonts. 661
Font Families 661
Using Generic Font Families 663

Using Quotation Marks 665
Using Custom Fonts 666
Using Font-Face Descriptors 667
Restricting Character Range 673
Working with Font Display 675
Combining Descriptors 677
Font Weights 680
How Weights Work 682
The font-weight Descriptor 685
Font Size 686
Using Absolute Sizes 687
Using Relative Sizes 689
Setting Sizes as Percentages 690
Automatically Adjusting Size 691
Font Style 695
The font-style Descriptor 697
Font Stretching 698
The font-stretch Descriptor 700
Font Synthesis 702
Font Variants 703
Capital Font Variants 705
Numeric Font Variants 707
Ligature Variants 709
Alternate Variants 711
East Asian Font Variants 713
Font Variant Position 713
Font Feature Settings 714
The font-feature-settings Descriptor 717
Font Variation Settings 718
Font Optical Sizing 719
Override Descriptors 720
Font Kerning 721
The font Property 722
Understanding font Property Limitations 724
Adding the Line Height 724
Using the Shorthand Properly 725
Using System Fonts 726
Font Matching 727
Summary 729

15. Text Properties. 731
Indentation and Inline Alignment 731

Indenting Text 732
Aligning Text 734
Aligning the Last Line 738
Word Spacing 739
Letter Spacing 741
Spacing and Alignment 742
Vertical Alignment 742
Adjusting the Height of Lines 743
Vertically Aligning Text 746
Text Transformation 751
Text Decoration 754
Setting Text Decoration Line Placement 754
Setting Text Decoration Color 756
Setting Text Decoration Thickness 756
Setting Text Decoration Style 757
Using the Text Decoration Shorthand Property 758
Offsetting Underlines 759
Skipping Ink 761
Understanding Weird Decorations 762
Text Rendering 764
Text Shadows 765
Text Emphasis 767
Setting Emphasis Style 767
Changing Emphasis Color 769
Placing Emphasis Marks 770
Using the text-emphasis Shorthand 771
Setting Text Drawing Order 771
Whitespace 772
Setting Tab Sizes 775
Wrapping and Hyphenation 776
Hyphenation 776
Word Breaking 778
Line Breaking 780
Wrapping Text 781
Writing Modes 783
Setting Writing Modes 783
Changing Text Orientation 784
Combining Characters 786
Declaring Direction 788
Summary 790

16. Lists and Generated Content. 791
Working with Lists 791
Types of Lists 792
List-Item Images 795
List-Marker Positions 798
List Styles in Shorthand 799
List Layout 800
The ::marker Pseudo-Element 802
Creating Generated Content 804
Inserting Generated Content 804
Specifying Content 807
Defining Counters 812
Defining Counting Patterns 819
Fixed Counting Patterns 821
Cyclic Counting Patterns 823
Symbolic Counting Patterns 826
Alphabetic Counting Patterns 829
Numeric Counting Patterns 830
Additive Counting Patterns 834
Extending Counting Patterns 836
Speaking Counting Patterns 838
Summary 840

17. Transforms. 841
Coordinate Systems 841
Transforming 845
The Transform Functions 848
Translation 848
Scaling 851
Element Rotation 854
Individual Transform Property Order 859
Skewing 860
Matrix Functions 861
Setting Element Perspective 864
More Transform Properties 866
Moving the Transform's Origin 866
Choosing the Transform's Box 869
Choosing a 3D Style 870
Changing Perspective 873
Dealing with Backfaces 876
Summary 878

18. Transitions. 879
CSS Transitions 879
Transition Properties 881
Limiting Transition Effects by Property 885
Setting Transition Duration 890
Altering the Internal Timing of Transitions 892
Delaying Transitions 898
Using the transition Shorthand 901
Reversing Interrupted Transitions 903
Animatable Properties and Values 904
How Property Values Are Interpolated 905
Interpolating Repeating Values 906
Printing Transitions 908
Summary 909

19. Animation. 911
Accommodating Seizure and Vestibular Disorders 912
Defining Keyframes 912
Setting Up Keyframe Animations 914
Defining Keyframe Selectors 915
Omitting from and to Values 916
Repeating Keyframe Properties 917
Animatable Properties 917
Using Nonanimatable Properties That Aren't Ignored 918
Scripting @keyframes Animations 919
Animating Elements 920
Invoking a Named Animation 920
Defining Animation Lengths 922
Declaring Animation Iterations 924
Setting an Animation Direction 925
Delaying Animations 927
Exploring Animation Events 929
Changing the Internal Timing of Animations 937
Setting the Animation Play State 947
Animation Fill Modes 948
Bringing It All Together 950
Animation, Specificity, and Precedence Order 953
Specificity and !important 953
Animation Iteration and display: none; 953
Animation and the UI Thread 953
Using the will-change Property 954
Printing Animations 955

Summary 956

20. Filters, Blending, Clipping, and Masking. 957
CSS Filters 957
Basic Filters 958
Color Filtering 960
Brightness, Contrast, and Saturation 961
SVG Filters 962
Compositing and Blending 963
Blending Elements 964
Blending Backgrounds 972
Blending in Isolation 974
Containing Elements 976
Float Shapes 979
Shaping with Image Transparency 981
Using Inset Shapes 982
Adding a Shape Margin 992
Clipping and Masking 994
Clipping 994
Clip Shapes 995
Clip Boxes 997
Clipping with SVG Paths 999
Masks 1000
Defining a Mask 1000
Changing the Mask's Mode 1003
Sizing and Repeating Masks 1005
Positioning Masks 1008
Clipping and Compositing Masks 1009
Bringing It All Together 1012
Setting Mask Types 1013
Border-Image Masking 1014
Object Fitting and Positioning 1020
Summary 1023

21. CSS At-Rules. 1025
Media Queries 1025
Basic Media Queries 1025
Complex Media Queries 1028
Special Value Types 1030
Keyword Media Features 1031
Forced Colors, Contrast, and Display Mode 1035
Ranged Media Features 1038

Deprecated Media Features 1040
Responsive Styling 1041
Paged Media 1043
Print Styles 1043
Differences Between Screen and Print 1043
Page Size 1045
Page Margins and Padding 1048
Named Page Types 1048
Page Breaking 1049
Orphans and Widows 1052
Page-Breaking Behavior 1054
Repeated Elements 1056
Elements Outside the Page 1057
Container Queries 1057
Defining Container Types 1058
Defining Container Names 1060
Using Container Shorthand 1061
Using Container At-Rules 1062
Defining Container Query Features 1065
Setting Container Length Units 1066
Feature Queries (@supports) 1067
Other At-Rules 1070
Defining a Character Set for a Stylesheet 1070
Defining a Namespace for Selectors 1071
Summary 1071

A. Additional Resources. 1073

Index. 1075

Preface

If you are a web designer or document author interested in sophisticated page styling, improved accessibility, and saving time and effort, this book is for you. All you really need to know before starting the book is HTML 4.0. The better you know HTML, the better prepared you'll be, but it is not a requirement. You will need to know very little else to follow this book.

This fifth edition of the book was finished at the end of 2022 and does its best to reflect the state of CSS at that time. Anything covered in detail either had wide browser support at the time of writing or was known to be coming soon after publication. CSS features that were still being developed or were known to have support dropping soon are not covered here.

Conventions Used in This Book

The following typographical conventions are used in this book (but make sure to read through "Value Syntax Conventions" on page xx to see how some of these are modified):

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Value Syntax Conventions

Throughout this book, boxes explain a given CSS property's details, including which values are permitted. This content has been reproduced practically verbatim from the CSS specifications, but some explanation of the syntax is in order.

The allowed values for each property are listed with a syntax like the following:

Value: <*family-name*>#

Value: <*url*> || <*color*>

Value: <*url*>? <*color*> [/ <*color*>]?

Value: [<*length*> | `thick` | `thin`]{1,4}

Value: [<*background*>,]* <*background-color*>

Any italicized words between < and > give a type of value, or a reference to another property's values. For example, the property `font` accepts values that originally belong to the property `font-family`. This is denoted by using the text <*font-family*>. Similarly, if a value type like a color is permitted, it will be represented using <*color*>.

Any words presented in `constant width` are keywords that must appear literally, without quotes. The forward slash (`/`) and the comma (`,`) must also be used literally.

Components of a value definition can be combined in numerous ways:

- Two or more keywords strung together with only space separating them means that all of them must occur in the given order. For example, `help me` would mean that the property must use those keywords in that order.
- If a vertical bar separates alternatives (*X* | *Y*), any one of them must occur, but only one. Given [*X* | *Y* | *Z*] any one of *X*, *Y*, or *Z* is permitted.

- A vertical double bar (*X* || *Y*) means that *X*, *Y*, or both must occur, but they may appear in any order. Thus: *X*, *Y*, *X Y*, and *Y X* are all valid interpretations.
- A double ampersand (*X* && *Y*) means both *X* and *Y* must occur, though they may appear in any order. Thus: *X Y* or *Y X* are both valid interpretations.
- Brackets ([…]) are for grouping things together. Thus `[please || help || me] do this` means that the words `please`, `help`, and `me` can appear in any order, though each appear only once. The words `do this` must always appear, in that order. Here are some examples: `please help me do this`, `help me please do this`, `me please help do this`.

Every component or bracketed group may (or may not) be followed by one of these modifiers:

- An asterisk (*) indicates that the preceding value or bracketed group is repeated zero or more times. Thus, `bucket*` means that the word `bucket` can be used any number of times, including zero. There is no upper limit defined on the number of times it can be used.
- A plus (+) indicates that the preceding value or bracketed group is repeated one or more times. Thus, `mop+` means that the word `mop` must be used at least once, and potentially many more times.
- A hash sign (#), formally called an octothorpe, indicates that the preceding value or bracketed group is repeated one or more times, separated by commas as needed. Thus, `floor#` can be `floor` or `floor, floor, floor`, and so on. This is most often used in conjunction with bracketed groups or value types.
- A question mark (?) indicates that the preceding value or bracketed group is optional. For example, `[pine tree]?` means that the words `pine tree` need not be used (although they must appear in that order if they are used).
- An exclamation point (!) indicates that the preceding value or bracketed group is required, and thus must result in at least one value, even if the syntax would seem to indicate otherwise. For example, `[ what? is? happening? ]!` must be at least one of the three terms marked optional.
- A pair of numbers in curly braces ({*M*,*N*}) indicates that the preceding value or bracketed group is repeated at least *M* and at most *N* times. For example, `ha{1,3}` means that there can be one, two, or three instances of the word `ha`.

The following are some examples:

`give || me || liberty`
: At least one of the three words must be used, and they can be used in any order. For example, `give liberty`, `give me`, `liberty me give`, and `give me liberty` are all valid interpretations.

`[ I | am ]? the || walrus`
: Either the word `I` or `am` may be used, but not both, and use of either is optional. In addition, either `the` or `walrus`, or both, must follow in any order. Thus you could construct `I the walrus`, `am walrus the`, `am the`, `I walrus`, `walrus the`, and so forth.

`koo+ ka-choo`
: One or more instances of `koo` must be followed by `ka-choo`. Therefore `koo koo ka-choo`, `koo koo koo ka-choo`, and `koo ka-choo` are all legal. The number of `koo`s is potentially infinite, although there are bound to be implementation-specific limits.

`I really{1,4}? [ love | hate ] [ Microsoft | Firefox | Opera | Safari | Chrome ]`
: The all-purpose web designer's opinion expresser. This can be interpreted as `I love Firefox`, `I really love Microsoft`, and similar expressions. Anywhere from zero to four `really`s may be used, though they may *not* be separated by commas. You also get to pick between `love` and `hate`, which really seems like some sort of metaphor.

`It's a [ mad ]# world`
: This gives the opportunity to put in as many comma-separated `mad`s as possible, with a minimum of one `mad`. If there is only one `mad`, no comma is added. Thus: `It's a mad world` and `It's a mad, mad, mad, mad, mad world` are both valid results.

`[ [ Alpha || Baker || Cray ], ]{2,3} and Delphi`
: Two to three of `Alpha`, `Baker`, and `Delta` must be followed by `and Delphi`. One possible result would be `Cray, Alpha, and Delphi`. In this case, the comma is placed because of its position within the nested bracket groups. (Some older versions of CSS enforced comma separation this way, instead of via the `#` modifier.)

Using Code Examples

Whenever you come across an icon that looks like ⏵, it means there is an associated code example. Live examples are available at *https://meyerweb.github.io/csstdg5figs*. If you are reading this book on a device with an internet connection, you can click the ⏵ icon to go directly to a live version of the code example referenced.

Supplemental material—in the form of the HTML, CSS, and image files that were used to produce nearly all of the figures in this book—is available for download at *https://github.com/meyerweb/csstdg5figs*. Please be sure to read the repository's *README.md* file for any notes regarding the contents of the repository.

If you have a technical question or a problem using the code examples, please send an email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*CSS: The Definitive Guide* by Eric A. Meyer and Estelle Weyl (O'Reilly). Copyright 2023 Eric A. Meyer and Estelle Weyl, 978-1-098-11761-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/css-the-definitive-guide-5e*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Eric Meyer

First, I want to thank all the technical reviewers of this edition, who lent their time and expertise to the arduous task of finding out all the places I'd been wrong, and for less recompense than they deserved. Alphabetically, by family name: Ire Aderinokun, Rachel Andrew, Adam Argyle, Amelia Bellamy-Royds, Chen Hui Jing, Stephanie Eckles, Eva Ferreira, Mandy Michael, Schalk Neethling, Jason Pamental, Janelle Pizarro, Eric Portis, Miriam Suzanne, Lea Verou, and Dan Wilson. Any errors are my fault, not theirs.

Thank you as well to all the technical reviewers of past editions, who are too many to name here, and all the people who have helped me understand various bits and bobs of CSS over the years, who are also far too many to name here. If you ever explained some CSS to me, please write your name in the following blank: _____, you have my gratitude.

Thank you to all the members of the CSS Working Group, past and present, who have shepherded an amazing language to astonishing heights…even as your work means we'll face a real production dilemma for the next edition of this book, which is already pushing the limits of what printing technology can reasonably manage.

Thank you to all the people who keep the Mozilla Developer Network (MDN) up and running as well as up-to-date.

Special thanks to all the fine people at Open Web Docs for your work on MDN, and for asking me to serve as a member of your steering committee.

To my coauthor, Estelle, thank you for all your contributions, expertise, and pushes to do what was needed.

To all the assorted friends, colleagues, coworkers, acquaintances, and passersby who have allowed space for my odd enthusiasms and strange demeanor, thank you for your understanding, patience, and kindness.

And as ever, my boundless gratitude to my family—my wife, Kat, and my children, Carolyn, Rebecca *z"l*, and Joshua. You are the home that shelters me, the suns in my sky, and the stars by which I steer. Thank you for everything you have taught me.

Cleveland Heights, OH
December 4, 2022

Estelle Weyl

I would like to acknowledge everyone who has worked to make CSS what it is today and all those who have helped improve diversity and inclusion in tech.

Many people work tirelessly with browser vendors and developers in writing the CSS specifications. Without the members of the CSS Working Group—past, current, and future—we would have no specifications, no standards, and no cross-browser compatibility. I am in awe of the thought process that goes into every CSS property and value added to, and omitted from, the specification. People like Tab Atkins, Elika Etimad, Dave Baron, Léonie Watson, and Greg Whitworth not only work on the specification, but also take their time to answer questions and explain nuances to the broader CSS public, notably me.

I also acknowledge all those who, whether they participate in the CSS Working Group or not, dive deep into CSS features and help translate the spec for the rest of us, including Sarah Drasner, Val Head, Sara Souidan, Chris Coyier, Jen Simmons, and Rachel Andrew. In addition, I thank the people who create tools that make all CSS developers' lives easier, especially Alexis Deveria for creating and maintaining the Can I Use tool.

I also appreciate all those who have contributed their time and effort to improve diversity and inclusion in all sectors of the developer community. Yes, CSS is awesome. But it's important to work with great people in a great community.

When I attended my first tech conference in 2007, the lineup was 93% male and 100% white. The audience had slightly less gender diversity and only slightly more ethnic diversity. I had picked that conference because the lineup was more diverse than most: it actually had a woman on it. Looking around the room, I knew things needed to change, and I realized it was something I needed to do. What I didn't realize then was how many unsung heroes I would meet over the next 10 years working for diversity and inclusion in all areas of the tech sector and life in general.

There are too many people—who work tirelessly, quietly, and often with little or no recognition—to name them all, but I would like to highlight some. I cannot express how much of a positive impact people like Erica Stanley of Women Who Code Atlanta, Carina Zona of Callback Women, and Jenn Mei Wu of Oakland Maker Space have had. Groups like The Last Mile, Black Girls Code, Girls Incorporated, Sisters Code, and so many others inspired me to create a Feeding the Diversity Pipeline list to help ensure that the path to a career in web development is not only for those with privilege.

Thank you to all of you. Thank you to everyone. Because of your efforts, more has been done than I ever could have imagined sitting in that conference 10 years ago.

San Francisco, CA
February 14, 2023

CHAPTER 1

CSS Fundamentals

Cascading Style Sheets (*CSS*), a powerful programming language that transforms the presentation of a document or a collection of documents, has spread to nearly every corner of the web as well as many ostensibly nonweb environments. For example, embedded-device displays often use CSS to style their user interfaces, many RSS clients let you apply CSS to feeds and feed entries, and some instant message clients use CSS to format chat windows. Aspects of CSS can be found in the syntax used by JavaScript (JS) frameworks and even in JS itself. It's everywhere!

A Brief History of (Web) Style

CSS was first proposed in 1994, just as the web was beginning to really catch on. At the time, browsers gave all sorts of styling power to the user—the presentation preferences in NCSA Mosaic, for example, permitted the user to define each element's font family, size, and color. None of this was available to document authors; all they could do was mark a piece of content as a paragraph, as a heading of some level, as preformatted text, or one of a dozen other element types. If a user configured their browser to make all level-one headings tiny and pink and all level-six headings huge and red, well, that was their lookout.

It was into this milieu that CSS was introduced. Its goal was to provide a simple, declarative styling language that was flexible for web page authors and, most importantly, provided styling power to authors and users alike. By means of the *cascade*, these styles could be combined and prioritized so that both site authors and readers had a say—though readers always had the last say.

Work quickly advanced, and by late 1996, CSS1 was finished. While the newly established CSS Working Group moved forward with CSS2, browsers struggled to implement CSS1 in an interoperable way. Although each piece of CSS was fairly simple on its own, the combination of those pieces created some surprisingly complex behaviors. Unfortunate missteps also occurred, such as the infamous discrepancy in box model implementations. These

problems threatened to derail CSS altogether, but fortunately some clever proposals were implemented, and browsers began to harmonize. Within a few years, thanks to increasing interoperability and high-profile developments such as the CSS-based redesign of *Wired* magazine and the CSS Zen Garden, CSS began to catch on.

Before all that happened, though, the CSS Working Group had finalized the CSS2 specification in early 1998. Once CSS2 was finished, work immediately began on CSS3, as well as a clarified version of CSS2 called CSS2.1. In keeping with the spirit of the times, what was initially coined *CSS3* was constructed as a series of (theoretically) standalone modules instead of a single monolithic specification. This approach reflected the then-active XHTML specification, which was split into modules for similar reasons.

The rationale for modularizing CSS was that each module could be worked on at its own pace, and particularly critical (or popular) modules could be advanced along the World Wide Web Consortium's (W3C's) progress track without being held up by others. Indeed, this has turned out to be the case. By early 2012, three CSS Level 3 modules (along with CSS1 and CSS 2.1) had reached full Recommendation status—CSS Color Level 3, CSS Namespaces, and Selectors Level 3. At that same time, seven modules were at Candidate Recommendation status, and several dozen others were in various stages of Working Draft-ness. Under the old approach, colors, selectors, and namespaces would have had to wait for every other part of the specification to be done or cut before they could be part of a completed specification. Thanks to modularization, they didn't have to wait.

So while we can't really point to a single tome and say, "This is CSS," we can talk of features by the module name under which they are introduced. The flexibility permitted by modules more than makes up for the semantic awkwardness they sometimes create. (If you want something approximating a single monolithic specification, the CSS Working Group publishes yearly "Snapshot" documents.)

With that established, we're ready to start understanding CSS. Let's start by covering the basics of what goes inside a stylesheet.

Stylesheet Contents

Inside a stylesheet, you'll find a number of *rules* that look a little something like this:

```
h1 {color: maroon;}
body {background: yellow;}
```

Styles such as these make up the bulk of any stylesheet—simple or complex, short or long. But which parts are which, and what do they represent?

Rule Structure

To illustrate the concept of rules in more detail, let's break down the structure.

Each *rule* has two fundamental parts: the selector and the declaration block. The declaration block is composed of one or more declarations, and each declaration is a pairing of a

property and a value. Every stylesheet is made up of a series of these rules. Figure 1-1 shows the parts of a rule.

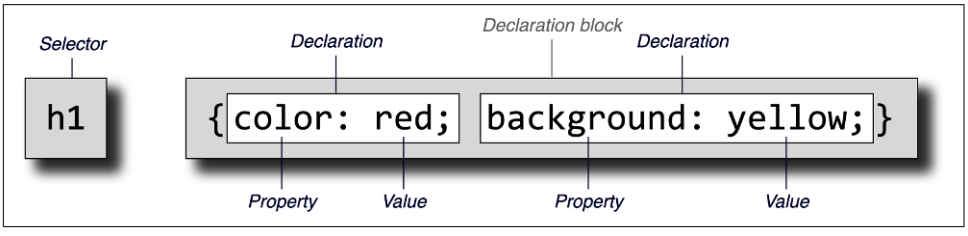


Figure 1-1. The structure of a rule

The *selector*, shown on the left side of the rule, defines which piece of the document will be selected for styling. In Figure 1-1, `<h1>` (heading level 1) elements are selected. If the selector were `p`, then all `<p>` (paragraph) elements would be selected.

The right side of the rule contains the *declaration block*, which is made up of one or more declarations. Each *declaration* is a combination of a CSS *property* and a *value* of that property. In Figure 1-1, the declaration block contains two declarations. The first states that this rule will cause parts of the document to have a `color` of `red`, and the second states that part of the document will have a `background` of `yellow`. So, all of the `<h1>` elements in the document (defined by the selector) will be styled in red text with a yellow background.

Vendor Prefixing

Sometimes you'll see pieces of CSS with hyphens and labels in front of them, like this: `-o-border-image`. These *vendor prefixes* were a way for browser vendors to mark properties, values, or other bits of CSS as being experimental or proprietary (or both). As of early 2023, a few vendor prefixes are in the wild, with the most common shown in Table 1-1.

Table 1-1. Some common vendor prefixes

Prefix	Vendor
`-epub-`	International Digital Publishing Forum ePub format
`-moz-`	Gecko-based browsers (e.g., Mozilla Firefox)
`-ms-`	Microsoft Internet Explorer
`-o-`	Opera-based browsers
`-webkit-`	WebKit-based browsers (e.g., Apple Safari and Google Chrome)

As Table 1-1 indicates, the generally accepted format of a vendor prefix is a hyphen, a label, and a hyphen, although a few prefixes erroneously omit the first hyphen.

The uses and abuses of vendor prefixes are long, tortuous, and beyond the scope of this book. Suffice to say that they started out as a way for vendors to test out new features, thus

helping speed interoperability without worrying about being locked into legacy behaviors that were incompatible with other browsers. This avoided a whole class of problems that nearly strangled CSS in its infancy. Unfortunately, prefixed properties were then publicly deployed by web authors and ended up causing a whole new class of problems.

As of early 2023, vendor-prefixed CSS features are nearly nonexistent, with old prefixed properties and values being slowly but steadily removed from browser implementations. You'll quite likely never write prefixed CSS, but you may encounter it in the wild or inherit it in a legacy codebase. Here's an example:

```
-webkit-transform-origin: 0 0;
-moz-transform-origin: 0 0;
-o-transform-origin: 0 0;
transform-origin: 0 0;
```

That's saying the same thing four times: once each for the WebKit, Gecko (Firefox), and Opera browser lines, and then finally the CSS-standard way. Again, this is no longer necessary. We're including it here only to give you an idea of what it might look like, should you come across this in the future.

Whitespace Handling

CSS is basically insensitive to whitespace between rules, and largely insensitive to whitespace within rules, although a few exceptions exist.

In general, CSS treats whitespace just like HTML does: any sequence of whitespace characters is collapsed to a single space for parsing purposes. Thus, you can format this hypothetical `rainbow` rule in the following ways,

```
rainbow: infrared  red  orange  yellow  green  blue  indigo  violet  ultraviolet;

rainbow:
   infrared  red  orange  yellow  green  blue  indigo  violet  ultraviolet;

rainbow:
   infrared
   red
   orange
   yellow
   green
   blue
   indigo
   violet
   ultraviolet
   ;
```

as well as any other separation patterns you can think up. The only restriction is that the separating characters be whitespace: an empty space, a tab, or a newline, alone or in combination, as many as you like.

Similarly, you can format series of rules with whitespace in any fashion you like. These are just five examples out of an effectively infinite number of possibilities:

```
html{color:black;}
body {background: white;}
p {
  color: gray;}
h2 {
     color : silver ;
   }
ol
   {
      color
         :
      silver
         ;
}
```

As you can see from the first rule, whitespace can be largely omitted. Indeed, this is usually the case with *minified* CSS, which is CSS that's had every last possible bit of extraneous whitespace removed, usually by an automated server-side script of some sort. The rules after the first two use progressively more extravagant amounts of whitespace until, in the last rule, pretty much everything that can be separated onto its own line has been.

All of these approaches are valid, so you should pick the formatting that makes the most sense—that is, is easiest to read—in your eyes, and stick with it.

CSS Comments

CSS does allow for comments. These are very similar to C/C++ comments in that they are surrounded by /* and */:

```
/* This is a CSS comment */
```

Comments can span multiple lines, just as in C++:

```
/* This is a CSS comment, and it
can be several lines long without
any problem whatsoever. */
```

It's important to remember that CSS comments cannot be nested. So, for example, this would *not* be correct:

```
/* This is a comment, in which we find
 another comment, which is WRONG
   /* Another comment */
 and back to the first comment, which is not a comment.*/
```



One way to create "nested" comments accidentally is to temporarily comment out a large block of a stylesheet that already contains a comment. Since CSS doesn't permit nested comments, the "outside" comment will end where the "inside" comment ends.

Unfortunately, there is no "rest of the line" comment pattern such as `//` or `#` (the latter of which is reserved for ID selectors anyway). The only comment pattern in CSS is `/* */`. Therefore, if you wish to place comments on the same line as markup, you need to be careful about how you place them. For example, this is the correct way to do it:

```
h1 {color: gray;}   /* This CSS comment is several lines */
h2 {color: silver;} /* long, but since it is alongside */
p {color: white;}   /* actual styles, each line needs to */
pre {color: gray;}  /* be wrapped in comment markers. */
```

Given this example, if each line isn't marked off, most of the stylesheet will become part of the comment and thus will not work:

```
h1 {color: gray;}   /* This CSS comment is several lines
h2 {color: silver;}  long, but since it is not wrapped
p {color: white;}    in comment markers, the last three
pre {color: gray;}   styles are part of the comment. */
```

In this example, only the first rule (`h1 {color: gray;}`) will be applied to the document. The rest of the rules, as part of the comment, are ignored by the browser's rendering engine.



CSS comments are treated by the CSS parser as if they do not exist at all, and so do not count as whitespace for parsing purposes. This means you can put them into the middle of rules—even right inside declarations!

Markup

There is no markup in stylesheets. This might seem obvious, but you'd be surprised. The one exception is HTML comment markup, which is permitted inside `<style>` elements for historical reasons:

```
<style><!--
h1 {color: maroon;}
body {background: yellow;}
--></style>
```

That's it, and even that isn't recommended anymore; the browsers that needed it have faded into near oblivion.

Speaking of markup, it's time to take a very slight detour to talk about the elements that our CSS will be used to style, and how those can be affected by CSS in the most fundamental ways.

Elements

Elements are the basis of document structure. In HTML, the most common elements are easily recognizable, such as `<p>`, `<table>`, `<span>`, `<a>`, and `<article>`. Every single element in a document plays a part in its presentation.

Replaced and Nonreplaced Elements

Although CSS depends on elements, not all elements are created equal. For example, images and paragraphs are not the same type of element. In CSS, elements generally take two forms: replaced and nonreplaced.

Replaced elements

Replaced elements are used to indicate content that is to be replaced by something not directly represented in the document. Probably the most familiar HTML example is the `<img>` element, which is replaced by an image file external to the document itself. In fact, `<img>` has no actual content, as you can see in this simple example:

```
<img src="howdy.gif" alt="Hello, friend!">
```

This markup fragment contains only an element name and an attribute. The element presents nothing unless you point it to external content (in this case, an image file whose location is given by the `src` attribute). If you point to a valid image file, the image will be placed in the document. If not, the browser will either display nothing or will show a "broken image" placeholder.

Similarly, the `input` element can also be replaced—by a radio button, checkbox, text input box, or other, depending on its type.

Nonreplaced elements

The majority of HTML elements are *nonreplaced elements*. Their content is presented by the user agent (generally a browser) inside a box generated by the element itself. For example, `<span>hi there</span>` is a nonreplaced element, and the text "hi there" will be displayed by the user agent. This is true of paragraphs, headings, table cells, lists, and almost everything else in HTML.

Element Display Roles

CSS has two basic display roles: *block formatting context* and *inline formatting context*. Many more display types exist, but these are the most basic, and the types to which most, if not all, other display types refer. The block and inline contexts will be familiar to authors who have spent time with HTML markup and its display in web browsers. The display roles are illustrated in Figure 1-2.

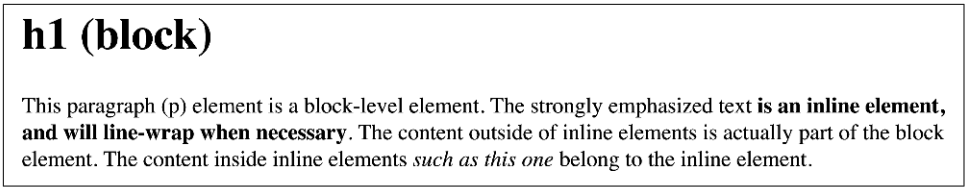


Figure 1-2. Block- and inline-level elements in an HTML document

Block-level elements

By default, *block-level elements* generate an element box that (by default) fills its parent element's content area and cannot have other elements at its sides. In other words, it generates "breaks" before and after the element box. The most familiar block elements from HTML are `<p>` and `<div>`. Replaced elements can be block-level elements, but usually they are not.

In CSS, this is referred to as an element generating a *block formatting context*. It also means that the element generates a block *outer display type*. The parts inside the element may have different display types.

Inline-level elements

By default, *inline-level elements* generate an element box within a line of text and do not break up the flow of that line. The best inline element example is the `<a>` element in HTML. Other candidates are `<strong>` and `<em>`. These elements do not generate a "break" before or after themselves, so they can appear within the content of another element without disrupting its display.

In CSS, this is referred to as an element generating an *inline formatting context*. It also means that the element generates an *inline outer display* type. The parts inside the element may have different display types. (In CSS, there is no restriction on how display roles can be nested within each other.)

To see how this works, let's consider the CSS property `display`.

display

Values	[*<display-outside>* ‖ *<display-inside>*] | *<display-listitem>* | *<display-internal>* | *<display-box>* | *<display-legacy>*
Definitions	See below
Initial value	`inline`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

```
<display-outside>
    block | inline | run-in

<display-inside>
    flow | flow-root | table | flex | grid | ruby

<display-listitem>
    list-item && <display-outside>? && [ flow | flow-root ]?

<display-internal>
    table-row-group | table-header-group | table-footer-group | table-row |
    table-cell | table-column-group | table-column | table-caption |
    ruby-base | ruby-text | ruby-base-container | ruby-text-container

<display-box>
    contents | none

<display-legacy>
    inline-block | inline-list-item | inline-table | inline-flex | inline-grid
```

You may have noticed that there are a *lot* of values here, only two of which we've mentioned: `block` and `inline`. Most of these values are dealt with elsewhere in the book; for example, `grid` and `inline-grid` are covered in Chapter 12, and the table-related values are all covered in Chapter 13.

For now, let's concentrate on `block` and `inline`. Consider the following markup:

```
<body>
<p>This is a paragraph with <em>an inline element</em> within it.</p>
</body>
```

Here we have two elements (`<body>` and `<p>`) that are generating block formatting contexts, and one element (`<em>`) with an inline formatting context. According to the HTML specification, `<em>` can descend from `<p>`, but the reverse is not true. Typically, the HTML hierarchy works out so that inlines descend from blocks, but not the other way around.

CSS, on the other hand, has no such restrictions. You can leave the markup as it is but change the display roles of the two elements like this:

```
p {display: inline;}
em {display: block;}
```

This causes the elements to generate a block box inside an inline box. This is perfectly legal and violates no part of CSS.

While changing the display roles of elements can be useful in HTML documents, it becomes downright critical for XML documents. An XML document is unlikely to have any inherent display roles, so it's up to the author to define them. For example, you might wonder how to lay out the following snippet of XML:

```
<book>
 <maintitle>The Victorian Internet</maintitle>
 <subtitle>The Remarkable Story of the Telegraph and the Nineteenth Century's
   On-Line Pioneers</subtitle>
 <author>Tom Standage</author>
 <publisher>Bloomsbury Pub Plc USA</publisher>
 <pubdate>February 25, 2014</pubdate>
 <isbn type="isbn-13">9781620405925</isbn>
 <isbn type="isbn-10">162040592X</isbn>
</book>
```

Since the default value of `display` is `inline`, the content would be rendered as inline text by default, as illustrated in Figure 1-3. This isn't a terribly useful display.

The Victorian Internet The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line Pioneers Tom Standage Bloomsbury Pub Plc USA February 25, 2014 9781620405925 162040592X

Figure 1-3. Default display of an XML document

You can define the basics of the layout with `display`:

```
book, maintitle, subtitle, author, isbn {display: block;}
publisher, pubdate {display: inline;}
```

We've now set five of the seven elements to be block and two to be inline. This means each of the block elements will generate its own block formatting context, and the two inlines will generate their own inline formatting contexts.

We could take the preceding rules as a starting point, add a few other styles for greater visual impact, and get the result shown in Figure 1-4.

The Victorian Internet

The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line Pioneers

Tom Standage
Bloomsbury Pub Plc USA (February 25, 2014)
ISBN-13 9781620405925
ISBN-10 162040592X

Figure 1-4. Styled display of an XML document

That said, before learning how to write CSS in detail, we need to look at how to associate CSS with a document. After all, without tying the two together, there's no way for the CSS to affect the document. We'll explore this in an HTML setting since it's the most familiar.

Bringing CSS and HTML Together

We've mentioned that HTML documents have an inherent structure, and that's a point worth repeating. In fact, that's part of the problem with web pages of old: too many of us forgot that documents are supposed to have an internal structure, which is altogether different from a visual structure. In our rush to create the coolest-looking pages on the web, we bent, warped, and generally ignored the idea that pages should contain information with some structural meaning.

That structure is an inherent part of the relationship between HTML and CSS; without it, there couldn't be a relationship at all. To understand it better, let's look at an example HTML document and break it down by pieces:

```
<!DOCTYPE html>
<html lang="en-us">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Eric's World of Waffles</title>
  <link rel="stylesheet" media="screen, print" href="sheet1.css">
  <style>
    /* These are my styles! Yay! */
    @import url(sheet2.css);
  </style>
</head>
<body>
  <h1>Waffles!</h1>
  <p style="color: gray;">The most wonderful of all breakfast foods is
  the waffle—a ridged and cratered slab of home-cooked, fluffy goodness
  that makes every child's heart soar with joy. And they're so easy to make!
  Just a simple waffle-maker and some batter, and you're ready for a morning
  of aromatic ecstasy!
  </p>
</body>
</html>
```

Figure 1-5 shows the result of this markup and the applied styles.

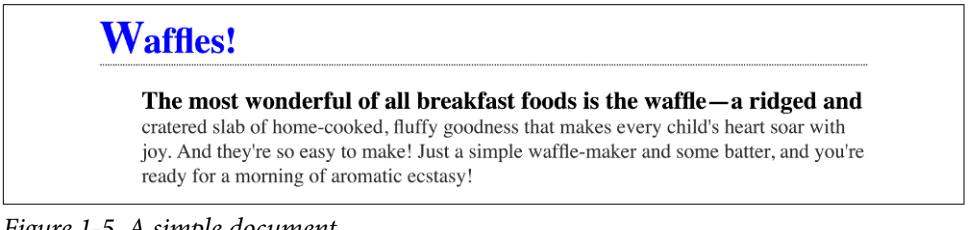


Figure 1-5. A simple document

Now, let's examine the various ways this document connects to CSS.

The <link> Tag

First, consider the use of the `<link>` tag:

```
<link rel="stylesheet" href="sheet1.css" media="screen, print">
```

The `<link>` tag's basic purpose is to allow HTML authors to associate other documents with the document containing the `<link>` tag. CSS uses it to link stylesheets to the document.

These stylesheets, which are not part of the HTML document but are still used by it, are referred to as *external stylesheets*. This is because they're stylesheets that are external to the HTML document. (Go figure.)

To successfully load an external stylesheet, `<link>` should be placed inside the `<head>` element, though it can also appear inside the `<body>` element. This will cause the web browser to locate and load the stylesheet and use whatever styles it contains to render the HTML document; Figure 1-6 depicts the stylesheet called *sheet1.css* being linked to the document.

Also shown in Figure 1-6 is the loading of the external *sheet2.css* via an `@import` declaration. Imports must be placed at the beginning of the stylesheet that contains them.

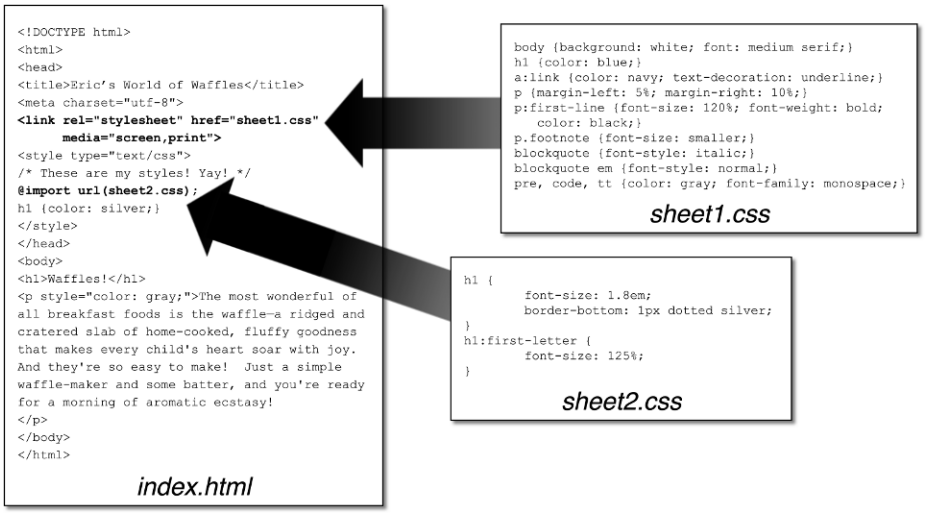


Figure 1-6. A representation of how external stylesheets are applied to documents

And what is the format of an external stylesheet? It's a list of rules, just like those you saw in the previous section and in the example HTML document; but in this case, the rules are saved into their own file. Just remember that no HTML or any other markup language can be included in the stylesheet—only style rules. Here are the contents of an external stylesheet:

```
h1 {color: red;}
h2 {color: maroon; background-color: white;}
h3 {color: white; background-color: black;
  font: medium Helvetica;}
```

That's all there is to it—no HTML markup or comments at all, just plain-and-simple style declarations. These are saved into a plain-text file and are usually given an extension of *.css*, as in *sheet1.css*.



An external stylesheet cannot contain any document markup at all, only CSS rules and CSS comments. The presence of markup in an external stylesheet can cause some or all of it to be ignored.

Attributes

For the rest of the `<link>` tag, the attributes and values are fairly straightforward. The `rel` attribute stands for *relation*, and in this case, the relation is `stylesheet`. Note that the `rel` attribute is *required*. CSS has an optional `type` attribute whose default value is `text/css`, so you can include `type="text/css"` or leave it out, whichever you prefer.

These attribute values describe the relationship and type of data that will be loaded using the `<link>` tag. That way, the web browser knows that the stylesheet is a CSS stylesheet, a fact that will determine how the browser will deal with the data it imports. (Other style languages may be used in the future. In such a future, if you are using a different style language, the `type` attribute will need to be declared.)

Next, we find the `href` attribute. The value of this attribute is the URL of your stylesheet. This URL can be either absolute or relative—that is, either relative to the URL of the document containing the URL, or else a complete URL that points to a unique location on the web. In our example, the URL is relative. It could have been something absolute, like *http://example.com/sheet1.css*.

Finally, we have a `media` attribute. The value of this attribute is one or more *media descriptors*, which are rules regarding media types and the features of those media, with each rule separated by a comma. Thus, for example, you can use a linked stylesheet in both screen and print media:

```
<link rel="stylesheet" href="visual-sheet.css" media="screen, print">
```

Media descriptors can get quite complicated and are explained in detail in Chapter 21. For now, we'll stick with the basic media types shown. The default value is `all`, which means the CSS will be applied in all media.

Note that more than one linked stylesheet can be associated with a document. In these cases, only those `<link>` tags with a `rel` of `stylesheet` will be used in the initial display of the document. Thus, if you wanted to link two stylesheets named *basic.css* and *splash.css*, it would look like this:

```
<link rel="stylesheet" href="basic.css">
<link rel="stylesheet" href="splash.css">
```

This will cause the browser to load both stylesheets, combine the rules from each, and apply them all to the document in all media types (because the `media` attribute is omitted, its default value `all` is used). For example:

```
<link rel="stylesheet" href="basic.css">
<link rel="stylesheet" href="splash.css">

<p class="a1">This paragraph will be gray only if styles from the
stylesheet 'basic.css' are applied.</p>
<p class="b1">This paragraph will be gray only if styles from the
stylesheet 'splash.css' are applied.</p>
```

The one attribute that isn't in this example markup, but could be, is `title`. This attribute is not often used but could become important in the future and, if used improperly, can have unexpected effects. Why? We'll explore that in the next section.

Alternate stylesheets

It's possible to define *alternate stylesheets* that users can select in some browsers. These are defined by making the value of the `rel` attribute `alternate stylesheet`, and they are used in document presentation only if selected by the user.

Should a browser be able to use alternate stylesheets, it will use the values of the `<link>` element's `title` attributes to generate a list of style alternatives. So you could write the following:

```
<link rel="stylesheet" href="sheet1.css" title="Default">
<link rel="alternate stylesheet" href="bigtext.css" title="Big Text">
<link rel="alternate stylesheet" href="zany.css" title="Crazy colors!">
```

Users could then pick the style they want to use, and the browser would switch from the first one, labeled `Default` in this case, to whichever the user picked. Figure 1-7 shows one way in which this selection mechanism might be accomplished (and in fact was, early in the resurgence of CSS).

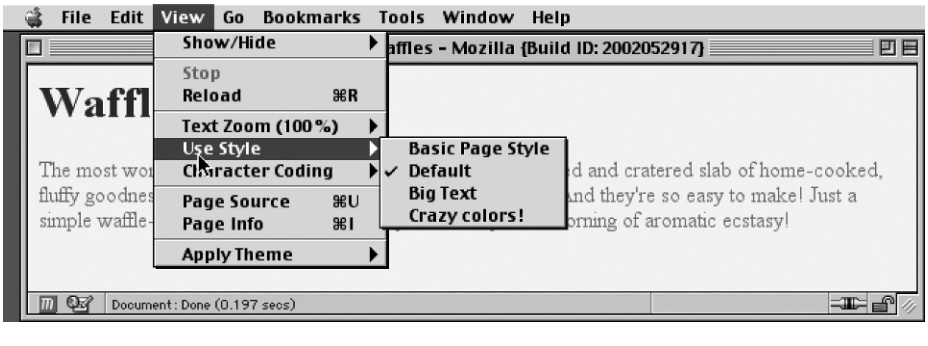


Figure 1-7. A browser offering alternate stylesheet selection



As of early 2023, alternate stylesheets are supported in most Gecko-based browsers like Firefox. The Chromium and WebKit families do not support selecting alternate stylesheets. Compare this to the build date of the browser shown in Figure 1-7, which is late 2002.

It's also possible to group alternate stylesheets together by giving them the same `title` value. Thus, you make it possible for the user to pick a different presentation for your site in both screen and print media:

```
<link rel="stylesheet"
   href="sheet1.css" title="Default" media="screen">
<link rel="stylesheet"
   href="print-sheet1.css" title="Default" media="print">
<link rel="alternate stylesheet"
   href="bigtext.css" title="Big Text" media="screen">
<link rel="alternate stylesheet"
   href="print-bigtext.css" title="Big Text" media="print">
```

If a user selects Big Text from the alternate stylesheet selection mechanism in a conforming user agent, *bigtext.css* will be used to style the document in the screen medium, and *print-bigtext.css* will be used in the print medium. Neither *sheet1.css* nor *print-sheet1.css* will be used in any medium.

Why is that? Because if you give a `<link>` with a `rel` of `stylesheet` a title, you are designating that stylesheet as a *preferred stylesheet*. Its use is preferred to alternate stylesheets, and it will be used when the document is first displayed. Once you select an alternate stylesheet, however, the preferred stylesheet will *not* be used.

Furthermore, if you designate a number of stylesheets as preferred, all but one of them will be ignored. Consider the following code example:

```
<link rel="stylesheet"
   href="sheet1.css" title="Default Layout">
<link rel="stylesheet"
   href="sheet2.css" title="Default Text Sizes">
<link rel="stylesheet"
   href="sheet3.css" title="Default Colors">
```

All three `<link>` elements now refer to preferred stylesheets, thanks to the presence of a `title` attribute on all three, but only one of them will actually be used in that manner. The other two will be ignored completely. Which two? There's no way to be certain, as HTML doesn't provide a method of determining which preferred stylesheets should be ignored and which should be used.

If you don't give a stylesheet a title, it becomes a *persistent stylesheet* and is always used in the display of the document. Often, this is exactly what an author wants, especially since alternate stylesheets are not widely supported and are almost completely unknown to users.

The <style> Element

The `<style>` element is one way to include a stylesheet, and it appears in the document itself:

```
<style>...</style>
```

The styles between the opening and closing `<style>` tags are referred to as the *document stylesheet* or the *embedded stylesheet* (because this kind of stylesheet is embedded within the document). It contains styles that apply to the document, but it can also contain multiple links to external stylesheets via the `@import` directive, discussed in the next section.

You can give `<style>` elements a `media` attribute, which functions in the same manner as it does on linked stylesheets. This, for example, will restrict an embedded stylesheet's rules to be applied in print media only:

```
<style media="print">…</style>
```

You can also label an embedded stylesheet with a `<title>` element, in the same manner and for the same reasons discussed in the previous section on alternate stylesheets.

As with the `<link>` element, the `<style>` element can use the attribute `type`; in the case of a CSS document, the correct value is `"text/css"`. The `type` attribute is optional in HTML as long as you're loading CSS, because the default value for the `type` attribute on the `<style>` element is `text/css`. It would be necessary to explicitly declare a `type` value only if you were using some other styling language, perhaps in a future where such a thing is supported. For the time being, though, the attribute remains wholly optional.

The @import Directive

Now we'll discuss the stuff that is found inside the `<style>` tag. First, we have something very similar to `<link>`, the `@import` directive:

```
@import url(sheet2.css);
```

Just like `<link>`, `@import` can be used to direct the web browser to load an external stylesheet and use its styles in the rendering of the HTML document. The only major difference is in the syntax and placement of the command. As you can see, `@import` is found inside the `<style>` element. It must be placed first, before the other CSS rules, or it won't work at all. Consider this example:

```
<style>
@import url(styles.css); /* @import comes first */
h1 {color: gray;}
</style>
```

As with `<link>`, a document can have more than one `@import` statement. Unlike `<link>`, however, the stylesheets of every `@import` directive will be loaded and used; there is no way to designate alternate stylesheets with `@import`. So, given the following markup:

```
@import url(sheet2.css);
@import url(blueworld.css);
@import url(zany.css);
```

…all three external stylesheets will be loaded, and all of their style rules will be used in the display of the document.

As with `<link>`, you can restrict imported stylesheets to one or more media by providing media descriptors after the stylesheet's URL:

```
@import url(sheet2.css) all;
@import url(blueworld.css) screen;
@import url(zany.css) screen, print;
```

As noted in "The <link> Tag" on page 12, media descriptors can get quite complicated and are explained in detail in Chapter 21.

The `@import` directive can be highly useful if you have an external stylesheet that needs to use the styles found in other external stylesheets. Since external stylesheets cannot contain any document markup, the `<link>` element can't be used—but `@import` can. Therefore, you might have an external stylesheet that contains the following:

```
@import url(http://example.org/library/layout.css);
@import url(basic-text.css);
@import url(printer.css) print;
body {color: red;}
h1 {color: blue;}
```

Well, maybe not those exact styles, but hopefully you get the idea. Note the use of both absolute and relative URLs in the previous example. Either URL form can be used, just as with `<link>`.

Note also that the `@import` directives appear at the beginning of the stylesheet, as they did in the example document. As we said previously, CSS requires the `@import` directives to come before any rules in a stylesheet, though they can be preceded by `@charset` and `@layer` declarations. An `@import` that comes after other rules (e.g., `body {color: red;}`) will be ignored by conforming user agents.



Some versions of Internet Explorer for Windows did not ignore any `@import` directive, even those that come after other rules, but all modern browsers do ignore improperly placed `@import` directives.

Another descriptor that can be added to an `@import` directive is a *cascade layer* identifier. This assigns all of the styles in the imported stylesheet to a cascade layer, which is a concept we'll explore in Chapter 4. It looks like this:

```
@import url(basic-text.css) screen layer(basic);
```

That assigns the styles from *basic-text.css* to the `basic` cascade layer. If you want to assign the styles to an unnamed layer, use `layer` without the parenthetical naming, like so:

```
@import url(basic-text.css) screen layer;
```

Note that this ability is a difference between `@import` and `<link>`, as the latter cannot be labeled with a cascade layer.

HTTP Linking

In another, far more obscure way to associate CSS with a document, you can link the two via HTTP headers.

Under Apache HTTP Server, this can be accomplished by adding a reference to the CSS file in a *.htaccess* file. For example:

```
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
```

This will cause supporting browsers to associate the referenced stylesheet with any documents served from under that *.htaccess* file. The browser will then treat it as if it were a linked stylesheet. Alternatively, and probably more efficiently, you can add an equivalent rule to the server's *httpd.conf* file:

```
<Directory /path/to/ /public/html/directory>
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
</Directory>
```

The effect is exactly the same in supporting browsers. The only difference is in where you declare the linking.

You probably noticed the use of the term "supporting browsers." As of late 2022, the widely used browsers that support HTTP linking of stylesheets are the Firefox family and Opera. That restricts this technique mostly to development environments based on one of those browsers. In such a situation, you can use HTTP linking on the test server to mark when you're on the development site as opposed to the public site. It's also an interesting way to hide styles from Chromium browsers, assuming you have a reason to do so.



Equivalents to this linking technique are used in common scripting languages such as PHP and IIS, both of which allow the author to emit HTTP headers. It's also possible to use such languages to explicitly write `link` elements into the document based on the server offering up the document. This is a more robust approach in terms of browser support: every browser supports the `link` element.

Inline Styles

If you want to just assign a few styles to one individual element, without the need for embedded or external stylesheets, you can employ the HTML attribute `style`:

```
<p style="color: gray;">The most wonderful of all breakfast foods is
the waffle—a ridged and cratered slab of home-cooked, fluffy goodness...
</p>
```

The `style` attribute can be associated with any HTML tag whatsoever, even tags found outside of `<body>` (`<head>` or `<title>`, for instance).

The syntax of a `style` attribute is fairly ordinary. In fact, it looks very much like the declarations found in the `<style>` container, except here the curly braces are replaced by double quotation marks. So `<p style="color: maroon; background: yellow;">` will set the text color to be maroon and the background to be yellow *for that paragraph only*. No other part of the document will be affected by this declaration.

Note that you can place only a declaration block, not an entire stylesheet, inside an inline `style` attribute. Therefore, you can't put an `@import` into a `style` attribute, nor can you include any complete rules. The only thing you can put into the value of a `style` attribute is what might go between the curly braces of a rule.

Use of the `style` attribute is discouraged. Many of the primary advantages of CSS—the ability to organize centralized styles that control an entire document's appearance or the appearance of all documents on a web server—are negated when you place styles into a `style` attribute. In many ways, inline styles are not much better than the ancient `<font>` tag, even if they do have a good deal more flexibility in terms of which visual effects they can apply.

Summary

With CSS, you can completely change the way elements are presented by a user agent. You can do this at a basic level with the `display` property, and in a different way by associating stylesheets with a document. The user will never know whether this is done via an external or embedded stylesheet, or even with an inline style. The real importance of external stylesheets is the way in which they allow you to put all of a site's presentation information in one place, and point all of the documents to that place. This not only makes site updates and maintenance a breeze, but also helps to save bandwidth, since all of the presentation is removed from documents.

To make the most of the power of CSS, you need to know how to associate a set of styles with the elements in a document. To fully understand how CSS can do all of this, you need a firm grasp of the way CSS selects pieces of a document for styling, which is the subject of the next few chapters.

CHAPTER 2

Selectors

One of the primary advantages of CSS is its ability to easily apply a set of styles to all elements of the same type. Unimpressed? Consider this: by editing a single line of CSS, you can change the colors of all your headings. Don't like the blue you're using? Change that one line of code, and they can all be purple, yellow, maroon, or any other color you desire.

This capability lets you, the author, focus on design and user experience rather than tedious find-and-replace operations. The next time you're in a meeting and someone wants to see headings with a different shade of green, just edit your style and hit Reload. Voilà! The results are accomplished in seconds and there for everyone to see.

Basic Style Rules

As stated, a central feature of CSS is its ability to apply certain rules to an entire set of element types in a document. For example, let's say that you want to make the text of all `<h2>` elements appear gray. Before we had CSS, you'd have to do this by inserting `<font color="gray">...</font>` tags inside all your `<h2>` elements. Applying inline styles using the `style` attribute, which is also bad practice, would require you to include `style="color: gray;"` in all your `<h2>` elements, like this:

```
<h2 style="color: gray;">This is h2 text</h2>
```

This will be a tedious process if your document contains a lot of `<h2>` elements. Worse, if you later decide that you want all those `<h2>`s to be green instead of gray, you'd have to start the manual tagging all over again. (Yes, this is really how it used to be done!)

CSS allows you to create rules that are simple to change, edit, and apply to all the text elements you define (the next section explains how these rules work). For example, you can write this rule once to make all your `<h2>` elements gray:

```
h2 {color: gray;}
```

Type Selectors

A *type selector*, previously known as an *element selector*, is most often an HTML element, but not always. For example, if a CSS file contains styles for an XML document, the type selectors might look something like this:

```
quote {color: gray;}
bib {color: red;}
booktitle {color: purple;}
myElement {color: red;}
```

In other words, the elements of the document are the node types being selected. In XML, a selector could be anything because XML allows for the creation of new markup languages that can have just about anything as an element name. If you're styling an HTML document, the selector will generally be one of the many defined HTML elements such as `<p>`, `<h3>`, `<em>`, `<a>`, or even `<html>` itself. For example:

```
html {color: black;}
h1 {color: gray;}
h2 {color: silver;}
```

Figure 2-1 shows the results of this stylesheet.

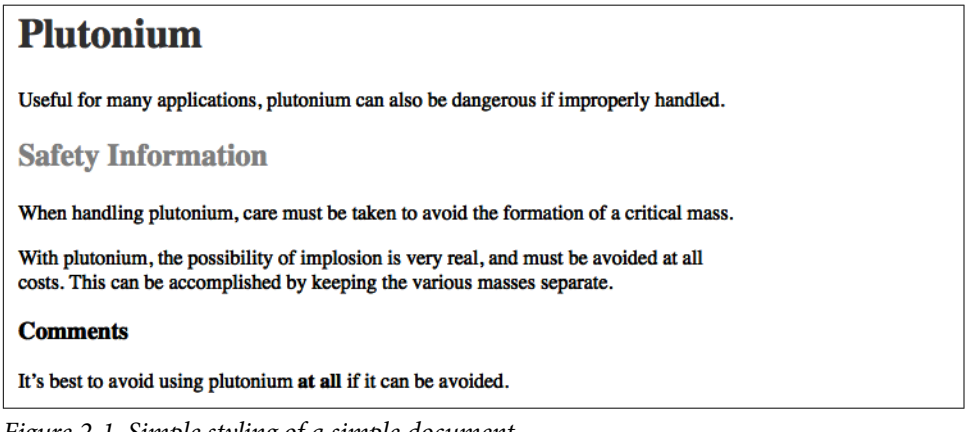


Figure 2-1. Simple styling of a simple document

Once you've globally applied styles directly to elements, you can shift those styles from one element to another. Let's say you decide that the paragraph text, not the `<h1>` elements, in Figure 2-1 should be gray. No problem. Just change the `h1` selector to `p`:

```
html {color: black;}
p {color: gray;}
h2 {color: silver;}
```

Figure 2-2 shows the results.

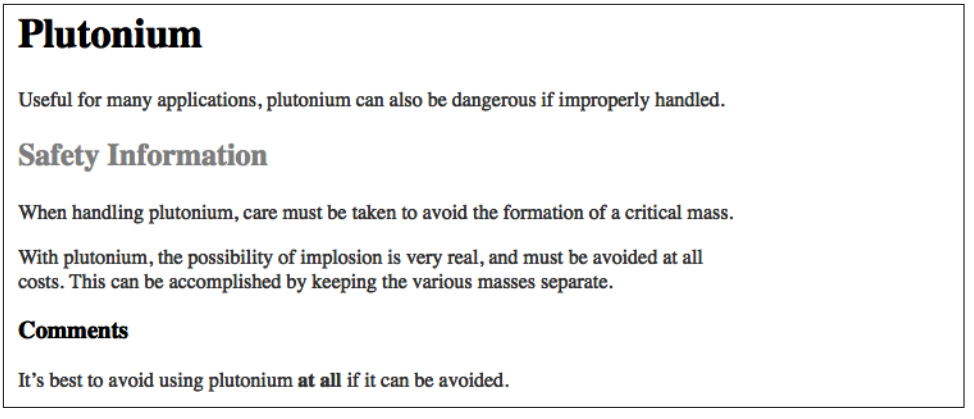


Figure 2-2. Moving a style from one element to another

Grouping

So far, you've seen fairly simple techniques for applying a single style to a single selector. But what if you want the same style to apply to multiple elements? *Grouping* allows an author to drastically compact certain types of style assignments, which makes for a shorter stylesheet.

Grouping Selectors

Let's say you want both `<h2>` elements and paragraphs to have gray text. The easiest way to accomplish this is to use the following declaration:

```
h2, p {color: gray;}
```

By placing the `h2` and `p` selectors at the beginning of the rule, before the opening curly brace, and separating them with a comma, you've defined a rule indicating that the style inside the curly braces (`color: gray;`) applies to the elements referenced by both selectors. The comma tells the browser that two different selectors are involved in the rule. Leaving out the comma would give the rule a completely different meaning, which we'll explore in "Defining Descendant Selectors" on page 45.

These alternatives produce exactly the same result, but one is a lot easier to type:

```
h1 {color: purple;}
h2 {color: purple;}
h3 {color: purple;}
h4 {color: purple;}
h5 {color: purple;}
h6 {color: purple;}

h1, h2, h3, h4, h5, h6 {color: purple;}
```

The second alternative, with one grouped selector, is also a lot easier to maintain over time.

The universal selector

The *universal selector*, displayed as an asterisk (`*`), matches any element at all, much like a wildcard. For example, to make every single element in a document bold, you would write this:

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

This declaration is equivalent to a grouped selector that lists every element contained within the document. The universal selector lets you assign the `font-weight` value `bold` to every element in the document in one efficient stroke. Beware, however: although the universal selector is convenient because it targets everything within its declaration scope, it can have unintended consequences, which are discussed in "Zeroed Selector Specificity" on page 111.

Grouping Declarations

Just as you can group selectors into a single rule, you can also group declarations. Assuming that you want all `<h1>` elements to appear in purple, 18-pixel-high Helvetica text on an aqua background (and you don't mind blinding your readers), you could write your styles like this:

```
h1 {font: 18px Helvetica;}
h1 {color: purple;}
h1 {background: aqua;}
```

But this method is inefficient—imagine creating such a list for an element that will carry 10 or 15 styles! Instead, you can group your declarations together:

```
h1 {font: 18px Helvetica; color: purple; background: aqua;}
```

This will have exactly the same effect as the three-line stylesheet just shown.

Note that using semicolons at the end of each declaration is crucial when you're grouping them. Browsers ignore whitespace in stylesheets, so the user agent must rely on correct syntax to parse the stylesheet. You can fearlessly format styles like the following:

```
h1 {
  font: 18px Helvetica;
  color: purple;
  background: aqua;
}
```

You can also minimize your CSS, removing all unrequired spaces:

```
h1{font:18px Helvetica;color:purple;background:aqua;}
```

The last three examples are treated equally by the server, but the second one is generally regarded as the most human-readable, and is the recommended method of writing your CSS during development. You might choose to minimize your CSS for network-performance reasons, but this is usually automatically handled by a build tool, server-side script, caching network, or other service, so you're usually better off writing your CSS in a human-readable fashion.

If the semicolon is omitted on the second statement, the user agent will interpret the stylesheet as follows:

```
h1 {
  font: 18px Helvetica;
  color: purple background: aqua;
}
```

Because `background:` is not a valid value for `color`, a user agent will ignore the `color` declaration entirely (including the `background: aqua` part). You might think the browser would at least render `<h1>`s as purple text without an aqua background, but not so. Instead, they will be the inherited color with a transparent background. The declaration `font: 18px Helvetica` will still take effect since it was correctly terminated with a semicolon.



Although following the last declaration of a rule with a semicolon is not technically necessary in CSS, doing so is generally good practice. First, it will keep you in the habit of terminating your declarations with semicolons, the lack of which is one of the most common causes of rendering errors. Second, if you decide to add another declaration to a rule, you won't have to worry about forgetting to insert an extra semicolon.

As with selector grouping, declaration grouping is a convenient way to keep your stylesheets short, expressive, and easy to maintain.

Grouping Everything

You now know that you can group selectors and you can group declarations. By combining both kinds of grouping in single rules, you can define very complex styles using only a few statements. Now, what if you want to assign some complex styles to all the headings in a document, and you want the same styles to be applied to all of them? Here's how:

```
h1, h2, h3, h4, h5, h6 {color: gray; background: white; padding: 0.5em;
  border: 1px solid black; font-family: Charcoal, sans-serif;}
```

Here we've grouped the selectors, so the styles inside the curly braces will be applied to all the headings listed; grouping the declarations means that all of the listed styles will be applied to the selectors on the left side of the rule. Figure 2-3 shows the result of this rule.

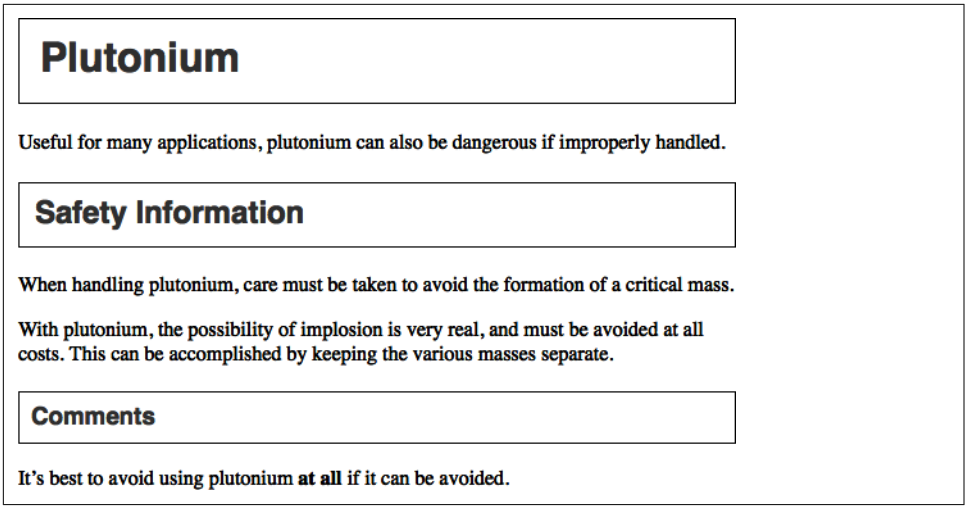


Figure 2-3. Grouping both selectors and rules

This approach is preferable to the drawn-out alternative, which would begin with something like this:

```
h1 {color: gray;}
h2 {color: gray;}
h3 {color: gray;}
h4 {color: gray;}
h5 {color: gray;}
h6 {color: gray;}
h1 {background: white;}
h2 {background: white;}
h3 {background: white;}
```

…and continue for many lines. You *can* write out your styles the long way, but we don't recommend it—editing them would be about as tedious as using `style` attributes everywhere!

Grouping allows for some interesting choices. For example, all the groups of rules in the following example are equivalent—each merely shows a different way of grouping both selectors and declarations:

```
/* group 1 */
h1 {color: silver; background: white;}
h2 {color: silver; background: gray;}
h3 {color: white; background: gray;}
h4 {color: silver; background: white;}
b {color: gray; background: white;}


/* group 2 */
h1, h2, h4 {color: silver;}
h2, h3 {background: gray;}
```

```
h1, h4, b {background: white;}
h3 {color: white;}
b {color: gray;}

/* group 3 */
h1, h4 {color: silver; background: white;}
h2 {color: silver;}
h3 {color: white;}
h2, h3 {background: gray;}
b {color: gray; background: white;}
```

Any of these three approaches to grouping selectors and declarations will yield the result shown in Figure 2-4.

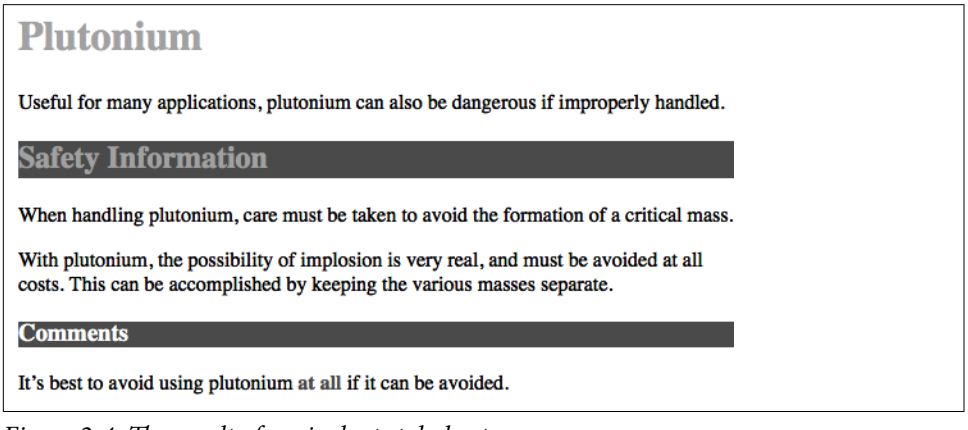


Figure 2-4. The result of equivalent stylesheets

Class and ID Selectors

So far, we've been grouping selectors and declarations together in a variety of ways, but the selectors we've been using are very simple ones that refer only to document elements. Type selectors are fine up to a point, but oftentimes you need something a little more focused.

In addition to type selectors, CSS has *class selectors* and *ID selectors*, which let you assign styles based on HTML attributes but independent of element type. These selectors can be used on their own or in conjunction with type selectors. However, they work only if you've marked up your document appropriately, so using them generally involves a little forethought and planning.

For example, say a document contains multiple warnings. You want each warning to appear in boldfaced text so that it will stand out. However, you don't know which element types contain this warning content. Some warnings could be entire paragraphs, while others could be a single item within a lengthy list or a few words in a section of text. So, you can't define a rule using type selectors of any kind. Suppose you tried this route:

```
p {
  font-weight: bold;
  color: red;
}
```

All paragraphs would be red and bold, not just those that contain warnings. You need a way to select only the text that contains warnings—or, more precisely, a way to select only those elements that are warnings. How do you do it? You apply styles to parts of the document that have been marked in a certain way, independent of the elements involved, by using class selectors.

Class Selectors

The most common way to apply styles without worrying about the elements involved is to use *class selectors*. Before you can use them, however, you need to modify your document markup so that the class selectors will work. Enter the `class` attribute:

```
<p class="warning">When handling plutonium, care must be taken to avoid
the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion is
very real, and must be avoided at all costs</span>. This can be accomplished
by keeping the various masses separate.</p>
```

To associate the styles of a class selector with an element, you must assign a `class` attribute the appropriate value. In the previous code block, a `class` value of `warning` is assigned to two elements: the first paragraph and the `<span>` element in the second paragraph.

To apply styles to these classed elements, you can use a compact notation in which the name of the class is preceded by a period (`.`):

```
*.warning {font-weight: bold;}
```

When combined with the example markup shown earlier, this simple rule has the effect shown in Figure 2-5. The declaration `font-weight: bold` will be applied to every element that carries a `class` attribute with a value of `warning`.

As Figure 2-5 illustrates, the class selector works by directly referencing a value that will be found in the `class` attribute of an element. This reference is *always* preceded by a period (`.`), which marks it as a class selector. The period helps keep the class selector separate from anything with which it might be combined, such as a type selector. For example, you may want boldfaced warning text only when an entire paragraph is a warning:

```
p.warning {font-weight: bold;}
```

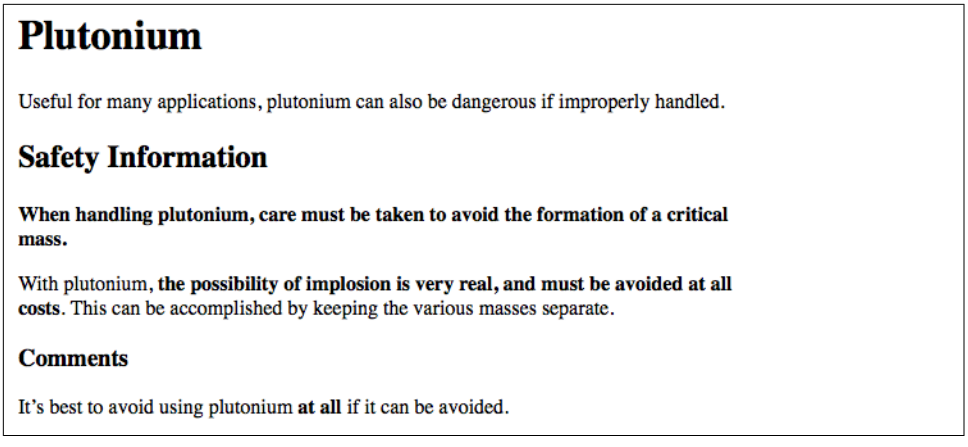


Figure 2-5. Using a class selector

The selector now matches any `<p>` elements that have a `class` attribute containing the word `warning`, but no other elements of any kind, classed or otherwise. Since the `<span>` element is not a paragraph, the rule's selector doesn't match it, and it won't be displayed using boldfaced text.

If you wanted to assign different styles to the `<span>` element, you could use the selector `span.warning`:

```
p.warning {font-weight: bold;}
span.warning {font-style: italic;}
```

In this case, the warning paragraph is boldfaced, while the warning `<span>` is italicized. Each rule applies only to a specific type of element/class combination, so it does not leak over to other elements.

Another option is to use a combination of a general class selector and an element-specific class selector to make the styles even more useful, as in the following markup:

```
.warning {font-style: italic;}
span.warning {font-weight: bold;}
```

Figure 2-6 shows the results.

In this situation, any warning text will be italicized, but only the text within a `<span>` element with a `class` of `warning` will be both boldfaced and italicized.

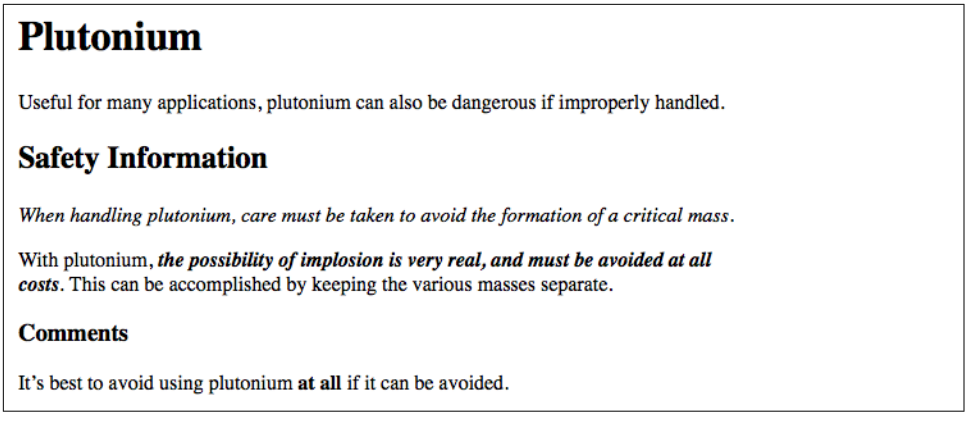


Figure 2-6. Using generic and specific selectors to combine styles



Notice the format of the general class selector used in the previous example: it's a class name preceded by a period, and without an element name or universal selector. If you want to select all elements that share a class name, you can omit the universal selector from a class selector without any ill effects. Thus, `*.warning` and `.warning` will have exactly the same effect.

Another thing about class names: they should *never* begin with a number. Browsers will allow you to get away with this, but CSS validators will complain, and it's a bad habit to get into. Thus, you should write `.c8675` in your CSS and `class="c8675"` in your HTML, rather than `.8675` and `class="8675"`. If you must refer to classes that begin with numbers, put a backslash between the period and the first number in your class selector, like so: `.\8675`.

Multiple Classes

In the previous section, we dealt with `class` values that contain a single word. In HTML, it's possible to have a space-separated list of words in a single `class` value. For example, if you want to mark a particular element as being both urgent and a warning, you could write this:

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion is
very real, and must be avoided at all costs</span>. This can be accomplished
by keeping the various masses separate.</p>
```

The order of the words doesn't matter; `warning urgent` would also work and would yield precisely the same results no matter how your CSS is written. Unlike HTML tags and type selectors, class selectors are case-sensitive.

Now let's say you want all elements with a `class` of `warning` to be boldfaced, those with a `class` of `urgent` to be italic, and those elements with both values to have a silver background. This would be written as follows:

```
.warning {font-weight: bold;}
.urgent {font-style: italic;}
.warning.urgent {background: silver;}
```

By chaining two class selectors together, you can select only those elements that have both class names, in any order. As you can see, the HTML source contains `class="urgent warning"`, but the CSS selector is written `.warning.urgent`. Regardless, the rule will still cause the "When handling plutonium… " paragraph to have a silver background, as illustrated in Figure 2-7. This happens because the order in which the words are written in the source document, or in the CSS, doesn't matter. (This is not to say the order of classes is always irrelevant, but we'll get to that later in the chapter.)

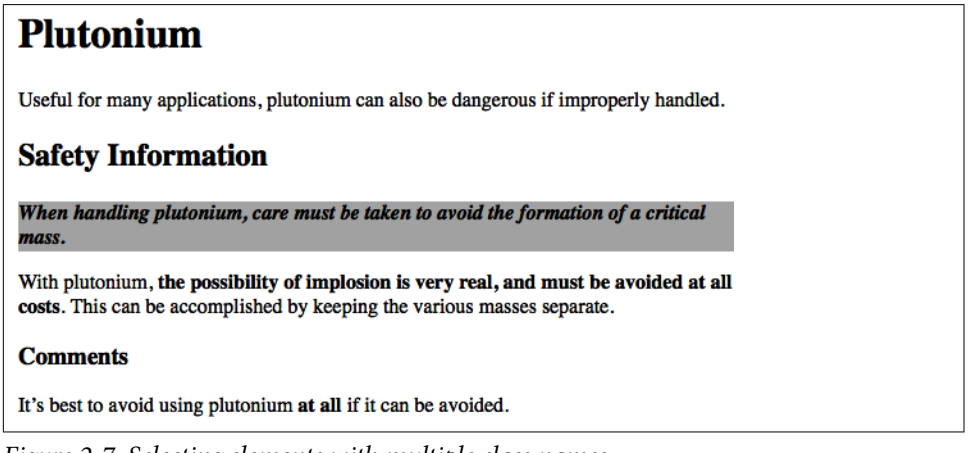


Figure 2-7. Selecting elements with multiple class names

If a multiple class selector contains a name that is not in the space-separated list, the match will fail. Consider the following rule:

```
p.warning.help {background: red;}
```

As you might expect, the selector will match only those `<p>` elements with a `class` containing the space-separated words `warning` and `help`. Therefore, it will not match a `<p>` element with just the words `warning` and `urgent` in its `class` attribute. It would, however, match the following:

```
<p class="urgent warning help">Help me!</p>
```

ID Selectors

In some ways, *ID selectors* are similar to class selectors, but a few crucial differences exist. First, ID selectors are preceded by a hash sign (#)—formally called an octothorpe and also known as a pound sign (in the United States), number sign, or tic-tac-toe board—instead of a period. Thus, you might see a rule like this one:

```
*#first-para {font-weight: bold;}
```

This rule produces boldfaced text in any element whose `id` attribute has a value of `first-para`.

The second difference is that instead of referencing values of the `class` attribute, ID selectors refer, sensibly enough, to values found in `id` attributes. Here's an example of an ID selector in action:

```
*#lead-para {font-weight: bold;}

<p id="lead-para">This paragraph will be boldfaced.</p>
<p>This paragraph will NOT be bold.</p>
```

Note that the value `lead-para` could have been assigned to any element within the document. In this particular case, it is applied to the first paragraph, but we could have applied it just as easily to the second or third paragraph. Or an unordered list. Or anything.

The third difference is that a document should have only one instance of a given ID value. If you find yourself wanting to apply the same ID to multiple elements in a document, make it a class instead.

As with class selectors, it is possible (and very much the norm) to omit the universal selector from an ID selector. In the previous example, we could also have written this with the exact same effect:

```
#lead-para {font-weight: bold;}
```

This is useful when you know that a certain ID value will appear in a document, but you don't know the element type on which it will appear. For example, you may know that in any given document, there will be an element with an ID value of `mostImportant`. You don't know whether that most important thing will be a paragraph, a short phrase, a list item, or a section heading. You know only that it will exist in each document, occur in an arbitrary element, and appear no more than once. In that case, you would write a rule like this:

```
#mostImportant {color: red; background: yellow;}
```

This rule would match any of the following elements (which, as noted before, should *not* appear together in the same document because they all have the same ID value):

```
<h1 id="mostImportant">This is important!</h1>
<em id="mostImportant">This is important!</em>
<ul id="mostImportant">This is important!</ul>
```

While HTML standards say each `id` must be unique in a document, CSS doesn't care. If we had erroneously included the HTML shown just now, all three would likely be red with a yellow background because all three match the `#mostImportant` selector.



As with class names, IDs should never start with numbers. If you must refer to an ID that begins with a number and cannot change the ID value in the markup, use a backslash before the first number, as in `#\309`.

Deciding Between Class and ID

You may assign classes to any number of elements, as demonstrated earlier; the class name `warning` was applied to both a `<p>` and a `<span>` element, and it could have been applied to many more elements. ID values, on the other hand, should be used once, and only once, within an HTML document. Therefore, if you have an element with an `id` value of `lead-para`, no other element in that document should have an `id` value of `lead-para`.

That's according to the HTML specification, anyway. As noted previously, CSS doesn't care if your HTML is valid or not: it should find however many elements a selector can match. That means that if you sprinkle an HTML document with several elements, all of which have the same value for their ID attributes, you should get the same styles applied to each.



Having more than one of the same ID value in a document makes DOM scripting more difficult, since functions like `getElementById()` depend on there being one, and only one, element with a given ID value.

Unlike class selectors, ID selectors can't be combined with other IDs, since ID attributes do not permit a space-separated list of words. An ID selector can be combined with itself, though: `#warning#warning` will match the element with an `id` value of `warning`. This should rarely, if ever, be done, but it is possible.

Another difference between `class` and `id` names is that IDs carry more weight when you're trying to determine which styles should be applied to a given element. This is explained in greater detail in Chapter 4.

Also note that HTML defines class and ID values to be case-sensitive, so the capitalization of your class and ID values must match what's found in your documents. Thus, in the following pairing of CSS and HTML, the element's text will not be boldfaced:

```
p.criticalInfo {font-weight: bold;}

<p class="criticalinfo">Don't look down.</p>
```

Because of the change in case for the letter *i*, the selector will not match the element shown.

On a purely syntactical level, the dot-class notation (e.g., `.warning`) is not guaranteed to work for XML documents. As of this writing, the dot-class notation works in HTML, Scalar Vector Graphics (SVG), and Mathematical Markup Language (MathML), and it may well be permitted in future languages, but it's up to each language's specification to decide that. The hash-ID notation (e.g., `#lead`) should work in any document language that has an attribute whose value is supposed to be unique within a document.

Attribute Selectors

With both class and ID selectors, what you're really doing is selecting values of elements' attributes. The syntax used in the previous two sections is particular to HTML, SVG, and MathML documents as of this writing. In other markup languages, these class and ID selectors may not be available (as, indeed, those attributes may not be present).

To address this situation, CSS2 introduced *attribute selectors*, which can be used to select elements based on their attributes and the values of those attributes. There are four general types of attribute selectors: simple attribute selectors, exact attribute value selectors, partial-match attribute value selectors, and leading-value attribute selectors.

Simple Attribute Selectors

If you want to select elements that have a certain attribute, regardless of that attribute's value, you can use a *simple attribute selector*. For example, to select all `<h1>` elements that have a `class` attribute with any value and make their text silver, write this:

```
h1[class] {color: silver;}
```

So, given the following markup,

```
<h1 class="hoopla">Hello</h1>
<h1>Serenity</h1>
<h1 class="fancy">Fooling</h1>
```

you get the result shown in Figure 2-8.

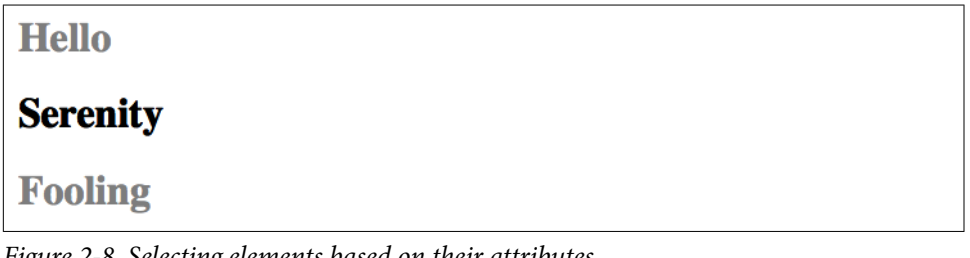


Figure 2-8. Selecting elements based on their attributes

This strategy is very useful in XML documents, as XML languages tend to have element and attribute names that are specific to their purpose. Consider an XML language that is used to describe planets of the solar system (we'll call it *PlanetML*). If you want to select all `<pml-planet>` elements with a `moons` attribute and make them boldface, thus calling attention to any planet that has moons, you would write this:

```
pml-planet[moons] {font-weight: bold;}
```

This would cause the text of the second and third elements in the following markup fragment to be boldfaced, but not the first:

```
<pml-planet>Venus</pml-planet>
<pml-planet moons="1">Earth</pml-planet>
<pml-planet moons="2">Mars</pml-planet>
```

In HTML documents, you can use this feature in creative ways. For example, you could style all images that have an `alt` attribute, thus highlighting those images that are correctly formed:

```
img[alt] {outline: 3px solid forestgreen;}
```

This particular example is generally useful more for diagnostic purposes—determining whether images are indeed correctly marked up—than for design purposes.

If you wanted to boldface any element that includes `title` information, which most browsers display as a tool tip when a cursor hovers over the element, you could write this:

```
*[title] {font-weight: bold;}
```

Similarly, you could style only those anchors (`<a>` elements) that have an `href` attribute, thus applying the styles to any hyperlink but not to any placeholder anchors.

It is also possible to select elements based on the presence of more than one attribute. You do this by chaining the attribute selectors together. For example, to boldface the text of any HTML hyperlink that has both an `href` and a `title` attribute, you would write the following:

```
a[href][title] {font-weight: bold;}
```

This would boldface the first link in the following markup, but not the second or third:

```
<a href="https://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="https://developer.mozilla.org">Standards Info</a><br />
<a title="Not a link">dead.letter</a>
```

Selection Based on Exact Attribute Value

You can further narrow the selection process to encompass only those elements whose attributes are a certain value. For example, let's say you want to boldface any hyperlink that points to a certain document on the web server. This would look something like the following:

```
a[href="http://www.css-discuss.org/about.html"] {font-weight: bold;}
```

This will boldface the text of any `a` element that has an `href` attribute with *exactly* the value `http://www.css-discuss.org/about.html`. Any change at all, even dropping the `www.` part or changing to a secure protocol with `https`, will prevent a match.

Any attribute and value combination can be specified for any element. However, if that exact combination does not appear in the document, the selector won't match anything. Again, XML languages can benefit from this approach to styling. Let's return to our PlanetML example. Suppose you want to select only those `planet` elements that have a value of `1` for the attribute `moons`:

```
planet[moons="1"] {font-weight: bold;}
```

This would boldface the text of the second element in the following markup fragment, but not the first or third:

```
<planet>Venus</planet>
<planet moons="1">Earth</planet>
<planet moons="2">Mars</planet>
```

As with attribute selection, you can chain together multiple attribute value selectors to select a single document. For example, to double the size of the text of any HTML hyperlink that has both an `href` with a value of `https://www.w3.org/` and a `title` attribute with a value of `W3C Home`, you would write this:

```
a[href="https://www.w3.org/"][title="W3C Home"] {font-size: 200%;}
```

This would double the text size of the first link in the following markup, but not the second or third:

```
<a href="https://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="https://developer.mozilla.org"
  title="Mozilla Developer Network">Standards Info</a><br />
<a href="http://www.example.org/" title="W3C Home">confused.link</a>
```

Figure 2-9 shows the results.

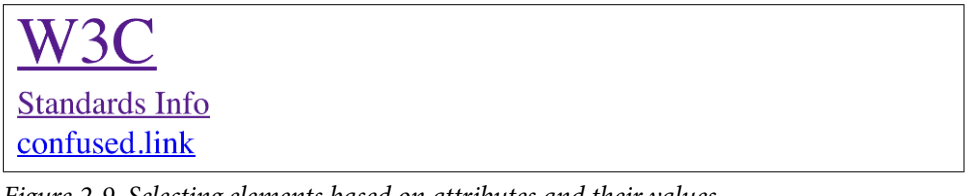


Figure 2-9. Selecting elements based on attributes and their values

Again, this format requires an *exact* match for the attribute's value. Matching becomes an issue when an attribute selector encounters values that can, in turn, contain a space-separated list of values (e.g., the HTML attribute `class`). For example, consider the following markup fragment:

```
<planet type="barren rocky">Mercury</planet>
```

The only way to match this element based on its exact attribute value is to write this:

```
planet[type="barren rocky"] {font-weight: bold;}
```

If you were to write `planet[type="barren"]`, the rule would not match the example markup and thus would fail. This is true even for the `class` attribute in HTML. Consider the following:

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
```

To select this element based on its exact attribute value, you would have to write this:

```
p[class="urgent warning"] {font-weight: bold;}
```

This is *not* equivalent to the dot-class notation covered earlier, as you will see in the next section. Instead, it selects any `p` element whose `class` attribute has *exactly* the value `urgent warning`, with the words in that order and a single space between them. It's effectively an exact string match, whereas when using a `class` selector, the class order doesn't matter.

Also, be aware that ID selectors and attribute selectors that target the `id` attribute are not precisely the same. In other words, a subtle but crucial difference exists between `h1#page-title` and `h1[id="page-title"]`. This difference is explained in Chapter 4.

Selection Based on Partial Attribute Values

Odds are that you'll sometimes want to select elements based on portions of their attribute values, rather than the full value. For such situations, CSS offers a variety of options for matching substrings in an attribute's value. These are summarized in Table 2-1.

Table 2-1. Substring matching with attribute selectors

Type	Description
`[foo~="bar"]`	Selects any element with an attribute `foo` whose value contains the word `bar` in a space-separated list of words
`[foo*="bar"]`	Selects any element with an attribute `foo` whose value *contains* the substring `bar`
`[foo^="bar"]`	Selects any element with an attribute `foo` whose value *begins* with `bar`
`[foo$="bar"]`	Selects any element with an attribute `foo` whose value *ends* with `bar`
`[foo|="bar"]`	Selects any element with an attribute `foo` whose value *starts* with `bar` followed by a hyphen (U+002D) or whose value is exactly equal to `bar`

The last of these attribute selectors that match on a partial subset of an element's attribute value is easier to show than it is to describe. Consider the following rule:

```
*[lang|="en"] {color: white;}
```

This rule will select any element whose `lang` attribute is equal to `en` or begins with `en-`. Therefore, the first three elements in the following example markup would be selected, but the last two would not:

```
<h1 lang="en">Hello!</h1>
<p lang="en-us">Greetings!</p>
<div lang="en-au">G'day!</div>
<p lang="fr">Bonjour!</p>
<h4 lang="cy-en">Jrooana!</h4>
```

In general, the form `[att|="val"]` can be used for any attribute and its values. Let's say you have a series of figures in an HTML document, each of which has a filename like *figure-1.gif* or *figure-3.jpg*. You can match all of these images by using the following selector:

```
img[src|="figure"] {border: 1px solid gray;}
```

Or, if you're creating a CSS framework or pattern library, instead of creating redundant classes like `"btn btn-small btn-arrow btn-active"`, you can declare `"btn-small-arrow-active"`, and target the class of elements with the following:

```
*[class|="btn"] { border-radius: 5px;}

<button class="btn-small-arrow-active">Click Me</button>
```

The most common use for this type of attribute selector is to match language values, as demonstrated in "The :lang() and :dir() Pseudo-Classes" on page 83.

Matching one word in a space-separated list

For any attribute that accepts a space-separated list of words, you can select elements based on the presence of any one of those words. The classic example in HTML is the `class` attribute, which can accept one or more words as its value. Consider our usual example text:

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
```

Let's say you want to select elements whose `class` attribute contains the word `warning`. You can do this with an attribute selector:

```
p[class~="warning"] {font-weight: bold;}
```

Note the presence of the tilde (~) in the selector. It is the key to selection based on the presence of a space-separated word within the attribute's value. If you omit the tilde, you would have an exact value-matching attribute selector, as discussed in the previous section.

This selector construct is equivalent to the dot-class notation discussed in "Deciding Between Class and ID" on page 33. Thus, `p.warning` and `p[class~="warning"]` are equivalent when applied to HTML documents. Here's an example that is an HTML version of the PlanetML markup seen earlier:

```
<span class="barren rocky">Mercury</span>
<span class="cloudy barren">Venus</span>
<span class="life-bearing cloudy">Earth</span>
```

To italicize all elements with the word `barren` in their `class` attribute, you write this:

```
span[class~="barren"] {font-style: italic;}
```

This rule's selector will match the first two elements in the example markup and thus italicize their text, as shown in Figure 2-10. This is the same result we would expect from writing `span.barren {font-style: italic;}`.



Figure 2-10. Selecting elements based on portions of attribute values

So why bother with the tilde-equals attribute selector in HTML? Because it can be used for any attribute, not just `class`. For example, you might have a document that contains numerous images, only some of which are figures. You can use a partial-match value attribute selector aimed at the `title` text to select only those figures:

```
img[title~="Figure"] {border: 1px solid gray;}
```

This rule selects any image whose `title` text contains the word `Figure` (but not `figure`, as title attributes are case-sensitive). Therefore, as long as all your figures have `title` text that looks something like "Figure 4. A bald-headed elder statesman," this rule will match those images. For that matter, the selector `img[title~="Figure"]` will also match a title attribute with the value "How to Figure Out Who's in Charge." Any image that does not have a `title` attribute, or whose `title` value doesn't contain the word `Figure`, won't be matched.

Matching a substring within an attribute value

Sometimes you want to select elements based on a portion of their attribute values, but the values in question aren't space-separated lists of words. In these cases, you can use the asterisk-equals substring matching form `[attr*="val"]` to match substrings that appear anywhere inside the attribute values. For example, the following CSS matches any `<span>` element whose `class` attribute contains the substring `cloud`, so both "cloudy" planets are matched, as shown in Figure 2-11:

```
span[class*="cloud"] {font-style: italic;}

<span class="barren rocky">Mercury</span>
<span class="cloudy barren">Venus</span>
<span class="life-bearing cloudy">Earth</span>
```

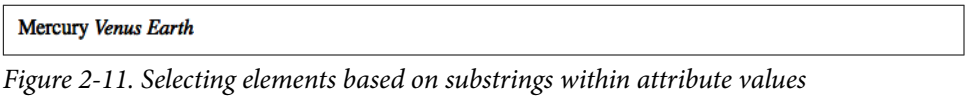


Figure 2-11. Selecting elements based on substrings within attribute values

Note the presence of the asterisk (`*`) in the selector. It's the key to selecting elements based on the presence of a substring within an attribute's value. To be clear, it is *not* related to the universal selector, other than it uses the same character.

As you can imagine, this particular capability has many useful applications. For example, suppose you want to specially style any links to the W3C's website. Instead of classing them all and writing styles based on that class, you could instead write the following rule:

```
a[href*="w3.org"] {font-weight: bold;}
```

You aren't confined to the `class` and `href` attributes. Any attribute is up for grabs here (`title`, `alt`, `src`, `id`…); if the attribute has a value, you can style based on a substring within that value. The following rule draws attention to any image with the string `space` in its source URL:

```
img[src*="space"] {outline: 5px solid red;}
```

Similarly, the following rule draws attention to `<input>` elements that have a title telling the user what to do, along with any other input whose title contains the substring `format` in its title:

```
input[title*="format"] {background-color: #dedede;}

<input type="tel"
    title="Telephone number should be formatted as XXX-XXX-XXXX"
    pattern="\d{3}\-\d{3}\-\d{4}">
```

A common use for the general substring attribute selector is to match a section of a class in pattern library class names. Elaborating on the preceding example, we can target any class name that starts with `btn` followed by a hyphen, and that contains the substring `arrow` preceded by a hyphen, by using the pipe-equals attribute selector:

```
*[class|="btn"][class*="-arrow"]:after { content: "▾";}

<button class="btn-small-arrow-active">Click Me</button>
```

The matches are exact: if you include whitespace in your selector, whitespace must also be present in an attribute's value. The attribute values are case-sensitive when the underlying document language requires case sensitivity. Class names, titles, URLs, and ID values are all case-sensitive, but enumerated HTML attribute values, such as input type keyword values, are not:

```
input[type="CHeckBoX"] {margin-right: 10px;}

<input type="checkbox" name="rightmargin" value="10px">
```

Matching a substring at the beginning of an attribute value

If you want to select elements based on a substring at the beginning of an attribute value, the caret-equals attribute selector pattern `[att^="val"]` is what you're seeking. This can be particularly useful when you want to style types of links differently, as illustrated in Figure 2-12:

```
a[href^="https:"] {font-weight: bold;}
a[href^="mailto:"] {font-style: italic;}
```

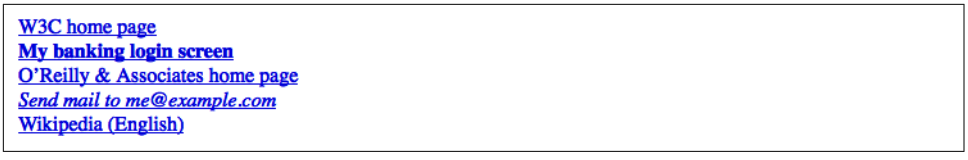


Figure 2-12. Selecting elements based on substrings that begin attribute values

In another use case, you may want to style all images in an article that are also figures, like the figures you see throughout this text. Assuming that the `alt` text of each figure begins with text in the pattern "Figure 5"—which is an entirely reasonable assumption in this case—you can select only those images with the caret-equals attribute selector:

```
img[alt^="Figure"] {border: 2px solid gray;  display: block; margin: 2em auto;}
```

The potential drawback here is that *any* `<img>` element whose `alt` starts with `Figure` will be selected, whether or not it's meant to be an illustrative figure. The likeliness of that occurring depends on the document in question.

Another use case is selecting all of the calendar events that occur on Mondays. In this case, let's assume all of the events have a `title` attribute containing a date in the format "Monday, March 5th, 2012." Selecting them all is just a simple matter of using `[title^="Monday"]`.

Matching a substring at the end of an attribute value

The mirror image of beginning-substring matching is ending-substring matching, which is accomplished using the `[att$="val"]` pattern. A very common use for this capability is to style links based on the kind of resource they target, such as separate styles for PDF documents, as illustrated in Figure 2-13:

```
a[href$=".pdf"] {font-weight: bold;}
```

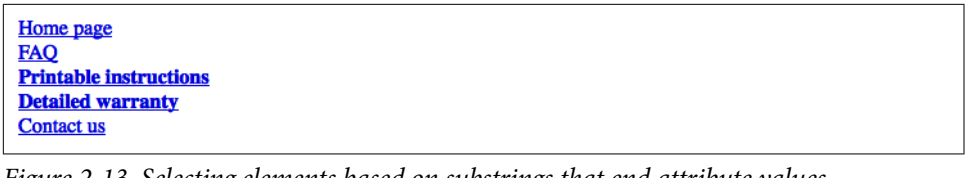


Figure 2-13. Selecting elements based on substrings that end attribute values

Similarly, you could (for whatever reason) select images based on their image format with the dollar-equals attribute selector:

```
img[src$=".gif"] {...}
img[src$=".jpg"] {...}
img[src$=".png"] {...}
```

To continue the calendar example from the previous section, it would be possible to select all of the events occurring within a given year by using a selector like `[title$="2015"]`.



You may have noticed that we've quoted all the attribute values in the attribute selectors. Quoting is required if the value includes any special characters, begins with a hyphen or digit, or is otherwise invalid as an identifier and needs to be quoted as a string. To be safe, we recommend always quoting attribute values in attribute selectors, even though it is required only to make strings out of invalid identifiers.

The Case-Insensitivity Identifier

Including an `i` before the closing bracket of an attribute selector will allow that selector to match attribute values case-insensitively, regardless of document language rules. For example, suppose you want to select all links to PDF documents, but you don't know if they'll end in *.pdf*, *.PDF*, or even *.Pdf*. Here's how:

```
a[href$='.PDF' i]
```

Adding that humble little `i` means the selector will match any `a` element whose `href` attribute's value ends in `.pdf`, regardless of the capitalization of the letters *P*, *D*, and *F*.

This case-insensitivity option is available for all the attribute selectors we've covered. Note, however, that this applies to only the *values* in the attribute selectors. It does not enforce case insensitivity on the attribute names themselves. Thus, in a case-sensitive language, `planet[type*="rock" i]` will match all of the following:

```
<planet type="barren rocky">Mercury</planet>
<planet type="cloudy ROCKY">Venus</planet>
<planet type="life-bearing Rock">Earth</planet>
```

It will *not* match the following element, because the attribute `TYPE` isn't matched by `type` in XML:

```
<planet TYPE="dusty rock">Mars</planet>
```

This is in languages that enforce case sensitivity in the element and attribute syntax. In languages that are case-insensitive, like HTML, this isn't an issue.



A proposed mirror identifier, `s`, enforces case sensitivity. As of early 2023, it is supported by only the Firefox family of browsers.

Using Document Structure

CSS is so capable because it uses the structure of documents to determine appropriate styles and how to apply them. Let's take a moment to discuss structure before moving on to more powerful forms of selection.

Understanding the Parent-Child Relationship

To understand the relationship between selectors and documents, we need to once again examine how documents are structured. Consider this very simple HTML document:

```
<!DOCTYPE html>
<html lang="en-us">
<head>
 <meta charset="utf-8">
 <meta name="viewport" content="width=device-width">
 <title>Meerkat Central</title>
</head>
<body>
 <h1>Meerkat <em>Central</em></h1>
 <p>
 Welcome to Meerkat <em>Central</em>, the <strong>best meerkat web site
 on <a href="inet.html">the <em>entire</em> Internet</a></strong>!</p>
 <ul>
  <li>We offer:
   <ul>
    <li><strong>Detailed information</strong> on how to adopt a meerkat</li>
    <li>Tips for living with a meerkat</li>
    <li><em>Fun</em> things to do with a meerkat, including:
     <ol>
      <li>Playing fetch</li>
      <li>Digging for food</li>
      <li>Hide and seek</li>
     </ol>
    </li>
   </ul>
  </li>
  <li>...and so much more!</li>
 </ul>
 <p>
 Questions? <a href="mailto:suricate@meerkat.web">Contact us!</a>
 </p>
</body>
</html>
```

Much of the power of CSS is based on the *parent-child relationship* of elements. HTML documents (and most structured documents of any kind) are based on a hierarchy of elements, which is visible in the "tree" view of the document (see Figure 2-14). In this hierarchy, each element fits somewhere into the overall structure of the document. Every element in the document is either the *parent* or the *child* of another element, and it's often both. If a parent has more than one child, those children are *siblings*.

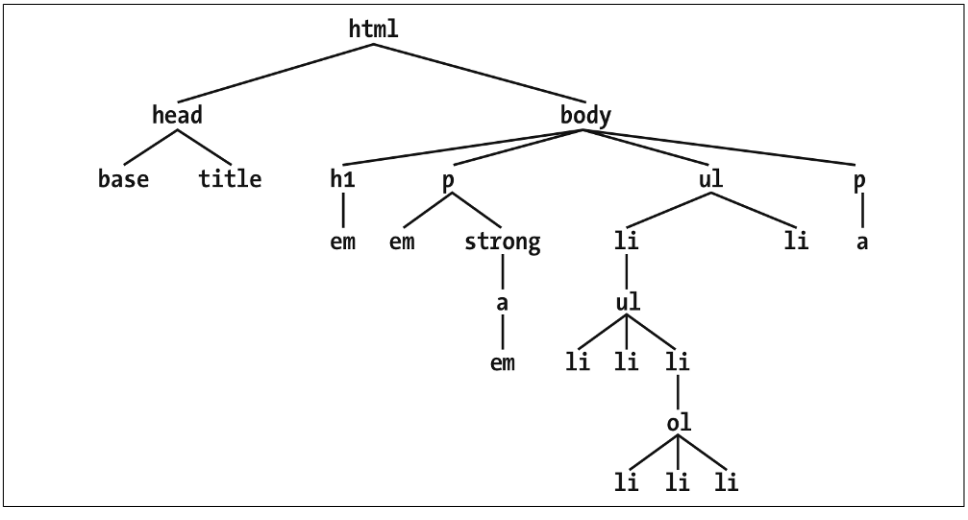


Figure 2-14. A document tree structure

An element is said to be the parent of another element if it appears directly above that element in the document hierarchy. For example, in Figure 2-14, the first `<p>` element from the left is parent to the `<em>` and `<strong>` elements, while `<strong>` is parent to an anchor (`<a>`) element, which is itself parent to another `<em>` element. Conversely, an element is the child of another element if it is directly beneath the other element. Thus, the anchor element on the far right side of Figure 2-14 is the child of a `<p>` element, which is in turn child to the `<body>` element, and so on.

The terms *parent* and *child* are specific applications of the terms *ancestor* and *descendant*, respectively. There is a difference between them: in the tree view, if an element is exactly one level above or below another, those elements have a *parent-child* relationship. If the path from one element to another is traced through two or more levels, the elements have an ancestor-descendant relationship, but not a parent-child relationship. (A child is also a descendant, and a parent is also an ancestor.) In Figure 2-14, the uppermost `<ul>` element is parent to two `<li>` elements, but the uppermost `<ul>` is also the ancestor of every element descended from its `<li>` element, all the way down to the most deeply nested `<li>` elements. Those `<li>` elements, children of the `<ol>`, are siblings.

Also, in Figure 2-14, there is an anchor that is a child of `<strong>`, but also a descendant of the `<p>`, `<body>`, and `<html>` elements. The `<body>` element is an ancestor of everything that the browser will display by default, and the `<html>` element is ancestor to the entire document. For this reason, in an HTML document, the `<html>` element is also called the *root element*.

Defining Descendant Selectors

The first benefit of understanding this model is the ability to define *descendant selectors*. Defining descendant selectors is the act of creating rules that operate in certain structural circumstances but not others. As an example, let's say you want to style only those `<em>` elements that are descended from `<h1>` elements. To do so, write the following:

```
h1 em {color: gray;}
```

This rule will make gray any text in an `<em>` element that is the descendant of an `<h1>` element. Other `<em>` text, such as that found in a paragraph or a block quote, will not be selected by this rule. Figure 2-15 illustrates the result.



Figure 2-15. Selecting an element based on its context

In a descendant selector, the selector side of a rule is composed of two or more space-separated selectors. The space between the selectors is an example of a *combinator*. Each space combinator can be translated as "found within," "which is part of," or "that is a descendant of," but only if you read the selector right to left. Thus, `h1 em` can be translated as, "Any `<em>` element that is a descendant of an `<h1>` element."

To read the selector left to right, you might phrase it like, "Any `<h1>` that contains an `<em>` will have the following styles applied to the `<em>`." That's much more verbose and confusing, and it's why we, like the browser, read selectors from right to left.

You aren't limited to two selectors. For example:

```
ul ol ul em {color: gray;}
```

In this case, as Figure 2-16 shows, any emphasized text that is part of an unordered list that is part of an ordered list that is itself part of an unordered list (yes, this is correct) will be gray. This is obviously a very specific selection criterion.

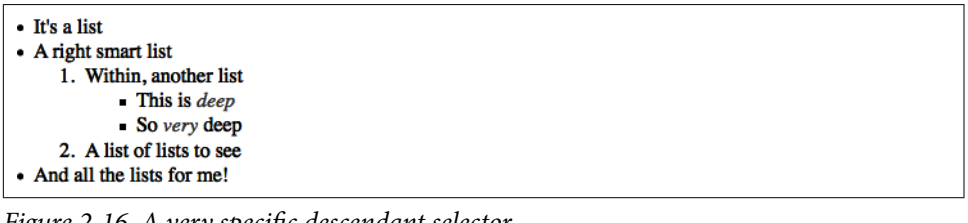


Figure 2-16. A very specific descendant selector

Descendant selectors can be extremely powerful. Let's consider a common example. Assume you have a document with a sidebar and a main area. The sidebar has a blue background, the main area has a white background, and both areas include lists of links. You can't set all links to be blue because they'd be impossible to read in the sidebar, and you also can't set all links to white because they'd disappear in the main part of the page.

The solution: descendant selectors. In this case, you give the element that contains your sidebar a class of `sidebar` and enclose the main part of the page in a `<main>` element. Then, you write styles like this:

```
.sidebar {background: blue;}
main {background: white;}
.sidebar a:any-link {color: white;}
main a:any-link {color: blue;}
```

Figure 2-17 shows the result.



Figure 2-17. Using descendant selectors to apply different styles to the same type of element



`:any-link` refers to both visited and unvisited links. We'll talk about it in detail in Chapter 3.

Here's another example: let's say that you want gray to be the text color of any `<b>` (bold-faced) element that is part of a `blockquote` and for any bold text that is found in a normal paragraph:

```
blockquote b, p b {color: gray;}
```

The result is that the text within `<b>` elements that are descended from paragraphs or block quotes will be gray.

One overlooked aspect of descendant selectors is that the degree of separation between two elements can be practically infinite. For example, if you write `ul em`, that syntax will select any `<em>` element descended from a `<ul>` element, no matter how deeply nested the `<em>` may be. Thus, `ul em` would select the `<em>` element in the following markup:

```
<ul>
  <li>List item 1
    <ol>
      <li>List item 1-1</li>
      <li>List item 1-2</li>
      <li>List item 1-3
        <ol>
          <li>List item 1-3-1</li>
```

```
          <li>List item <em>1-3-2</em></li>
          <li>List item 1-3-3</li>
        </ol>
      </li>
      <li>List item 1-4</li>
    </ol>
  </li>
</ul>
```

A more subtle aspect of descendant selectors is that they have no notion of element proximity. In other words, the closeness of two elements within the document tree has no bearing on whether a rule applies. This is important when it comes to specificity (which we'll cover in the next chapter) and when considering rules that might appear to cancel each other out.

For example, consider the following (which contains `:not()`, a selector type we'll discuss in "The negation pseudo-class" on page 84):

```
div:not(.help) span {color: gray;}
div.help span {color: red;}
```

```
<div class="help">
   <div class="aside">
      This text contains <span>a span element</span> within.
   </div>
</div>
```

What the CSS says, in effect, is "any `<span>` inside a `<div>` that doesn't have a `class` containing the word `help` should be gray" in the first rule, and "any `<span>` inside a `<div>` whose `class` contains the word `help`" in the second rule. In the given markup fragment, *both* rules apply to the `<span>` shown.

Because the two rules have equal specificity weight and the `red` rule is written last, it wins out, and the `<span>` is red. The fact that `div class="aside"` is "closer to" `<span>` than `div class="help"` is irrelevant. Again: descendant selectors have no notion of element proximity. Both rules match, only one color can be applied, and because of the way CSS works, red is the winner here. (We'll discuss why that's so in the next chapter.)



As of early 2023, proposals have been made to add element-proximity awareness to CSS via *selector scoping*, but the proposals are still being actively revised and may not come to fruition.

Selecting Children

In some cases, you don't want to select an arbitrarily descended element. Rather, you want to narrow your range to select an element that is specifically a child of another element. You might, for example, want to select a `<strong>` element only if it is a child (as opposed

to any other level of descendant) of an <h1> element. To do this, you use the *child combinator*, which is the greater-than symbol (>):

```
h1 > strong {color: red;}
```

This rule will make red the element shown in the first <h1>, but not the second:

```
<h1>This is <strong>very</strong> important.</h1>
<h1>This is <em>really <strong>very</strong></em> important.</h1>
```

Read right to left, the selector `h1 > strong` translates as, "Selects any element that is a direct child of an <h1> element." The child combinator can be optionally surrounded by whitespace. Thus, `h1 > strong`, `h1> strong`, and `h1>strong` are all equivalent. You can use or omit whitespace as you wish.

When viewing the document as a tree structure, we can see that a child selector restricts its matches to elements that are directly connected in the tree. Figure 2-18 shows part of a document tree.

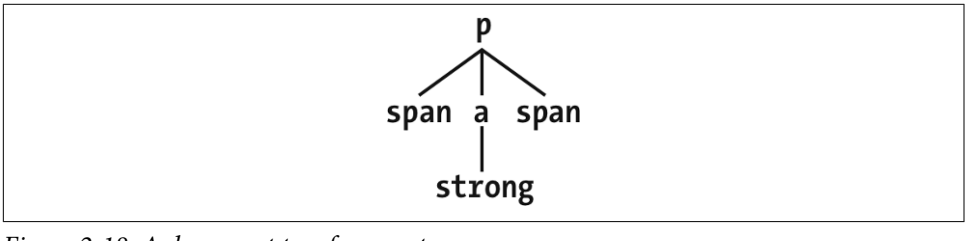


Figure 2-18. A document tree fragment

In this tree fragment, you can pick out parent-child relationships. For example, the <a> element is a parent to and is also a child of the <p> element. You could match elements in this fragment with the selectors `p > a` and `a > strong`, but not `p > strong`, since is a descendant of <p> but not its child.

You can also combine descendant and child combinators in the same selector. Thus, `table.summary td > p` will select any <p> element that is a *child* of a <td> element that is itself *descended* from a <table> element that has a `class` attribute containing the word `summary`.

Selecting Adjacent-Sibling Elements

Let's say you want to style the paragraph immediately after a heading, or give a special margin to a list that immediately follows a paragraph. To select an element that immediately follows another element with the same parent, you use the *adjacent-sibling combinator*, represented as a plus symbol (+). As with the child combinator, the symbol can be surrounded by whitespace, or not, at your discretion.

To remove the top margin from a paragraph immediately following an <h1> element, write this:

```
h1 + p {margin-top: 0;}
```

The selector is read as, "Select any <p> element that immediately follows an <h1> element that *shares a parent* with the <p> element."

To visualize how this selector works, let's once again consider a fragment of a document tree, shown in Figure 2-19.

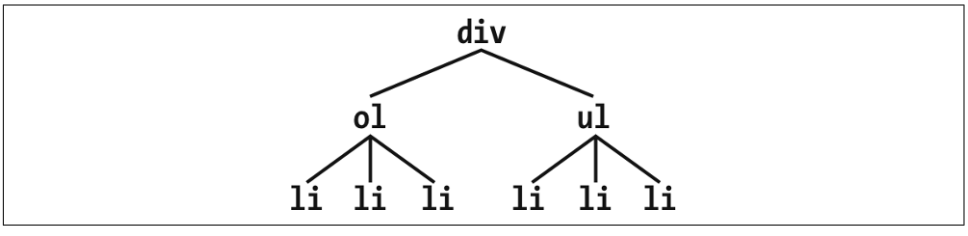


Figure 2-19. Another document tree fragment

In this fragment, a pair of lists descends from a element, one ordered and the other not, each containing three list items. Each list is an adjacent sibling, and the list items themselves are also adjacent siblings. However, the list items from the first list are *not* siblings of the second, as the two sets of list items do not share the same parent element. (At best, they're cousins, and CSS has no cousin selector.)

Remember that you can select the second of two adjacent siblings only with a single combinator. Thus, if you write `li + li {font-weight: bold;}`, only the second and third items in each list will be boldfaced. The first list items will be unaffected, as illustrated in Figure 2-20.

1. List item 1
2. **List item 1**
3. **List item 1**

This is some text that is part of the 'div'.

- A list item
- **Another list item**
- **Yet another list item**

Figure 2-20. Selecting adjacent siblings

To work properly, CSS requires that the two elements appear in *source order*. In our example, an <ol> element is followed by a <ul> element. This allows us to select the second element with `ol + ul`, but we cannot select the first by using the same syntax. For `ul + ol` to match, an ordered list must immediately follow an unordered list.

Keep in mind that text content between two elements does *not* prevent the adjacent-sibling combinator from working. Consider this markup fragment, whose tree view would be the same as that shown in Figure 2-18:

```
<div>
  <ol>
    <li>List item 1</li>
```

```
    <li>List item 1</li>
    <li>List item 1</li>
  </ol>
  This is some text that is part of the 'div'.
  <ul>
    <li>A list item</li>
    <li>Another list item</li>
    <li>Yet another list item</li>
  </ul>
</div>
```

Even though we have text between the two lists, we can still match the second list with the selector `ol + ul`. That's because the intervening text is not contained within a sibling element, but is instead part of the parent `<div>`. If we wrapped that text in a paragraph element, it would then prevent `ol + ul` from matching the second list. Instead, we might have to write something like `ol + p + ul`.

As the following example illustrates, the adjacent-sibling combinator can be used in conjunction with other combinators:

```
html > body table + ul{margin-top: 1.5em;}
```

The selector translates as, "Selects any `<ul>` element that immediately follows a sibling `<table>` element that is descended from a `<body>` element that is itself a child of an `<html>` element."

As with all combinators, you can place the adjacent-sibling combinator in a more complex setting, such as `div#content h1 + div ol`. That selector is read as, "Selects any `<ol>` element that is descended from a `<div>` when the `<div>` is the adjacent sibling of an `<h1>` that is itself descended from a `<div>` whose `id` attribute has a value of `content`."

Selecting Following Siblings

The *general sibling combinator* lets you select any element that follows another element when both elements share the same parent, represented using the tilde (~) combinator.

As an example, to italicize any `<ol>` that follows an `<h2>` and also shares a parent with the `<h2>`, you'd write `h2 ~ ol {font-style: italic;}`. The two elements do not have to be adjacent siblings, although they can be adjacent and still match this rule. The result of applying this rule to the following markup is shown in Figure 2-21:

```
<div>
  <h2>Subheadings</h2>
  <p>It is the case that not every heading can be a main heading.  Some headings
  must be subheadings.  Examples include:</p>
  <ol>
    <li>Headings that are less important</li>
    <li>Headings that are subsidiary to more important headlines</li>
    <li>Headings that like to be dominated</li>
  </ol>
  <p>Let's restate that for the record:</p>
```

```
    <ol>
      <li>Headings that are less important</li>
      <li>Headings that are subsidiary to more important headlines</li>
      <li>Headings that like to be dominated</li>
    </ol>
  </div>
```

As you can see, both ordered lists are italicized. That's because both of them are `<ol>` elements that follow an `<h2>` with which they share a parent (the `<div>`).

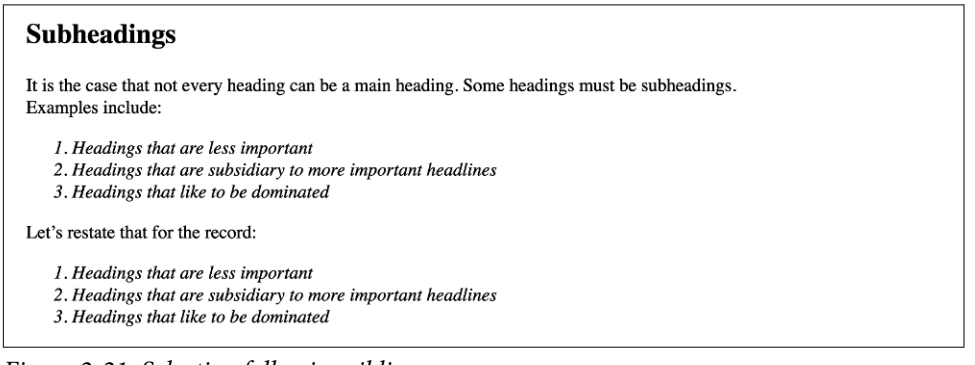


Figure 2-21. Selecting following siblings

Summary

By using selectors based on the document's language, you can create CSS rules that apply to a large number of similar elements just as easily as you can construct rules that apply in very narrow circumstances. The ability to group together both selectors and rules keeps stylesheets compact and flexible, which incidentally leads to smaller file sizes and faster download times.

Selectors are the one thing that user agents usually must get right, because the inability to correctly interpret selectors pretty much prevents a user agent from using CSS at all. On the flip side, it's crucial for authors to correctly write selectors because errors can prevent the user agent from applying the styles as intended. An integral part of correctly understanding selectors and how they can be combined is having a strong grasp of how selectors relate to document structure and how mechanisms—such as inheritance and the cascade itself—come into play when determining how an element will be styled.

The selectors we covered in this chapter aren't the end of the story, though. They're not even half the story. In the next chapter, we'll dive into the powerful and ever-expanding world of pseudo-class and pseudo-element selectors.

CHAPTER 3

Pseudo-Class and -Element Selectors

In the previous chapter, you saw how selectors can match a single element or a collection of elements, using fairly simple expressions that match HTML attributes in the document. Those are great if your need is just to style based on attributes, but what if you need to style part of a document based on its current state or structure? Or if you want to select all the form elements that are disabled, or those that are required for form submission to be allowed? For those things, and a great deal more, CSS has the pseudo-class and pseudo-element selectors.

Pseudo-Class Selectors

Pseudo-class selectors let you assign styles to what are, in effect, phantom classes inferred by the state of certain elements, or markup patterns within the document, or even by the state of the document itself.

The term *phantom classes* might seem a little odd, but it really is the best way to think of how pseudo-classes work. For example, suppose you want to highlight every other row of a data table. You could do that by marking up every other row with something like `class="even"` and then writing CSS to highlight rows with that class—or (as you'll soon see) you could use a pseudo-class selector to achieve the same effect, one that will act as if you've added all those classes to the markup even though you haven't.

One aspect of pseudo-classes needs to be made explicit here: pseudo-classes always refer to the element to which they're attached, and to no other. Seems like a weirdly obvious thing to say, right? The reason we make it explicit is that for some pseudo-classes, it's a common error to think they are descriptors that refer to descendant elements.

To illustrate this, Eric would like to share a personal anecdote:

> When my first child was born in 2003, I announced it online, as one does. A number of people responded with congratulations and CSS jokes, chief among them the selector `#ericmeyer:first-child` (we'll get to `:first-child` in just a bit). But that selector

would select me, not my daughter, and only if I were the first child of my own parents (which, as it happens, I am). To properly select *my* first child, that selector would need to be `#ericmeyer > :first-child`.

The confusion is understandable, which is why we're addressing it here. Reminders are found throughout the following sections. Just always keep in mind that the effect of pseudo-classes is to apply a sort of phantom class to the element to which they're attached, and you should be OK.

All pseudo-classes, without exception, are a word or hyphenated term preceded by a single colon (`:`), and they can appear anywhere in a selector.

Combining Pseudo-Classes

Before we really get started, a word about chaining. CSS makes it possible to combine (*chain*) pseudo-classes together. For example, you can make unvisited links red when they're hovered and visited links maroon when they are hovered:

```
a:link:hover {color: red;}
a:visited:hover {color: maroon;}
```

The order you specify doesn't matter; you could also write `a:hover:link` to the same effect as `a:link:hover`. It's also possible to assign separate hover styles to unvisited and visited links that are in another language—for example, German:

```
a:link:hover:lang(de) {color: gray;}
a:visited:hover:lang(de) {color: silver;}
```

Be careful not to combine mutually exclusive pseudo-classes. For example, a link cannot be both visited and unvisited, so `a:link:visited` doesn't make any sense and will never match anything.

Structural Pseudo-Classes

The first set of pseudo-classes we'll explore are structural in nature; that is, they refer to the markup structure of the document. Most of them depend on patterns within the markup, such as choosing every third paragraph, but others allow you to address specific types of elements.

Selecting the root element

This is the quintessence of structural simplicity: the pseudo-class `:root` selects the root element of the document. In HTML, this is *always* the `<html>` element. The real benefit of this selector is found when writing stylesheets for XML languages, as the root element may be different in every language—for example, in SVG it's the `<svg>` element, and in our earlier PlanetML examples it was the `<pml>` element—or even when you have more than one possible root element within a single language (though not a single document!).

Here's an example of styling the root element in HTML, as illustrated in Figure 3-1:

```
:root {border: 10px dotted gray;}
body {border: 10px solid black;}
```

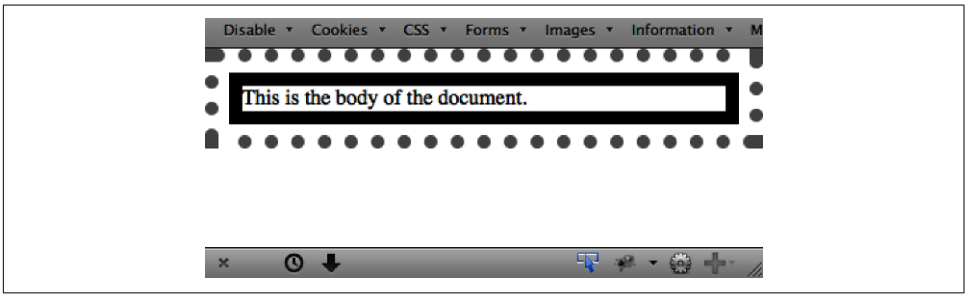


Figure 3-1. Styling the root element

In HTML documents, you can always select the `<html>` element directly, without having to use the `:root` pseudo-class. The two selectors differ in terms of specificity, which we'll cover in Chapter 4, but otherwise they'll have the same effect.

Selecting empty elements

With the pseudo-class `:empty`, you can select any element that has no children of any kind, *including* text nodes, which covers both text and whitespace. This can be useful in suppressing elements that a content management system (CMS) has generated without filling in any actual content. Thus, `p:empty {display: none;}` would prevent the display of any empty paragraphs.

Note that in order to be matched, an element must be, from a parsing perspective, truly empty—no whitespace, visible content, or descendant elements. Of the following elements, only the first and last would be matched by `p:empty`:

```
<p></p>
<p> </p>
<p>
</p>
<p><!--a comment--></p>
```

The second and third paragraphs are not matched by `:empty` because they are not empty: they contain, respectively, a single space and a single newline character. Both are considered text nodes and thus prevent a state of emptiness. The last paragraph matches because comments are not considered content, not even whitespace. But put even one space or newline to either side of that comment, and `p:empty` would fail to match.

You might be tempted to just style all empty elements with something like `*:empty {display: none;}`, but there's a hidden catch: `:empty` matches HTML's empty elements, like `<img>`, `<hr>`, `<br>`, and `<input>`. It could even match `<textarea>`, unless you insert some default text into the `<textarea>` element.

Thus, in terms of matching elements, `img` and `img:empty` are effectively the same. (They are different in terms of specificity, which we'll cover in the next chapter.)

Selecting only children

If you've ever wanted to select all the images that are wrapped by a hyperlink element, the `:only-child` pseudo-class is for you. It selects elements when they are the only child element of another element. So let's say you want to add a border to any image that's the only child of another element. You'd write the following:

```
img:only-child {border: 1px solid black;}
```

This would match any image that meets those criteria. Therefore, if you had a paragraph that contained an image and no other child elements, the image would be selected regardless of all the text surrounding it. If what you're really after is images that are sole children and found inside hyperlinks, you just modify the selector like so (which is illustrated in Figure 3-2):

```
a[href] img:only-child {border: 2px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><img src="w3.png" alt=""> The W3C</a>
<a href="http://w3.org/"><img src="w3.png" alt=""> <em>The W3C</em></a>
```

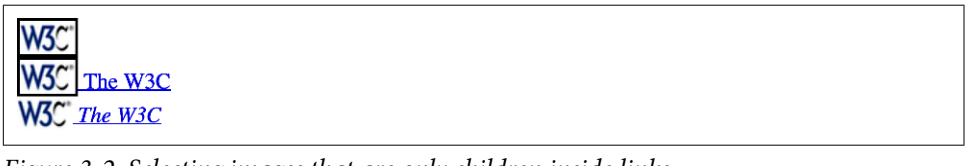


Figure 3-2. Selecting images that are only children inside links

You should remember two things about `:only-child`. The first is that you *always* apply it to the element you want to be an only child, not to the parent element, as explained earlier. That brings up the second thing to remember, which is that when you use `:only-child` in a descendant selector, you aren't restricting the elements listed to a parent-child relationship.

To go back to the hyperlinked-image example, `a[href] img:only-child` matches any image that is an only child and is descended from an `a` element, whether or not it's a *child* of an `a` element. To match, the element image must be the only child of its direct parent and also a descendant of an `a` element with an `href` attribute, but that parent can itself be a descendant of the same `<a>` element. Therefore, all three of the images in the following would be matched, as shown in Figure 3-3:

```
a[href] img:only-child {border: 5px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><img src="w3.png" alt="W3C"></span></a>
<a href="http://w3.org/">A link to <span>the <img src="w3.png" alt="W3C">
   web</span> site</a>
```



Figure 3-3. Selecting images that are only children inside links, redux

In each case, the image is the only child element of its parent, and it is also descended from an `<a>` element. Thus, all three images are matched by the rule shown. If you want to restrict the rule so that it matches images that are the only children of `<a>` elements, you add the child combinator to yield `a[href] > img:only-child`. With that change, only the first of the three images shown in Figure 3-3 would be matched.

Using only-of-type selection

That's all great, but what if you want to match images that are the only images inside hyperlinks, but other elements may be in there with them? Consider the following:

```
<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
```

In this case, we have an `a` element that has two children: `<b>` and `<img>`. That image, no longer the only child of its parent (the hyperlink), can never be matched using `:only-child`. However, it *can* be matched using `:only-of-type`. This is illustrated in Figure 3-4:

```
a[href] img:only-of-type {border: 5px solid black;}

<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><b>•</b><img src="w3.png" alt="W3C"></span></a>
```

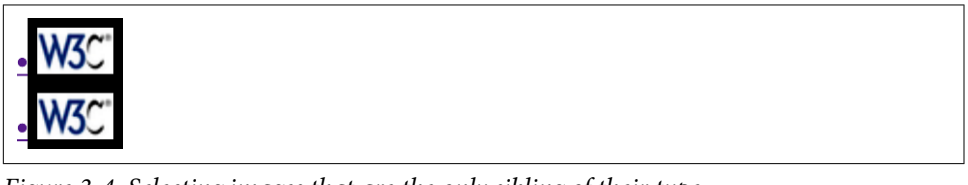


Figure 3-4. Selecting images that are the only sibling of their type

The difference is that `:only-of-type` will match any element that is the only one of its type among all its siblings, whereas `:only-child` will match only if an element has no siblings at all.

This can be very useful in cases such as selecting images within paragraphs without having to worry about the presence of hyperlinks or other inline elements:

```
p > img:only-of-type {float: right; margin: 20px;}
```

As long as there aren't multiple images that are children of the same paragraph, the image will be floated to the right.

You can also use this pseudo-class to apply extra styles to an <h2> when it's the only one in a given section of a document, like this:

```
section > h2 {margin: 1em 0 0.33em; font-size: 1.8rem; border-bottom: 1px solid
   gray;}
section > h2:only-of-type {font-size: 2.4rem;}
```

Given those rules, any <section> that has only one child <h2> will have that <h2> appear larger than usual. If a section has two or more <h2> children, neither will be larger than the other. The presence of other children—whether they are other heading levels, tables, paragraphs, lists, and so on—will not interfere with matching.

One more point to make clear is that :only-of-type refers to elements and nothing else. Consider the following:

```
p.unique:only-of-type {color: red;}

<div>
  <p class="unique">This paragraph has a 'unique' class.</p>
  <p>This paragraph doesn't have a class at all.</p>
</div>
```

In this case, neither of the paragraphs will be selected. Why not? Because two paragraphs are descendants of the , neither can be the only one of their type.

The class name is irrelevant here. We can be fooled into thinking that *type* is a generic description, because of the way we parse language. *Type*, in the way :only-of-type means it, refers only to the element type, as with type selectors. Thus, p.unique:only-of-type means, "Select any <p> element that is the only <p> element among its siblings if it also has a class of unique." It does *not* mean, "Select any <p> element whose class attribute contains the word unique when it's the only sibling paragraph to meet that criterion."

Selecting first children

It's pretty common to want to apply special styling to the first or last child of an element. A typical example is styling a bunch of navigation links in a tab bar and wanting to put special visual touches on the first or last tab (or both). If we didn't have structural selectors, this could be done by applying special classes to those elements. We have pseudo-classes to carry the load for us, removing the need to manually figure out which elements are the first and last.

The pseudo-class :first-child is used to select elements that are the first children of other elements. Consider the following markup:

```
<div>
  <p>These are the necessary steps:</p>
  <ul>
    <li>Insert key</li>
    <li>Turn key <strong>clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
```

```
    Do <em>not</em> push the brake at the same time as the accelerator.
  </p>
</div>
```

In this example, the elements that are first children are the first `<p>`, the first `<li>`, and the `<strong>` and `<em>` elements, which are all the first children of their respective parents. Given the following two rules,

```
p:first-child {font-weight: bold;}
li:first-child {text-transform: uppercase;}
```

we get the result shown in Figure 3-5.

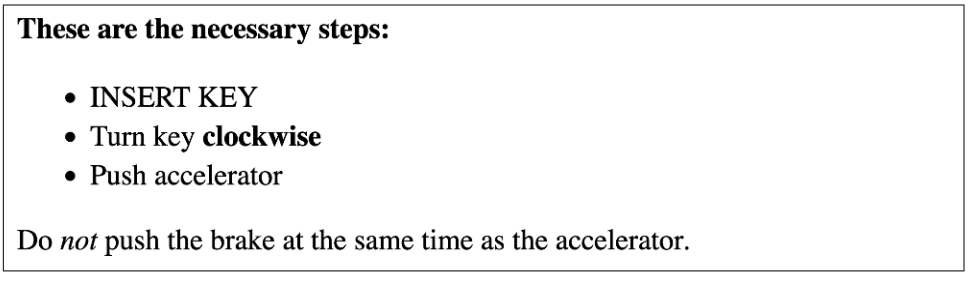


Figure 3-5. Styling first children

The first rule boldfaces any `<p>` element that is the first child of another element. The second rule uppercases any `<li>` element that is the first child of another element (which, in HTML, must be either an `<ol>` or `<ul>` element).

As has been mentioned, the most common error is assuming that a selector like `p:first-child` will select the first child of a `<p>` element. Remember the nature of pseudo-classes, which is to attach a sort of phantom class to the anchor element, the element associated with the pseudo-class. If you were to add actual classes to the markup, it would look like this:

```
<div>
  <p class="first-child">These are the necessary steps:</p>
  <ul>
    <li class="first-child">Insert key</li>
    <li>Turn key <strong class="first-child">clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
    Do <em class="first-child">not</em> push the brake at the same time as the
  accelerator.
  </p>
</div>
```

Therefore, if you want to select those `<em>` elements that are the first child of another element, you write `em:first-child`.

Selecting last children

The mirror image of `:first-child` is `:last-child`. If we take the previous example and just change the pseudo-classes, we get the result shown in Figure 3-6:

```
p:last-child {font-weight: bold;}
li:last-child {text-transform: uppercase;}

<div>
  <p>These are the necessary steps:</p>
  <ul>
    <li>Insert key</li>
    <li>Turn key <strong>clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
    Do <em>not</em> push the brake at the same time as the accelerator.
  </p>
</div>
```

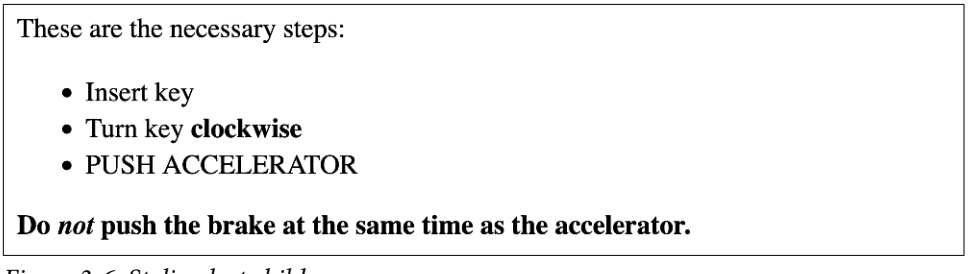


Figure 3-6. Styling last children

The first rule boldfaces any `<p>` element that is the last child of another element. The second rule uppercases any `<li>` element that is the last child of another element. If you wanted to select the `<em>` element inside that last paragraph, you could use the selector `p:last-child em`, which selects any `<em>` element that descends from a `<p>` element that is itself the last child of another element.

Interestingly, you can combine these two pseudo-classes to create a version of `:only-child`. The following two rules will select the same elements:

```
p:only-child {color: red;}
p:first-child:last-child {background-color: red;}
```

Either way, we get paragraphs with red foreground and background colors (not a good idea, to be clear).

Selecting the first and last of a type

In a manner similar to selecting the first and last children of an element, you can select the first or last of a type of element within another element. This permits actions like selecting the first `<table>` inside a given element, regardless of whatever elements come before it:

```
table:first-of-type {border-top: 2px solid gray;}
```

Note that this does *not* apply to the entire document; the rule shown will not select the first table in the document and skip all the others. It will instead select the first `<table>` element within each element that contains one, and skip any sibling `<table>` elements that come after the first. Thus, given the document structure shown in Figure 3-7, the circled nodes are the ones that are selected.

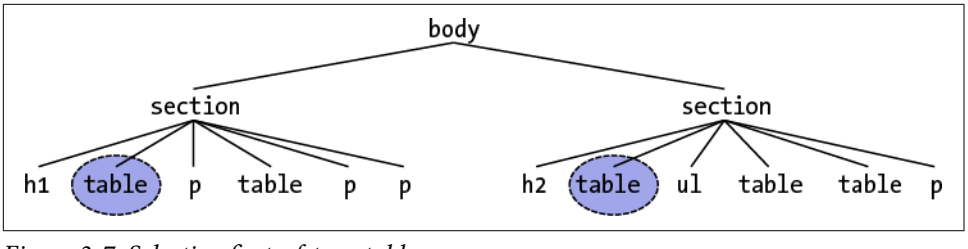


Figure 3-7. Selecting first-of-type tables

Within the context of tables, a useful way to select the first data cell within a row regardless of whether a header cell comes before it in the row is as follows:

```
td:first-of-type {border-left: 1px solid red;}
```

That would select the first data cell in each of the following table rows (that is, the cells containing `7` and `R`):

```
<tr>
  <th scope="row">Count</th><td>7</td><td>6</td><td>11</td>
</tr>
<tr>
  <td>R</td><td>X</td><td>-</td>
</tr>
```

Compare that to the effects of `td:first-child`, which would select the first `<td>` element in the second row, but not in the first row.

The flip side is `:last-of-type`, which selects the last instance of a given type from among its sibling elements. In a way, it's just like `:first-of-type`, except you start with the last element in a group of siblings and walk backward toward the first element until you reach an instance of the type. Given the document structure shown in Figure 3-8, the circled nodes are the ones selected by `table:last-of-type`.

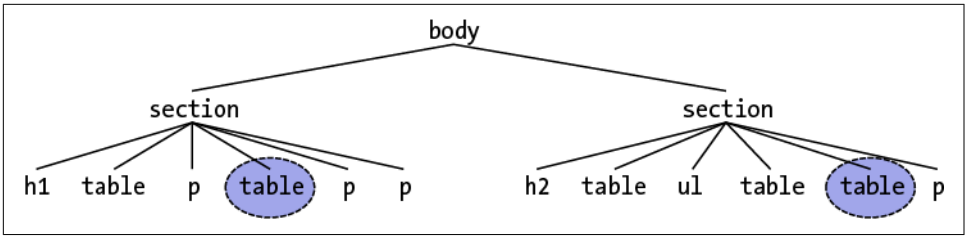


Figure 3-8. Selecting last-of-type tables

As was noted with `:only-of-type`, remember that you are selecting elements of a type from among their sibling elements; thus, every set of siblings is considered separately. In other words, you are *not* selecting the first (or last) of all the elements of a type within the entire document as a single group. Each set of elements that shares a parent is its own group, and you can select the first (or last) of a type within each group.

Similar to what was noted in the previous section, you can combine these two pseudo-classes to create a version of `:only-of-type`. The following two rules will select the same elements:

```
table:only-of-type{color: red;}
table:first-of-type:last-of-type {background: red;}
```

Selecting every nth child

If you can select elements that are the first, last, or only children of other elements, how about every third child? All even children? Only the ninth child? Rather than try to define a literally infinite number of named pseudo-classes, CSS has the `:nth-child()` pseudo-class. By filling integers or even basic algebraic expressions into the parentheses, you can select any arbitrarily numbered child element you like.

Let's start with the `:nth-child()` equivalent of `:first-child`, which is `:nth-child(1)`. In the following example, the selected elements will be the first paragraph and the first list item:

```
p:nth-child(1) {font-weight: bold;}
li:nth-child(1) {text-transform: uppercase;}

<div>
  <p>These are the necessary steps:</p>
  <ul>
    <li>Insert key</li>
    <li>Turn key <strong>clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
    Do <em>not</em> push the brake at the same time as the accelerator.
  </p>
</div>
```

If we change the numbers from `1` to `2`, however, then no paragraphs will be selected, and the middle (or second) list item will be selected, as illustrated in Figure 3-9:

```
p:nth-child(2) {font-weight: bold;}
li:nth-child(2) {text-transform: uppercase;}
```

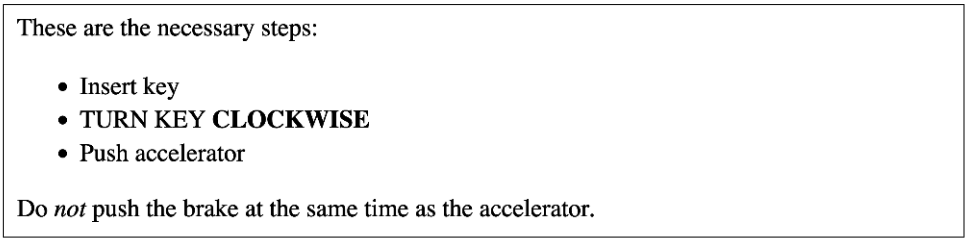


Figure 3-9. Styling second children

You can insert any integer you choose. If you have a use case for selecting any ordered list that is the 93rd child element of its parent, `ol:nth-child(93)` is ready to serve. This will match the 93rd child of any parent as long as that child is an ordered list. (This does not mean the 93rd ordered list among its siblings; see "Selecting every nth of a type" on page 66 for that.)

Is there a reason to use `:nth-child(1)` rather than `:first-child`? No. In this case, use whichever you prefer. There is literally no difference between them.

More powerfully, you can use simple algebraic expressions in the form *a*`n` + *b* or *a*`n` - *b* to define recurring instances, where *a* and *b* are integers and `n` is present as itself. Furthermore, the + *b* or – *b* part is optional and thus can be dropped if it isn't needed.

Let's suppose we want to select every third list item in an unordered list, starting with the first. The following makes that possible, selecting the first and fourth items, as shown in Figure 3-10:

```
ul > li:nth-child(3n + 1) {text-transform: uppercase;}
```

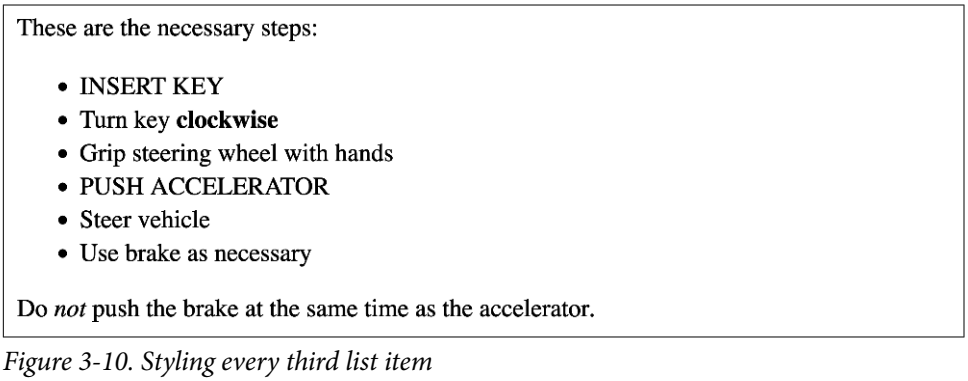


Figure 3-10. Styling every third list item

The way this works is that `n` represents the series 0, 1, 2, 3, 4, and on into infinity. The browser then solves for `3n + 1`, yielding 1, 4, 7, 10, 13, and so on. Were we to drop the `+ 1`, thus leaving us with simply `3n`, the results would be 0, 3, 6, 9, 12, and so on. Since there is no zeroth list item—all element counting starts with 1, to the likely chagrin of array-slingers everywhere—the first list item selected by this expression would be the third list item in the list.

Given that element counting starts with 1, it is a minor trick to deduce that `:nth-child(2n)` will select even-numbered children, and either `:nth-child(2n+1)` or `:nth-child(2n-1)` will select odd-numbered children. You can commit that to memory, or you can use the two special keywords that `:nth-child()` accepts: `even` and `odd`. Want to highlight every other row of a table, starting with the first? Here's how you do it, with the results shown in Figure 3-11:

```
tr:nth-child(odd) {background: silver;}
```

Montana	MT	Helena	Western Meadowlark
Nebraska	NE	Lincoln	Western Meadowlark
Nevada	NV	Carson City	Mountain Bluebird
New Hampshire	NH	Concord	Purple Finch
New Jersey	NJ	Trenton	Eastern Goldfinch
New Mexico	NM	Santa Fe	Roadrunner
New York	NY	Albany	Eastern Bluebird
North Carolina	NC	Raleigh	Northern Cardinal
North Dakota	ND	Bismarck	Western Meadowlark
Ohio	OH	Columbus	Northern Cardinal
Oklahoma	OK	Oklahoma City	Scissor-Tailed Flycatcher
Oregon	OR	Salem	Western Meadowlark
Pennsylvania	PA	Harrisburg	Ruffed Grouse
Rhode Island	RI	Providence	Rhode Island Red Chicken

Figure 3-11. Styling every other table row

Anything more complex than every-other-element requires an *an* + *b* expression.

Note that when you want to use a negative number for *b*, you have to remove the + sign, or else the selector will fail entirely. Of the following two rules, only the first will do anything. The second will be dropped by the parser and the entire declaration block will be ignored:

```
tr:nth-child(4n - 2) {background: silver;}
tr:nth-child(3n + -2) {background: red;}  /* INVALID */
```

You can also use a negative value for *a* in the expression, which will effectively count backward from the term you use in *b*. Selecting the first five list items in a list can be done like this:

```
li:nth-child(-n + 5) {font-weight: bold;}
```

This works because negative `n` goes 0, –1, –2, –3, –4, and so on. Add 5 to each of those, and you get 5, 4, 3, 2, 1, and so on. Put in a negative number for a multiplier on `n`, and you can get every second, third, or whatever-number-you-want element, like so:

```
li:nth-child(-2n + 10) {font-weight: bold;}
```

That will select the 10th, 8th, 4th, and 2nd list items in a list.

As you might expect, a corresponding pseudo-class is `:nth-last-child()`. This lets you do the same thing as `:nth-child()`, except with `:nth-last-child()` you start from the last element in a list of siblings and count backward toward the beginning. If you're intent on highlighting every other table row *and* making sure the very last row is one of the rows in the highlighting pattern, either one of these will work for you:

```
tr:nth-last-child(odd) {background: silver;}
tr:nth-last-child(2n+1) {background: silver;} /* equivalent */
```

If the Document Object Model (DOM) is updated to add or remove table rows, there is no need to add or remove classes. By using structural selectors, these selectors will always match the odd rows of the updated DOM.

Any element can be matched using both `:nth-child()` and `:nth-last-child()` if it fits the criteria. Consider these rules, the results of which are shown in Figure 3-12:

```
li:nth-child(3n + 3) {border-left: 5px solid black;}
li:nth-last-child(4n - 1) {border-right: 5px solid black; background: silver;}
```

Again, using negative terms for *a* will essentially count backward, except since this pseudo-class is already counting from the end, a negative term counts forward. That is to say, you can select the last five list items in a list like so:

```
li:nth-last-child(-n + 5) {font-weight: bold;}
```



An extension of `:nth-child()` and `:nth-last-child()` allows selecting from among elements matched by a simple or compound selector; for example, `:nth-child(2n + 1 of p.callout)`. As of early 2023, this is supported in Safari and Chrome beta releases. With it being included in Interop 2023, there are plans to have it fully supported in the near future.

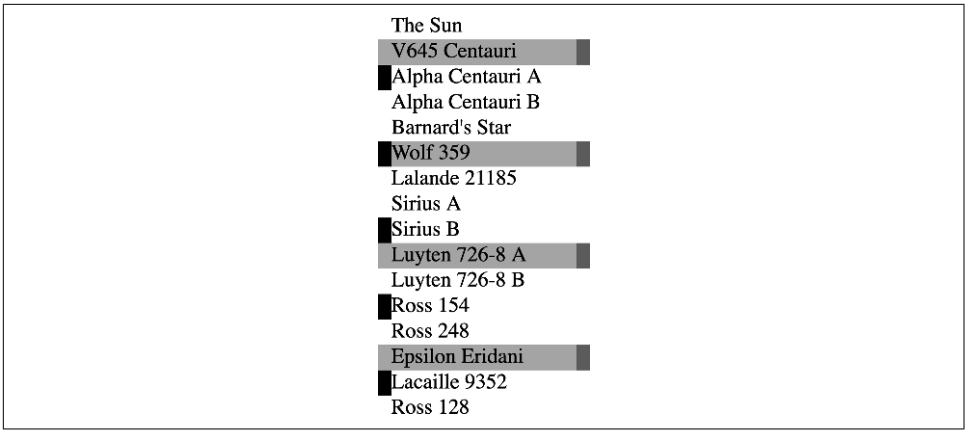


Figure 3-12. Combining patterns of `:nth-child()` and `:nth-last-child()`

You also can string these two pseudo-classes together as `:nth-child(1):nth-last-child(1)`, thus creating a more verbose restatement of `:only-child`. There's no real reason to do so other than to create a selector with a higher specificity (discussed in Chapter 4), but the option is there.

You can use CSS to determine the number of items in a list and style them accordingly:

```
li:only-child {width: 100%;}
li:nth-child(1):nth-last-child(2),
li:nth-child(2):nth-last-child(1) {width: 50%;}
li:nth-child(1):nth-last-child(3),
li:nth-child(1):nth-last-child(3) ~ li {width: 33.33%;}
li:nth-child(1):nth-last-child(4),
li:nth-child(1):nth-last-child(4) ~ li {width: 25%;}
```

In these examples, if a list item is the only list item, the width is 100%. If a list item is the first item as well as the second-from-the-last item, that means there are two items, and the width is 50%. If an item is the first item as well as the third-from-the-last item, we make it, and the two sibling list items following it, 33% wide. Similarly, if a list item is the first item as well as the fourth-from-the-last item, it means that there are exactly four items, so we make it, and its three siblings, 25% of the width. (Note: this sort of thing is a lot easier with the `:has()` pseudo-class, covered in "The :has() Pseudo-Class" on page 89.)

Selecting every nth of a type

In what may have become a familiar pattern, the `:nth-child()` and `:nth-last-child()` pseudo-classes have analogues in `:nth-of-type()` and `:nth-last-of-type()`. You can, for example, select every other hyperlink that's a child of any given paragraph, starting with the second, using `p > a:nth-of-type(even)`. This will ignore all other elements (`<span>`, `<strong>`, etc.) and consider only the links, as demonstrated in Figure 3-13:

```
p > a:nth-of-type(even) {background: blue; color: white;}
```

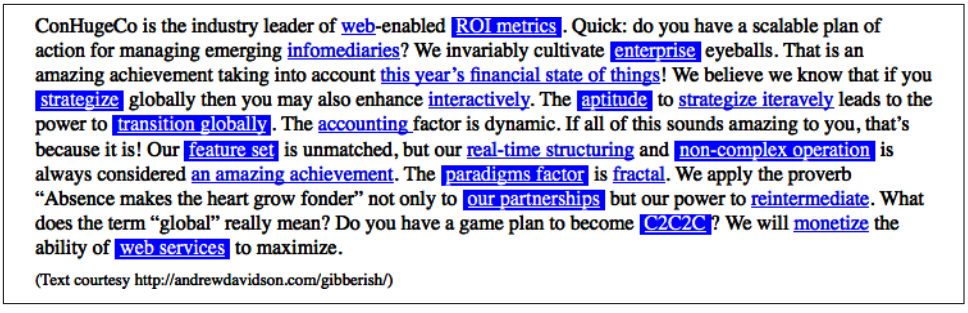


Figure 3-13. Selecting the even-numbered links

If you want to work from the last hyperlink backward, then you'd use `p > a:nth-last-of-type(even)`.

As before, these pseudo-classes select elements of a type from among their sibling elements, *not* from among all the elements of a type within the entire document as a single group. Each element has its own list of siblings, and selections happen within each group.

The difference between `:nth-of-type()` and `nth-child()` is that `:nth-of-type()` counts the instances of whatever you're selecting, and does its counting within that collection of elements. Take, for example, the following markup:

```
<tr>
   <th scope="row">Count</th>
   <td>7</td>
   <td>6</td>
   <td>11</td>
   <td>17</td>
   <td>3</td>
   <td>21</td>
</tr>
<tr>
   <td>R</td>
   <td>X</td>
   <td>-</td>
   <td>C</td>
   <td>%</td>
   <td>A</td>
   <td>I</td>
</tr>
```

If you wanted to select every table cell in a row when it's in an even-numbered column, you would use `td:nth-child(even)`. But if you want to select every even-numbered instance of a table cell, that would be `td:nth-of-type(even)`. You can see the difference in Figure 3-14, which shows the result of the following CSS:

```
td:nth-child(even) {background: silver;}
td:nth-of-type(even) {text-decoration: underline;}
```

Count	7	6	11	17	3	21
R	X	-	C	%	A	I

Figure 3-14. Selecting both `nth-child` and `nth-of-type` table cells

In the first row, every other table data cell (`td`) is selected, starting with the first cell that comes after the table header cell (`th`). In the second row, since all the cells are `td` cells, that means all the cells in that row are of the same type and thus the counting starts at the first cell.

As you might expect, you can use `:nth-of-type(1):nth-last-of-type(1)` together to restate `:only-of-type`, only with higher specificity. (We *will* explain specificity in Chapter 4, we promise.)

Location Pseudo-Classes

With the *location pseudo-classes*, we cross into the territory of selectors that match pieces of a document based on something in addition to the structure of the document—something that cannot be precisely deduced simply by studying the document's markup.

This may sound like we're applying styles at random, but not so. Instead, we're applying styles based on somewhat ephemeral conditions that can't be predicted in advance. Nevertheless, the circumstances under which the styles will appear are, in fact, well-defined.

Think of it this way: during a sporting event, whenever the home team scores, the crowd will cheer. You don't know exactly when during a game the team will score, but when it does, the crowd will cheer, just as predicted. The fact that you can't predict the exact moment of the cheer doesn't make it any less expected.

Now consider the anchor element (`<a>`), which (in HTML and related languages) establishes a link from one document to another. Anchors are always anchors, but some anchors refer to pages that have already been visited, while others refer to pages that have yet to be visited. You can't tell the difference by simply looking at the HTML markup, because in the markup, all anchors look the same.

The only way to tell which links have been visited is by comparing the links in a document to the user's browser history. So there are actually two basic types of links: visited and unvisited.

Hyperlink-specific pseudo-classes

CSS defines a few pseudo-classes that apply only to hyperlinks. In HTML, hyperlinks are any `<a>` elements with an `href` attribute; in XML languages, a hyperlink is any element that acts as a link to another resource. Table 3-1 describes the pseudo-classes you can apply to them.

Table 3-1. Link pseudo-classes

Name	Description
`:link`	Refers to any anchor that is a hyperlink (i.e., has an `href` attribute) and points to an address that has not been visited.
`:visited`	Refers to any anchor that is a hyperlink to an already visited address. For security reasons, the styles that can be applied to visited links are severely limited; see "Visited Links and Privacy" on page 70 for details.
`:any-link`	Refers to any element that would be matched by either `:link` or `:visited`.
`:local-link`	Refers to any link that points at the same URL as the page being styled. One example would be skip-links within a document. *Note: not supported as of early 2023.*

The first of the pseudo-classes in Table 3-1 may seem a bit redundant. After all, if an anchor hasn't been visited, it must be unvisited, right? If that's the case, all we should need is the following:

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

Although this format seems reasonable, it's not quite enough. The first of the rules shown here applies not only to unvisited links, but also to any `<a>` element, even those without an `href` attribute such as this one:

```
<a id="section004">4. The Lives of Meerkats</a>
```

The resulting text would be blue, because the `<a>` element will match the rule `a {color: blue;}`. Therefore, to avoid applying your link styles to placeholder links, use the `:link` and `:visited` pseudo-classes:

```
a:link {color: blue;}    /* unvisited links are blue */
a:visited {color: red;}   /* visited links are red */
```

This is a good place to revisit attribute and class selectors and show how they can be combined with pseudo-classes. For example, let's say you want to change the color of links that point outside your own site. In most circumstances, we can use the starts-with attribute selector. However, some CMSs set all links to be absolute URLs, in which case you could assign a class to each of these anchors. It's easy:

```
<a href="/about.html">My About page</a>
<a href="https://www.site.net/" class="external">An external site</a>
```

To apply different styles to the external link, all you need is a rule like this:

```
a.external:link, a[href^="http"]:link { color: slateblue;}
a.external:visited, a[href^="http"]:visited  {color: maroon;}
```

This rule will make the second anchor in the preceding markup slate blue by default and maroon once visited, while the first anchor will remain the default color for hyperlinks (usually blue when not visited and purple once visited). For improved usability and accessibility, visited links should be easily distinguished from unvisited links.



Styled visited links enable visitors to know where they have been and what they have yet to visit. This is especially important on large websites, where it may be difficult to remember which pages have been visited, especially for those with cognitive disabilities. Not only is highlighting visited links one of the W3C Web Content Accessibility Guidelines, but it makes searching for content faster, more efficient, and less stressful for everyone.

The same general syntax is used for ID selectors as well:

```
a#footer-copyright:link {background: yellow;}
a#footer-copyright:visited {background: gray;}
```

If you want to select all links, regardless of whether they're visited or not, use `:any-link`:

```
a#footer-copyright:any-link {text-decoration: underline;}
```

Visited Links and Privacy

For well over a decade, it was possible to style visited links with any CSS properties available, just as you could unvisited links. However, in the mid-2000s several people demonstrated that visual styling and simple DOM scripting could be used to determine if a user had visited a given page.

For example, given the rule `:visited {font-weight: bold;}`, a script could find all of the boldfaced links and tell the user which of those sites they'd visited—or, worse still, report those sites back to a server. A similar, nonscripted tactic uses background images to achieve the same result.

While this might not seem terribly serious, it can be utterly devastating for a web user in a country that jails people for visiting certain sites—opposition parties, unsanctioned religious organizations, "immoral" or "corrupting" sites, and so on. These techniques can also be used by phishing sites to determine which online banks a user has visited. Thus, two steps were taken.

The first step is that only color-related properties can be applied to visited links: `color`, `background-color`, `column-rule-color`, `outline-color`, `border-color`, and the individual-side border color properties (e.g., `border-top-color`). Attempts to apply any other property to a visited link will be ignored. Furthermore, any styles defined for `:link` will be applied to visited links as well as unvisited links, which effectively makes `:link` "style any hyperlink," instead of "style any unvisited hyperlink."

The second step is that if a visited link has its styles queried via the DOM, the resulting value will be as if the link were not visited. Thus, if you've defined visited links to be purple rather than unvisited links' blue, even though the link will appear purple onscreen, a DOM query of its color will return the blue value, not the purple one.

This behavior is present throughout all browsing modes, not just "private browsing" modes. Even though we're limited in how we can use CSS to differentiate visited links from unvisited links, it is important for usability and accessibility to use the limited styles supported by visited links to differentiate them from unvisited links.

Nonhyperlink location pseudo-classes

Hyperlinks aren't the only elements that can be related to location. CSS also provides a few pseudo-classes that relate to the targets of hyperlinks, summarized in Table 3-2.

Table 3-2. Nonlink location pseudo-classes

Name	Description
`:target`	Refers to an element whose `id` attribute value matches the fragment selector in the URL used to load the page—that is, the element specifically targeted by the URL.
`:target-within`	Refers to an element that is the target of the URL, or that contains an element that is so targeted. *Note: not supported as of early 2023.*
`:scope`	Refers to elements that are a reference point for selector matching.

Let's talk about target selection. When a URL includes a fragment identifier, the piece of the document at which it points is called (in CSS) the *target*. Thus, you can uniquely style any element that is the target of a URL fragment identifier with the `:target` pseudo-class.

Even if you're unfamiliar with the term *fragment identifier*, you've probably seen them in action. Consider this URL:

```
http://www.w3.org/TR/css3-selectors/#target-pseudo
```

The `target-pseudo` portion of the URL is the fragment identifier, which is marked by the `#` symbol. If the referenced page (*http://www.w3.org/TR/css3-selectors/*) has an element with an ID of `target-pseudo`, that element becomes the target of the fragment identifier.

Thanks to `:target`, you can highlight any targeted element within a document, or you can devise different styles for various types of elements that might be targeted—say, one style for targeted headings, another for targeted tables, and so on. Figure 3-15 shows an example of `:target` in action:

```
*:target {border-left: 5px solid gray; background: yellow url(target.png)
    top right no-repeat;}
```

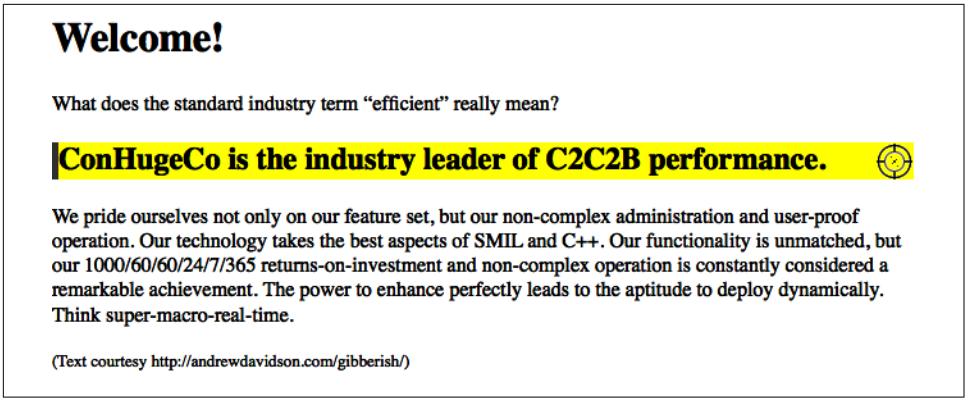


Figure 3-15. Styling a fragment identifier target

The `:target` styles will not be applied in three circumstances:

- The page is accessed via a URL that does not have a fragment identifier.
- The page is accessed via a URL that has a fragment identifier, but the identifier does not match any elements within the document.
- The page's URL is updated in such a way that a scroll state is not created, which happens most often via JS shenanigans. (This isn't a CSS rule, but it is how browsers behave.)

More interestingly, though, what happens if multiple elements within a document can be matched by the fragment identifier—for example, if the author erroneously includes three separate instances of `<div id="target-pseudo">` in the same document?

The short answer is that CSS doesn't have or need rules to cover this case, because all CSS is concerned with is styling targets. Whether the browser picks just one of the three elements to be the target or designates all three as coequal targets, `:target` styles should be applied to anything that is a valid target.

Closely related to the `:target` pseudo-class is the `:target-within` pseudo-class. The difference is that `:target-within` will match not only elements that are targets, but also elements that are the ancestors of targets. Thus, the following CSS would match any `<p>` element containing a target, or that was itself a target:

```
p:target-within {border-left: 5px solid gray; background: yellow url(target.png)
    top right no-repeat;}
```

Or it would, anyway, if any browser supported it. As of early 2023, this is not the case.

Finally, we consider the `:scope` pseudo-class. This is quite widely supported, but at present, it comes in handy only in scripting situations. Consider the following JS and HTML, which we'll explain after the code:

```
var output = document.getElementById('output');
var registers = output.querySelectorAll(':scope > div');
```

```
<section id="output">
  <h3>Results</h3>
  <div></div>
  <div></div>
</section>
```

The JS portion says, in effect, "Find the element with an ID of `output`. Then, find all the `<div>` elements that are children of the `output` element you just found." (Yes, CSS selectors can be used in JS!) The `:scope` in that bit of JS refers to the scope of the thing that had been found, thus keeping the selection confined to just that instead of the whole document. The result is that, in the JS program's memory, it now has a structure holding references to the two `<div>` elements in the HTML.

If you use `:scope` in straight CSS, it will refer to the *scoping root*, which (at present) means the `<html>` element, assuming the document is HTML. Neither HTML nor CSS provides a way to set scoping roots other than the root element of the document. So, outside of JS, `:scope` is essentially equivalent to `:root`. That may change in the future, but for now, you should use `:scope` only in JS contexts.

JS and CSS

CSS has influenced the evolution of JS in a few ways, and one is the ability to use the CSS selection engine from within JS via `.querySelectorAll()`. This method can take any CSS selector as a string, and will return a collection of all the elements within the DOM that are matched by the selector. The `.querySelector()` method also accepts any CSS selector as a string, but will return only the first element found, so it's not always as useful.

You may come across older JS methods for collecting elements, such as `.getElementByID()` and `.getElementsByTagName()`. These are from the time before `.querySelectorAll()` was added to JS, and while they may be marginally more performant than `.querySelectorAll()` in some situations, they're mostly found in legacy codebases these days. Both are now more simply handled with `.querySelectorAll()`. For example, the following two lines would have almost the same result:

```
var subheads = Document.getElementsByTagName('h2');
var subheads = Document.querySelectorAll('h2');
```

Similarly, a `.getElementById('summary')` can be equivalently replaced with `.querySelectorAll('#summary')`.

The advantage in `.querySelectorAll()` is that it can take any selector, no matter how complex, including grouped selectors. Thus, you could get all of the level-two and -three headings in a single call: `Document.querySelectorAll('h2, h3')`. Or you

could grab a more complex set of elements with something like `.querySelectorAll('h2 + p, pre + p, table + *, thead th:nth-child(even)')`.

Note, though, that the list of elements returned by `.querySelectorAll()` is static, and therefore is not updated when the DOM is dynamically changed. If another part of the JS adds a section with an `<h2>` element in it, the elements previously collected with `.querySelectorAll('h2, h3')` will *not* be updated to include the newly added `<h2>`. You'd need to either add it yourself manually or do a new `.querySelectorAll()` call.

User Action Pseudo-Classes

CSS defines a few pseudo-classes that can change a document's appearance based on actions taken by the user. These *dynamic pseudo-classes* have traditionally been used to style hyperlinks, but the possibilities are much wider. Pseudo-classes are described in Table 3-3.

Table 3-3. User action pseudo-classes

Name	Description
`:hover`	Refers to any element over which the mouse pointer is placed—e.g., a hyperlink over which the mouse pointer is hovering
`:active`	Refers to any element that has been activated by user input—e.g., a hyperlink on which a user clicks during the time the mouse button is held down, or an element a user has tapped via touch screen
`:focus`	Refers to any element that currently has the input focus—i.e., can accept keyboard input or otherwise be activated in some way
`:focus-within`	Refers to any element that currently has the input focus—i.e., can accept keyboard input or be activated in some way—or an element containing an element that is so focused
`:focus-visible`	Refers to any element that currently has the input focus, but only if the user agent thinks it is an element type that should have visible focus

Elements that can become `:active` or have `:focus` include links, buttons, menu items, any element with a `tabindex` value, and all other interactive elements, including form controls and elements containing content that can be edited (by having the attribute, `contenteditable`, added to the element's opening tag).

As with `:link` and `:visited`, these pseudo-classes are most familiar in the context of hyperlinks. Many web pages have styles that look like this:

```
a:link {color: navy;}
a:visited {color: gray;}
a:focus {color: orange;}
a:hover {color: red;}
a:active {color: yellow;}
```



The order of the pseudo-classes is more important than it might seem at first. The usual recommendation is `link`, `visited`, `focus`, `hover`, and `active`. The next chapter explains why this particular order is important and discusses several reasons you might choose to change or even ignore the recommendation.

Notice that the dynamic pseudo-classes can be applied to any element, which is good since it's often useful to apply dynamic styles to elements that aren't links. Consider this example:

```
input:focus {background: silver; font-weight: bold;}
```

By using this markup, you could highlight a form element that is ready to accept keyboard input, as shown in Figure 3-16.



Figure 3-16. Highlighting a form element that has focus

Two relatively new additions to the user-action pseudo-classes are `:focus-within` and `:focus-visible`. Let's take the second one first.

The :focus-visible pseudo-class

The `:focus-visible` class is very much like `:focus` in that it applies to elements that have focus, but there's a big difference: it will match only if the element that has focus is an element that the user agent thinks should be given visible focus styles in a given situation.

For example, consider HTML buttons. When a button is clicked via a mouse, that button is given focus, the same as if we had used a keyboard interface to move the focus to it. As authors who care about accessibility and aesthetics, we want the button to have focus when it's focused via the keyboard or another assistive technology, but we might not like it getting focus styles when it's clicked or tapped.

We can split this difference by using CSS such as the following:

```
button:focus-visible {outline: 5px solid maroon;}
```

This will put a thick dark-red outline around the button when tabbing to it via keyboard, but the rule won't be applied when the button is clicked with the mouse.

The :focus-within pseudo-class

Building on that, `:focus-within` applies to any element that has focus, or any element that has a descendant with focus. Given the following CSS and HTML, we'll get the result shown in Figure 3-17:

```
nav:focus-within {border: 3px solid silver;}
a:focus-visible {outline: 2px solid currentcolor;}

<nav>
  <a href="home.html">Home</a>
  <a href="about.html">About</a>
  <a href="contact.html">Contact</a>
</nav>
```

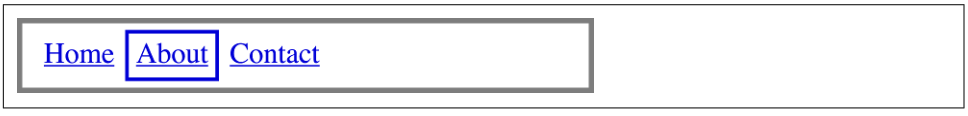


Figure 3-17. Selecting elements by using `:focus-within`

The third link currently has focus, having received it by the user tabbing to that link, and is styled with a 2-pixel outline. The `<nav>` element that contains it is also being given focus styling via `:focus-within`, because an element within itself (that is, an element descended from it) currently has focus. This adds a little more visual weight to that area of the page, which can be helpful. Be careful of overdoing it, though. Too many focus styles can create visual overload, potentially confusing users.



While you can style elements with `:focus` any way you like, do *not* remove all styling from focused elements. Differentiating which element currently has focus is vital for accessibility, especially for those navigating your site or application with a keyboard.

Real-world issues with dynamic styling

Dynamic pseudo-classes present some interesting issues and peculiarities. For example, you can set visited and unvisited links to one font size and make hovered links a larger size, as shown in Figure 3-18:

```
a:link, a:visited {font-size: 13px;}
a:hover, a:active {font-size: 20px;}
```



Figure 3-18. Changing layout with dynamic pseudo-classes

As you can see, the user agent increases the size of the anchor while the mouse pointer hovers over it—or, thanks to the `:active` setting, when a user touches it on a touch screen. Because we are changing a property that impacts line height, a user agent that supports this behavior must redraw the document while an anchor is in the hover state, which could force a reflow of all the content that follows the link.

UI-State Pseudo-Classes

Closely related to the dynamic pseudo-classes are the *user-interface (UI) state pseudo-classes*, which are summarized in Table 3-4. These pseudo-classes allow for styling based on the current state of UI elements such as checkboxes.

Table 3-4. UI-state pseudo-classes

Name	Description
`:enabled`	Refers to UI elements (such as form elements) that are enabled—that is, available for input
`:disabled`	Refers to UI elements (such as form elements) that are disabled—that is, not available for input
`:checked`	Refers to radio buttons or checkboxes that have been selected, either by the user or by defaults within the document itself
`:indeterminate`	Refers to radio buttons or checkboxes that are neither checked nor unchecked; this state can be set only via DOM scripting, and not by user input
`:default`	Refers to the radio button, checkbox, or option that was selected by default
`:autofill`	Refers to a user input that has been autofilled by the browser
`:placeholder-shown`	Refers to a user input that has placeholder (not value) text prefilled
`:valid`	Refers to a user input that meets all of its data validity requirements
`:invalid`	Refers to a user input that does not meet all of its data validity requirements
`:in-range`	Refers to a user input whose value is between the minimum and maximum values
`:out-of-range`	Refers to a user input whose value is below the minimum or above the maximum values allowed by the control
`:required`	Refers to a user input that must have a value set
`:optional`	Refers to a user input that does not need to have a value set
`:read-write`	Refers to a user input that is editable by the user
`:read-only`	Refers to a user input that is not editable by the user

Although the state of a UI element can certainly be changed by user action—for example, a user checking or unchecking a checkbox—UI-state pseudo-classes are not classified as purely dynamic because they can also be affected by the document structure or scripting.

Enabled and disabled UI elements

Thanks to both DOM scripting and HTML, you can mark a UI element (or group of UI elements) as being disabled. A disabled element is displayed but cannot be selected, activated, or otherwise interacted with by the user. Authors can set an element to be disabled either through DOM scripting or by adding a `disabled` attribute to the element's markup.

Any element that can be disabled, but hasn't been, is by definition enabled. You can style these two states by using the `:enabled` and `:disabled` pseudo-classes. It's much more common to style disabled elements and leave enabled elements alone, but both have their uses, as illustrated in Figure 3-19:

```
:enabled {font-weight: bold;}
:disabled {opacity: 0.5;}
```



Figure 3-19. Styling enabled and disabled UI elements

Check states

In addition to being enabled or disabled, certain UI elements can be checked or unchecked—in HTML, the input types `checkbox` and `radio` fit this definition. CSS offers a `:checked` pseudo-class to handle elements in that state. In addition, the `:indeterminate` pseudo-class matches any checkable UI element that is neither checked nor unchecked. These states are illustrated in Figure 3-20:

```
:checked {background: silver;}
:indeterminate {border: red;}
```

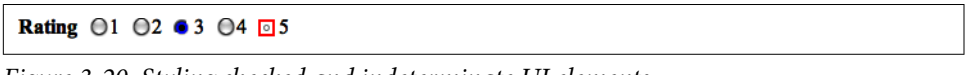


Figure 3-20. Styling checked and indeterminate UI elements

Although checkable elements are unchecked by default, an HTML author can toggle them on by adding the `checked` attribute to an element's markup. An author can also use DOM scripting to flip an element's checked state to checked or unchecked, whichever they prefer.

As of early 2023, the indeterminate state can be set only through DOM scripting or by the user agent itself; no markup-level method exists to set elements to an indeterminate state. The purpose of styling an indeterminate state is to visually indicate that the element needs to be checked (or unchecked) by the user. However, this is purely a visual effect: it does not affect the underlying state of the UI element, which is either checked or unchecked, depending on document markup and the effects of any DOM scripting.

Although the previous examples show styled radio buttons, remember that direct styling of radio buttons and checkboxes with CSS is actually very limited. Nevertheless, that shouldn't limit your use of the selected-option pseudo-classes. As an example, you can style the labels associated with your checkboxes and radio buttons by using a combination of `:checked` and the adjacent sibling combinator:

```
input[type="checkbox"]:checked + label {
  color: red;
  font-style: italic;
}

<input id="chbx" type="checkbox"> <label for="chbx">I am a label</label>
```

If you need to select all checkboxes that are not checked, use the negation pseudo-class (which is covered later in the chapter) like this: `input[type="checkbox"]:not(:checked)`. Only radio buttons and checkboxes can be checked. Note that every element, and these two when not checked, are `:not(:checked)`. This approach does not fill the gap left by the absence of an `:unchecked` pseudo-class, and such a pseudo-class should match only elements that should be checkable.

Default-value pseudo-classes

Three pseudo-classes relate to default values and filler text: `:default`, `:placeholder-shown`, and `:autofill`.

The `:default` pseudo-class matches the UI elements that are the default among a set of similar elements. This typically applies to context menu items, buttons, and select lists/menus. If there are several same-named radio buttons, the one that was originally selected (if any) matches `:default`, even if the UI has been updated by the user so that it no longer matches `:checked`. If a checkbox was checked on page load, `:default` matches it. Any initially selected option(s) in a `select` element will match:

```
[type="checkbox"]:default + label { font-style: italic; }

<input type="checkbox" id="chbx" checked name="foo" value="bar">
<label for="chbx">This was checked on page load</label>
```

The `:default` pseudo-class will also match a form's default button, which is generally the first `button` element in DOM order that is a member of a given form. This could be used to indicate to users which button will be activated if they just hit Enter, instead of explicitly selecting a button to activate.

The `:placeholder-shown` pseudo-class is similar in that it will select any input that has placeholder text defined at the markup level while that placeholder text is visible. The placeholder is no longer shown when the input has a value. For example:

```
<input type="text" id="firstName" placeholder="Your first name">
<input type="text" id="lastName"  placeholder="Your last name">
```

By default, the value of a `placeholder` attribute will be placed into the input fields in a browser, usually in a lighter color than normal text. If you want to style those input elements in a consistent way, you can do something like this:

```
input:placeholder-shown {opacity: 0.75;}
```

This selects the input as a whole, not the placeholder text. (To style the placeholder text itself, see "The Placeholder Text Pseudo-Element" on page 97.)

The `:autofill` pseudo-class is a little bit different from the other two: it matches any element that has had its value automatically filled in or auto-completed by the browser. This may be familiar to you if you've ever filled out a form by having the browser fill in stored values of your name, email, mailing address, and so on. The input fields that are filled in usually get a distinct style, like a yellowish background. You can add to that using `:autofill`, perhaps like this:

```
input:autofill {border: thick solid maroon;}
```



While you can add to default browser styling of autofilled text, overriding the browser's built-in styles for things such as background colors is difficult. This is because the browsers' styles for autofilled fields are set to override just about anything else, largely as a way to provide users with a consistent experience of autofilled content and to protect the user.

Optionality pseudo-classes

The `:required` pseudo-class matches any user-input element that is required, as denoted by the presence of the `required` attribute. The `:optional` pseudo-class matches user-input elements that do not have the `required` attribute, or whose `required` attribute has a value of `false`.

A user-input element is `:required` if the user must provide a value for before submitting the form to which it belongs. All other user-input elements are matched by `:optional`. For example:

```
input:required { border: 1px solid #f00;}
input:optional { border: 1px solid #ccc;}

<input type="email" placeholder="enter an email address" required>
<input type="email" placeholder="optional email address">
<input type="email" placeholder="optional email address" required="false">
```

The first email input will match the `:required` pseudo-class because of the presence of the `required` attribute. The second input is optional and therefore will match the `:optional` pseudo-class. The same is true for the third input, which has a `required` attribute, but the value is `false`.

Elements that are not user-input elements can be neither required nor optional. Including the `required` attribute on a non-user-input element won't lead to an optionality pseudo-class match.

Validity pseudo-classes

The `:valid` pseudo-class refers to a user input that meets all of its data validity requirements. The `:invalid` pseudo-class, on the other hand, refers to a user input that does not meet all of its data validity requirements.

The validity pseudo-classes `:valid` and `:invalid` apply only to elements having the capacity for data validity requirements: a `<div>` will never match either selector, but an `<input>` could match either, depending on the current state of the interface.

In the following example, an image is dropped into the background of any email input that has focus, with one image being used when the input is invalid and another used when the input is valid, as illustrated in Figure 3-21:

```
input[type="email"]:focus {
  background-position: 100% 50%;
  background-repeat: no-repeat;
}
input[type="email"]:focus:invalid {
  background-image: url(warning.jpg);
}
input[type="email"]:focus:valid {
  background-image: url(checkmark.jpg);
}

<input type="email">
```



Figure 3-21. Styling valid and invalid UI elements

Keep in mind that these pseudo-class states may not act as you might expect. For example, as of late 2022, any empty email input that isn't required matches `:valid`. Even though a null input is not a valid email address, failing to enter an email address is a valid response for an optional input. If you try to fill in a malformed address or just some random text, that will be matched by `:invalid` because it isn't a valid email address.

Range pseudo-classes

The range pseudo-classes include `:in-range`, which refers to a user input whose value is between the minimum and maximum values set by HTML's `min` and `max` attributes,

and `:out-of-range`, which refers to a user input whose value is below the minimum or above the maximum values allowed by the control.

For example, consider a number input that accepts numbers in the range 0 to 1,000:

```
input[type="number"]:focus {
  background-position: 100% 50%;
  background-repeat: no-repeat;
}
input[type="number"]:focus:out-of-range {
  background-image: url(warning.jpg);
}
input[type="number"]:focus:in-range {
  background-image: url(checkmark.jpg);
}

<input id="grams" type="number" min="0" max="1000" />
```

In this example, a value from 0 to 1,000, inclusive, would mean the `input` element is matched by `:in-range`. Any value outside that range, whether input by the user or assigned via the DOM, will cause the `input` to match `:out-of-range` instead.

The `:in-range` and `:out-of-range` pseudo-classes apply *only* to elements with range limitations. User inputs that don't have range limitations, like links for inputs of type `tel`, will not be matched by either pseudo-class.

HTML also has a `step` attribute. If a value is invalid because it does not match the `step` value, but is still between or equal to the `min` and `max` values, it will match `:invalid` while *also* still matching `:in-range`. A value can be in range while also being invalid.

Thus, in the following scenario, the input's value will be both red and boldfaced, because the value 23 is in range but is not evenly divisible by 10:

```
input[type="number"]:invalid {color: red;}
input[type="number"]:in-range {font-weight: bold;}

<input id="by-tens" type="number" min="0" max="1000" step="10" value="23" />
```

Mutability pseudo-classes

The mutability pseudo-classes include `:read-write`, which refers to a user input that is editable by the user; and `:read-only`, which matches user inputs that are not editable, including radio buttons and checkboxes. Only elements that can have their values altered by user input can match `:read-write`.

For example, in HTML, a nondisabled, non-read-only `input` element is `:read-write`, as is any element with the `contenteditable` attribute. Everything else matches `:read-only`.

By default, neither of the following rules would ever match, because `<textarea>` elements are read-write, and `<pre>` elements are read-only:

```
textarea:read-only {opacity: 0.75;}
pre:read-write:hover {border: 1px dashed green;}
```

However, each can be made to match as follows:

```
<textarea disabled></textarea>
<pre contenteditable>Type your own code!</pre>
```

Because `<textarea>` is given a `disabled` attribute, it becomes read-only, and so will have the first rule apply. Similarly, the `<pre>` here has been made editable via the `contenteditable` attribute, so now it is a read-write element. This will be matched by the second rule.

The :lang() and :dir() Pseudo-Classes

When you want to select an element based on its language, you can use the `:lang()` pseudo-class. In terms of its matching patterns, this pseudo-class is similar to the `|=` attribute selector (see "Selection Based on Partial Attribute Values" on page 37). For example, to italicize elements whose content is written in French, you could write either of the following:

```
*:lang(fr) {font-style: italic;}
*[lang|="fr"] {font-style: italic;}
```

The primary difference between the pseudo-class selector and the attribute selector is that language information can be derived from multiple sources, some of which are outside the element itself. For the attribute selector, the element must have the attribute present to match. The `:lang()` pseudo-class, on the other hand, matches descendants of an element with the language declaration. As Selectors Level 3 states:

> In HTML, the language is determined by a combination of the `lang` attribute, and possibly information from the `meta` elements and the protocol (such as HTTP headers). XML uses an attribute called `xml:lang`, and there may be other document language-specific methods for determining the language.

The pseudo-class will operate on all of that information, whereas the attribute selector can work only if a `lang` attribute is present in the element's markup. Therefore, the pseudo-class is more robust than the attribute selector and is probably a better choice in most cases where language-specific styling is needed.

CSS also has a `:dir()` pseudo-class, which selects elements based on the HTML direction of an element. So you could, for example, select all the elements whose direction is right to left like this:

```
*:dir(rtl) {border-right: 2px solid;}
```

The thing to watch out for here is that the `:dir()` pseudo-class selects elements based on their directionality in HTML, and not the value of the `direction` property in CSS that may be applied to them. Thus, the only two values you can really use for selection are `ltr` (left to right) and `rtl` (right to left) because those are the only direction values that HTML supports.

Logical Pseudo-Classes

Beyond structure and language, some pseudo-classes are intended to bring a touch of logic and flexibility to CSS selectors.

The negation pseudo-class

Every selector we've covered thus far has had one thing in common: they're all positive selectors. They are used to identify the things that should be selected, thus excluding by implication all the things that don't match and are thus not selected.

For those times when you want to invert this formulation and select elements based on what they are *not*, CSS provides the negation pseudo-class, `:not()`. It's not quite like any other selector, fittingly enough, and it does have some restrictions on its use, but let's start with an example.

Let us suppose you want to apply a style to every list item that does not have a `class` of `moreinfo`, as illustrated in Figure 3-22. That used to be very difficult, and in certain cases impossible, to make happen. Now we can declare the following:

```
li:not(.moreinfo) {font-style: italic;}
```

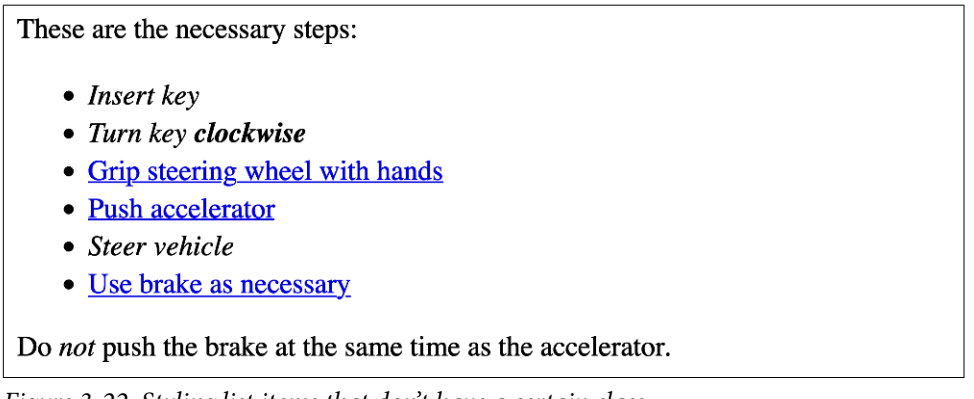


Figure 3-22. Styling list items that don't have a certain class

The way `:not()` works is that you attach it to a selector, and then in the parentheses you fill in a selector or group of selectors describing what the original selector cannot match.

Let's flip around the previous example and select all elements with a class of `moreinfo` that are not list items. This is illustrated in Figure 3-23:

```
.moreinfo:not(li) {font-style: italic;}
```

These are the necessary steps:

- Insert key
- Turn key **clockwise**
- Grip steering wheel with hands

Do *not* push the brake at the same time as the accelerator. Doing so can cause what *computer scientists* might term a "*race condition*" except you won't be racing so much as burning out the engine. This can cause a fire, lead to *a traffic accident*, or worse.

Figure 3-23. Styling elements with a certain class that aren't list items

Translated into English, the selector would say, "Select all elements with a class whose value contains the word `moreinfo` as long as they are not `<li>` elements." Similarly, the translation of `li:not(.moreinfo)` would be, "Select all `<li>` elements as long as they do not have a class whose value contains the word `moreinfo`."

You can also use the negation pseudo-class at any point in a more complex selector. Thus, to select all tables that are not children of a `<section>` element, you would write `*:not(section) > table`. Similarly, to select table header cells that are not part of the table header, you'd write something like `table *:not(thead) > tr > th`, with a result like that shown in Figure 3-24.

State	Post	Capital	State Bird
Alabama	AL	Montgomery	Yellowhammer
Alaska	AK	Juneau	Willow Ptarmigan
Arizona	AZ	Phoenix	Cactus Wren
Arkansas	AR	Little Rock	Mockingbird
California	CA	Sacramento	California Quail
Colorado	CO	Denver	Lark Bunting
Connecticut	CT	Hartford	American Robin
Delaware	DE	Dover	Blue Hen Chicken
Florida	FL	Tallahassee	Northern Mockingbird
Georgia	GA	Atlanta	Brown Thrasher
State	**Post**	**Capital**	**State Bird**

Figure 3-24. Styling header cells outside the table's head area

What you cannot do is nest negation pseudo-classes; thus, `p:not(:not(p))` is not valid and will be ignored. It's also, logically, the equivalent of just writing `p`, so there's no point anyway. Furthermore, you cannot reference pseudo-elements (which we'll cover shortly) inside the parentheses.

Technically, you can put a universal selector into the parentheses, but there's little point. After all, `p:not(*)` would mean "select any `<p>` element as long as it isn't any element," and there's no such thing as an element that is not an element. Similarly, `p:not(p)` would also select nothing. It's also possible to write things like `p:not(div)`, which will select any

<p> element that is not a element—in other words, all of them. Again, there is little reason to do this.

On the other hand, it's possible to chain negations together to create a sort of "and also not this" effect. For example, you might want to select all elements with a class of `link` that are neither list items nor paragraphs:

```
*.link:not(li):not(p) {font-style: italic;}
```

That translates to "select all elements with a class whose value contains the word `link` as long as they are neither <li> nor <p> elements." This used to be the only way to exclude a group of elements, but CSS (and browsers) support selector lists in negations. That allows us to rewrite the previous example like so:

```
*.link:not(li, p) {font-style: italic;}
```

Along with this came the ability to use more complex selectors, such as those using descendant combinators. If you need to select all elements that are descended from a <form> element but do not immediately follow a <p> element, you could write it as follows:

```
form *:not(p + *)
```

Translated, that's "select any element that is not the adjacent sibling <p> element, and is also the descendant of a <form> element." And you can put these into groups, so if you also want to exclude list items and table-header cells, it would go something like this:

```
form *:not(p + *, li, thead > tr > th)
```



The ability to use complex selectors in `:not()` came to browsers in only early 2021, so exercise caution when using it, especially in legacy settings.

One thing to watch out for with `:not()` is that in some situations rules can combine in unexpected ways, mostly because we're not used to thinking of selection in the negative. Consider this test case:

```
div:not(.one) p {font-weight: normal;}
div.one p {font-weight: bold;}

<div class="one">
   <div class="two">
      <p>I'm a paragraph!</p>
   </div>
</div>
```

The paragraph will be boldfaced, not normal weight. This is because both rules match: the <p> element is descended from a whose class does not contain the word `one` ( ), but it is *also* descended from a whose class contains the word `one`. Both rules match, so both apply. Since a conflict exists, the cascade (which is explained in

Chapter 4) is used to resolve the conflict, and the second rule wins. The structural arrangement of the markup, with the `div.two` being "closer" to the paragraph than `div.one`, is irrelevant.

The :is() and :where() pseudo-classes

CSS has two pseudo-classes that allow for group matching within a complex selector, `:is()` and `:where()`. These are almost identical to each other, with just a minor difference that we'll cover once you understand how they work. Let's start with `:is()`.

Suppose you want to select all list items, whether or not they are part of an ordered or an unordered list. The traditional way to do that is shown here:

```
ol li, ul li {font-style: italic;}
```

With `:is()`, we can rewrite that like so:

```
:is(ol, ul) li {font-style: italic;}
```

The matched elements will be exactly the same: all list items that are part of either ordered or unordered lists.

This might seem slightly pointless: the syntax is not only slightly less clear, but also one character longer. And it's true that in simple situations like that, `:is()` isn't terribly compelling. The more complex the situation, though, the more likely `:is()` will really shine.

For example, what if we want to style all list items that are at least two levels deep in nested lists, no matter what combination of ordered and unordered lists are above them? Compare the following rules, both of which will have the effect shown in Figure 3-25, except one uses the traditional approach and the other uses `:is()`:

```
ol ol li, ol ul li, ul ol li, ul ul li {font-style: italic;}

:is(ol, ul) :is(ol, ul) li {font-style: italic;}
```

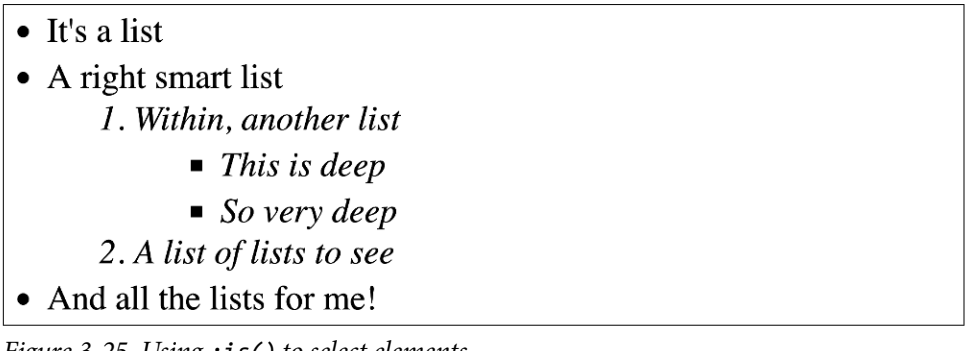


Figure 3-25. Using `:is()` to select elements

Now consider what the traditional approach would look like for three, four, or even more levels deep of nested lists!

The `:is()` pseudo-class can be used in all sorts of situations; selecting all links inside lists that are themselves inside headers, footers, and `<nav>` elements could look like this:

```
:is(header,footer,nav,#header,#footer) :is(ol,ul) a[href] {font-style: italic;}
```

Even better: the list of selectors inside `:is()` is what's called a *forgiving selector list*. By default, if any one thing in a selector is invalid, the whole rule is marked invalid. Forgiving selector lists, on the other hand, will throw any part that's invalid and honor the rest.

So, given all that, what's the difference between `:is()` and `:where()`? The sole difference is that `:is()` takes the specificity of the most-specific selector in its selector list, whereas `:where()` has zero specificity. If that last sentence didn't make sense to you, don't worry! We haven't discussed specificity yet but will in the next chapter.



`:is()` and `:where()` came to browsers in only early 2021, so exercise caution when using them, especially in legacy settings.

Selecting defined elements

As the web has advanced, it's added more and more capabilities. One of the more recent is the ability to add custom HTML elements to your markup in a standardized way. This happens a lot with pattern libraries, which often define Web Components based on elements that are specific to the library.

One thing such libraries do to be more efficient is hold off on defining an element until it's needed, or it's ready to be populated with whatever content is supposed to go into it. Such a custom element might look like this in markup:

```
<mylib-combobox>options go here</mylib-combobox>
```

The actual goal is to fill that combobox (a drop-down list that also allows users to enter arbitrary values) with whatever options the backend CMS provides for it, downloaded via a script that requests the latest data in order to build the list locally, and removing the placeholder text in the process. However, what happens if the server fails to respond, leaving the custom element undefined and stuck with its placeholder text? Without taking steps, the text "options go here" will get inserted into the page, probably with minimal styling.

That's where `:defined` comes in. You can use it to select any defined element, and combine it with `:not()` to select elements that aren't yet defined. Here's a simple way to hide undefined comboboxes, as well as to apply styles to defined comboboxes:

```
mylib-combobox:not(:defined) {display: none;}
mylib-combobox:defined {display: inline-block;}
mylib-combobox {font-size: inherit; border: 1px solid gray;
   outline: 1px solid silver;}
```

The :has() Pseudo-Class

The `:has()` pseudo-class is a little bit tricky, because it doesn't quite follow all the rules we've been working under until now—but as a result, it's also *insanely* powerful.

Imagine you want to apply special styles to any `<div>` element that contains an image. In other words, if a `<div>` element *has* an `<img>` element inside it, you want to apply certain styles to the `<div>`. And that's exactly what `:has()` makes possible.

The previous example would be written something like this, with the result illustrated in Figure 3-26:

```
div:has(img) {
        border: 3px double gray;
}

<div>
   <img src="masthead.jpg" alt="">
</div>
<div>
   <p>No image here!</p>
</div>
<div>
   <p>This has text and <img src="redbox.gif" alt="an image">.
</div>
```

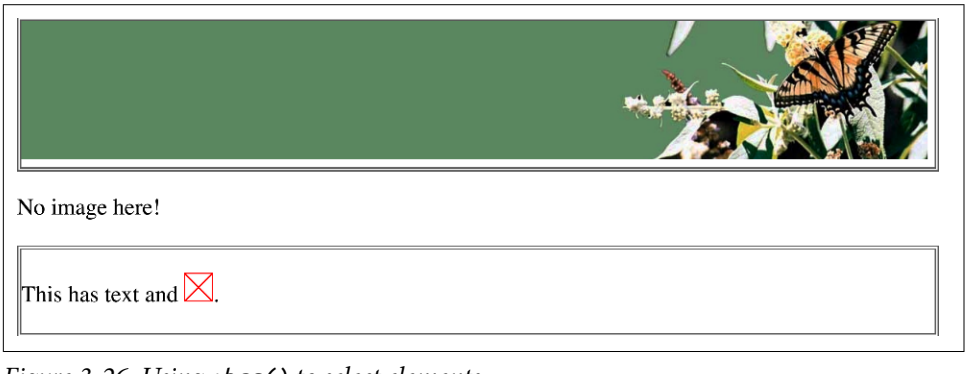


Figure 3-26. Using `:has()` to select elements

The second `<div>`, which does not have an `<img>` element as a descendant, doesn't get the border. If you wanted only the first `<div>` to get the border, because you actually wanted to style only `<div>` elements that have images as direct children, just modify the selector to use the child combinator, like this: `div:has(> img)`. That would prevent the third `<div>` from getting the border.

The `:has()` pseudo-class is, in one real sense, the mythical "parent selector" CSS authors have wished for since the beginning of CSS itself. Except it isn't just for parent selection, because you can select based on siblings, or make the selection happen as far up the

ancestry chain as you like. And if all that didn't quite make sense to you, hang on: we'll explain further.

We have two facts to note right off the bat:

- Inside the parentheses of `:has()`, you can provide a comma-separated list of selectors, and each of those selectors can be simple, compound, or complex.
- Those selectors are considered relative to the anchor element, the element to which `:has()` is attached.

Let's take those in order. All of the following are valid `:has()` uses:

```
table:has(tbody th) {…}
/* tables whose body contains table headers */

a:any-link:has(img:only-child) {…}
/* links containing only an image */

header:has(nav, form.search) {…}
/* headers containing either nav or a form classed search */

section:has(+ h2 em, table + table, ol ul ol ol) {…}
/* sections immediately followed by an 'h2' that contains an 'em'
   OR that contain a table immediately followed by another table
   OR that contain an 'ol' inside an 'ol' inside a 'ul' inside an 'ol' */
```

That last example might be a bit overwhelming, so let's break it down a bit further. We could restate in a longer way, like this:

```
section:has(+ h2 em),
section:has(table + table),
section:has(ol ul ol ol) {…}
```

And here are two examples of the markup patterns that would be selected:

```
<section>(…section content…)</section>
<h2>I'm an h2 with an <em>emphasis element</em> inside, which means
    the section right before me gets selected!</h2>

<section>
<h2>This h2 doesn't get the section selected, because it's a child of
    the section, not its immediately following sibling</h2>
<p>This paragraph is just here.</p>
<aside>
<h3>Q1 Results</h3>
<table>(…table contents…)</table>
<table>(…table contents…)</table>
</aside>
<p>Those adjacent-sibling tables mean this paragraph's parent section element
   DOES get selected!</p>
</section>
```

In the first example, the selection isn't based on parentage or any other ancestry; instead, the `<section>` is selected because its immediate sibling (the `<h2>`) has an `<em>` element as one of its descendants. In the second, the `<section>` is selected because it has a descendant `<table>` that's immediately followed by another `<table>`, both of which happen in this case to be inside an `<aside>` element. That makes this specific example one of grandparent selection, not parent selection, because the `<section>` is a grandparent to the tables.

Right, so that's the first point that we raised earlier. The second is that the selectors inside the parentheses are relative to the element bearing the `:has()`. That means, for example, that the following selector is never going to match anything:

```
div:has(html body h1)
```

That's because while an `<h1>` can certainly be a descendant of a `<div>`, the `<html>` and `<body>` elements cannot. What that selector means, translated into English, is "select any `<div>` that has a descendant `<html>` which itself has a descendant `<body>` which has a descendant `<h1>`." The `<html>` element will never be a descendant of `<div>`, so this selector can't match.

To pick something a little more realistic, here's a bit of markup showing lists nested inside one another, which has the document structure shown in Figure 3-27:

```
<ol>
<li>List item</li>
<li>List item
        <ul>
        <li>List item</li>
        <li>List item</li>
        <li>List item</li>
        </ul>
</li>
<li>List item</li>
<li>List item</li>
<li>List item
        <ul>
        <li>List item</li>
        <li>List item
                <ol>
                <li>List item</li>
                <li>List item</li>
                <li>List item</li>
                </ol>
        </li>
        <li>List item</li>
        </ul>
</li>
</ol>
```

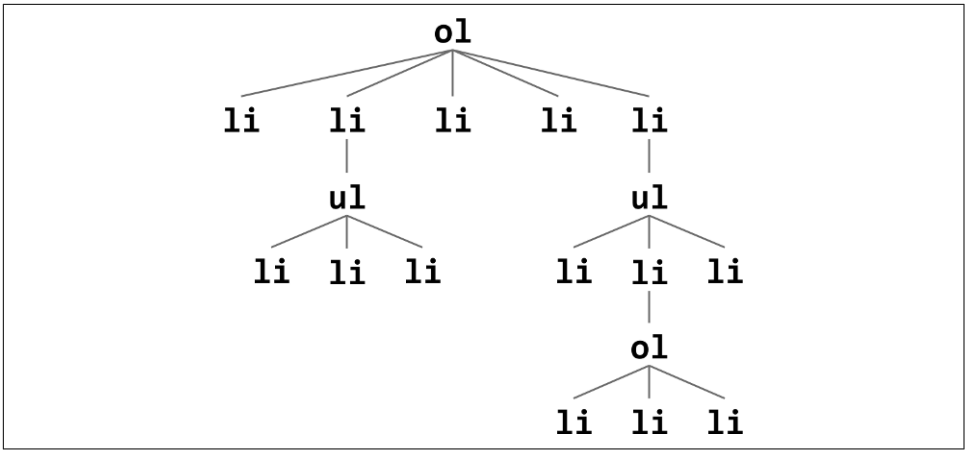


Figure 3-27. A fragment of a document's structure

To that structure, we'll apply the following rules. Spoiler alert: one of them will match an element, and the other will not:

```
ul:has(li ol) {border: 1px solid red;}
ul:has(ol ul ol) {font-style: italic;}
```

The first causes the browser to look at all the `<ul>` elements. For any `<ul>` it finds, it looks at the structure of the elements that descend from that `<ul>`. If it finds an `li ol` relationship in the elements that descend from the `<ul>`, then the `<ul>` is matched, and in this case will be given a red border.

If we study the markup structure, either in the code or in Figure 3-27, we can see two `<ul>` elements. The first has `<li>` descendants but not any `<ol>` descendants, so it won't be matched. The second `<ul>` also has `<li>` descendants, and one of them has an `<ol>` descendant. It's a match! The `<ul>` will be given a red border.

The second rule also causes the browser to look at all the `<ul>` elements. In this case, for any `<ul>` it finds, the browser looks for an `ol ul ol` relationship within the descendants of the `<ul>`. Elements outside the `<ul>` don't count: only those within it are considered. Of the two `<ul>` elements in the document, neither has an `<ol>` inside a `<ul>` that's inside another `<ol>` that is itself descended from the `<ul>` being considered. There's no match, so neither of the `<ul>` elements will be italicized.

Even more powerfully, you're free to mix `:has()` with other pseudo-classes. You might, for example, want to select any heading level if it has an image inside. You can do this in two ways: the long, clumsy way or the compact way. Both are shown here:

```
h1:has(img), h2:has(img), h3:has(img), h4:has(img), h5:has(img), h6:has(img)

:is(h1, h2, h3, h4, h5, h6):has(img)
```

The two selectors have the same outcome: if an element *is* one of the listed heading elements, and that element *has* among its descendant elements an `<img>` element, then the heading will be selected.

For that matter, you could select any headings that *don't* have images inside:

```
:is(h1, h2, h3, h4, h5, h6):not(:has(img))
```

Here, if an element *is* one of the listed heading levels, but an `<img>` element is *not* one of the descendants it has, then the heading will be selected. If we bring them together and apply them to numerous headings, we get the results shown in Figure 3-28.

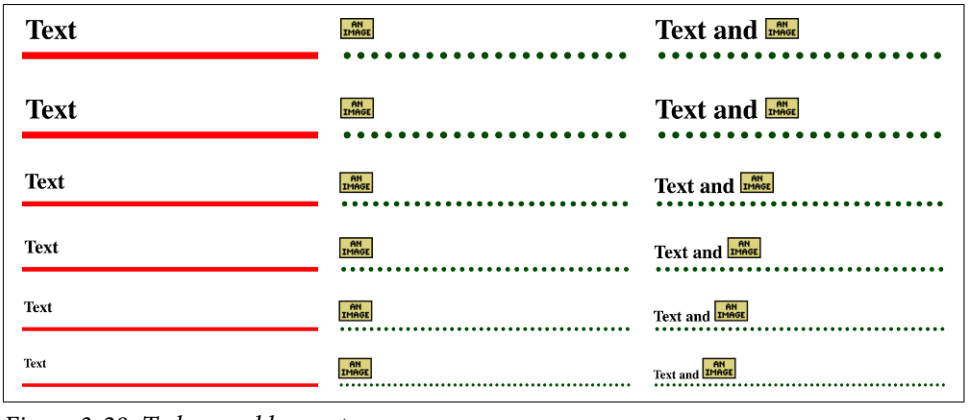


Figure 3-28. To has and has not

As you can already see, this selector has a lot of power. Dangers also exist: it is entirely possible to write selectors that cause major performance hits to the browser, especially in settings where scripting may be used to modify the document structure. Consider the following:

```
div:has(*.popup) {…}
```

This is saying, "Apply these styles to any `<div>` that has an element with a class of `popup` as a descendant." When the page is loaded into the browser, it has to check all the `<div>` elements to see if they match this selector. That could mean a few trips up and down the document's structural tree, but ideally it would resolve in a few milliseconds, and the page can then be displayed.

But suppose we have a script that can add `.popup` to an element, or even several elements, on the page. As soon as the class values change, the browser has to check not only whether any styles apply to `.popup` elements and their descendants, but also whether any ancestor or sibling elements are affected by this change. Instead of looking only down the document tree, the browser now has to look up as well. And any change triggered by this could mean changes all throughout the page's layout, both when an element is marked as `.popup` and whenever a `.popup` element loses that class value, potentially affecting elements in entirely different parts of the document.

This sort of performance hit is why there hasn't been a "parent selector" or anything like it before. Computers are getting fast enough, and browser engines smart enough, that this is much less of a worry than it was in the past—but it's still something to keep in mind and test out thoroughly.



It is not possible to nest pseudo-elements like `::first-line` or `::selection` in `has()`. (We'll discuss pseudo-elements shortly.)

Other Pseudo-Classes

Even more pseudo-classes are defined in the CSS Selectors specification, but they are partially supported in browsers, or in some cases not supported at all as of early 2023, or else are things we'll cover elswhere in the book. We're listing them in Table 3-5 for the sake of completeness, and to point you toward pseudo-classes that might be supported between this edition of the book and the next one. (Or could be replaced with an equivalent pseudo-class with a different name; that happens sometimes.)

Table 3-5. Other pseudo-classes

Name	Description
`:nth-col()`	Refers to table cells or grid items that are in an nth column, which is found using the *an* + *b* pattern; essentially the same as `:nth-child()`, but refers specifically to table or grid columns
`:nth-last-col()`	Refers to table cells or grid items that are in an nth-last column, which is found using the *an* + *b* pattern; essentially the same as `:nth-last-child()`, but refers specifically to table or grid columns
`:left`	Refers to any lefthand page in a printed document; see Chapter 21 for more
`:right`	Refers to any righthand page in a printed document; see Chapter 21 for more
`:fullscreen`	Refers to an element that is being displayed full-screen (e.g., a video that's in full-screen mode)
`:past`	Refers to an element that appeared before (in time) an element being matched by `:current`
`:current`	Refers to an element, or the ancestor of an element, that is currently being displayed in a time-based format like a video (e.g., an element containing closed-caption text)
`:future`	Refers to an element that will appear after (in time) an element being matched by `:current`
`:paused`	Refers to any element that can have the states "playing" or "paused" (e.g., audio, video, etc.) when it is in the "paused" state
`:playing`	Refers to any element that can have the states "playing" or "paused" (e.g., audio, video, etc.) when it is in the "playing" state
`:picture-in-picture`	Refers to an element that is used as a picture-in-picture display

Pseudo-Element Selectors

Much as pseudo-classes assign phantom classes to anchors, *pseudo-elements* insert fictional elements into a document in order to achieve certain effects.

Unlike the single colon of pseudo-classes, pseudo-elements employ a double-colon syntax, like `::first-line`. This is meant to distinguish pseudo-elements from pseudo-classes. This was not always the case—in CSS2, both selector types used a single colon—so for backward compatibility, browsers may accept some single-colon pseudo-type selectors. Don't take this as an excuse to be sloppy, though! Use the proper number of colons at all times to future-proof your CSS; after all, there is no way to predict when browsers will stop accepting single-colon pseudo-type selectors.

Styling the First Letter

The `::first-letter` pseudo-element styles the first letter, or a leading punctuation character and the first letter (if the text starts with punctuation), of any non-inline element. This rule causes the first letter of every paragraph to be colored red:

```
p::first-letter {color: red;}
```

The `::first-letter` pseudo-element is most commonly used to create an initial-cap or drop-cap typographic effect. You could make the first letter of each `<p>` twice as big as the rest of the heading, though you may want to apply this styling to only the first letter of the first paragraph:

```
p:first-of-type::first-letter {font-size: 200%;}
```

Figure 3-29 illustrates the result of this rule.

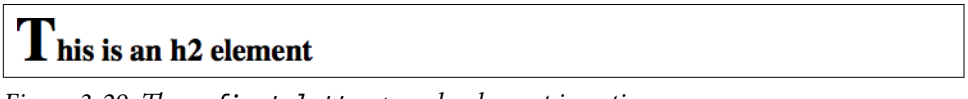


Figure 3-29. The `::first-letter` pseudo-element in action

This rule effectively causes the user agent to style a fictional, or faux, element that encloses the first letter of each `<p>`. It would look something like this:

```
<p><p-first-letter>T</p-first-letter>his is a p element, with a styled first
    letter</h2>
```

The `::first-letter` styles are applied only to the contents of the fictional element shown in the example. This `<p-first-letter>` element does *not* appear in the document source, nor even in the DOM tree. Instead, its existence is constructed on the fly by the user agent and is used to apply the `::first-letter` style(s) to the appropriate bit of text. In other words, `<p-first-letter>` is a pseudo-element. Remember, you don't have to add any new tags. The user agent styles the first letter for you as if you had encased it in a styled element.

The first letter is defined as the first typographic letter unit of the originating element, if it is not preceded by other content, like an image. The specifications use the term letter unit because some languages have letters made up of more than one character, like *æ* in Old West Norse. Punctuation that precedes or follows the first letter unit, even if there are several such symbols, should be included in the `::first-letter` pseudo-element. The browser does this for you.

Styling the First Line

Similarly, `::first-line` can be used to affect the first line of text in an element. For example, you could make the first line of each paragraph in a document large and purple:

```
p::first-line {
  font-size: 150%;
  color: purple;
}
```

In Figure 3-30, the style is applied to the first displayed line of text in each paragraph. This is true no matter how wide or narrow the display region is. If the first line contains only the first five words of the paragraph, only those five words will be big and purple. If the first line contains the first 30 words of the element, all 30 will be big and purple.

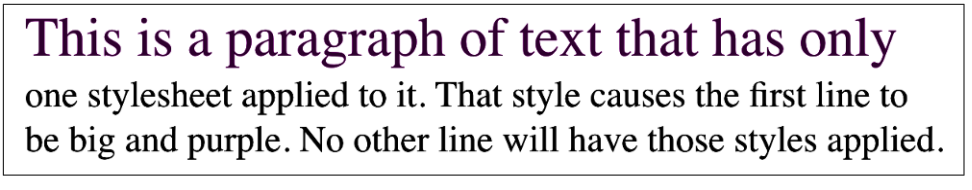


Figure 3-30. The `::first-line` pseudo-element in action

Because the text from "This" to "only" should be big and purple, the user agent employs a fictional markup that looks something like this:

```
<p>
<p-first-line>This is a paragraph of text that has only</p-first-line>
one stylesheet applied to it. That style causes the first line to
be big and purple. No other line will have those styles applied.
</p>
```

If the first line of text were edited to include only the first seven words of the paragraph, the fictional `</p-first-line>` would move back and occur just after the word "that." If the user were to increase or decrease the font-size rendering, or expand or contract the browser window causing the width of the text to change, thereby causing the number of words on the first line to increase or decrease, the browser automatically sets only the words in the currently displayed first line to be both big and purple.

The length of the first line depends on multiple factors, including the font size, letter spacing, and width of the parent container. Depending on the markup and the length of that first line, the end of the first line could come in the middle of a nested element. If

the `::first-line` breaks up a nested element, such as an `em` or a hyperlink, the properties attached to the `::first-line` will apply to only the portion of that nested element that is displayed on the first line.

Restrictions on ::first-letter and ::first-line

The `::first-letter` and `::first-line` pseudo-elements currently can be applied only to block-display elements such as headings or paragraphs, and not to inline-display elements such as hyperlinks. There are also limits on the CSS properties that may be applied to `::first-line` and `::first-letter`. Table 3-6 gives an idea of these limitations. Like all pseudo-elements, neither can be included in `:has()` or `:not()`.

Table 3-6. Properties permitted on pseudo-elements

`::first-letter`	`::first-line`
• All font properties	• All font properties
• All background properties	• All background properties
• All text decoration properties	• All margin properties
• All inline typesetting properties	• All padding properties
• All inline layout properties	• All border properties
• All border properties	• All text decoration properties
• `box-shadow`	• All inline typesetting properties
• `color`	• `color`
• `opacity`	• `opacity`

The Placeholder Text Pseudo-Element

As it happens, the restrictions on what styles can be applied via `::first-line` are exactly the same as the restrictions on styles applied via `::placeholder`. This pseudo-element matches any placeholder text placed into text inputs and textareas. You could, for example, italicize text input placeholder text and turn textarea placeholder text a dusky blue like this:

```
input::placeholder {font-style: italic;}
textarea::placeholder {color: cornflowerblue;}
```

For both `<input>` and `<textarea>` elements, this text is defined by the `placeholder` attribute in HTML. The markup will look something very much like this:

```
<input type="text" placeholder="(XXX) XXX-XXXX" id="phoneno">
<textarea placeholder="Tell us what you think!"></textarea>
```

If text is prefilled using the `value` attribute on `<input>` elements, or by placing content inside the `<textarea>` element, that will override the value of any `placeholder` attribute, and the resulting text won't be selected with the `::placeholder` pseudo-element.

The Form Button Pseudo-Element

Speaking of form elements, it's also possible to directly select the file-selector button—and *only* the file-selector button—in an `<input>` element that has a `type` of `file`. This gives you a way to call attention to the button a user needs to click to open the file-selection dialog, even if no other part of the input can be directly styled.

If you've never seen a file-selection input, it usually looks like this:

```
<label for="uploadField">Select file from computer</label>
<input id="uploadField" type="file">
```

That second line gets replaced with a control whose appearance is dependent on the combination of operating system and browser, so it tends to look at least a little different (sometimes a lot different) from one user to the next. Figure 3-31 shows one possible rendering of the input, with the button styled by the following CSS:

```
input::file-selector-button {
   border: thick solid gray;
   border-radius: 2em;
}
```

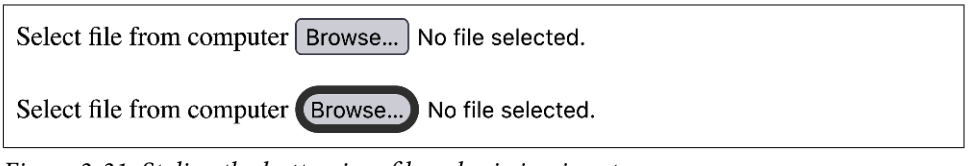


Figure 3-31. Styling the button in a file-submission input

Generating Content Before and After Elements

Let's say you want to preface every `<h2>` element with a pair of silver square brackets as a typographical effect:

```
h2::before {content: "]]"; color: silver;}
```

CSS lets you insert *generated content*, and then style it directly using the pseudo-elements `::before` and `::after`. Figure 3-32 illustrates an example.

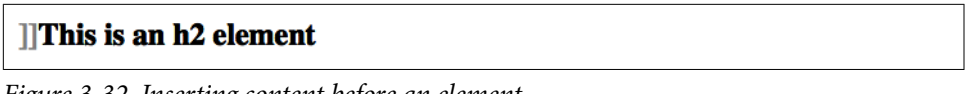


Figure 3-32. Inserting content before an element

The pseudo-element is used to insert the generated content and to style it. To place content at the end of an element, right before the closing tag, use the pseudo-element `::after`. You could end your documents with an appropriate finish:

```
body::after {content: "The End.";}
```

Conversely, if you want to insert some content at the beginning of an element, right after the opening tag, use `::before`. Just remember that in either case, you have to use the `content` property in order to insert something to style.

Generated content is its own subject, and the entire topic (including more detail on `::before`, `::after`, and `content`) is covered more thoroughly in Chapter 16.

Highlight Pseudo-Elements

A relatively new concept in CSS is the ability to style pieces of content that have been highlighted, either by user selection or by the user agent itself. These are summarized in Table 3-7.

Table 3-7. Highlight pseudo-elements

Name	Description
`::selection`	Refers to any part of a document that has been highlighted for user operation (e.g., text that has been drag-selected with a mouse)
`::target-text`	Refers to the text of a document that has been targeted; this is distinct from the `:target` pseudo-class, which refers to a targeted element as a whole, not a fragment of text
`::spelling-error`	Refers to the part of a document that has been marked by the user agent as a misspelling
`::grammar-error`	Refers to the part of a document that has been marked by the user agent as a grammar error

Of the four pseudo-elements in Table 3-7, only one, `::selection`, has any appreciable support as of early 2023. So we'll explore it and leave the others for a future edition.

When someone uses a mouse pointer to click-hold-and-drag in order to highlight some text, that's a selection. Most browsers have default styles set for text selection. Authors can apply a limited set of CSS properties to such selections, overriding the browser's default styles, by styling the `::selection` pseudo-element. Let's say you want selected text to be white on a navy-blue background. The CSS would look like this:

```
::selection {color: white; background-color: navy;}
```

The primary use cases for `::selection` are specifying a color scheme for selected text that doesn't clash with the rest of the design, or defining different selection styles for different parts of a document. For example:

```
::selection {color: white; background-color: navy;}
form::selection {color: silver; background-color: maroon;}
```

Be careful in styling selection highlights: users generally expect text they select to look a certain way, usually defined by settings in their operating system. Thus, if you get too clever with selection styling, you could confuse users. That said, if you know that selected text can be difficult to see because your design's colors tend to obscure it, defining more obvious highlight styles is probably a good idea.

Note that selected text can cross element boundaries, and that multiple selections can occur within a given document. Imagine that a user selects text starting from the middle of one paragraph to the middle of the next. In effect, each paragraph will get its own selection pseudo-element nested inside, and selection styling will be handled as appropriate for the context. Given the following CSS and HTML, you'll get a result like that shown in Figure 3-33:

```
.p1::selection {color: silver; background-color: black;}
.p2::selection {color: black; background-color: silver;}

<p class="p1">This is a paragraph with some text that can be selected,
   one of two.</p>
<p class="p2">This is a paragraph with some text that can be selected,
   two of two.</p>
```

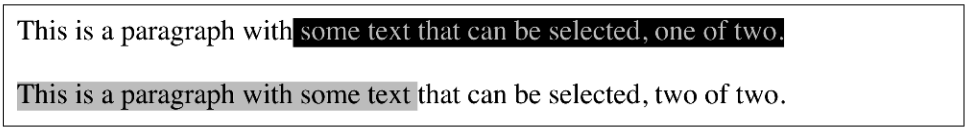


Figure 3-33. Selection styling

This underscores a point made earlier: *be careful* with your selection styling. It is all too easy to make text unreadable for some users, particularly if your selection styles interact badly with the user's default selection styles.

Furthermore, for user privacy reasons, you can apply only a limited number of CSS properties to selections: `color`, `background-color`, `text-decoration` and related properties, `text-shadow`, and the `stroke` properties (in SVG).



As of early 2023, selections do not have their styles inherited: selecting text containing some inline elements will apply the selection styling to the text outside the inline elements, but not within the inline elements. It is not clear if this behavior is intended, but it is consistent across major browsers.

Beyond `::selection`, there will likely be increasing support for `::target-text`. As of early 2023, this is supported in only Chromium browsers, which introduced a feature that needs it. With this feature, text can be added to the end of a URL as part of the fragment identifier for highlighting, in order to draw attention to one or more parts of the page.

For example, a URL might look something like: `https://example.org/#:~:text=for%20use%20in%20illustrative%20examples`. The part at the end says to the browser, "Once you've loaded the page, highlight any examples of this text." The text is encoded for use in URLs, which is why it's filled with `%20` strings—they represent spaces. The result will look something like Figure 3-34.

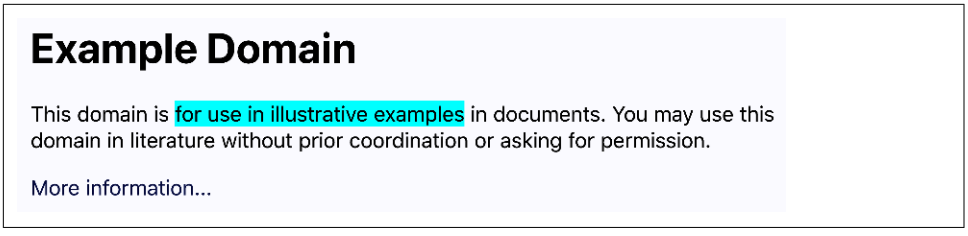


Figure 3-34. Targeted text styling

If you wanted to suppress this content highlighting on your own pages, you might do something like this:

```
::target-text {color: inherit; background-color: inherit;}
```

As for `::spelling-error` and `::grammar-error`, these are meant to apply highlighting of some sort to any spelling or grammar errors within a document. You can see the utility for something like Google Docs or the editing fields of CMSs like WordPress or Craft. For most other applications, though, they seem unlikely to be very popular. Regardless, as of this writing, there is no browser support for either, and the Working Group is still hashing out the details of how they should work.

The Backdrop Pseudo-Element

Suppose you have an element that's being presented full-screen, like a video. Furthermore, suppose that element doesn't neatly fill the screen all the way to the edges, perhaps because the aspect ratio of the element doesn't match the aspect ratio of the screen. What should be filled in for the parts of the screen that the element doesn't cover? And how would you select that non-element region with CSS?

Enter the `::backdrop` pseudo-element. This represents a box that's the exact size of the full-screen viewport, and it is always drawn beneath a full-screen element. So you might put a dark-gray backdrop behind any full-screen video like this:

```
video::backdrop {background: #111;}
```

CSS doesn't restrict which styles can be applied to backdrops, but since they're essentially empty boxes placed behind a full-screen element, most of the time you'll probably be setting background colors or images.

An important thing to remember is that backdrops do *not* participate in inheritance. They can't inherit styles from ancestor elements, nor do they pass any of their styles on to any children. Whatever styles you apply to the backdrop will exist in their own little pocket universe.

The Video-Cue Pseudo-Element

On the subject of videos, videos often have Web Video Text Tracks (WebVTT) data containing the text captions enabling accessibility. These captions are known as *cues* and can be styled with the `::cue` pseudo-element.

Let's say you have a video that's mostly dark, with a few light segments. You might style the cues to be a lightish-gray text on a translucent dark background, as follows:

```
::cue {
  color: silver;
  background: rgba(0,0,0,0.5);
}
```

This will always apply to the currently visible cue.

You can also select parts of individual cues by using a selector pattern inside parentheses. This can be used to style specific elements defined in the WebVTT data, drawn from a small list allowed by the WebVTT specification. For example, any italicized cue text could be selected as follows:

```
::cue(i) {…}
```

You could use structural pseudo-classes like `:nth-child`, but these will apply only within a given cue, not across cues. You can't select every other cue for styling, but you can select every other element within a given cue. Assume the following WebVTT data:

```
00:00:01.500 --> 00:00:02.999
<v Hildy>Tell me, is the lord of the universe in?</v>

00:00:03.000 --> 00:00:04.299
- Yes, he's in.
- In a bad humor.
```

The second cue includes two lines of text. These are treated as separate elements, in effect, even though no elements are specified. Thus, we could make the line Hildy says (indicated with `<v Hildy>`, which is the WebVTT equivalent of `<v voice="Hildy">`) boldface, and give alternate colors to the two lines of dialog in the second cue, like so:

```
::cue(v[voice="Hildy"]) {font-weight: bold;}
::cue(:nth-child(odd)) {color: yellow;}
::cue(:nth-child(even)) {color: white;}
```

As of early 2023, a limited range of properties can be applied to cues:

- `color`
- `background` and its associated longhand properties (e.g., `background-color`)
- `text-decoration` and its associated longhand properties (e.g., `text-decoration-thickness`)
- `text-shadow`
- `text-combine-upright`

- `font` and its associated longhand properties (e.g., `font-weight`)
- `ruby-position`
- `opacity`
- `visibility`
- `white-space`
- `outline` and its associated longhand properties (e.g., `outline-width`)

Shadow Pseudo-Classes and -Elements

Another recent innovation in HTML has been the introduction of the shadow DOM, which is a deep and complex subject we don't have the space to explore here. At a basic level, the *shadow DOM* allows developers to create encapsulated markup, style, and scripting within the regular (or *light*) DOM. This keeps the styles and scripts of one shadow DOM from affecting any other part of the document, whether those parts are in the light or shadow DOM.

We're bringing this up here because CSS does provide ways to hook into shadow DOMs, as well as to reach up from within a shadow DOM to select the piece of the light DOM that hosts the shadow. (This all sounds very panel-van artistic, doesn't it?)

Shadow Pseudo-Classes

To see what this means, let's bring back the combobox example from earlier in the chapter. It looks like this:

```
<mylib-combobox>options go here</mylib-combobox>
```

All the CSS (and JS) within this custom element apply *only* to the `<mylib-combobox>` element. Even if the CSS within the custom element says something like `li {color: red;}`, that will apply to only `<li>` elements constructed within the `<mylib-combobox>`. It can't leak out to turn list items elsewhere on the page red.

That's all good, but what if you want to style the host element in a certain way from within the custom element? The host element, more generally called the *shadow host*, is in this case `<mylib-combobox>`. From within the shadow host, CSS can select the host by using the `:host` pseudo-class. For example:

```
:host {border: 2px solid red;}
```

That will reach up, so to speak, "pierce through the shadow boundary" (to use an evocative phrase from the specification), and select the `<mylib-combobox>` element, or whatever the name of the custom element containing the shadow DOM CSS is.

Now, suppose there can be different kinds of comboboxes, each with its own class. Something like this:

```
<mylib-combobox class="countries">options go here</mylib-combobox>
<mylib-combobox class="regions">options go here</mylib-combobox>
<mylib-combobox class="cities">options go here</mylib-combobox>
```

You might want to style each class of combobox differently. For that, the `:host()` pseudo-class exists:

```
:host(.countries) {border: 2px solid red;}
:host(.regions) {border: 1px solid silver;}
:host(.cities) {border: none; background: gray;}
```

These rules could then be included in the styles that are loaded by all comboboxes, using the presence of classes on the shadow hosts to style as appropriate.

But wait! What if, instead of latching on to classes, we want to style our shadow hosts based on where they appear in the light DOM? In that case, `:host-context()` has you covered. Thus, we can style our comboxes one way if they're part of a form, and a different way if they're part of a header navigation element:

```
:host-context(form) {border: 2px solid red;}
:host-context(header nav) {border: 1px solid silver;}
```

The first of these means "if the shadow host is the descendant of a `<form>` element, apply these styles." The second means "if the shadow host is the descendant of a `<nav>` element that is itself descended from a `<header>` element, apply these styles." To be clear, `form` and `<nav>` are *not* the shadow hosts in these situations! The selector in `:host-context()` is describing only the context in which the host needs to be placed in order to be selected.



The four selectors that cross the shadow DOM/light DOM boundary—`:host`, `:host()`, and `:host-content()`, along with the `:slotted()` selector discussed next—are supported only when declared within the context of the shadow DOM. As of early 2023, `:host-context()` isn't supported in Safari or Firefox and is at risk of being removed from the specification.

Shadow Pseudo-Elements

In addition to having hosts, shadow DOMs can also define *slots*. These are elements that are meant to have other things slotted into them, much as you would place an expansion card into an expansion slot. Let's expand the markup of the combobox a little bit:

```
<mylib-combobox>
     <span slot="label">Country</span>
     ["shadow-tree"]
          <slot name="label"></slot>
     [/"shadow tree"]
</mylib-combobox>
```

Now, to be clear, the `shadow tree` isn't actual markup. It's just there to represent the shadow DOM that gets constructed by whatever script builds it. So please don't go writing square-bracketed quoted element names into your documents; they will fail.

That said, given a setup like the preceding one, `<span>` would be slotted into the `slot` element, because the names match. You could try applying styles to the slot, but what if you'd rather style the thing that got plugged into the slot? That's represented by the `::slotted()` pseudo-element, which accepts a selector as needed.

Thus, if you want to style all slotted elements one way and then add some extra style if the slotted element is a `<span>`, you would write something like this:

```
::slotted(*) {outline: 2px solid red;}
::slotted(span) {font-style-italic;}
```

More practically, you could style all slots red, and then remove that red from any slot that's been slotted with content, thus making the slots that failed to get any content stand out. Something like this:

```
slot {color: red;}
::slotted(*) {color: black;}
```



The shadow DOM and its use is a complex topic, and one that we have not even begun to scratch the surface of in this section. Our only goal is to introduce the pseudo-classes and -elements that pertain to the shadow DOM, not explain the shadow DOM or illustrate best practices.

Summary

As you saw in this chapter, pseudo-classes and pseudo-elements bring a whole lot of power and flexibility to the table. Whether selecting hyperlinks based on their visited state, matching elements based on their placement in the document structure, or styling pieces of the shadow DOM, there's a pseudo selector for nearly every taste.

In this chapter and the preceding one, we've mentioned the concepts of specificity and the cascade a few times and promised to talk about them soon. Well, soon is now. That's exactly what we'll do in the next chapter.

CHAPTER 4

Specificity, Inheritance, and the Cascade

Chapters 2 and 3 showed how document structure and CSS selectors allow you to apply a wide variety of styles to elements. Knowing that every valid document generates a structural tree, you can create selectors that target elements based on their ancestors, attributes, sibling elements, and more. The structural tree is what allows selectors to function and is also central to a similarly crucial aspect of CSS: inheritance.

Inheritance is the mechanism by which some property values are passed on from an element to a descendant element. When determining which values should apply to an element, a user agent must consider not only inheritance but also the *specificity* of the declarations, as well as the origin of the declarations themselves. This process of consideration is what's known as the *cascade*.

We will explore the interrelation between these three mechanisms—specificity, inheritance, and the cascade—in this chapter. For now, the difference between the latter two can be summed up this way: when we write `h1 {color: red; color: blue;}`, the `<h1>` becomes blue because of the cascade, and any `<span>` inside the `<h1>` also becomes blue because of inheritance.

Above all, regardless of how abstract things may seem, keep going! Your perseverance will be rewarded.

Specificity

You know from Chapters 2 and 3 that you can select elements by using a wide variety of means. In fact, the same element can often be selected by two or more rules, each with its own selector. Let's consider the following three pairs of rules. Assume that each pair will match the same element:

```
h1 {color: red;}
body h1 {color: green;}
```

```
h2.grape {color: purple;}
h2 {color: silver;}

html > body table tr[id="totals"] td ul > li {color: maroon;}
li#answer {color: navy;}
```

Only one of the two rules in each pair can be applied, or "win," since the matched elements can be only one color at a time. How do we know which one will win?

The answer is found in the *specificity* of each selector. For every rule, the user agent (i.e., a web browser) evaluates the specificity of the selector and attaches the specificity to each declaration in the rule within the cascade layer that has precedence. When an element has two or more conflicting property declarations, the one with the highest specificity will win out.



This isn't the whole story in terms of conflict resolution, which is a bit more complicated than a single paragraph can cover. For now, just keep in mind that selector specificity is compared only to other selectors that share the same origin and cascade layer. We'll cover those terms, and more in "The Cascade" on page 116.

A selector's specificity is determined by the components of the selector itself. A specificity value can be expressed in three parts, like this: `0,0,0`. The actual specificity of a selector is determined as follows:

- For every ID attribute value given in the selector, add `1,0,0`.
- For every class attribute value, attribute selection, or pseudo-class given in the selector, add `0,1,0`.
- For every element and pseudo-element given in the selector, add `0,0,1`.
- Combinators do not contribute anything to the specificity.
- Anything listed inside a `:where()` pseudo-class, and the universal selector, adds `0,0,0`. (While they do not contribute anything to the specificity weight, they do match elements, unlike combinators.)
- The specificity of an `:is()`, `:not()`, or `:has()` pseudo-class is equal to the specificity of the most specific selector in its selector list argument.

For example, the following rules' selectors result in the indicated specificities:

```
h1 {color: red;}                    /* specificity = 0,0,1 */
p em {color: purple;}               /* specificity = 0,0,2 */
.grape {color: purple;}             /* specificity = 0,1,0 */
*.bright {color: yellow;}           /* specificity = 0,1,0 */
p.bright em.dark {color: maroon;}   /* specificity = 0,2,2 */
#id216 {color: blue;}               /* specificity = 1,0,0 */
*:is(aside#warn, code) {color: red;} /* specificity = 1,0,1 */
div#sidebar *[href] {color: silver;} /* specificity = 1,1,1 */
```

If an element is matched by both the second and fifth rules in this example, that element will be maroon because the sixth rule's specificity outweighs the second's.

Take special note of the next-to-last selector, `*:is(aside#warn, code)`. The `:is()` pseudo-class is one of a small group of pseudo-classes for which the specificity is equal to the most specific selector in the selector list. Here, the selector list is `aside#warn, code`. The `aside#warn` compound selector has a specificity of `1,0,1`, and the `code` selector has a specificity of `0,0,1`. Thus, the whole `:is()` portion of the selector is set to the specificity of the `aside#warn` selector.

Now, let's return to the pairs of rules from earlier in the section and fill in the specificities:

```
h1 {color: red;}          /* 0,0,1 */
body h1 {color: green;}   /* 0,0,2 (winner)*/

h2.grape {color: purple;}  /* 0,1,1 (winner) */
h2 {color: silver;}        /* 0,0,1 */

html > body table tr[id="totals"] td ul > li {color: maroon;}  /* 0,1,7 */
li#answer {color: navy;}                                      /* 1,0,1
   (winner) */
```

We've indicated the winning rule in each pair; in each case, it's because the specificity is higher. Notice how they're listed and that the order of the rules doesn't matter here.

In the second pair, the selector `h2.grape` wins because it has an extra class: `0,1,1` beats out `0,0,1`. In the third pair, the second rule wins because `1,0,1` wins out over `0,1,7`. In fact, the specificity value `0,1,0` would win out over the value `0,0,13`.

This happens because the values are compared from left to right. A specificity of `1,0,0` will win out over any specificity that begins with a `0`, no matter what the rest of the numbers might be. So `1,0,1` wins over `0,1,7` because the `1` in the first value's first position beats the `0` in the second value's first position.

Declarations and Specificity

Once the specificity of a selector has been determined, the specificity value will be conferred on all of its associated declarations. Consider this rule:

```
h1 {color: silver; background: black;}
```

For specificity purposes, the user agent must treat the rule as if it were "ungrouped" into separate rules. Thus, the previous example would become the following:

```
h1 {color: silver;}
h1 {background: black;}
```

Both have a specificity of `0,0,1`, and that's the value conferred on each declaration. The same splitting-up process happens with a grouped selector as well. Given the rule,

```
h1, h2.section {color: silver; background: black;}
```

the user agent treats it as if it were the following:

```
h1 {color: silver;}             /* 0,0,1 */
h1 {background: black;}         /* 0,0,1 */
h2.section {color: silver;}     /* 0,1,1 */
h2.section {background: black;} /* 0,1,1 */
```

This becomes important when multiple rules match the same element and some of the declarations clash. For example, consider these rules:

```
h1 + p {color: black; font-style: italic;}              /* 0,0,2 */
p {color: gray; background: white; font-style: normal;} /* 0,0,1 */
*.callout {color: black; background: silver;}           /* 0,1,0 */
```

When applied to the following markup, the content will be rendered as shown in Figure 4-1:

```
<h1>Greetings!</h1>
<p class="callout">
It's a fine way to start a day, don't you think?
</p>
<p>
There are many ways to greet a person, but the words are not as important
as the act of greeting itself.
</p>
<h1>Salutations!</h1>
<p>
There is nothing finer than a hearty welcome from one's neighbor.
</p>
<p class="callout">
Although a steaming pot of fresh-made jambalaya runs a close second.
</p>
```

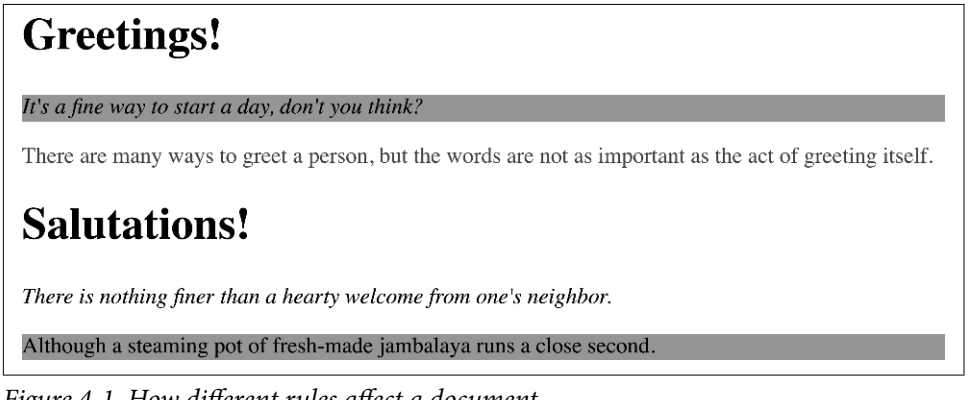


Figure 4-1. How different rules affect a document

In every case, the user agent determines which rules match a given element, calculates all of the associated declarations and their specificities, determines which rules win out, and then applies the winners to the element to get the styled result. These machinations must be performed on every element, selector, and declaration. Fortunately, the user agent does

it all automatically and nearly instantly. This behavior is an important component of the cascade, which we'll discuss later in this chapter.

Resolving Multiple Matches

When an element is matched by more than one selector in a grouped selector, the most specific selector is used. Consider the following CSS:

```
li,            /* 0,0,1 */
.quirky,       /* 0,1,0 */
#friendly,     /* 1,0,0 */
li.happy.happy.happy#friendly { /* 1,3,1 */
   color: blue;
}
```

Here we have one rule with a grouped selector, and each of the individual selectors has a very different specificity. Now suppose we find this in our HTML:

```
<li class="happy quirky" id="friendly">This will be blue.</li>
```

Every one of the selectors in the grouped selector applies to the list item! Which one is used for specificity purposes? The most specific. Thus, in this example, the blue is applied with a specificity of `1,3,1`.

You might have noticed that we repeated the `happy` class name three times in one of the selectors. This is a bit of hack that can be used with classes, attributes, pseudo-classes, and even ID selectors to increase specificity. Do be careful with it, since artificially inflating specificity can create problems in the future: you might want to override that rule with another, and that rule will need even more classes chained together.

Zeroed Selector Specificity

The universal selector does not contribute to specificity. It has a specificity of `0,0,0`, which is different from having no specificity (as we'll discuss in "Inheritance" on page 113). Therefore, given the following two rules, a paragraph descended from a `<div>` will be black, but all other elements will be gray:

```
div p {color: black;} /* 0,0,2 */
* {color: gray;}      /* 0,0,0 */
```

This means the specificity of a selector that contains a universal selector along with other selectors is not changed by the presence of the universal selector. The following two selectors have exactly the same specificity:

```
div p         /* 0,0,2 */
body * strong /* 0,0,2 */
```

The same is true for the `:where()` pseudo-class, regardless of whatever selectors might be in its selector list. Thus, `:where(aside#warn, code)` has a specificity of `0,0,0`.

Combinators, including `~`, `>`, `+`, and the space character, have no specificity at all—not even zero specificity. Thus, they have no impact on a selector's overall specificity.

ID and Attribute Selector Specificity

It's important to note the difference in specificity between an ID selector and an attribute selector that targets an `id` attribute. Returning to the third pair of rules in the example code, we find the following:

```
html > body table tr[id="totals"] td ul > li {color: maroon;} /* 0,1,7 */
li#answer {color: navy;}                                     /* 1,0,1 (wins) */
```

The ID selector (`#answer`) in the second rule contributes `1,0,0` to the overall specificity of the selector. In the first rule, however, the attribute selector (`[id="totals"]`) contributes `0,1,0` to the overall specificity. Thus, given the following rules, the element with an `id` of `meadow` will be green:

```
#meadow {color: green;}      /* 1,0,0 */
*[id="meadow"] {color: red;} /* 0,1,0 */
```

Importance

Sometimes a declaration is so important that it outweighs all other considerations. CSS calls these *important declarations* (for hopefully obvious reasons) and lets you mark them by inserting the flag `!important` just before the terminating semicolon in a declaration:

```
p.dark {color: #333 !important; background: white;}
```

Here, the color value of `#333` is marked with the `!important` flag, whereas the background value of `white` is not. If you wish to mark both declarations as important, each declaration needs its own `!important` flag:

```
p.dark {color: #333 !important; background: white !important;}
```

You must place the `!important` flag correctly, or the declaration may be invalidated: `!important` *always* goes at the end of a declaration, just before the semicolon. This placement is especially critical when it comes to properties that allow values containing multiple keywords, such as `font`:

```
p.light {color: yellow; font: smaller Times, serif !important;}
```

If `!important` were placed anywhere else in the `font` declaration, the entire declaration would likely be invalidated and none of its styles applied.



We realize that to those of you who come from a programming background, the syntax of this token instinctively translates to "not important." For whatever reason, the bang (`!`) was chosen as the delimiter for important flags, and it does *not* mean "not" in CSS, no matter how many other languages give it that very meaning. This association is unfortunate, but we're stuck with it.

Declarations that are marked `!important` do not have a special specificity value, but are instead considered separately from unimportant declarations. In effect, all `!important`

declarations are grouped together, and specificity conflicts are resolved within that group. Similarly, all unimportant declarations are considered as a group, with any conflicts within the unimportant group as described previously. Thus, in any case where an important and an unimportant declaration conflict, an important declaration will always win (unless the user agent or user have declared the same property as important, which you'll see later in the chapter).

Figure 4-2 illustrates the result of the following rules and markup fragment:

```
h1 {font-style: italic; color: gray !important;}
.title {color: black; background: silver;}
* {background: black !important;}

<h1 class="title">NightWing</h1>
```



Figure 4-2. Important rules always win



It's generally bad practice to use `!important` in your CSS, and it is rarely needed. If you find yourself reaching for `!important`, stop and look for other ways to get the same result without using `!important`. Cascade layers are one such possibility; see "Sorting by Cascade Layer" on page 119 for more details.

Inheritance

Another key concept in understanding how styles are applied to elements is *inheritance*. Inheritance is the mechanism by which some styles are applied not only to a specified element, but also to its descendants. If a color is applied to an `<h1>` element, for example, that color is applied to all text inside the `<h1>`, even the text enclosed within child elements of that `<h1>`:

```
h1 {color: gray;}

<h1>Meerkat <em>Central</em></h1>
```

Both the ordinary `<h1>` text and the `<em>` text are colored gray because the `<em>` element inherits the value of `color` from the `<h1>`. If property values could not be inherited by descendant elements, the `<em>` text would be black, not gray, and we'd have to color the elements separately.

Consider an unordered list. Let's say we apply a style of `color: gray;` for `<ul>` elements:

```
ul {color: gray;}
```

We expect that style applied to a `<ul>` will also be applied to its list items, as well as to any content of those list items, including the marker (i.e., the bullet next to each list item). Thanks to inheritance, that's exactly what happens, as Figure 4-3 demonstrates.

- Oh, don't you wish
- That you could be a fish
- And swim along with me
- Underneath the sea

1. Strap on some fins
2. Adjust your mask
3. Dive in!

Figure 4-3. Inheritance of styles

It's easier to see how inheritance works by turning to a tree diagram of a document. Figure 4-4 shows the tree diagram for a document much like the very simple document shown in Figure 4-3.

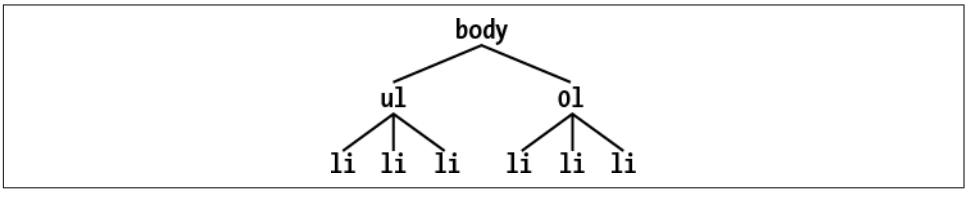


Figure 4-4. A simple tree diagram

When the declaration `color: gray;` is applied to the `<ul>` element, that element takes on that declaration. The value is then propagated down the tree to the descendant elements and continues on until no more descendants remain to inherit the value. Values are *never* propagated upward; an element never passes values up to its ancestors.



The upward propagation rule in HTML has a notable exception: background styles applied to the `<body>` element can be passed to the `<html>` element, which is the document's root element and therefore defines its canvas. This happens only if the `<body>` element has a defined background and the `<html>` element does not. A few other properties share this body-to-root behavior, such as `overflow`, but it happens only with the `<body>` element. No other elements risk inheriting properties from a descendant.

Inheritance is one of those things about CSS that is so basic that you almost never think about it unless you have to. However, you should still keep a couple of things in mind.

First, note that many properties are not inherited—generally in order to avoid undesirable outcomes. For example, the property `border` (which is used to set borders on elements) does not inherit. A quick glance at Figure 4-5 reveals why this is the case. If borders were

inherited, documents would become much more cluttered—unless the author took the extra effort to turn off the inherited borders.

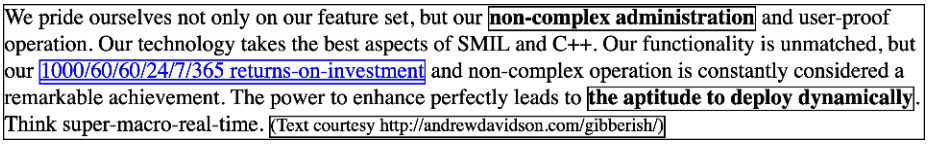


Figure 4-5. Why borders aren't inherited

As it happens, most of the box-model properties—including margins, padding, backgrounds, and borders—are not inherited for the same reason. After all, you likely wouldn't want all of the links in a paragraph to inherit a 30-pixel left margin from their parent element!

Second, inherited values have no specificity at all, not even zero specificity. This seems like an academic distinction until you work through the consequences of the lack of inherited specificity. Consider the following rules and markup fragment and compare them to the result shown in Figure 4-6:

```
* {color: gray;}
h1#page-title {color: black;}

<h1 id="page-title">Meerkat <em>Central</em></h1>
<p>
Welcome to the best place on the web for meerkat information!
</p>
```

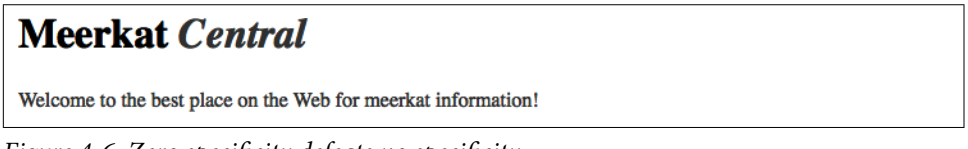


Figure 4-6. Zero specificity defeats no specificity

Since the universal selector applies to all elements and has zero specificity, its color declaration's value of `gray` wins out over the inherited value of `black`, which has no specificity at all. (And now you may understand why we listed `:where()` and the universal selector as having `0,0,0` specificity: they add no weight, but do match elements.) Therefore, the `<em>` element is rendered gray instead of black.

This example vividly illustrates one of the potential problems of using the universal selector indiscriminately. Because it can match *any* element or pseudo-element, the universal selector often has the effect of short-circuiting inheritance. This can be worked around, but it's usually more sensible to avoid the problem in the first place by not using the universal selector by itself indiscriminately.

The complete lack of specificity for inherited values is not a trivial point. For example, assume that a stylesheet has been written such that all text in a toolbar is to be white on black:

```
#toolbar {color: white; background: black;}
```

This will work so long as the element with an `id` of `toolbar` contains nothing but plain text. If, however, the text within this element is all hyperlinks (`a` elements), then the user agent's styles for hyperlinks will take over. In a web browser, this means they'll likely be colored blue, since the browser's internal stylesheet probably contains an entry like this:

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

To overcome this problem, you must declare something like this:

```
#toolbar {color: white; background: black;}
#toolbar a:any-link {color: white;}
```

By targeting a rule directly at the `a` elements within the toolbar, you'll get the result shown in Figure 4-7.



Figure 4-7. Directly assigning styles to the relevant elements

Another way to get the same result is to use the value `inherit`, covered in the next chapter. We can alter the previous example like so:

```
#toolbar {color: white; background: black;}
#toolbar a:link {color: inherit;}
```

This also leads to the result shown in Figure 4-7, because the value of `color` is explicitly inherited thanks to an assigned rule whose selector has specificity.

The Cascade

Throughout this chapter, we've skirted one rather important issue: what happens when two rules of equal specificity apply to the same element? How does the browser resolve the conflict? For example, consider the following rules:

```
h1 {color: red;}
h1 {color: blue;}
```

Which one wins? Both have a specificity of `0,0,1`, so they have equal weight and should both apply. That can't be the case because the element can't be both red and blue. So which will it be?

At last, the name *Cascading Style Sheets* comes into focus: CSS is based on a method of causing styles to *cascade* together, which is made possible by combining inheritance and specificity with a few rules. The cascade rules for CSS are as follows:

1. Find all rules containing a selector that matches a given element.
2. Sort all declarations applying to the given element by *explicit weight*.
3. Sort all declarations applying to the given element by *origin*. There are three basic origins: author, reader, and user agent. Under normal circumstances, the author's styles (that is, your styles as the author of the page) win out over the reader's styles, and both author and reader styles override the user agent's default styles. This is reversed for rules marked `!important`, where user agent styles override author styles, and both override reader styles.
4. Sort all declarations applying to the given element by *encapsulation context*. If a style is assigned via a shadow DOM, for example, it has an encapsulation context for all elements within that same shadow DOM and does not apply to elements outside that shadow DOM. This allows encapsulated styles to override styles that are inherited from outside the shadow DOM.
5. Sort all declarations by whether they are *element attached*. Styles assigned via a `style` attribute are element attached. Styles assigned from a stylesheet, whether external or embedded, are not.
6. Sort all declarations by *cascade layer*. For normal-weight styles, the later a cascade layer first appears in the CSS, the greater the precedence. Styles without a layer are considered to be part of a "default" final pseudo-layer, one that has higher precedence than styles in explicitly created layers. For important-weight styles, the *earlier* a cascade layer appears in the CSS, the greater its precedence, and all important-weight styles in explicitly created layers win out over styles in the default layer, important or otherwise. Cascade layers can appear in any origin.
7. Sort all declarations applying to the given element by *specificity*. Those elements with a higher specificity have more weight than those with lower specificity.
8. Sort all declarations applying to the given element by *order of appearance*. The later a declaration appears in the stylesheet or document, the more weight it is given. Declarations that appear in an imported stylesheet are considered to come before all declarations within the stylesheet that imports them.

To be clear about how this all works, let's consider examples that illustrate some of the cascade rules.

Sorting by Importance and Origin

If two rules apply to an element, and one is marked `!important`, the important rule wins out:

```
p {color: gray !important;}

<p style="color: black;">Well, <em>hello</em> there!</p>
```

Even though a color is assigned in the `style` attribute of the paragraph, the `!important` rule wins out, and the paragraph is gray. This occurs because sorting by `!important` has higher precedence than sorting by element-attached styles (`style=""`). The gray is inherited by the `<em>` element as well.

Note that if `!important` is added to the inline style in this situation, *it* will be the winner. Thus, given the following, the paragraph (and its descendant element) will be black:

```
p {color: gray !important;}

<p style="color: black !important;">Well, <em>hello</em> there!</p>
```

If the importance is the same, the origin of a rule is considered. If an element is matched by normal styles in both the author's stylesheet and the reader's stylesheet, the author's styles are used. For example, assume that the following styles come from the indicated origins:

```
p em {color: black;}    /* author's stylesheet */

p em {color: yellow;}   /* reader's stylesheet */
```

In this case, emphasized text within paragraphs is colored black, not yellow, because the author styles win out over the reader styles. However, if both rules are marked `!important`, the situation changes:

```
p em {color: black !important;}    /* author's stylesheet */

p em {color: yellow !important;}   /* reader's stylesheet */
```

Now the emphasized text in paragraphs will be yellow, not black.

As it happens, the user agent's default styles—which are often influenced by the user preferences—are figured into this step. The default style declarations are the least influential of all. Therefore, if an author-defined rule applies to anchors (e.g., declaring them to be `white`), then this rule overrides the user agent's defaults.

To sum up, CSS has eight basic levels to consider in terms of declaration precedence. In order of most to least precedence, these are as follows:

1. Transition declarations (see Chapter 18)
2. User agent important declarations
3. Reader important declarations
4. Author important declarations

5. Animation declarations (see Chapter 19)
6. Author normal declarations
7. Reader normal declarations
8. User agent declarations

Thus, a transition style will override all other rules, regardless of whether those other rules are marked `!important` or from what origin the rules come.

Sorting by Element Attachment

Styles can be attached to an element by using a markup attribute such as `style`. These are called *element-attached* styles, and they are outweighed only by considerations of origin and weight.

To understand this, consider the following rule and markup fragment:

```
h1 {color: red;}

<h1 style="color: green;">The Meadow Party</h1>
```

Given that the rule is applied to the `<h1>` element, you would still probably expect the text of the `<h1>` to be green. This happens because every inline declaration is element attached, and so has a higher weight than styles that aren't element attached, like the `color: red` rule.

This means that even elements with `id` attributes that match a rule will obey the inline style declaration. Let's modify the previous example to include an `id`:

```
h1#meadow {color: red;}

<h1 id="meadow" style="color: green;">The Meadow Party</h1>
```

Thanks to the inline declaration's weight, the text of the `<h1>` element will still be green.

Just remember that inline styles are generally a bad practice, so try not to use them if at all possible.

Sorting by Cascade Layer

Cascade layers allow authors to group styles together so that they share a precedence level within the cascade. This might sound like `!important`; in some ways they are similar—but in others, very different. This is easier to demonstrate than it is to describe. The ability to create cascade layers means authors can balance various needs, such as the needs of a component library, against the needs of a specific page or part of a web app.



Cascade layers were introduced to CSS at the end of 2021, so browser support for them exists only in browsers released from that point forward.

If conflicting declarations apply to an element and all have the same explicit weight and origin, and none are element attached, they are next sorted by cascade layer. The order of precedence for layers is set by the order in which the layers are first declared or used, with later declared layers taking precedence over earlier declared layers for normal styles. Consider the following:

```
@layer site {
     h1 {color: red;}
}
@layer page {
     h1 {color: blue;}
}
```

These `<h1>` elements will be colored blue. This is because the `page` layer comes later in the CSS than the `site` layer, and so has higher precedence.

Any style not part of a named cascade layer is assigned to an implicit "default" layer, one that has higher precedence than any named layer for unimportant rules. Suppose we alter the previous example as follows:

```
h1 {color: maroon;}
@layer site {
     h1 {color: red;}
}
@layer page {
     h1 {color: blue;}
}
```

The `<h1>` elements will now be maroon, because the implicit "default" layer to which the `h1 {color: maroon;}` belongs has higher precedence than any named layer.

You can also define a specific precedence order for named cascade layers. Consider the following CSS:

```
@layer site, page;

@layer page {
   h1 {color: blue;}
}

@layer site {
   h1 {color: red;}
}
```

Here, the first line defines an order of precedence for the layers: the `page` layer will be given higher precedence than the `site` layer for normal-weight rules like those shown in the example. Thus, in this case, `<h1>` elements will be blue, because when the layers are sorted, `page` is given more precedence than `site`. For important-flagged rules, the order of precedence is reversed. Thus, if both rules were marked `!important`, the precedence would flip and `<h1>` elements would be red.

Let's talk a little bit more about how cascade layers specifically work, especially since they're so new to CSS. Let's say you want to define three layers: one for the basic site styles, one for individual page styles, and one for a component library whose styles are imported from an external stylesheet. The CSS might look like this:

```
@layer site, page;
@import url(/assets/css/components.css) layer(components);
```

This ordering will have normal-weight `components` styles override and `page` and `site` normal-weight styles, and normal-weight `page` styles will override only `site` normal-weight styles. Conversely, important `site` styles will override all `page` and `components` styles, whether they're important or normal weight, and `page` important styles will override all `components` styles.

Here's a small example of how layers might be managed:

```
@layer site, component, page;
@import url(/c/lib/core.css) layer(component);
@import url(/c/lib/widgets.css) layer(component);
@import url(/c/site.css) layer(site);

@layer page {
   h1 {color: maroon;}
   p {margin-top: 0;}
}

@layer site {
   body {font-size: 1.1rem;}
   h1 {color: orange;}
   p {margin-top: 0.5em;}
}

p {margin-top: 1em;}
```

This example has three imported stylesheets, one of which is assigned to the `site` layer and two of which are in the `component` layer. Then some rules are assigned to the `page` layer, and a couple of rules are placed in the `site` layer. The rules in the `@layer site {}` block will be combined with the rules from `/c/site.css` into a single `site` layer.

After that, there's a rule outside the explicit cascade layers, which means it's part of the implicit "default" layer. Rules in this default layer will override the styles of any of the other layers. So, given the code shown, paragraphs will have top margins of `1em`.

But before all of that, a directive sets the precedence order of the named layers: `page` overrules `component` and `site`, and `component` overrules `site`. Here's how those various rules are grouped as far as the cascade is concerned, with comments to describe their placement in the sorting:

```
/* 'site' layer is the lowest weighted */
@import url(/c/site.css) layer(site);
@layer site {
   body {font-size: 1.1rem;}
```

```
    h1 {color: orange;}
    p {margin-top: 0.5em;}
}

/* 'component' layer is the next-lowest weighted */
@import url(/c/lib/core.css) layer(component);
@import url(/c/lib/widgets.css) layer(component);

/* 'page' layer is the next-highest weighted */
@layer page {
    h1 {color: maroon;}
    p {margin-top: 0;}
}

/* the implicit layer is the highest weighted */
p {margin-top: 1em;}
```

As you can see, the later a layer comes in the ordering of the layers, the more weight it's given by the cascade's sorting algorithm.

To be clear, cascade layers don't have to be named. Naming just keeps things a lot clearer in terms of setting an order for them, and it also enables adding styles to the layer. Here are some examples of using unnamed cascade layers:

```
@import url(base.css) layer;

p {margin-top: 1em;}

@layer {
    h1 {color: maroon;}
    body p {margin-top: 0;}
}
```

In this case, the rules imported from *base.css* are assigned to an unnamed layer. Even though this layer doesn't actually have a name, let's think of it as CL1. Then a rule outside the layers sets paragraph top margins to `1em`. Finally, an unnamed layer block has a couple of rules; let's think of this layer as CL2.

So now we have rules in three layers: CL1, CL2, and the implicit layer. And that's the order they're considered in, so in the case of any conflicting normal rules, the rules in the implicit default layer (which comes last in the ordering) will win over conflicting rules in the other two layers, and rules in CL2 will win over conflicting rules in CL1.

At least, that's the case for normal-weight rules. For `!important` rules, the order of precedence is flipped, so those in CL1 will win over conflicting important rules in the other two layers, and important rules in CL2 win over conflicting important rules in the implicit layer. Strange but true!

This sorting by order will come up again in just a little bit, but first let's bring specificity into the cascade.

Sorting by Specificity

If conflicting declarations apply to an element and those declarations all have the same explicit weight, origin, element attachment (or lack thereof), and cascade layer, they are then sorted by specificity. The most specific declaration wins out, like this:

```
@layer page {
  p#bright#bright#bright {color: grey;}
}
p#bright {color: silver;}
p {color: black;}

<p id="bright">Well, hello there!</p>
```

Given these rules, the text of the paragraph will be silver, as illustrated in Figure 4-8. Why? Because the specificity of `p#bright` (`1,0,1`) overrides the specificity of `p` (`0,0,1`), even though the latter rule comes later in the stylesheet. The styles from the `page` layer, even though they have the strongest selector (`3,0,1`), aren't even compared. Only the declarations from the layer with precedence are in contention.

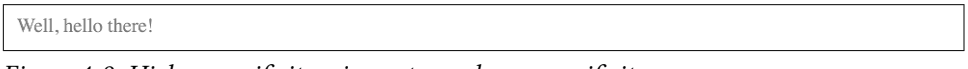


Figure 4-8. Higher specificity wins out over lower specificity

Remember that this rule applies only if the rules are part of the same cascade layer. If not, specificity doesn't matter: a `0,0,1` selector in the implicit layer will win over any unimportant rule in an explicitly created cascade layer, no matter how high the latter's specificity gets.

Sorting by Order

Finally, if two rules have exactly the same explicit weight, origin, element attachment, cascade layer, and specificity, then the one that appears later in the stylesheet wins out, similar to the way cascade layers are sorted in order so that later layers win over earlier layers.

Let's return to an earlier example, where we find the following two rules in the document's stylesheet:

```
body h1 {color: red;}
html h1 {color: blue;}
```

In this case, the value of `color` for all `<h1>` elements in the document will be `blue`, not `red`. This is because the two rules are tied with each other in terms of explicit weight and origin, are in the same cascade layer, and the selectors have equal specificity, so the last one declared is the winner. It doesn't matter how close together the elements are in the document tree; even though `<body>` and `<h1>` are closer together than `<html>` and `<h1>`, the later one wins. The only thing that matters (when the origin, cascade layer, layer, and specificity are the same) is the order in which the rules appear in the CSS.

So what happens if rules from completely separate stylesheets conflict? For example, suppose the following:

```
@import url(basic.css);
h1 {color: blue;}
```

What if `h1 {color: red;}` appears in *basic.css*? In this case, since there are no cascade layers in play, the entire contents of *basic.css* are treated as if they were pasted into the stylesheet at the point where the `@import` occurs. Thus, any rule contained in the document's stylesheet occurs later than those from the `@import`. If they tie in terms of explicit weight and specificity, the document's stylesheet contains the winner. Consider the following:

```
p em {color: purple;}  /* from imported stylesheet */

p em {color: gray;}    /* rule contained within the document */
```

In this case, the second rule wins out over the imported rule because it is the last one specified, and both are in the implicit cascade layer.

Order sorting is the reason behind the often-recommended ordering of link styles. The recommendation is that you write your link styles in the order `link`, `visited`, `focus`, `hover`, `active`, or LVFHA, like this:

```
a:link {color: blue;}
a:visited {color: purple;}
a:focus {color: green;}
a:hover {color: red;}
a:active {color: orange;}
```

Thanks to the information in this chapter, you now know that the specificity of all of these selectors is the same: `0,1,1`. Because they all have the same explicit weight, origin, and specificity, the last one that matches an element will win out. An unvisited link that is being clicked or otherwise activated, such as via the keyboard, is matched by four of the rules—`:link`, `:focus`, `:hover`, and `:active`—so the last one of those four will win out. Given the LVFHA ordering, `:active` will win, which is likely what the author intended.

Assume for a moment that you decide to ignore the common ordering and alphabetize your link styles instead. This would yield the following:

```
a:active {color: orange;}
a:focus {color: green;}
a:hover {color: red;}
a:link {color: blue;}
a:visited {color: purple;}
```

Given this ordering, no link would ever show `:hover`, `:focus`, or `:active` styles because the `:link` and `:visited` rules come after the other three. Every link must be either visited or unvisited, so those styles will always override the others.

Let's consider a variation on the LVFHA order that an author might want to use. In this ordering, only unvisited links will get a hover style; visited links will not. Both visited and unvisited links will get an active style:

```
a:link {color: blue;}
a:hover {color: red;}
a:visited {color: purple;}
a:focus {color: green;}
a:active {color: orange;}
```

Such conflicts arise only when all the states attempt to set the same property. If each state's styles address a different property, the order does not matter. In the following case, the link styles could be given in any order and would still function as intended:

```
a:link {font-weight: bold;}
a:visited {font-style: italic;}
a:focus {color: green;}
a:hover {color: red;}
a:active {background: yellow;}
```

You may also have realized that the order of the `:link` and `:visited` styles doesn't matter. You could order the styles LVFHA or VLFHA with no ill effect.

The ability to chain pseudo-classes together eliminates all these worries. The following could be listed in any order without any overrides, as the specificity of the latter two is greater than that of the first two:

```
a:link {color: blue;}
a:visited {color: purple;}
a:link:hover {color: red;}
a:visited:hover {color: gray;}
```

Because each rule applies to a unique set of link states, they do not conflict. Therefore, changing their order will not change the styling of the document. The last two rules do have the same specificity, but that doesn't matter. A hovered unvisited link will not be matched by the rule regarding hovered visited links, and vice versa. If we were to add active-state styles, order would start to matter again. Consider this:

```
a:link {color: blue;}
a:visited {color: purple;}
a:link:hover {color: red;}
a:visited:hover {color: gray;}
a:link:active {color: orange;}
a:visited:active {color: silver;}
```

If the active styles were moved before the hover styles, they would be ignored. Again, this would happen because of specificity conflicts. The conflicts could be avoided by adding more pseudo-classes to the chains, like this:

```
a:link:hover:active {color: orange;}
a:visited:hover:active {color: silver;}
```

This does have the effect of raising the specificity of the selectors—both have a specificity value of `0,3,1`—but they don't conflict because the actual selection states are mutually

exclusive. A link can't be both a visited hovered active link *and* an unvisited hovered active link: only one of the two rules will match.

Working with Non-CSS Presentational Hints

A document could contain presentational hints that are not CSS—for example, the deprecated `<font>` element, or the still very much used `height`, `width`, and `hidden` attributes. Such presentational hints will be overridden by any author or reader styles, but not by the user agent's styles. In modern browsers, presentational hints from outside CSS are treated as if they belong to the user agent's stylesheet.

Summary

Perhaps the most fundamental aspect of Cascading Style Sheets is the cascade itself—the process used to sort out conflicting declarations and determine the final document presentation. Integral to this process is the specificity of selectors and their associated declarations, and the mechanism of inheritance.

CHAPTER 5

Values and Units

In this chapter, we'll tackle features that are the basis for almost everything you can do with CSS: the units that affect the colors, distances, and sizes of a whole host of properties, as well as the units that help define those values. Without units, you couldn't declare that an image should have 10 pixels of blank space around it, or that a heading's text should be a certain size. By understanding the concepts put forth here, you'll be able to learn and use the rest of CSS much more quickly.

Keywords, Strings, and Other Text Values

Everything in a stylesheet is text, but certain value types directly represent strings of text as opposed to, say, numbers or colors. Included in this category are URLs and, interestingly enough, images.

Keywords

For those times when a value needs to be described with a word of some kind, CSS has *keywords*. A common example is the keyword `none`, which is distinct from `0` (zero). Thus, to remove the underline from links in an HTML document, you would write the following:

```
a[href] {text-decoration: none;}
```

Similarly, if you want to force underlines on the links, you would use the keyword `underline` instead of `none`.

If a property accepts keywords, its keywords will be defined only for the scope of that property. If two properties use the same word as a keyword, the behavior of the keyword for one property will not necessarily be shared with the other. As an example, `normal`, as defined for `letter-spacing`, means something very different from the `normal` defined for `font-style`.

Global keywords

CSS defines five *global* keywords that are accepted by every property in the specification: `inherit`, `initial`, `unset`, `revert`, and `revert-layer`.

inherit. The keyword `inherit` makes the value of a property on an element the same as the value of that property on its parent element. In other words, it forces inheritance to occur even in situations where it would not normally operate. In many cases, you don't need to specify inheritance, since many properties inherit naturally. Nevertheless, `inherit` can still be useful.

For example, consider the following styles and markup:

```
#toolbar {background: blue; color: white;}

<div id="toolbar">
<a href="one.html">One</a> | <a href="two.html">Two</a> |
<a href="three.html">Three</a>
</div>
```

The `<div>` itself will have a blue background and a white foreground, but the links will be styled according to the browser's preference settings. They'll most likely end up as blue text on a blue background, with white vertical bars between them.

You could write a rule that explicitly sets the links in the toolbar to be white, but you can make things a little more robust by using `inherit`. You just add the following rule to the stylesheet:

```
#toolbar a {color: inherit;}
```

This will cause the links to use the inherited value of `color` in place of the user agent's default styles.

Ordinarily, directly assigned styles override inherited styles, but `inherit` can undo that behavior. It might not always be a good idea—for example, here links might blend into surrounding text too much, and become a usability and accessibility concern—but it can be done.

Similarly, you can pull a property value down from a parent even if it wouldn't happen normally. Take `border`, for example, which is (rightfully) not inherited. If you want a `<span>` to inherit the border of its parent, all you need is `span {border: inherit;}`. More likely, though, you just want the border on a `<span>` to use the same border color as its parent. In that case, `span {border-color: inherit;}` will do the trick.

initial. The keyword `initial` sets the value of a property to the defined initial value, which, in a way, means it "resets" the value. For example, the default value of `font-weight` is `normal`. Therefore, declaring `font-weight: initial` is the same as declaring `font-weight: normal`.

This might seem a little bit silly until you consider that not all values have explicitly defined initial values. For example, the initial value for `color` "depends on user agent." That's not a funky keyword you should type! What it means is that the default value of `color` depends on things like the preference settings in a browser. While almost nobody changes the default text color setting from black, someone might set it to a dark gray or even a bright red. By declaring `color: initial;`, you're telling the browser to set the color of the element to whatever the user's default color is set to be.

Another benefit of `initial` is that you can set a property back to its initial value without having to know that initial value. This can be especially useful when resetting a lot of properties all at once, either via JS or CSS.

unset. The keyword `unset` acts as a universal stand-in for both `inherit` and `initial`. If the property is inherited, `unset` has the same effect as if `inherit` were used. If the property is *not* inherited, `unset` has the same effect as if `initial` were used. This makes `unset` useful for resetting a property by canceling out any other styles that might be applied to it.

revert. The keyword `revert` sets the value of a property to the value the property would have had if no changes had been made by the current style origin. In effect, `revert` lets you say, "All property values for this element should be as if the author styles don't exist, but user agent and user styles do exist."

Thus, given the following basic example, `p` elements will be rendered as gray text with a transparent background:

```
p {background: lime; color: gray;}
p {background: revert;}
```

This does mean that any property whose value is inherited will be given the same value as that of its parent. The `revert` keyword is useful when you have a bunch of site-wide styles applying to an element, and you want to strip them all away so as to apply a set of one-off styles to just that element. Rather than having to override all those properties, you can revert them to defaults—and you can do it with a single property, `all`, which is the topic of the next section.

revert-layer. If you're using cascade layers (see "Sorting by Cascade Layer" on page 119) and want to "undo" whatever styles might be applied by the current layer, the `revert-layer` value is here to help. The difference here is that `revert-layer` effectively means, "All property values for this element should be as if the author styles *in the current cascade layer* don't exist, but other author cascade layers (including the default), user agent, and user styles do exist."

Thus, given the following, paragraphs with a `class` containing the word `example` will be rendered as red text on a yellow background:

```
@layer site, system;

p {color: red;}
```

```
@layer system {
        p {background: yellow; color: fuchsia;}
}
@layer site {
        p {background: lime; color: gray;}
        p.example {background: revert; color: revert;}
}
```

For the background, the browser looks at the assigned values in previous cascade layers and picks the one with the highest weight. Only one layer (`system`) sets a background color, so that's what's used instead of `lime`. The same is done for the foreground color, and since a color is assigned in the default layer, and the default layer overrides all explicitly created layers, `red` is used instead of `gray`.



As of late 2023, only Firefox supports `revert-layer`, but we anticipate it being widely supported in the near future.

The all Property

These global values are usable on all properties, but one special property accepts *only* the global keywords: `all`.

all

Values	`inherit` | `initial` | `unset` | `revert`
Initial value	See individual properties

The `all` property is a stand-in for all properties *except* `direction`, `unicode-bidi`, and any custom properties (see "Custom Properties" on page 168). Thus, if you declare `all: inherit` on an element, you're saying that you want all properties except `direction`, `unicode-bidi`, and custom properties to inherit their values from the element's parent. Consider the following:

```
section {color: white; background: black; font-weight: bold;}
#example {all: inherit;}

<section>
    <div id="example">This is a div.</div>
</section>
```

You might think this causes the `<div>` element to inherit the values of `color`, `background`, and `font-weight` from the `<section>` element. And it does do that, yes—but it will *also*

force inheritance of the values of *every single other property in CSS* (minus the two exceptions) from the `<section>` element.

Maybe that's what you want, in which case, great. But if you just want to inherit the property values you wrote out for the `<section>` element, the CSS would need to look more like this:

```
section {color: white; background: black; font-weight: bold;}
#example {color: inherit; background: inherit; font-weight: inherit;}
```

Odds are that what you really want in these situations is `all: unset`, but your stylesheet may vary.

Strings

A *string value* is an arbitrary sequence of characters wrapped in either single or double quotes, and is represented in value definitions with *<string>*. Here are two simple examples:

```
"I like to play with strings."
'Strings are fun to play with.'
```

Note that the quotes balance, which is to say that you always start and end with the same kind of quotes. Getting this wrong can lead to all kinds of parsing problems, since starting with one kind of quote and trying to end with the other means the string won't actually be terminated. You could accidentally incorporate subsequent rules into the string that way!

If you want to put quote marks inside strings, that's OK, as long as they're either not the kind you used to enclose the string or are escaped using a backslash:

```
"I've always liked to play with strings."
'He said to me, "I like to play with strings."'
"It's been said that \"haste makes waste.\""
'There\'s never been a "string theory" that I\'ve liked.'
```

Note that the only acceptable string delimiters are `'` and `"`, sometimes called *straight quotes*. That means you can't use *curly* or *smart* quotes to begin or end a string value. You can use them inside a string value, as in this code example, though, and they don't have to be escaped:

```
"It’s been said that “haste makes waste.”"
'There’s never been a “string theory” that I’ve liked.'
```

This requires that you use Unicode encoding (using the Unicode standard) for your documents, but you should be doing that regardless.

If you have some reason to include a newline in your string value, you can do that by escaping the newline itself. CSS will then remove it, making things as if it had never been there. Thus, the following two string values are identical from a CSS point of view:

```
"This is the right place \
for a newline."
"This is the right place for a newline."
```

If, on the other hand, you actually want a string value that includes a newline character, use the Unicode reference `\A` where you want the newline to occur:

```
"This is a better place \Afor a newline."
```

Identifiers

One-word, case-sensitive strings that should not be quoted are known as *identifiers*, represented in the CSS syntax as <*ident*> or <*custom-ident*>, depending on the specification and context. Identifiers are used as animation names, grid-line names, and counter names, among others. In addition, <*dashed-ident*> is used for custom properties. Rules for creating a custom identifier include not starting the word with a number, a double hyphen, or a single hyphen followed by a number. Other than that, really any character is valid, including emojis, but if you use certain characters, including a space or a backslash, you need to escape them with a backslash.

Identifiers themselves are words and are case-sensitive; thus, `myID` and `MyID` are, as far as CSS is concerned, completely distinct and unrelated to each other. If a property accepts both an identifier and one or more keywords, the author should take care to never define an identifier that is identical to a valid keyword, including the global keywords `initial`, `inherit`, `unset`, and `revert`. Using `none` is also a really bad idea.

URLs

If you've written web pages, you're almost certainly familiar with uniform resource locators (URLs). Whenever you need to refer to one—as in the `@import` statement, which is used when importing an external stylesheet—here is the general format:

```
url(protocol://server/pathname/filename)
url("<string>")   /* can use single or double quotes, or no quotes. */
```

This example defines an *absolute URL*. This URL will work no matter where (or rather, in what page) it's found, because it defines an absolute location in web space. Let's say that you have a server called *web.waffles.org*. On that server is a directory called *pix*, and in this directory is an image *waffle22.gif*. In this case, the absolute URL of that image would be as follows:

```
https://web.waffles.org/pix/waffle22.gif
```

This URL is valid no matter where it is written, whether the page containing it is located on the server *web.waffles.org* or *web.pancakes.com*.

The other type of URL is a *relative URL*, so named because it specifies a location that is relative to the document that uses it. If you're referring to a relative location, such as a file in the same directory as your web page, the general format is as follows:

```
url(pathname)
url("<string>")   /* can use single or double quotes. */
```

This works only if the image is on the same server as the page that contains the URL. For argument's sake, assume that you have a web page located at *http://web.waffles.org/syrup.html* and that you want the image *waffle22.gif* to appear on this page. In that case, the URL would be the following:

```
pix/waffle22.gif
```

This path works because the web browser knows it should start with the place it found the web document and then add the relative URL. In this case, the pathname *pix/waffle22.gif* added to the server name *http://web.waffles.org* equals *http://web.waffles.org/pix/waffle22.gif*. You can almost always use an absolute URL in place of a relative URL; it doesn't matter which you use, as long as it defines a valid location.

In CSS, relative URLs are relative to the stylesheet itself, not to the HTML document that uses the stylesheet. For example, you may have an external stylesheet that imports another stylesheet. If you use a relative URL to import the second stylesheet, it must be relative to the first stylesheet. In fact, if you have a URL in any imported stylesheet, it needs to be relative to the imported stylesheet.

As an example, consider an HTML document at *http://web.waffles.org/toppings/tips.html*, which has a `<link>` to the stylesheet *http://web.waffles.org/styles/basic.css*:

```
<link rel="stylesheet" type="text/css"
    href="http://web.waffles.org/styles/basic.css">
```

Inside the file *basic.css* is an `@import` statement referring to another stylesheet:

```
@import url(special/toppings.css);
```

This `@import` will cause the browser to look for the stylesheet at *http://web.waffles.org/styles/special/toppings.css*, not at *http://web.waffles.org/toppings/special/toppings.css*. If you have a stylesheet at the latter location, the `@import` in *basic.css* should read one of the two following ways:

```
@import url(http://web.waffles.org/toppings/special/toppings.css);

@import url("../special/toppings.css");
```

Note that there cannot be a space between the `url` and the opening parenthesis:

```
body {background: url(http://www.pix.web/picture1.jpg);}   /* correct */
body {background: url  (images/picture2.jpg);}          /* INCORRECT */
```

If the space is present, the entire declaration will be invalidated and thus ignored.



As of late 2022, the CSS Working Group is planning to introduce a new function called `src()`, which will accept only strings and not unquoted URLs. This is meant to allow custom properties to be used inside `src()`, which will let authors define which file should be loaded based on the value of a custom property.

Images

An *image value* is a reference to an image, as you might have guessed. Its syntax representation is <*image*>.

At the most basic level of support, which is to say the one every CSS engine on the planet would understand, an <*image*> value is a <*url*> value. In more modern user agents, <*image*> stands for one of the following:

<*url*>
: A URL identifier of an external resource—in this case, the URL of an image.

<*gradient*>
: Refers to either a linear, radial, or conic gradient image, either singly or in a repeating pattern. Gradients are fairly complex and are covered in detail in Chapter 9.

<*image-set*>
: A set of images, chosen based on a set of conditions embedded into the value, which is defined as `image-set()` but is more widely recognized with the `-webkit-` prefix. For example, `-webkit-image-set()` could specify that a larger image be used for desktop layouts, whereas a smaller image (both in pixel size and file size) be used for a mobile design. This value is intended to at least approximate the behavior of the `srcset` attribute for `<picture>` elements. As of early 2023, `-webkit-image-set` is basically universally supported, with most browsers other than Safari also accepting `image-set()` (without the prefix).

<*cross-fade*>
: Used to blend two (or more) images together, with a specific transparency given to each image. Use cases include blending two images together, blending an image with a gradient, and so on. As of early 2023, this is supported as `-webkit-cross-fade()` in Blink- and WebKit-based browsers, and not supported at all in the Firefox family, with or without the prefix.

There are also the `image()` and `element()` functions, but as of early 2023, neither is supported by any browser, except for a vendor-prefixed version of `element()` supported by Firefox. Finally, `paint()` refers to an image painted by CSS Houdini's PaintWorklet. As of early 2023, this is supported in only a basic form by Blink-based browsers like Chrome.

Numbers and Percentages

Numbers and percentages serve as the foundation for many other values types. For example, font sizes can be defined using the `em` unit (covered later in this chapter) preceded by a number. But what kind of number? Understanding the types of numbers here will help you better grasp defining other value types later.

Integers

An *integer value* is about as simple as it gets: one or more numbers, optionally prefixed by a `+` or `–` (plus or minus) sign to indicate a positive or negative value. That's it. Integer values are represented in value syntax as <*integer*>. Examples include `712`, `13`, `–42`, and `1066`.

Some properties define a range of acceptable integer values. Integer values that fall outside a defined range are, by default, considered invalid and cause the entire declaration to be ignored. However, some properties define behavior that causes values outside the accepted range to be set to the accepted value closest to the declared value, known as *clamping*.

In cases (such as the property `z-index`) where there is no restricted range, user agents must support values up to ±1,073,741,824 ($\pm 2^{30}$).

Numbers

A *number value* is either an <*integer*> or a real number, which is to say an integer followed by a dot and then some number of following integers. Additionally, it can be prefixed by either `+` or `-` to indicate positive or negative values. Number values are represented in value syntax as <*number*>. Examples include `5`, `2.7183`, `–3.1416`, `6.2832`, and `1.0218e29` (scientific notation).

The reason a <*number*> can be an <*integer*> and yet they are separate value types is that some properties will accept only integers (e.g., `z-index`), whereas others will accept any real number (e.g., `flex-grow`).

As with integer values, number values may have limits imposed on them by a property definition; for example, `opacity` restricts its value to be any valid <*number*> in the range `0` to `1`, inclusive. Some properties define behavior that causes values outside the accepted range to be clamped to an acceptable value closest to the declared value; e.g., `opacity: 1.7` would be clamped to `opacity: 1`. For those that do not, number values that fall outside a defined range are considered invalid and cause the entire declaration to be ignored.

Percentages

A *percentage value* is a <*number*> followed by a percentage sign (`%`), and is represented in value syntax as <*percentage*>. Examples include `50%` and `33.333%`. Percentage values are always relative to another value, which can be anything—the value of another property of the same element, a value inherited from the parent element, or a value of an ancestor element. Properties that accept percentage values will define any restrictions on the range of allowed percentage values, as well as the way in which the percentage is relatively calculated.

Fractions

A *fraction value* (or *flexible ratio*) is a <*number*> followed by the `fr` unit label. Thus, one fractional unit is `1fr`. The `fr` unit represents a fraction of the leftover space, if any, in a grid container.

As with all CSS dimensions, there is no space between the unit and the number. Fraction values are not lengths (nor are they compatible with <*length*> values, unlike some <*percentage*> values), so they cannot be used with other unit types in `calc()` functions.



Fraction values are mostly used in grid layout (see Chapter 12), but there are plans to use it in more contexts, such as the planned (as of early 2023) `stripes()` function.

Distances

Many CSS properties, such as margins, depend on length measurements to properly display various page elements. It's likely no surprise, then, CSS provides multiple ways to measure length.

All length units can be expressed as either positive or negative numbers followed by a label, although note that some properties will accept only positive numbers. You can also use real numbers—that is, numbers with decimal fractions, such as 10.5 or 4.561.

All length units are followed by a short abbreviation that represents the actual unit of length being specified, such as `in` (inches) or `pt` (points). The only exception to this rule is a length of `0` (zero), which need not be followed by a unit when describing lengths.

These length units are divided into two types: *absolute length units* and *relative length units*.

Absolute Length Units

We'll start with absolute units because they're easiest to understand. The seven types of absolute units are as follows:

Inches (`in`)
: As you might expect, this notation refers to the inches you'd find on a ruler in the US. (The fact that this unit is in the specification, even though almost the entire world uses the metric system, is an interesting insight into the pervasiveness of US interests on the internet—but let's not get into virtual sociopolitical theory right now.)

Centimeters (`cm`)
: Refers to the centimeters that you'd find on rulers the world over. There are 2.54 centimeters to an inch, and one centimeter equals 0.394 inches.

Millimeters (`mm`)
: For those Americans who are metric challenged, 10 millimeters are in 1 centimeter, so an inch equals 25.4 millimeters, and a millimeter equals 0.0394 inches.

Quarter-millimeters (`Q`)
: There are 40 Q units in a centimeter; thus, setting an element to be 1/10 of a centimeter wide—which is also a millimeter wide—would mean a value of `4Q`.

Points (`pt`)
: Points are standard typographical measurements that have been used by printers and typesetters for decades and by word processing programs for many years. Traditionally, there are 72 points to an inch. Therefore the capital letters of text set to 12 points should be one-sixth of an inch tall. For example, `p {font-size: 18pt;}` is equivalent to `p {font-size: 0.25in;}`.

Picas (`pc`)
: A pica, which is another typographical term, is equivalent to 12 points, which means there are 6 picas to an inch. As just shown, the capital letters of text set to 1 pica should be one-sixth of an inch tall. For example, `p {font-size: 1.5pc;}` would set text to the same size as the example declarations found in the definition of points.

Pixels (`px`)
: A pixel is a small box onscreen, but CSS defines pixels more abstractly. In CSS terms, a pixel is defined to be the size required to yield 96 pixels per inch. Many user agents ignore this definition in favor of simply addressing the pixels on the screen. Scaling factors are brought into play when page zooming or printing, where an element `100px` wide can be rendered more than 100 device dots wide.

These units are really useful only if the browser knows all the details of the screen on which your page is displayed, the printer you're using, or whatever other user agent might apply. On a web browser, display is affected by the size of the screen and the resolution to which the screen is set; there isn't much that you, as the author, can do about these factors. If nothing else, measurements should be consistent in relation to each other—that is, a setting of `1.0in` should be twice as large as `0.5in`, as shown in Figure 5-1.

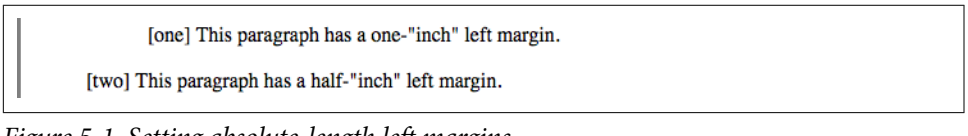


Figure 5-1. Setting absolute-length left margins

Let's make the (fairly suspect) assumption that your computer knows enough about its display system to accurately reproduce real-world measurements. You could make sure every paragraph has a top margin of half an inch by declaring `p {margin-top: 0.5in;}` in that case.

Absolute units are much more useful in defining stylesheets for printed documents, where measuring things in terms of inches, points, and picas is common.

Pixel lengths

On the face of things, pixels are straightforward. If you look at a screen closely enough, you can see that it's broken into a grid of tiny little boxes. Each box is a pixel. Say you define an element to be a certain number of pixels tall and wide, as in the following markup:

```
<p>
The following image is 20 pixels tall and wide: <img src="test.gif"
  style="width: 20px; height: 20px;" alt="" />
</p>
```

Then it follows that the element will be that many screen elements tall and wide, as shown in Figure 5-2.



Figure 5-2. Using pixel lengths

The problem is, thanks to high-density displays like those found on mobile devices and modern laptops, the individual screen elements aren't treated as pixels anymore. Instead, the pixels used in your CSS are translated into something that aligns with human expectations, which is covered in the next section.

Pixel theory

In its discussion of pixels, the CSS specification recommends that, when a display's resolution density is significantly different from 96 pixels per inch (ppi), user agents should scale pixel measurements to a reference pixel.

The W3C defines *reference pixel* as follows:

> The visual angle of one pixel on a device with a device pixel density of 96dpi and a distance from the reader of an arm's length. For a nominal arm's length of 28 inches, the visual angle is therefore about 0.0213 degrees. For reading at arm's length, 1px thus corresponds to about 0.26 mm (1/96 inch).

On most modern displays, the actual number of pixels per inch (ppi) is higher than 96—sometimes much higher. The Retina display on an iPhone 13, for example, is physically 326 ppi, and the display on the iPad Pro is physically 264 ppi. As long as a browser on one of those devices sets the reference pixel such that an element set to be `10px` tall appears to be 2.6 millimeters tall on the screen, the physical display density isn't something you have to worry about, any more than having to worry about the number of dots per inch on a printout.

Resolution Units

Some unit types are based on display resolution:

Dots per inch (`dpi`)
: The number of display dots per linear inch. This can refer to the dots on a paper printer's output, the physical pixels on an LED screen or other device, or the elements in an e-ink display such as that used by a Kindle.

Dots per centimeter (`dpcm`)
: Same as `dpi`, except the linear measure is 1 centimeter instead of 1 inch.

Dots per pixel unit (`dppx`)
: The number of display dots per CSS `px` unit, with `1dppx` being equivalent to `96dpi` because CSS defines pixel units at that ratio. Just bear in mind that ratio could change in future versions of CSS.

These units are most often used in the context of media queries. As an example, an author can create a media block to be used only on displays that have higher than 500 dpi:

```
@media (min-resolution: 500dpi) {
    /* rules go here */
}
```

Again, it's important to remember that CSS pixels are *not* device resolution pixels. Text with `font-size: 16px` will be a relatively consistent size whether the device has 96 dpi or 470 dpi. While a reference pixel is defined to appear to be 1/96th of an inch in size, when a device has more than 96 dpi, the content will not look smaller. Zooming is created by expanding CSS pixels as much as is needed; an image will appear larger, but the image size doesn't actually change: rather, the width of the screen, in terms of reference pixels, gets smaller.

Relative Length Units

Relative units are so called because they are measured in relation to other things. The actual (or absolute) distance they measure can change because of factors beyond their control, such as screen resolution, the width of the viewing area, the user's preference settings, and a whole host of other things. In addition, for some relative units, their size is almost always relative to the element that uses them and will thus change from element to element.

First, let's consider the character-based length units, including `em`, `ex`, and `ch`, which are closely related. Two other font-relative units, `cap` and `ic`, are discussed later in the chapter.

The em unit

In CSS, `1em` is defined to be the value of `font-size` for a given font. If the `font-size` of an element is 14 pixels, then for that element, `1em` is equal to 14 pixels.

As you may suspect, this value can change from element to element. For example, let's say you have an `<h1>` with a font size of 24 pixels, an `<h2>` element with a font size of 18 pixels, and a paragraph with a font size of 12 pixels. If you set the left margin of all three at `1em`, they will have left margins of 24 pixels, 18 pixels, and 12 pixels, respectively:

```
h1 {font-size: 24px;}
h2 {font-size: 18px;}
p {font-size: 12px;}
h1, h2, p {margin-left: 1em;}
small {font-size: 0.8em;}

<h1>Left margin = <small>24 pixels</small></h1>
<h2>Left margin = <small>18 pixels</small></h2>
<p>Left margin = <small>12 pixels</small></p>
```

When setting the size of the font, on the other hand, the value of `em` is relative to the font size of the parent element, as illustrated in Figure 5-3.



Figure 5-3. Using `em` for margins and font sizing

In theory, `1em` is equal to the width of a lowercase *m* in the font used—that's where the name comes from, in fact. It's an old typographer's term. However, this is not assured in CSS.

The ex unit

The `ex` unit refers to the height of a lowercase *x* in the font being used. Therefore, if two paragraphs use text that is 24 points in size, but each paragraph uses a different font, then the value of `ex` could be different for each paragraph. This is because different fonts have different heights for *x*, as you can see in Figure 5-4. Even though the examples use 24-point text—and therefore each example's `em` value is 24 points—the x-height for each is different.

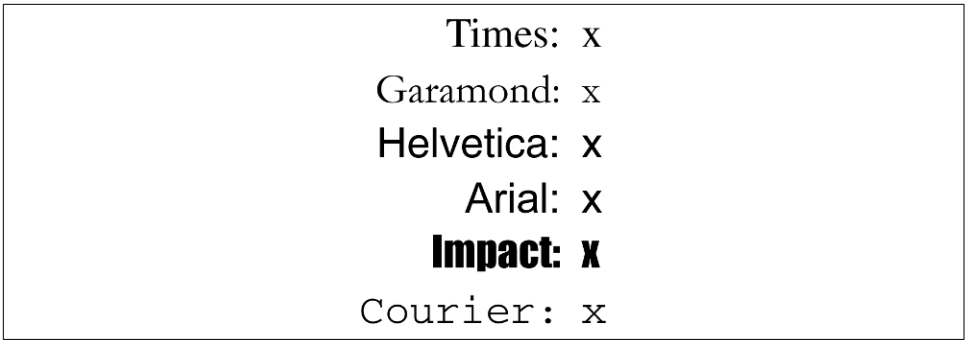


Figure 5-4. Varying x heights

The ch unit

The ch unit is broadly meant to represent *one character*. CSS Values and Units Level 4 defines ch as follows:

> Equal to the advance measure of the "0" (ZERO, U+0030) glyph found in the font used to render it.

The term *advance measure* is a CSS-ism that corresponds to the term *advance width* in Western typography. CSS uses the term *measure* because some scripts are not right to left or left to right, but instead top to bottom or bottom to top, and so may have an advance height rather than an advance width.

Without getting into too many details, a character glyph's advance width is the distance from the start of a character glyph to the start of the next. This generally corresponds to the width of the glyph itself plus any built-in spacing to the sides (although that built-in spacing can be either positive or negative).

The easiest way to demonstrate the ch unit is to run a bunch of zeros together and then set an image to have a width with the same number of ch units as the number of zeros, as shown in Figure 5-5:

```
img {height: 1em; width: 25ch;}
```

Given a monospace font like Courier, all characters are by definition 1ch wide. In any proportional face type, which is what the vast majority of Western typefaces are, characters may be wider or narrower than the 0 and so cannot be assumed to be exactly 1ch wide.

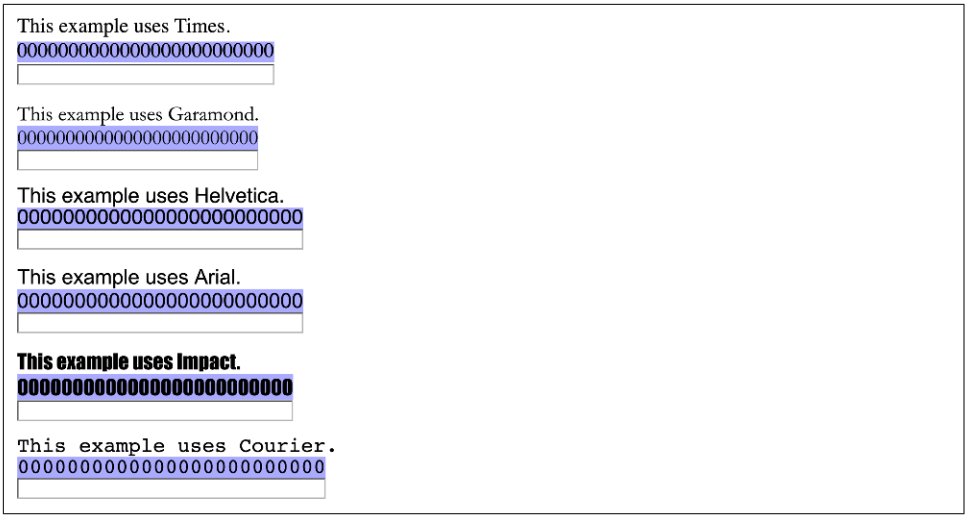


Figure 5-5. Character-relative sizing

Other relative length units

We have a few other relative length units to mention:

`ic`
: The advance measure of 水 glyph (Chinese, Japanese, and Korean water ideograph, U+6C34) found in the first font that can render it. This is like `ch` in that it uses an advance measure, but defines a measure more useful for ideographic languages than the 0 character. If `ic` can't be calculated for a given situation, it's assumed to be equal to `1em`.

`cap`
: The cap-height is approximately equal to the height of a capital Latin letter, even in fonts that do not contain Latin letters. If it can't be calculated for a given situation, it's assumed to be equal to the font's ascent height.

`lh`
: Equal to the computed value of the `line-height` property of the element on which it is used.

At the time of this writing, only Firefox supports `cap`, and only Chromium-based browsers support `lh`.

Root-Relative Length Units

Most of the character-based length units discussed in the previous section have a corresponding root-relative value. A *root-relative value* is one that is calculated with respect to the root element of the document, and thus provides a uniform value no matter what

context it's used in. We will discuss the most widely supported such unit and then summarize the rest.

The rem unit

The `rem` unit is calculated using the font size of the document's root element. In HTML, that's the `<html>` element. Thus, declaring any element to have `font-size: 1rem;` sets it to have the same font-size value as the root element of the document.

As an example, consider the following markup fragment. It will have the result shown in Figure 5-6:

```
<p> This paragraph has the same font size as the root element thanks to
    inheritance.</p>
<div style="font-size: 30px; background: silver;">
  <p style="font-size: 1em;">This paragraph has the same font size as its parent
     element.</p>
  <p style="font-size: 1rem;">This paragraph has the same font size as the root
     element.</p>
</div>
```

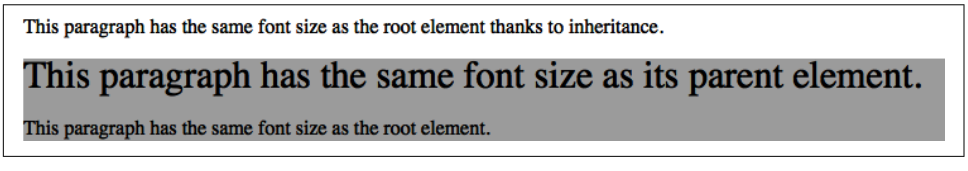


Figure 5-6. Using the `em` unit (middle sentence) versus the `rem` unit (bottom)

In effect, `rem` acts as a reset for font size: no matter what relative font sizing has happened to the ancestors of an element, giving it `font-size: 1rem;` will put it right back where the root element is set. This will usually be the user's default font size, unless you (or the user) have set the root element to a specific font size.

For example, given this declaration, `1rem` will always be equivalent to `13px`:

```
html {font-size: 13px;}
```

However, given *this* declaration, `1rem` will always be equivalent to three-quarters the user's default font size:

```
html {font-size: 75%;}
```

In this case, if the user's default is 16 pixels, `1rem` will equal `12px`. If the user has set their default to 12 pixels—and yes, some people do this—then `1rem` will equal `9px`. If the default setting is 20 pixels, `1rem` equals `15px`. And so on.

You are not restricted to the value `1rem`. Any real number can be used, just as with the `em` unit, so you can do fun things like set all of your headings to be multiples of the root element's font size:

```
h1 {font-size: 2rem;}
h2 {font-size: 1.75rem;}
```

```
h3 {font-size: 1.4rem;}
h4 {font-size: 1.1rem;}
h5 {font-size: 1rem;}
h6 {font-size: 0.8rem;}
```



`font-size: 1rem` is equivalent to `font-size: initial` as long as no font size is set for the root element.

Other root-relative units

As mentioned previously, `rem` is not the only root-relative unit defined by CSS. Table 5-1 summarizes the other root-relative units.

Table 5-1. Root-relative equivalent units

Length	Root-relative unit	Relative to
`em`	`rem`	Computed `font-size`
`ex`	`rex`	Computed x-height
`ch`	`rch`	Advance measure of the `0` character
`cap`	`rcap`	Height of a Roman capital letter
`ic`	`ric`	Advance measure of the 水 ideograph
`lh`	`rlh`	Computed `line-height`

Of all the root-relative units, only `rem` is supported as of late 2022, but it is supported by essentially all browsers.

Viewport-Relative Units

CSS provides six *viewport-relative size units*. These are calculated with respect to the size of the viewport—browser window, printable area, mobile device display, etc.:

Viewport width unit (`vw`)
: Equal to the viewport's width divided by 100. Therefore, if the viewport is 937 pixels wide, `1vw` is equal to `9.37px`. If the viewport's width changes (say, by dragging the browser window wider or narrower), the value of `vw` changes along with it.

Viewport height unit (`vh`)
: Equal to the viewport's height divided by 100. Therefore, if the viewport is 650 pixels tall, `1vh` is equal to `6.5px`. If the viewport's height changes (say, by dragging the browser window taller or shorter), the value of `vh` changes along with it.

Viewport block unit (`vb`)
: Equal to the size of the viewport along the block axis, divided by 100. The block axis is explained in Chapter 6. In top-to-bottom languages like English or Arabic, `vb` will be equal to `vh` by default.

Viewport inline unit (`vi`)
: Equal to the size of the viewport along the inline axis, divided by 100. The inline axis is explained in Chapter 6. In horizontally written languages like English or Arabic, `vi` will be equal to `vw` by default.

Viewport minimum unit (`vmin`)
: Equal to 1/100th of the viewport's width or height, whichever is *less*. Thus, given a viewport that is 937 pixels wide by 650 pixels tall, `1vmin` is equal to `6.5px`.

Viewport maximum unit (`vmax`)
: Equal to 1/100th of the viewport's width or height, whichever is *greater*. Thus, given a viewport that is 937 pixels wide by 650 pixels tall, `1vmax` is equal to `9.37px`.

Because these are length units like any other, they can be used anywhere a length unit is permitted. You can scale the font size of a heading in terms of the viewport height, for example, with something like `h1 {font-size: 10vh;}`. This sets the font size to be 1/10th the height of the viewport—a technique potentially useful for article titles and the like.

These units can be particularly handy for creating full-viewport interfaces, such as those we expect to find on a mobile device, because the units allow elements to be sized compared to the viewport and not to any of the elements within the document tree. It's thus very simple to fill up the entire viewport, or at least major portions of it, and not have to worry about the precise dimensions of the actual viewport in any particular case.

A basic example of viewport-relative sizing is illustrated in Figure 5-7:

```
div {width: 50vh; height: 33vw; background: gray;}
```

An interesting (though perhaps not useful) fact about these units is that they aren't bound to their own primary axis. Thus, for example, you can declare `width: 25vh;` to make an element as wide as one-quarter the height of the viewport.

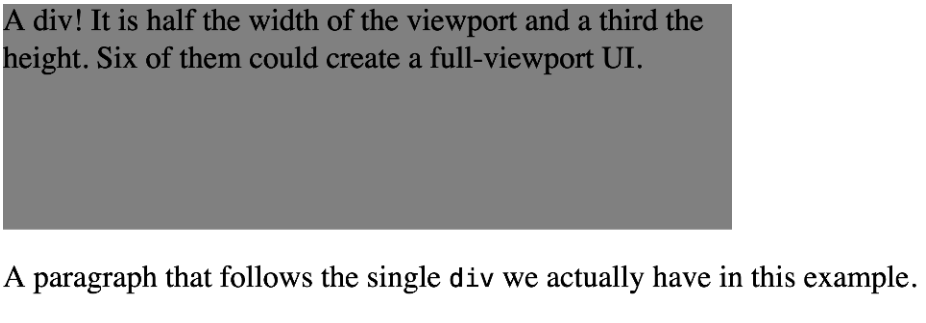


Figure 5-7. Viewport-relative sizing

Variants of these units accommodate the vagaries of viewports and how they can be sized, particularly on devices where the UI may expand and contract based on user input. These variants are based on four viewport types:

Default
: The default viewport size, as defined by the user agent (browser). This viewport type is expected to correspond to the units `vw`, `vh`, `vb`, `vi`, `vmin`, and `vmax`. The default viewport may correspond to one of the other viewport types; e.g., the default viewport could be the same as the large viewport, but that's up to each browser to decide.

Large
: The largest possible viewport after any user-agent interfaces are contracted to their fullest extent. For example, on a mobile device, the browser *chrome* (the browser's address bar, navigation bar, and so on) may be minimized or hidden most of the time so that the maximum screen area can be used to show page content. This is the state described by the large viewport. If you want an element's size to be determined by the full viewport area, even if that will lead to it being overlapped by the UI, the large-viewport units are the way to go. The units corresponding to this viewport type are `lvw`, `lvh`, `lvb`, `lvi`, `lvmin`, and `lvmax`.

Small
: The smallest possible viewport after any user-agent interfaces are expanded to their fullest extent. In this state, the browser's chrome takes up as much screen space as it possibly can, leaving a minimum space for the page content. If you want to be sure an element's sizing will take into account any possible interface actions, use these units. The units corresponding to this viewport type are `svw`, `svh`, `svb`, `svi`, `svmin`, and `svmax`.

Dynamic
: The area in which content is visible, which can change as the UI expands or contracts. As an example, consider how the browser interface can appear or disappear on mobile devices, depending on how the content is scrolled or where on the screen the user taps. If you want to set lengths based on the size of the viewport at every moment, regardless of how it changes, these are the units for you. The units corresponding to this viewport type are `dvw`, `dvh`, `dvb`, `dvi`, `dvmin`, and `dvmax`.

As of late 2022, scrollbars (if any) are ignored for the purposes of calculating all of the previous units. Thus, the calculated size of `svw` or `dvw` will *not* change if scrollbars appear or disappear, or at least shouldn't.

Function Values

One of the more recent developments in CSS is an increase in the number of values that are effectively functions. These values can range from doing math calculations to clamping value ranges to pulling values out of HTML attributes. CSS has, in fact, a *lot* of these, listed here:

- abs()
- acos()
- annotation()
- asin()
- atan()
- atan2()
- attr()
- blur()
- brightness()
- calc()
- character-variant()
- circle()
- clamp()
- color-contrast()
- color-mix()
- color()
- conic-gradient()
- contrast()
- cos()
- counter()
- counters()
- cross-fade()
- device-cmyk()
- drop-shadow()
- element()
- ellipse()
- env()
- exp()
- fit-content()
- grayscale()
- hsl()
- hsla()
- hue-rotate()
- hwb()
- hypot()
- image-set()
- image()
- inset()
- invert()
- lab()
- lch()
- linear-gradient()
- log()
- matrix()
- matrix3d()
- max()
- min()
- minmax()
- mod()
- oklab()
- oklch()
- opacity()
- ornaments()
- paint()
- path()
- perspective()
- polygon()
- pow()
- radial-gradient()
- rem()
- repeat()
- repeat-conic-gradiant()
- repeating-linear-gradiant()
- repeating-radial-gradient()
- rgb()
- rgba()
- rotate()
- rotate3d()
- rotateX()
- rotateY()
- rotateZ()
- round()
- saturate()
- scale()
- scale3d()
- scaleX()
- scaleY()
- scaleZ()
- sepia()
- sign()
- sin()
- skew()
- skewX()
- skewY()
- sqrt()
- styleset()
- stylistic()
- swash()
- symbols()
- tan()
- translate()
- translate3d()
- translateX()

- `translateY()`
- `translateZ()`
- `url()`
- `var()`

That's 97 different function values. We'll cover some in the rest of this chapter. The rest are covered in other chapters, as appropriate for their topics (e.g., the filter functions are described in Chapter 20).

Calculation Values

When you need to do a little math, CSS provides a `calc()` value. Inside the parentheses, you can construct simple mathematical expressions. The permitted operators are `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), as well as parentheses. These follow the traditional precedence order of parentheses, exponents, multiplication, division, addition, and subtraction (PEMDAS), although in this case it's really just PMDAS since exponents are not permitted in `calc()`.

As an example, suppose you want your paragraphs to have a width that's 2 em less than 90% the width of their parent element. Here's how you express that with `calc()`:

```
p {width: calc(90% - 2em);}
```

The `calc()` value can be used with any property that permits one of the following value types: <*length*>, <*frequency*>, <*angle*>, <*time*>, <*percentage*>, <*number*>, or <*integer*>. You can also use all these unit types within a `calc()` value, though CSS has some limitations to keep in mind.

The basic limitation is that `calc()` does basic type checking to make sure that units are, in effect, compatible. The checking works like this:

1. To either side of a `+` or `–` sign, both values must have the same unit type, or be a <*number*> and <*integer*> (in which case, the result is a <*number*>). Thus, `5 + 2.7` is valid, and results in `7.7`. On the other hand, `5em + 2.7` is invalid, because one side has a length unit and the other does not. Note that `5em + 20px` *is* valid, because `em` and `px` are both length units.
2. Given a `*`, one of the values involved must be a <*number*> (which, remember, includes integer values). So `2.5rem * 2` and `2 * 2.5rem` are both valid, and each results in `5rem`. On the flip side, `2.5rem * 2rem` is *not* valid, because the result would be `5rem`^{2}, and length units cannot be area units.
3. Given a `/`, the value on the *right* side must be a <*number*>. If the left side is an <*integer*>, the result is a <*number*>. Otherwise, the result is of the unit type used on the left side. This means that `30em / 2.75` is valid, but `30 / 2.75em` is not.
4. Furthermore, any circumstance that yields division by zero is invalid. This is easiest to see in a case like `30px/0`, but there are other ways to get there.

One more notable limitation is that whitespace is *required* on both sides of the `+` and `-` operators, while it is not for `*` and `/`. This was done to allow future development of `calc()` values to support keywords that contain hyphens (e.g., `max-content`).

Furthermore, it's valid (and supported) to nest `calc()` functions inside each other. Thus you can say something like this:

```
p {width: calc(90% - calc(1em + 0.1vh));}
```

Beyond that, the CSS specification requires that user agents support a *minimum* of 20 terms inside any single `calc()` function, where a term is a number, percentage, or dimension (e.g., a length). If the number of terms somehow exceeds the user agent's term limits, the entire function is treated as invalid.

Maximum Values

Calculation is nice, but sometimes you just want to make sure a property is set to one of a number of values, whichever is smallest. In those cases, the `min()` function value comes in very handy. Yes, this is confusing at first, but give us a minute and hopefully it will make sense.

Suppose you want to make sure that an element is never wider than a certain amount; say, an image should be one-quarter the width of the viewport or 200 pixels wide, whichever is *smaller*. This allows it to be constrained to 200 pixels of width on wide viewports, but take up to a quarter the width of smaller viewports. For that, you'd say the following:

```
.figure {width: min(25vw, 200px);}
```

The browser will compute the width of `25vw` and compare that to `200px`, and use whichever is smaller. If `200px` is smaller than 25% the width of the viewport, then `200px` will be used. Otherwise, the element will be 25% as wide as the viewport, which could easily be smaller than `1em`. Note that *smaller* in this case means closest to negative infinity, not closest to zero. Thus, if you compare two terms that compute to (say) `-1500px` and `-2px`, `min()` will pick `-1500px`.

You can nest `min()` inside `min()`, or throw a mathematical expression in there for one of the values, without having to wrap it in `calc()`. For that matter, you can put in `max()` and `clamp()`, which we haven't even discussed yet. You can supply as many terms as you like: if you want to compare four ways of measuring something and then pick the minimum, just separate them with commas. Here's a slightly contrived example:

```
.figure {width: min(25vw, 200px, 33%, 50rem - 30px);}
```

Whichever of those values is computed to be the minimum (closest to negative infinity) will be used, thus defining a maximum for the `width` value. The order you list them in doesn't matter, since the minimum value will always be picked regardless of where it appears in the function.

In general, `min()` can be used in any property value that permits <*`length`*>, <*`frequency`*>, <*`angle`*>, <*`time`*>, <*`percentage`*>, <*`number`*>, or <*`integer`*>.



Remember that setting a maximum value on font sizes is an accessibility concern. You should *never* set a maximum font size using pixels, because that would likely prevent text zooming by users. You probably shouldn't use `min()` for font sizing in any case, but if you do, keep `px` lengths out of the values!

Minimum Values

The mirror image of `min()` is `max()`, which can be used to set a minimum value for a property. It can appear in the same places and can be nested in the same ways as `min()`, and is generally just the same except that it picks the largest (closest to positive infinity) value from among the alternatives given.

As an example, perhaps the top of a page's design should be a minimum of 100 pixels tall, but it can be taller if conditions permit. In that case, you could use something like this:

```
header {height: max(100px, 15vh, 5rem);}
```

Whichever of the values is largest will be used. For a desktop browser window, that would probably be `15vh`, unless the base size text is really enormous. For a handheld display, it's more likely that `5rem` or `100px` will be the largest value. In effect, this sets a minimum size of 100 pixels tall, since getting either `15vh` or `5rem` below that value is easily possible.

Remember that setting even a minimum value on font sizes can create an accessibility problem, since a too-small minimum is still too small. A good way to handle this is to always include `1rem` in your `max()` expressions for font sizes. Use something like this:

```
.sosumi {font-size: max(1vh, 0.75em, 1rem);}
```

Alternatively, you could not use `max()` for font sizing at all. It's probably best left to box sizing and other such uses.

Clamping Values

If you've already been thinking about ways to nest `min()` and `max()` to set upper and lower bounds on a value, there's a way to not only do that, but set an "ideal" value as well: `clamp()`. This function value takes three parameters representing, in order, the minimum allowed value, preferred value, and maximum allowed value.

For example, consider some text you want to be about 5% the height of the viewport, while keeping its minimum the base font size and its maximum three times the text around it. That would be expressed like so:

```
footer {font-size: clamp(1rem, 2vh, 3em);}
```

Given those styles and assuming the base font size is 16 pixels, as it is by default in most browsers, then the footer text will be equal to the base font size up to a viewport height of 800 pixels (16 divided by 0.02). If the viewport gets taller, the text will start to get bigger, unless doing so would make it bigger than `3em`. If the text ever gets to the same size as `3em`, it will stop growing. (This is fairly unlikely, but one never knows.)

If the maximum value of a `clamp()` is ever computed to be smaller than the minimum value, the maximum is ignored and the minimum value is used instead.

You can use `clamp()` anywhere you can use `min()` and `max()`, including nesting them inside each other. For example:

```
footer {font-size: clamp(1rem, max(2vh, 1.5em), 3em);}
```

This is basically the same as the previous example, except in this case the preferred value is either 2% the height of the viewport or 1.5 times the size of the parent element's text, whichever is larger.

Attribute Values

In a few CSS properties, it's possible to pull in the value of an HTML attribute defined for the element being styled. You do this with the `attr()` function.

For example, with generated content, you can insert the value of any attribute. It looks something like this (don't worry about understanding the exact syntax, which we'll explore in Chapter 16):

```
p::before {content: "[" attr(id) "]";}
```

That expression would prefix any paragraph that has an `id` attribute with the value of that `id`, enclosed in square brackets. Therefore, applying the previous style to the following paragraphs would have the result shown in Figure 5-8:

```
<p id="leadoff">This is the first paragraph.</p>
<p>This is the second paragraph.</p>
<p id="conclusion">This is the third paragraph.</p>
```

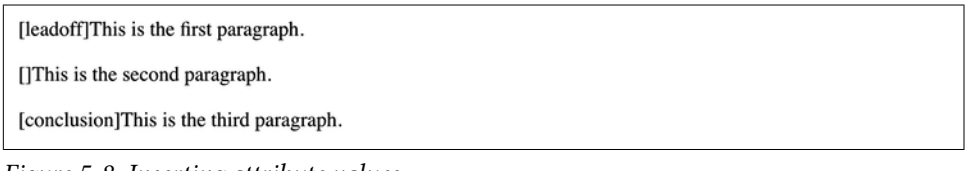


Figure 5-8. Inserting attribute values

While `attr()` is supported in the `content` property value, it isn't parsed. In other words, if the `attr()` returns an image URL from an attribute value, the generated content will be the URL written out as text, and not the image that lives at that URL. This is true as of late 2022, anyway; there are plans for changes such that `attr()` can be parsed (and also be used for all properties, not just `content`).

Color

One of the first questions every starting web author asks is, "How do I set colors on my page?" Under HTML, you have two choices: you could use one of a large but limited number of colors with names, such as `red` or `purple`, or employ a vaguely cryptic method

using hexadecimal codes. Both methods for describing colors remain in CSS, along with several—and, we think, more intuitive—methods.

Named Colors

Over the years, CSS has added a set of 148 colors that are identified by human-readable names like `red` or `firebrickred`. CSS calls these, logically enough, *named colors*. In the early days, CSS used only the 16 basic color keywords defined in HTML 4.01:

- `aqua`
- `gray`
- `navy`
- `silver`
- `black`
- `green`
- `olive`
- `teal`
- `blue`
- `lime`
- `purple`
- `white`
- `fuchsia`
- `maroon`
- `red`
- `yellow`

So, let's say you want all first-level headings to be maroon. The best declaration would be as follows:

```
h1 {color: maroon;}
```

Simple enough, isn't it? Figure 5-9 shows a few more examples:

```
h1 {color: silver;}
h2 {color: gray;}
h3 {color: black;}
```



Figure 5-9. Named colors

You've probably seen (and maybe even used) color names other than the ones listed earlier. For example, you could say

```
h1 {color: lightgreen;}
```

and get a light-green (but not exactly lime) color applied to `<h1>` elements.

The CSS color specification includes those original 16 named colors in a longer list of 148 color keywords. This extended list is based on the standard X11 RGB values that have been in use for decades and recognized by browsers for many years, with the addition of

some color names from SVG (mostly involving variants of "gray" and "grey") and a memorial color.

Color Keywords

CSS has two special keywords that can be used anywhere a color value is permitted: `transparent` and `currentcolor`.

As its name suggests, `transparent` defines a completely transparent color. The CSS Color Module defines it to be equivalent to `rgb(0 0 0 / 0%)`, and that's its computed value. This keyword is not often used to set text color, for example, but it is the default value for element background colors. It can also be used to define element borders that take up space but are not visible, and is often used when defining gradients—all topics we'll cover in later chapters.

By contrast, `currentcolor` means "whatever the computed value of `color` is for this element." Consider the following:

```
main {color: gray; border-color: currentcolor;}
```

The first declaration causes any `<main>` elements to have a foreground color of `gray`. The second declaration uses `currentcolor` to copy the computed value of `color`—in this case `gray`—and apply it to any borders the `<main>` elements might have. Incidentally, `currentcolor` is actually the default value for `border-color`, which we'll cover in Chapter 7.

As with all the named colors, these color names are case-insensitive. We show `currentcolor` with mixed capitalization because it is generally written that way for legibility.

Fortunately, CSS has more detailed and precise ways to specify colors. The advantage is that, with these methods, you can specify any color in the color spectrum, not just a limited list of named colors.

Colors by RGB and RGBa

Computers create colors by combining different levels of the primary colors red, green, and blue—a combination often referred to as *RGB color*. So, it makes sense that you should be able to specify your own combinations of these primary colors in CSS. That solution is a bit complex, but possible, and the payoffs are worth it because CSS has very few limits on which colors you can produce. You can produce color in this manner in four ways, detailed in this section.

Functional RGB colors

Two color value types use *functional RGB notation* as opposed to hexadecimal notation. The generic syntax for this type of color value is `rgb(`*`color`*`)`, where *`color`* is expressed using a triplet of either percentages or numbers. The percentage values can be in the range `0%`–`100%`, and the integers can be in the range `0`–`255`.

Thus, to specify white and black, respectively, using percentage notation, the values would be as follows:

```
rgb(100%,100%,100%)
rgb(0%,0%,0%)
```

Using the integer-triplet notation, the same colors would be represented as follows:

```
rgb(255,255,255)
rgb(0,0,0)
```

An important thing to remember is that you can't mix integers and percentages in the same color value. Thus, `rgb(255,66.67%,50%)` would be invalid and thus ignored.



In more recent browsers, the separating commas in RGB values can be replaced with simple whitespace. Thus, black can be represented as `rgb(0 0 0)` or `rgb(0% 0% 0%)`. This is true of all the color values that allow commas that we'll see throughout the chapter. Bear in mind that some of the newer color functions do not allow commas.

Assume you want your `<h1>` elements to be a shade of red that lies between the values for red and maroon. The `red` value is equivalent to `rgb(100%,0%,0%)`, whereas `maroon` is equal to `(50%,0%,0%)`. To get a color between those two, you might try this:

```
h1 {color: rgb(75%,0%,0%);}
```

This makes the red component of the color lighter than `maroon`, but darker than `red`. If, on the other hand, you want to create a pale-red color, you would raise the green and blue levels:

```
h1 {color: rgb(75%,50%,50%);}
```

The closest equivalent color using integer-triplet notation is shown here:

```
h1 {color: rgb(191,127,127);}
```

The easiest way to visualize how these values correspond to color is to create a table of gray values. The result is shown in Figure 5-10:

```
p.one {color: rgb(0%,0%,0%);}
p.two {color: rgb(20%,20%,20%);}
p.three {color: rgb(40%,40%,40%);}
p.four {color: rgb(60%,60%,60%);}
p.five {color: rgb(80%,80%,80%);}
p.six {color: rgb(0,0,0);}
p.seven {color: rgb(51,51,51);}
p.eight {color: rgb(102,102,102);}
p.nine {color: rgb(153,153,153);}
p.ten {color: rgb(204,204,204);}
```



Figure 5-10. Text set in shades of gray

Since we're dealing in shades of gray, all three RGB values are the same in each statement. If any one were different from the others, a color hue would start to emerge. If, for example, `rgb(50%,50%,50%)` were modified to be `rgb(50%,50%,60%)`, the result would be a medium-dark color with just a hint of blue.

You can use fractional numbers in percentage notation. You might, for some reason, want to specify that a color be exactly 25.5% red, 40% green, and 98.6% blue:

```
h2 {color: rgb(25.5%,40%,98.6%);}
```

Values that fall outside the allowed range for each notation are *clipped* to the nearest range edge, meaning that a value that is greater than `100%` or less than `0%` will default to those allowed extremes. Thus, the following declarations would be treated as if they were the values indicated in the comments:

```
P.one {color: rgb(300%,4200%,110%);}   /*  100%,100%,100%  */
P.two {color: rgb(0%,-40%,-5000%);}   /*  0%,0%,0%  */
p.three {color: rgb(42,444,-13);}    /* 42,255,0  */
```

Conversion between percentages and integers may seem arbitrary, but there's no need to guess at the integer you want—there's a simple formula for calculating them. If you know the percentages for each of the RGB levels you want, you need only apply them to the number 255 to get the resulting values. Let's say you have a color of 25% red, 37.5% green, and 60% blue. Multiply each of these percentages by 255, and you get 63.75, 95.625, and 153. Round these values to the nearest integers, and voilà: `rgb(64,96,153)`.

If you already know the percentage values, there isn't much point in converting them into integers. Integer notation is more useful for people who use programs such as Adobe Photoshop, which can display integer values in the Info dialog, or for those who are so familiar with the technical details of color generation that they normally think in values of 0–255.

RGBa colors

RGB notations can include a fourth parameter defining the alphatransparency value. By adding an alpha value at the end of the RGB triplet, `rgb()` accepts a red-green-blue-alpha, or RGBa, value, with the alpha value being a measurent of opacity.

While the `rgb()` notation allows for three or four values, the alpha value must be present in the legacy `rgba()` function to be valid.

For example, suppose you want an element's text to be half-opaque white. That way, any background color behind the text would "shine through," mixing with the half-transparent white. You could write one of the following two values:

```
rgb(255 255 255 / 0.5)
rgba(100% 100% 100% / 0.5)  /* commas would also be allowed */
```

To make a color completely transparent, you set the alpha value to `0`; to be completely opaque, the correct value is `1`. Thus `rgb(0,0,0)` and `rgba(0,0,0,1)` will yield precisely the same result (black). Figure 5-11 shows a series of paragraphs set in increasingly transparent black, which is the result of the following rules:

```
p.one {color: rgb(0,0,0,1);}
p.two {color: rgba(0%,0%,0%,0.8);}
p.three {color: rgb(0 0 0 / 0.6);}
p.four {color: rgba(0% 0% 0% / 0.4);}
p.five {color: rgb(0,0,0,0.2);}
```

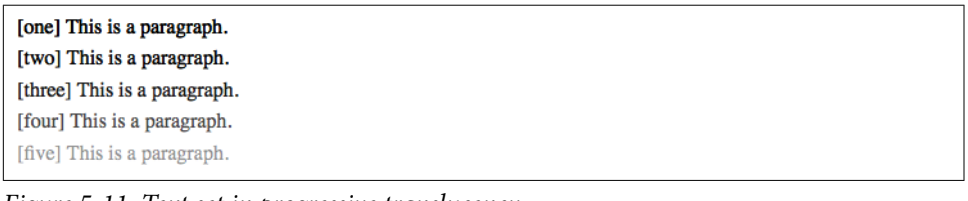


Figure 5-11. Text set in progressive translucency

Alpha values are always real numbers in the range `0` to `1`, or percentages in the range `0%` to `100%`. Any value outside that range will either be ignored or reset to the nearest valid alpha value.

Hexadecimal RGB colors

CSS allows you to define a color using the same *hexadecimal color notation* so familiar to old-school HTML web authors:

```
h1 {color: #FF0000;}   /* set H1s to red */
h2 {color: #903BC0;}   /* set H2s to a dusky purple */
h3 {color: #000000;}   /* set H3s to black */
h4 {color: #808080;}   /* set H4s to medium gray */
```

Computers have been using hex notation for quite some time now, and programmers are typically either trained in its use or pick it up through experience. Their familiarity with

hexadecimal notation likely led to its use in setting colors in HTML. That practice was carried over to CSS.

Here's how it works: by stringing together three hexadecimal numbers in the range `00` through `FF`, you can set a color. The generic syntax for this notation is `#RRGGBB`. Note that there are no spaces, commas, or other separators between the three numbers.

Hexadecimal notation is mathematically equivalent to integer-pair notation. For example, `rgb(255,255,255)` is precisely equivalent to `#FFFFFF`, and `rgb(51,102,128)` is the same as `#336680`. Feel free to use whichever notation you prefer—it will be rendered identically by most user agents. If you have a calculator that converts between decimal and hexadecimal, making the jump from one to the other should be pretty simple.

For hexadecimal numbers that are composed of three matched pairs of digits, CSS permits a shortened notation. The generic syntax of this notation is `#RGB`:

```
h1 {color: #000;}   /* set H1s to black */
h2 {color: #666;}   /* set H2s to dark gray */
h3 {color: #FFF;}   /* set H3s to white */
```

As you can see from the markup, each color value has only three digits. However, since hexadecimal numbers between `00` and `FF` need two digits each, and you have only three total digits, how does this method work?

The answer is that the browser takes each digit and replicates it. Therefore, `#F00` is equivalent to `#FF0000`, `#6FA` would be the same as `#66FFAA`, and `#FFF` would come out `#FFFFFF`, which is the same as `white`. Not every color can be represented in this manner. Medium gray, for example, would be written in standard hexadecimal notation as `#808080`. This cannot be expressed in shorthand; the closest equivalent would be `#888`, which is the same as `#888888`.

Hexadecimal RGBa colors

Hexadecimal notation can have a fourth hex value to represent the alpha channel value. The following rules style the series of paragraphs in Figure 5-12, which are set in increasingly transparent black, just as you saw in the previous section:

```
p.one {color: #000000FF;}
p.two {color: #000000CC;}
p.three {color: #00000099;}
p.four {color: #00000066;}
p.five {color: #00000033;}
```

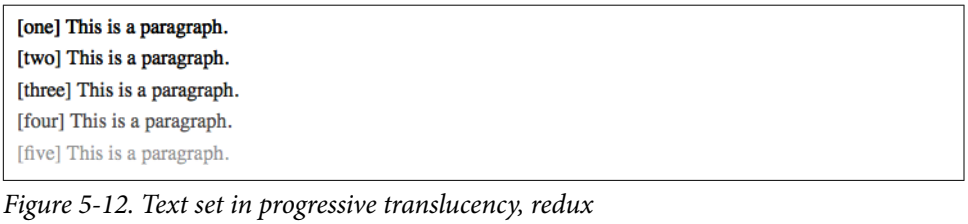


Figure 5-12. Text set in progressive translucency, redux

As with non-alpha hexadecimal values, you can shorten a value composed of matched pairs to a four-digit value. Thus, a value of `#663399AA` can be written as `#639A`. If the value has any pairs that are not repetitive, the entire eight-digit value must be written out: `#663399CA` cannot be shortened to `#639CA`.

HSL and HSLa Colors

Hue, saturation, and lightness (HSL) color notation is similar to hue, saturation, and brightness (HSB), the color system in image editing software like Photoshop, and just as intuitive. The hue is expressed as an angle value, saturation is a percentage value from 0% (no saturation) to 100% (full saturation), and lightness is a percentage value from 0% (completely dark) to 100% (completely light). If you're intimately familiar with RGB, HSL may be confusing at first. (But then, RGB is confusing for people familiar with HSL.)

The *hue* angle is expressed in terms of a circle around which the full spectrum of colors progresses. It starts with red at 0 degrees and then proceeds through the rainbow until it comes to red again at 360 degrees. When the hue value is a unitless number, it is interpreted as degrees.

Saturation measures the intensity of a color. A saturation of `0%` always yields a shade of gray, no matter what hue angle you have set, and a saturation of `100%` creates the most vivid possible shade of that hue (in the HSL color space) for a given lightness.

Similarly, *lightness* defines how dark or light the color appears. A lightness of `0%` is always black, regardless of the other hue and saturation values, just as a lightness of `100%` always yields white. Consider the results of the following styles, illustrated on the left side of Figure 5-13.

```
p.one {color: hsl(0,0%,0%);}
p.two{color: hsl(60 0% 25%);}
p.three {color: hsl(120deg,0%,50%);}
p.four {color: hsl(180deg 0% 75%);}
p.five {color: hsl(0.667turn,0%,0%);}
p.six {color: hsl(0.833turn 0% 25%);}
p.seven {color: hsl(400grad 0% 50%);}
```



Remember that in more recent browsers, the commas in `hsl()` values can be replaced with whitespace.

The gray you see on the left side isn't just a function of the limitations of print: every one of those paragraphs is a shade of gray, because every color value has `0%` in the saturation (middle) position. The degree of lightness or darkness is set by the lightness (third) position. In all seven examples, the hue angle changes, and in none of them does it matter.

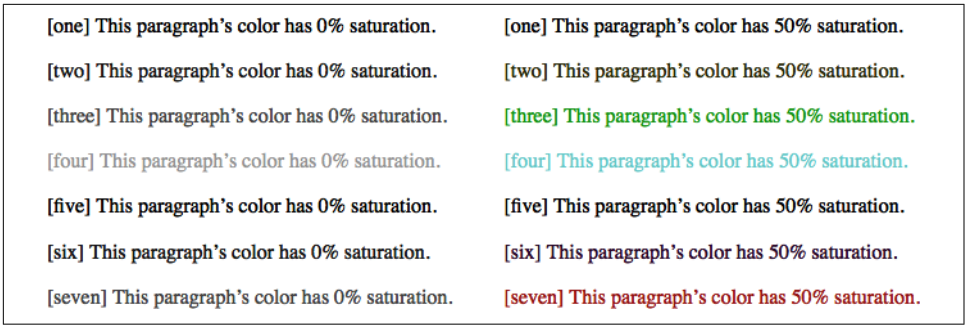


Figure 5-13. Varying lightness and hues

But that's only so long as the saturation remains at `0%`. If that value is raised to, say, `50%`, then the hue angle will become very important, because it will control what sort of color you see. Consider the same set of values that we saw before, but all set to `50%` saturation; this is illustrated on the right side of Figure 5-13, although the color is not visible in the print version of this book.

Just as RGB has a legacy RGBa counterpart, HSL has an HSLa counterpart. This is an HSL triplet followed by an alpha value in the range 0–1. The following HSLa values are all black with varying shades of transparency, just as in "Hexadecimal RGBa colors" on page 157 (and illustrated in Figure 5-12):

```
p.one {color: hsl(0,0%,0%,1);}
p.two {color: hsla(0,0%,0%,0.8);}
p.three {color: hsl(0 0% 0% / 0.6);}
p.four {color: hsla(0 0% 0% / 0.4);}
p.five {color: hsl(0rad 0% 0% / 0.2);}
```

Colors with HWB

Colors can also be represented in terms of their *hue*, *white* level, and *black* level by using the `hwb()` functional value. This function value accepts hue values expressed as an angle value. After the hue angle, instead of lightness and saturation, whiteness and blackness values are specified as percentages.

Unlike HSL, however, there is no legacy `hwba()` function. Instead, the value syntax for `hwb()` allows an opacity to be defined after the HWB values, separated from them by a forward slash (`/`). The opacity can be expressed either as a percentage or as a real value from 0 to 1, inclusive. Also unlike HSL, commas are not supported: the HWB values can only be separated by whitespace.

Here are some examples of using HWB notation:

```
/* Varying shades of red */
hwb(0 40% 20%)
hwb(360 50% 10%)
hwb(0deg 10% 10%)
```

```
hwb(0rad 60% 0%)
hwb(0turn 0% 40%)

/* Partially translucent red */
hwb(0 10% 10% / 0.4)
hwb(0 10% 10% / 40%)
```

Lab Colors

Historically, all CSS colors were defined in the sRGB color space, which encompassed more colors than older display monitors could represent. Modern displays, on the other hand, can handle about 150% of the sRGB color space, which still isn't the full range of color humans can perceive, but it's a lot closer.

In 1931, the *Commission Internationale de l'Éclairage* (International Commission on Illumination, or CIE) defined a scientific system for defining colors created via light, as opposed to those created with paint or dyes. Now, almost a century later, CSS has brought the work of the CIE into its repertoire.

It does this using the `lab()` function value to express color in the CIE L*a*b* (hereafter shortened as *Lab*) color space. Lab is designed to represent the entire range of color that humans can see. The `lab()` function accepts three to four parameters: `lab(`*`L a b / A`*`)`. Similar to HWB, the parameters must be space-separated (no commas allowed) and a forward slash (`/`) precedes the alpha value, if provided.

The *`L`* (Lightness) component specifies the CIE lightness, and is a <*`percentage`*> from `0%` representing black to `100%` representing white, or a <*`number`*> from `0` to `1`. The second component, *a*, is the distance along the a-axis in the Lab color space. This axis runs from a purplish red in the positive direction to a shade of green in the negative direction. The third component, *b*, is the distance along the b-axis in the Lab color space. This axis runs from a yellow in the positive direction to a blue-violet in the negative direction.

The fourth, optional parameter is the opacity, with a value from 0 to 1 inclusive, or 0% to 100% inclusive. If omitted, the opacity defaults to 1 (100%), or full opacity.

Here are some examples of Lab color expressed in CSS:

```
lab(29.2345% 39.3825 -20.0664);
lab(52.2345% 40.1645 59.9971);
lab(52.2345% 40.1645 59.9971 / .5);
```

The main reason to bring Lab (and LCH, which we'll discuss in a moment) colors into CSS is that they are systematically designed to be *perceptually uniform*: color values that share a given coordinate will seem consistent in terms of that coordinate. Two colors with different hues but the same lightness will appear to have similar lightnesses. Two colors with the same hue but different lightnesses will appear to be shades of a single hue. This is often not the case with RGB and HSL values, so Lab and LCH represent a big improvement.

They're also defined to be device independent, so you should be able to specify colors in these color spaces and get a visually consistent result from one device to another.



As of late 2022, only WebKit supports `lab()`.

LCH Colors

Lightness Chroma Hue (LCH) is a version of Lab designed to represent the entire spectrum of human vision. It does this using a different notation: `lch(`*`L C H / A`*`)`. The main difference is that *`C`* and *`H`* are polar coordinates, rather than linear values along color axes.

The *`L`* (Lightness) component is the same as the CIE Lightness, and is a *<percentage>* from `0%` representing black to `100%` representing white.

The *`C`* (Chroma amount) component roughly represents the amount of color. Its minimum value is `0`, and no maximum is defined. Negative `C` values are clamped to 0.

The *`H`* (Hue angle) component is essentially a combination of the *`a`* and *`b`* values in `lab()`. The value `0` points along the positive a-axis (toward purplish red), `90` points along the positive b-axis (toward mustard yellow), `180` points along the negative a-axis (toward greenish cyan), and `270` points along the negative b-axis (toward sky blue). This component loosely corresponds to HSL's Hue, but the hue angles differ.

The optional *`A`* (alpha) component can be a *<number>* from 0 to 1, or else a *<percentage>*, where the number 1 corresponds to 100% (full opacity). If present, it is preceded by a forward slash (`/`). Here are some examples:

```
lch(56% 132 331)
lch(52% 132 8)
lch(52% 132 8 / 50%)
```

To give an example of the capabilities of LCH, `lch(52% 132 8)` is a very bright magenta equivalent to `rgb(118.23% -46.78% 40.48%)`. Notice the large red value and the negative green value, which places the color outside the sRGB color space. If you supplied that RGB value to a browser, it would clamp the value to `rgb(100% 0% 40.48%)`. This is within the sRGB color space, but it is visually quite distinct from the color that is defined by `lch(52% 132 8)`.



As of late 2022, only Safari supports `lch()` values.

Oklab and Oklch

Improved versions of Lab and LCH, called *Oklab* and *Oklch*, will be supported by CSS via the `oklab()` and `oklch()` functional values. Oklab was developed by taking a large set of visually similar colors and performing a numerical optimization on them, yielding a color space with better hue linearity and uniformity, and better chroma uniformity, than the CIE color spaces. Oklch is a polar-coordinate version of Oklab, just as LCH is to Lab.

Because of this improved uniformity, Oklab and Oklch will be the default for color-interpolation calculations in CSS going forward. However, as of late 2022, only Safari supports the `oklab()` and `oklch()` CSS functional values.

Using color()

The `color()` function value allows a color to be specified in a named color space rather than the implicit sRGB color space. It accepts four space-separated parameters, as well as an optional fifth opacity value preceded by a forward slash (/).

The first parameter is a predefined, named color space. Possible values as of late 2022 include `srgb`, `srgb-linear`, `display-p3`, `a98-rgb`, `prophoto-rgb`, `rec2020`, `xyz`, `xyz-d50`, and `xyz-d65`. The three values that follow are specific to the color space declared in the first parameter. Some color spaces may allow these values to be percentages, while others may not.

As an example, the following values should yield the same color:

```
#7654CD
rgb(46.27% 32.94% 80.39%)
lab(44.36% 36.05 -58.99)
color(xyz-d50 0.2005 0.14089 0.4472)
color(xyz-d65 0.21661 0.14602 0.59452)
```

You could easily declare a color that lies outside the gamut of a given color space. For example, `color(display-p3 -0.6112 1.0079 -0.2192);` is outside the `display-p3` gamut. It's still a valid color, just not one that can be expressed in that color space. When a color value is valid but outside the gamut, it will be mapped to the closest color that lies inside the color space's gamut.

If a color's value is straight-up invalid, opaque black is used.



As of late 2022, only Safari supports `color()`.

Applying Color

Since we've just gone through all the possible color formats, let's take a brief detour to talk about the property that uses color values the most often: `color`. This property sets the color of an element's text and the value of `currentcolor`.

color

Values	*<color>*
Initial value	User-agent specific
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

This property accepts as a value any valid color type, such as `#FFCC00` or `rgb(100% 80% 0% / 0.5)`.

For nonreplaced elements like paragraphs or `<em>` elements, `color` sets the color of the text in the element. The following code results in Figure 5-14:

```
<p style="color: gray;">This paragraph has a gray foreground.</p>
<p>This paragraph has the default foreground.</p>
```

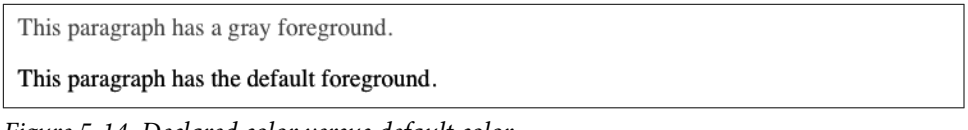


Figure 5-14. Declared color versus default color

In this example, the default foreground color is black. That doesn't have to be the case, since the user might have set their browser (or other user agent) to use a different foreground (text) color. If the browser's default text color was set to `green`, the second paragraph in the preceding example would be green, not black—but the first paragraph would still be gray.

You need not restrict yourself to such basic operations. There are plenty of ways to use `color`. You might have some paragraphs that contain text warning the user of a potential problem. To make this text stand out more than usual, you might decide to color it red. Just apply a class of `warn` to each paragraph that contains warning text (`<p class="warn">`) and the following rule:

```
p.warn {color: red;}
```

In the same document, you might decide that any unvisited hyperlinks within a warning paragraph should be green:

```
p.warn {color: red;}
p.warn a:link {color: green;}
```

Then you change your mind, deciding that warning text should be dark red, and that unvisited links in such text should be medium purple. The preceding rules need only be changed to reflect the new values. The following code results in Figure 5-15:

```
p.warn {color: #600;}
p.warn a:link {color: #400040;}
```

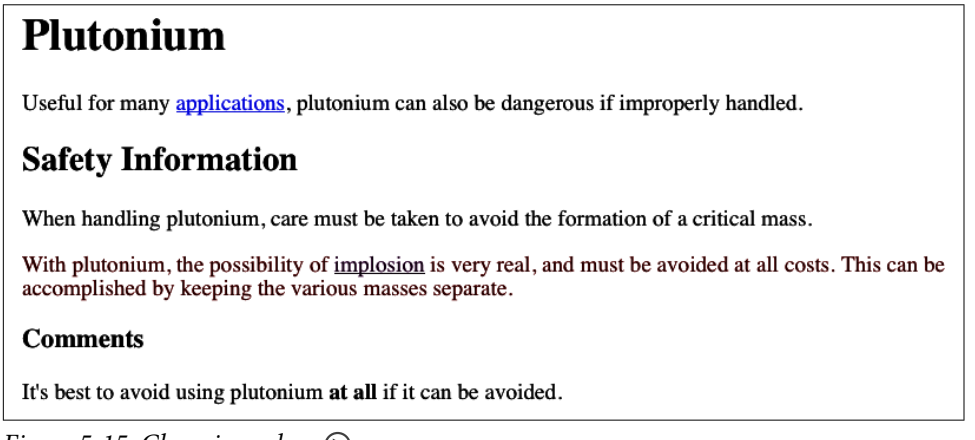


Figure 5-15. Changing colors

Another use for `color` is to draw attention to certain types of text. For example, boldfaced text is already fairly obvious, but you could give it a different color to make it stand out even further—let's say, maroon:

```
b, strong {color: maroon;}
```

Then you decide that you want all table cells with a class of `highlight` to contain light yellow text:

```
td.highlight {color: #FF9;}
```

If you don't set a background color for any of your text, you run the risk that a user's setup won't combine well with your own. For example, if a user has set their browser's background to be a pale yellow, like `#FFC`, then the previous rule would generate light-yellow text on a pale-yellow background. Far more likely is that it's still the default background of white, against which light yellow is still going to be hard to read. It's therefore generally a good idea to set foreground and background colors together. (We'll talk about background colors shortly.)

Affecting Form Elements

Setting a value for `color` should (in theory, anyway) apply to form elements. Declaring `<select>` elements to have dark-gray text should be as simple as this:

```
select {color: rgb(33%,33%,33%);}
```

This might also set the color of the borders around the edge of the `<select>` element, or it might not. It all depends on the user agent and its default styles.

You can also set the foreground color of input elements—although, as you can see in Figure 5-16, doing so would apply that color to all inputs, from text to radio buttons to checkbox inputs:

```
select {color: rgb(33%,33%,33%);}
input {color: red;}
```

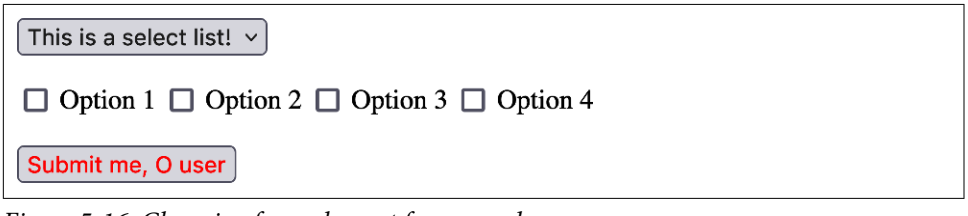


Figure 5-16. Changing form element foregrounds

Note in Figure 5-16 that the text color next to the checkboxes is still black. This is because the rules shown assign styles only to elements like `<input>` and `<select>`, not normal paragraph (or other) text.

Also note that the checkmark in the checkbox is black. This is due to the way form elements are handled in some web browsers, which typically use the form widgets built into the base operating system. Thus, when you see a checkbox and checkmark, they really aren't content in the HTML document—they're UI widgets that have been inserted into the document, much as an image would be. In fact, form inputs are, like images, replaced elements. In theory, CSS does not style the contents of form elements (though this may change in the future).

In practice, the line is a lot blurrier than that, as Figure 5-16 demonstrates. Some form inputs have the color of their text and even portions of their UI changed, while others do not. And since the rules aren't explicitly defined, behavior is inconsistent across browsers. In short, form elements are deeply tricky to style and should be approached with extreme caution.

Inheriting Color

As the definition of `color` indicates, the property is inherited. This makes sense, since if you declare `p {color: gray;}`, you probably expect that any text within that paragraph will also be gray, even if it's emphasized or boldfaced or whatever. If you *want* such

elements to be different colors, that's easy enough. The following code, for example, results in Figure 5-17:

```
em {color: red;}
p {color: gray;}
```

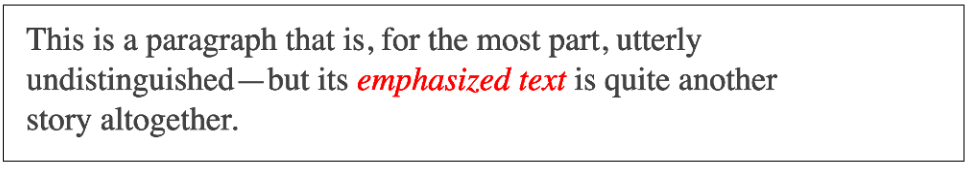


Figure 5-17. Different colors for different elements

Since color is inherited, it's theoretically possible to set all of the ordinary text in a document to a color, such as red, by declaring `body {color: red;}`. This should make all text that is not otherwise styled (such as anchors, which have their own color styles) red.

Angles

Since we just recently finished talking about hue angles in a number of color value types, this is a good time to talk about angle units. Angles in general are represented as <*angle*>, which is a <*number*> followed by one of four unit types:

`deg`
: Degrees, of which there are 360 in a full circle.

`grad`
: Gradians, of which there are 400 in a full circle. Also known as *grades* or *gons*.

`rad`
: Radians, of which there are 2π (approximately 6.28) in a full circle.

`turn`
: Turns, of which there is one in a full circle. This unit is mostly useful when animating a rotation and you wish to have it turn multiple times, such as `10turn` to make it spin 10 times. (Sadly, the pluralization `turns` is invalid, at least as of early 2023, and will be ignored.)

To help understand the relationships among these angle types, Table 5-2 shows how some angles are expressed in the various angle units. Unlike for length values, when including angles, the unit is always required, even if the value is `0deg`.

Table 5-2. Angle equivalents

Degrees	Gradians	Radians	Turns
0deg	0grad	0rad	0turn
45deg	50grad	0.785rad	0.125turn
90deg	100grad	1.571rad	0.25turn
180deg	200grad	3.142rad	0.5turn
270deg	300grad	4.712rad	0.75turn
360deg	400grad	6.283rad	1turn

Time and Frequency

When a property needs to express a period of time, the value is represented as <*time*> and is a <*number*> followed by either `s` (seconds) or `ms` (milliseconds.) Time values are most often used in transitions and animations, either to define durations or delays. The following two declarations will have exactly the same result:

```
a[href] {transition-duration: 2.4s;}
a[href] {transition-duration: 2400ms;}
```

Time values are also used in aural CSS, again to define durations or delays, but support for aural CSS is extremely limited as of this writing.

Another value type historically used in aural CSS is <*frequency*>, which is a <*number*> followed by either `Hz` (hertz) or `kHz` (kilohertz). As usual, the unit identifiers are case-insensitive, so `Hz` and `hz` are equivalent. The following two declarations will have exactly the same result:

```
h1 {pitch: 128hz;}
h1 {pitch: 0.128khz;}
```

Unlike with length values, for time and frequency values the unit type is *always* required, even when the value is `0s` or `0hz`.

Ratios

When you need to express a ratio of two numbers, you use a <*ratio*> value. These values are represented as two positive <*number*> values separated by a forward slash (`/`), plus optional whitespace.

The first integer refers to the width (inline size) of an element, and the second to the height (block size). Thus, to express a height-to-width ratio of 16 to 9, you can write `16/9` or `16 / 9`.

As of late 2022, there is no facility to express a ratio as a single real number (e.g., `1.777` instead of `16/9`), nor to use a colon separator instead of a forward slash (e.g., `16:9`).

Position

You use a *position value*, represented as <*position*>, to specify the placement of an origin image in a background area. Its syntactical structure is rather complicated:

```
[
  [ left | center | right | top | bottom | <percentage> | <length> ] |
  [ left | center | right | <percentage> | <length> ]
  [ top | center | bottom | <percentage> | <length> ] |
  [ center | [ left | right ] [ <percentage> | <length> ]? ] &&
  [ center | [ top | bottom ] [ <percentage> | <length> ]? ]
]
```

That might seem a little nutty, but it's all down to the subtly complex patterns that this value type has to allow.

If you declare only one value, such as `left` or `25%`, the second value is set to `center`. Thus, `left` is the same as `left center`, and `25%` is the same as `25% center`.

If you declare two values (either implicitly, as in the previous example, or explicitly), and the first one is a <*length*> or <*percentage*>, then it is *always* considered to be the horizontal value. Therefore, given `25% 35px`, the `25%` is a horizontal distance and the `35px` is a vertical distance. If you swap them to say `35px 25%`, then `35px` is horizontal and `25%` is vertical. If you write `25% left` or `35px right`, the entire value is invalid because you have supplied two horizontal distances and no vertical distance. (Similarly, a value of `right left` or `top bottom` is invalid and will be ignored.) On the other hand, if you write `left 25%` or `right 35px`, there is no problem because you've given a horizontal distance (with the keyword) and a vertical distance (with the percentage or length).

If you declare four values (we'll deal with three in just a moment), you must have two lengths or percentages, each of which is preceded by a keyword. In this case, each length or percentage specifies an offset distance, and each keyword defines the edge from which the offset is calculated. Thus, `right 10px bottom 30px` means an offset of 10 pixels to the left of the right edge, and an offset of 30 pixels up from the bottom edge. Similarly, `top 50% left 35px` means a 50% offset from the top and a 35-pixels-to-the-right offset from the left.

You can declare only three position values with the `background-position` property. If you declare three values, the rules are the same as for four, except the fourth offset is set to be 0 (no offset). Thus `right 20px top` is the same as `right 20px top 0`.

Custom Properties

If you've used a preprocessor like Less or Sass, you've probably created variables to hold values. CSS itself has this capability as well. The technical term for this is *custom properties*, even though what these really do is create something like variables in your CSS.

Here's a basic example, with the result shown in Figure 5-18 (though color won't be visible in the printed version):

```
html {
    --base-color: #639;
    --highlight-color: #AEA;
}

h1 {color: var(--base-color);}
h2 {color: var(--highlight-color);}
```



Figure 5-18. Using custom values to color headings

There are two things to absorb here. The first is the definition of the custom values `--base-color` and `--highlight-color`. These are not some sort of special color types. They're just names that we picked to describe what the values contain. We could just as easily have said this:

```
html {
    --alison: #639;
    --david: #AEA;
}

h1 {color: var(--alison);}
h2 {color: var(--david);}
```

You probably shouldn't do that sort of thing, unless you're literally defining colors that specifically correspond to people named Alison and David. (Perhaps on an "About Our Team" page.) It's always better to define custom identifiers that are self-documenting—things like `main-color` or `accent-color` or `brand-font-face`.

The important point is that any custom identifier of this type begins with *two* hyphens (`--`). It can then be invoked later by using a `var()` value type. Note that these names are case-sensitive, so `--main-color` and `--Main-color` are completely separate identifiers.

These custom identifiers are often referred to as *CSS variables*, which explains the `var()` pattern. An interesting feature of custom properties is their ability to scope themselves to a given portion of the DOM. If that sentence made any sense to you, it probably gave a little thrill. If not, here's an example to illustrate scoping, with the result shown in Figure 5-19:

```
html {
    --base-color: #666;
}
aside {
    --base-color: #CCC;
}

h1 {color: var(--base-color);}
```

```
<body>

<h1>Heading 1</h1><p>Main text.</p>

<aside>
    <h1>Heading 1</h1><p>An aside.</p>
</aside>

<h1>Heading 1</h1><p>Main text.</p>

</body>
```



Figure 5-19. Scoping custom values to certain contexts

Notice that the headings are a dark gray outside the `<aside>` element and a light gray inside. That's because the variable `--base-color` was updated for `<aside>` elements. The new custom value applies to any `<h1>` inside an `<aside>` element.

A great many patterns are possible with CSS variables, even if they are confined to value replacement. Here's an example suggested by Chriztian Steinmeier combining variables with the `calc()` function to create a regular set of indents for unordered lists:

```
html {
    --gutter: 3ch;
    --offset: 1;
}
```

```
ul li {margin-left: calc(var(--gutter) * var(--offset));}
ul ul li {--offset: 2;}
ul ul ul li {--offset: 3;}
```

This particular example is basically the same as writing the following:

```
ul li {margin-left: 3ch;}
ul ul li {margin-left: 6ch;}
ul ul ul li {margin-left: 9ch;}
```

The difference is that with variables, it's simple to update the `--gutter` multiplier in one place and have everything adjust automatically, rather than having to retype three values and make sure all the math is correct.

Custom Property Fallbacks

When you're setting a value by using `var()`, you can specify a fallback value. For example, you could say that if a custom property isn't defined, you want a regular value used instead, like so:

```
ol li {margin-left: var(--list-indent, 2em);}
```

Given that, if `--list-indent` isn't defined, is determined to be invalid, or is explicitly set to `initial`, `2em` will be used instead. You get just the one fallback, and it can't be another custom property name.

That said, it *can* be another `var()` expression, and that nested `var()` can contain another `var()` as its fallback, and so on to infinity. So let's say you're using a pattern library that defines colors for various interface elements. If those aren't available for some reason, you could fall back to a custom property value defined by your basic site stylesheet. Then, if that's also not available, you could fall back to a plain color value. It would look something like this:

```
.popup {color: var(--pattern-modal-color, var(--highlight-color, maroon));}
```

The thing to watch out for here is that if you manage to create an invalid value, the whole thing gets blown up and the value is either inherited or set to its initial value, depending on whether the property in question is usually inherited or not, as if it were set to `unset` (see "unset" on page 129).

Suppose we wrote the following invalid `var()` values:

```
:root {
        --list-color: hsl(23, 25%, 50%);
        --list-indent: 5vw;
}

li {
        color: var(--list-color, --base-color, gray);
        margin-left: var(--list-indent, --left-indent, 2em);
}
```

In the first case, the fallback is `--base-color, gray` as a single string, not something that's parsed, so it's invalid. Similarly, in the second case, the fallback `--left-indent` was never declared. In either case, if the first custom property is valid, the invalid fallback doesn't matter, because the browser never gets to it. But if, say, `--list-indent` doesn't have a value, the browser will go to the fallback, and here that's invalid. So what happens next?

For the color, since the property `color` is inherited, the list items will inherit their color from their parent, almost certainly an `<ol>` or `<ul>` element. If the parent's `color` value is `fuchsia`, the list items will be fuchsia. For the left margin, the property `margin-left` is not inherited, so the left margins of the list items will be set to the initial value of `margin-left`, which is `0`. So the list items will have no left margin.

This also happens if you try to apply a value to a property that can't accept those kinds of values. Consider the following:

```
:root {
        --list-color: hsl(23, 25%, 50%);
        --list-indent: 5vw;
}

li {
        color: var(--list-indent, gray);
        margin-left: var(--list-color, 2em);
}
```

Here, everything looks fine at first glance, except the `color` property is being given a length value, and the `margin-left` property is being given a color value. As a result, the fallbacks of `gray` and `2em` are not used. This is because the `var()` syntax is valid, so the result is the same as if we declared `color: 5vw` and `margin-left: hsl(23, 25%, 50%)`, both of which are tossed out as invalid.

This means the outcome will be the same as we saw before: the list items will inherit the color value from their parents, and their left margins will be set to the initial value of 0, just as if the given values were `unset`.

Summary

As you've seen, CSS provides a wide range of value and unit types. These units can have advantages and drawbacks, depending on the circumstances in which they're used. You've already seen some of those circumstances, and their nuances will be discussed throughout the rest of the book, as appropriate.

CHAPTER 6

Basic Visual Formatting

You've likely experienced the frustration of having your intended layout not render as expected. After adding 27 style rules to get it perfect, you still might not know which rule solved your problem. With a model as open and powerful as that contained within CSS, no book could hope to cover every possible way of combining properties and effects. You will undoubtedly go on to discover new ways of using CSS. With a thorough grasp of how the visual rendering model works, however, you'll be better able to determine whether a behavior is a correct (if unexpected) consequence of the rendering engine CSS defines.

Basic Boxes

At its core, CSS assumes that every element generates one or more rectangular boxes, called *element boxes*. (Future versions of the specification may allow for nonrectangular shapes, and indeed changes have been proposed, but for now all boxes are still rectangular.)

Each element box has a *content area* at its center. This content area is surrounded by optional amounts of padding, borders, outlines, and margins. These areas are considered optional because they could all be set to a size of 0, effectively removing them from the element box. Figure 6-1 shows an example content area, along with the surrounding regions of padding, borders, and margins.

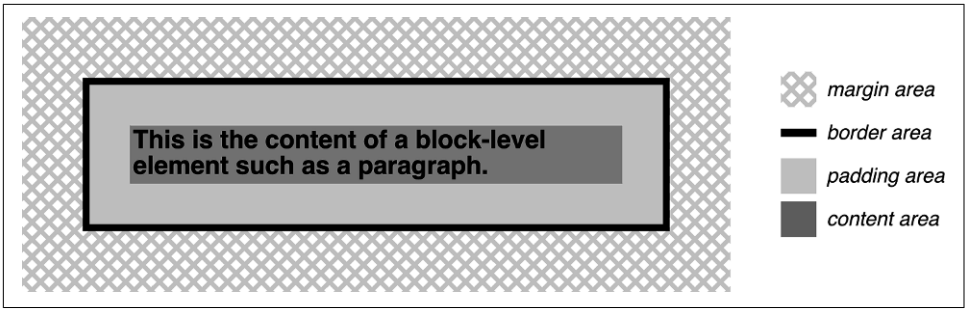


Figure 6-1. The content area and its surroundings

Before looking at the properties that can alter the space taken up by elements, let's cover the vocabulary needed to fully understand how elements are laid out and take up space.

A Quick Primer

First, we'll quickly review the kinds of boxes we'll be discussing, as well as some important terms that are needed to understand the explanations to come:

Block flow direction
: Also known as the *block axis*, this is the direction along which block-level element boxes are stacked. In many languages, including all European and European-derived languages, this direction is from top to bottom. In Chinese/Japanese/Korean (CJK) languages, this can be either right to left or top to bottom. The actual block flow direction is set by the writing mode, which is discussed in Chapter 15.

Inline base direction
: Also known as the *inline axis*, this is the direction along which lines of text are written. In Romanic languages, among others, this is left to right. In languages such as Arabic or Hebrew, the inline base direction is right to left. In CJK languages, this can be either top to bottom or left to right. As with block flow, the inline base direction is set by the writing mode.

Normal flow
: The default system by which elements are placed inside the browser's viewport, based on the parent's writing mode. Most elements are in the normal flow, and the only way for an element to leave the normal flow is to be floated, positioned, or made into a flexible box, grid layout, or table element. The discussions in this chapter cover elements in the normal flow, unless otherwise stated.

Block box
: This is a box generated by an element such as a paragraph, heading, or `<div>`. These boxes generate blank spaces both before and after their boxes when in the normal flow so that block boxes in the normal flow stack along the block flow axis, one after another. Pretty much any element can be made to generate a block box by declaring

`display: block`, though there are other ways to make elements generate block boxes (e.g., float them or make them flex items).

Inline box
: This is a box generated by an element such as `<strong>` or `<span>`. These boxes are laid out along the inline base direction, and do not generate line breaks before or after themselves. An inline box longer than the inline size of its element will (by default, if it's nonreplaced) wrap to multiple lines. Any element can be made to generate an inline box by declaring `display: inline`.

Nonreplaced element
: This is an element whose content is contained within the document. For example, a paragraph (`<p>`) is a nonreplaced element because its textual content is found within the element itself.

Replaced element
: This is an element that serves as a placeholder for something else. The classic example of a replaced element is `<img>`, which simply points to an image file that is inserted into the document's flow at the point where the `<img>` element itself is found. Most form elements are also replaced (e.g., `<input type="radio">`).

Root element
: This is the element at the top of the document tree. In HTML documents, this is the element `<html>`. In XML documents, it can be whatever the language permits: for example, the root element of RSS files is `<rss>`, whereas in an SVG document, the root element is `<svg>`.

The Containing Block

We need to examine one more kind of box in detail, and in this case enough detail that it merits its own section: the *containing block*.

Every element's box is laid out with respect to its containing block. In a very real way, the containing block is the *layout context* for a box. CSS defines a series of rules for determining a box's containing block.

For a given element, the containing block forms from the *content edge* of the nearest ancestor element that generates a list item or block box, which includes all table-related boxes (e.g., those generated by table cells). Consider the following:

```
<body>
    <div>
        <p>This is a paragraph.</p>
    </div>
</body>
```

In this simple markup, the containing block for the `<p>` element's block box is the `<div>` element's block box, as that is the closest ancestor element box that has a block or a list item box (in this case, it's a block box). Similarly, the `<div>`'s containing block is the

<body>'s box. Thus, the layout of the <p> is dependent on the layout of the , which is in turn dependent on the layout of the <body> element.

And above that in the document tree, the layout of the <body> element is dependent on the layout of the <html> element, whose box creates what is called the *initial containing block*. It's unique in that the viewport—the browser window in screen media, or the printable area of the page in print media—determines the dimensions of the initial containing block, not the size of the content of the root element. This matters because the content can be shorter, and usually longer, than the size of the viewport. Most of the time it doesn't make a difference, but when it comes to things such as fixed positioning or viewport units, the difference is real.

Now that you understand some of the terminology, we can address the properties that make up Figure 6-1. The various margin, border, and padding features, such as `border-style`, can be set using various side-specific longhand properties, such as `margin-inline-start` or `border-bottom-width`. (The outline properties do not have side-specific properties; a change to an outline property affects all four sides.)

The content's background—a color or tiled image, for example—is applied within the padding and border areas by default, but this can be changed. The margins are always transparent, allowing the background(s) of any parent element(s) to be visible. Padding and borders cannot be of a negative length, but margins can. We'll explore the effects of negative margins in "Negative Margins and Collapsing" on page 195.

Borders are most often generated using defined styles, with a `border-style` such as `solid`, `dotted`, or `inset`, and their colors are set using the `border-color` property. If no color is set, the value defaults to `currentcolor`. Borders can also be generated from images. If a border style has gaps of some type, as with `border-style: dashed` or with a border generated from a partially transparent image, then the element's background is visible through those gaps by default, though it is possible to clip the background to stay inside the border (or the padding).

Altering Element Display

You can affect the way a user agent displays by setting a value for the `display` property.

display

Values	[*<display-outside>* ‖ *<display-inside>*] | *<display-listitem>* | *<display-internal>* | *<display-box>* | *<display-legacy>*
Definitions	See below
Initial value	`inline`
Applies to	All elements
Computed value	As specified

Inherited	No
Animatable	No

<display-outside>
    `block` | `inline` | `run-in`

<display-inside>
    `flow` | `flow-root` | `table` | `flex` | `grid` | `ruby`

<display-listitem>
    `list-item` && *<display-outside>*? && [`flow` | `flow-root`]?

<display-internal>
    `table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-cell` | `table-column-group` | `table-column` | `table-caption` | `ruby-base` | `ruby-text` | `ruby-base-container` | `ruby-text-container`

<display-box>
    `contents` | `none`

<display-legacy>
    `inline-block` | `inline-list-item` | `inline-table` | `inline-flex` | `inline-grid`

We're going to ignore the `ruby`- and `table`-related values, since they're far too complex for this chapter. We'll also temporarily ignore the value `list-item`, since it's very similar to block boxes and is explored in detail in Chapter 16. For now, we'll spend a moment talking about how altering an element's display role can alter layout.

Changing Roles

When it comes to styling a document, it's sometimes handy to be able to change the type of box an element generates. For example, suppose we have a series of links in a `<nav>` that we'd like to lay out as a vertical sidebar:

```
<nav>
    <a href="index.html">WidgetCo Home</a>
    <a href="products.html">Products</a>
    <a href="services.html">Services</a>
    <a href="fun.html">Widgety Fun!</a>
    <a href="support.html">Support</a>
    <a href="about.html" id="current">About Us</a>
    <a href="contact.html">Contact</a>
</nav>
```

By default, the links will generate inline boxes, and thus get sort of mushed together into what looks like a short paragraph of nothing but links. We could put all the links into their own paragraphs or list items, or we could just make them all block-level elements, like this:

```
nav a {display: block;}
```

This will make every `<a>` element within the navigation element `<nav>` generate a block box, instead of its usual inline box. If we add on a few more styles, we could have a result like that shown in Figure 6-2.



Figure 6-2. Changing the display role from inline to block

Changing display roles can be useful when you want the navigation links to be inline elements if the CSS isn't available (perhaps by failing to load), but to lay out the same links as block-level elements in CSS-aware contexts. You could also present the links as inline on desktop displays and block on mobile devices, or vice versa. With the links laid out as blocks, you can style them as you would `<div>` or `<p>` elements, with the advantage that the entire element box becomes part of the link.

You may also want to take elements and make them inline. Suppose we have an unordered list of names:

```
<ul id="rollcall">
    <li>Bob C.</li>
    <li>Marcio G.</li>
    <li>Eric M.</li>
    <li>Kat M.</li>
    <li>Tristan N.</li>
    <li>Arun R.</li>
    <li>Doron R.</li>
    <li>Susie W.</li>
</ul>
```

Given this markup, say we want our display to show a series of inline names with vertical bars between them (and on each end of the list). The only way to do so is to change their display role. The following rules will have the effect shown in Figure 6-3:

```
#rollcall li {display: inline; border-right: 1px solid; padding: 0 0.33em;}
#rollcall li:first-child {border-left: 1px solid;}
```

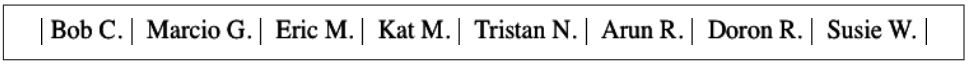


Figure 6-3. Changing the display role from `list-item` to `inline`

Understand that you are, for the most part, changing the display role of elements—not changing their inherent nature. In other words, causing a paragraph to generate an inline box does *not* turn that paragraph into an inline element. In HTML, for example, some elements are block while others are inline. While a `<span>` can easily be placed inside a paragraph, a `<span>` should not be wrapped around a paragraph.

We say "for the most part" because while CSS mostly impacts presentation and not content, CSS properties can impact accessibility in more ways than just color contrast. For example, changing the `display` value can impact the way an element is perceived by assistive technologies. Setting an element's `display` property to `none` removes the element from the accessibility tree. Setting the `display` property on a `<table>` to `grid` may cause the table to be interpreted as something other than a data table, removing normal table keyboard navigation, and making the table inaccessible as a data table to screen-reader users. (This shouldn't happen, but it does in some browsers.)

This can be mitigated by setting the Accessible Rich Internet Applications (ARIA) `role` attribute for the table and all its descendants. However, in general, anytime a change you make in CSS forces you to make changes in ARIA roles, you should take a moment to consider whether there isn't a better way to accomplish your goal.

Handling Block Boxes

Block boxes behave in predictable, yet sometimes surprising, ways. The handling of box placement along the block and inline axes can differ, for example. To fully understand how block boxes are handled, you must clearly understand several aspects of these boxes. They are shown in detail in Figure 6-4, which illustrates placement in two different writing modes.

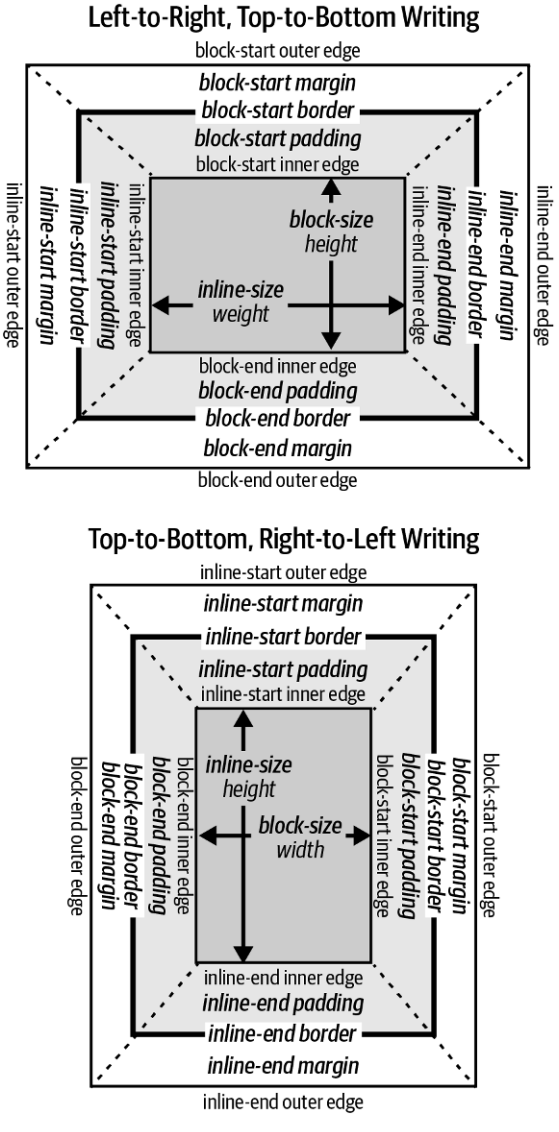


Figure 6-4. The complete box model in two different writing modes

As shown in Figure 6-4, CSS deals with the block directions and inline directions, as well as block sizes and inline sizes. Block and inline sizes are descriptions of the size of the content area (by default) along the block and inline axes.

By contrast, the *width* (sometimes referred to as the *physical width*) of a block box is defined as the distance between the inner edges of the content area (again, by default) along the horizontal axis (left to right), regardless of the writing direction, and the *height* (*physical height*) is the distance along the vertical axis (top to bottom). Properties are available to set all these sizes, which we'll talk about shortly.

Something important to note in Figure 6-4 is the use of *start* and *end* to describe various parts of the element box. For example, you'll see a block-start margin and a block-end margin. The *start edge* is the edge that you come to first as you move along an axis.

This may be clearer if you look at Figure 6-5 and trace your finger along each axis from arrow tail to tip. As you move along a block axis, the first edge you come to for each element is that element's block-start edge. As you pass out of the element, you move through the block-end edges. Similarly, as you move along an inline axis, you go through the inline-start edges, across the inline dimension of the content, and then out the inline-end edges. Try it for each of the three examples.

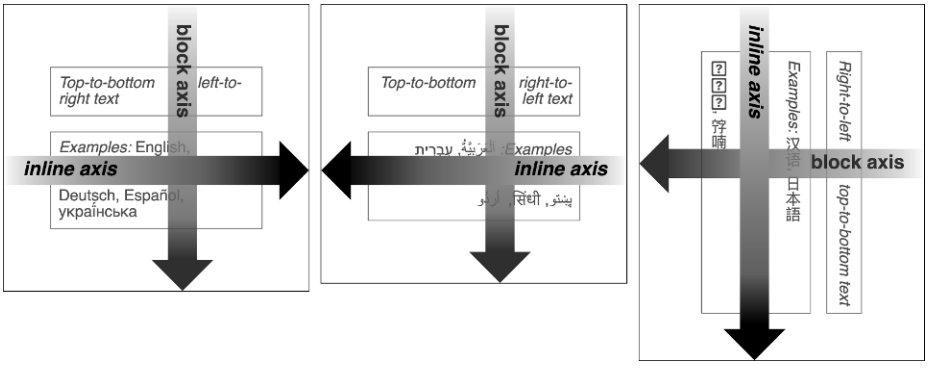


Figure 6-5. The block- and inline-axis directions for three common writing modes

Logical Element Sizing

Because CSS recognizes block and inline axes for elements, it provides properties that let you set an explicit element size along each axis.

block-size, inline-size

Values	<*length*> | <*percentage*> | `min-content` | `max-content` | `fit-content` | `auto`
Initial value	`auto`
Applies to	All elements except nonreplaced inline elements, table rows, and row groups
Percentages	Calculated with respect to the length of the element's containing block along the block-flow axis (for `block-size`) or inline-flow axis (for `inline-size`)

Computed value	For `auto` and percentage values, as specified; otherwise, an absolute length, unless the property does not apply to the element (then `auto`)
Inherited	No
Animatable	Yes

These properties allow you to set the size of an element along its block axis, or to constrain the lengths of lines of text along the inline axis, regardless of the direction of text flow. If you write `block-size: 500px`, the element's block size will be 500 pixels wide, even if that leads to content spilling out of the element box. (We'll discuss that in more detail later in the chapter.)

Consider the following, which has the results shown in Figure 6-6 when applied in various writing modes:

```
p {inline-size: 25ch;}
```

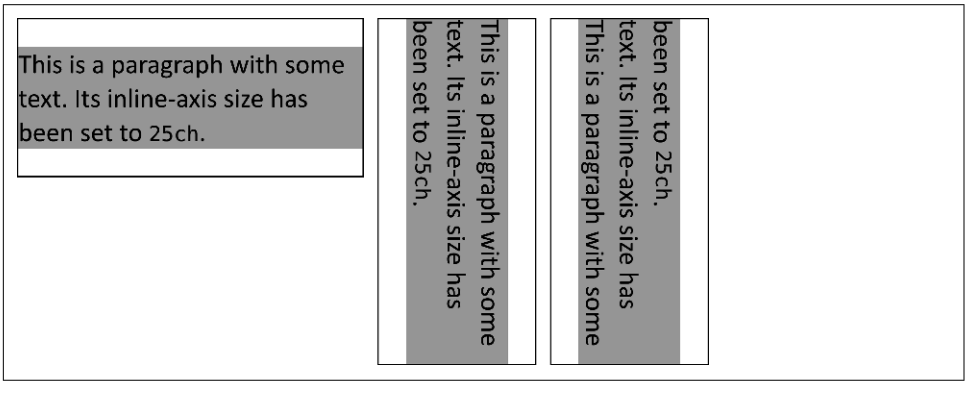


Figure 6-6. Sizing elements along their inline axis

As seen in Figure 6-6, the elements are sized consistently along their inline axis, regardless of the writing direction. If you tilt your head to the side, you can see that the lines wrap in exactly the same places. This yields a consistent line length across all writing modes.

Similarly, you can set a block size for elements. This is used a bit more often for replaced elements like images, but it can be used in any circumstance that makes sense. Take this as an example:

```
p img {block-size: 1.5em;}
```

Having done that, any `<img>` element found inside a `<p>` element will have its block size set to one and a half times the size of the surrounding text. (This works on images because they're inline replaced elements; it wouldn't work on inline nonreplaced elements.) You could also use `block-size` to constrain the block length of grid layout items to be a minimum or maximum size, such as this:

```
#maingrid > nav {block-size: clamp(2rem, 4em, 25vh);}
```

It should be said that usually block size is determined automatically, because elements in the normal flow often don't have an explicitly set block size. For example, if an element's block flow is top to bottom and it's 8 lines long, and each line is 1/8th of an inch tall, then its block size will be 1 inch. If it's 10 lines tall, the block size is instead 1.25 inches. In either case, as long as `block-size` is `auto`, the block size is determined by the content of the element, not by the author. This is usually what you want, particularly for elements containing text. When `block-size` is explicitly set and there isn't enough content to fill the box, empty space will appear inside the box; if there is more content than can fit, the content may overflow the box or scrollbars may appear.

Content-Based Sizing Values

Beyond the lengths and percentages you saw previously for setting block and inline sizes, a few keywords provide content-based sizing:

`max-content`
: Take up the most space possible to fit in the content, even suppressing line wrapping in the case of text content.

`min-content`
: Take up the least space possible to fit in the content.

`fit-content`
: Take up the amount of space determined by calculating the values of `max-content`, `min-content`, and regular content sizing, taking the maximum of `min-content` and regular sizing, and then taking the minimum of `max-content` and whichever value was the greater of `min-content` and regular sizing. Yes, that all sounds a bit confusing, but we'll explain it in a moment.

If you've worked at all with CSS Grid (covered in Chapter 12), you may recognize these keywords, as they were originally defined as ways to size grid items. Now they're making their way into other areas of CSS. Let's consider the first two keywords, which are demonstrated in Figure 6-7.

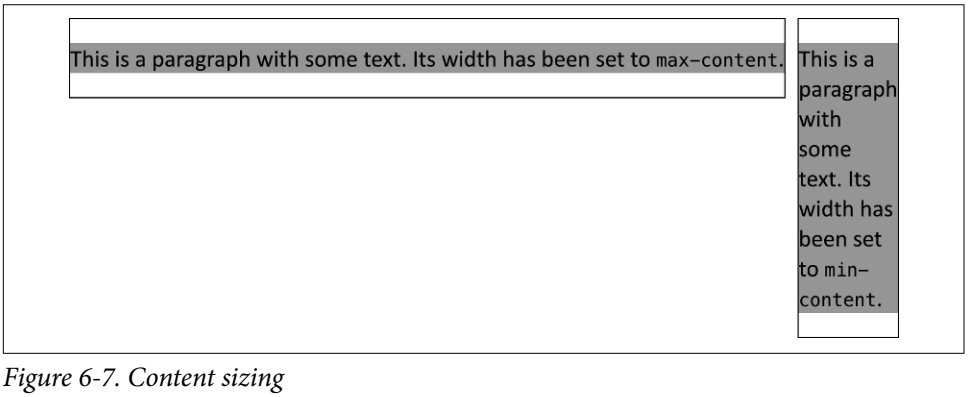


Figure 6-7. Content sizing

The paragraph on the left is set to `max-content`, and that's what happens: the paragraph is made as wide as needed to fit all of the content. It's as narrow as it is only because there isn't much content. Had we added another three sentences, the single line of text would have just kept going and going with no line wrapping, even if it ran right off the page (or out of the browser window).

In the paragraph on the right, the content is as narrow as possible without forcing breaks or hyphens inside words. In this particular case, the element is *just* wide enough to fit the word "paragraph," which is the longest word in the content. For every other line of text in the example, the browser places as many words as will fit into the space needed for "paragraph," and goes to the next line when it runs out of room. If we added "antidisestablishmentarianism" to the text, the element would become just wide enough to fit *that* word, and every other line of text would very likely contain multiple words.

Notice that, at the end of the `min-content` example in Figure 6-7, the browser took advantage of the presence of the hyphen in `min-content` to trigger a line wrap there. Had it not made that choice, `min-content` would almost certainly have been the longest piece of content in the paragraph, and the element's width would have been set to that length. This means that if your content contains symbols that browsers understand to be natural line-wrapping points (e.g., spaces and hyphens), they'll likely be considered in the `min-content` calculations. If you want to squeeze the element width even further, you can enable auto-hyphenating of words with the `hyphens` property (see Chapter 15).

For some more examples of `min-content` sizing, see Figure 6-8.

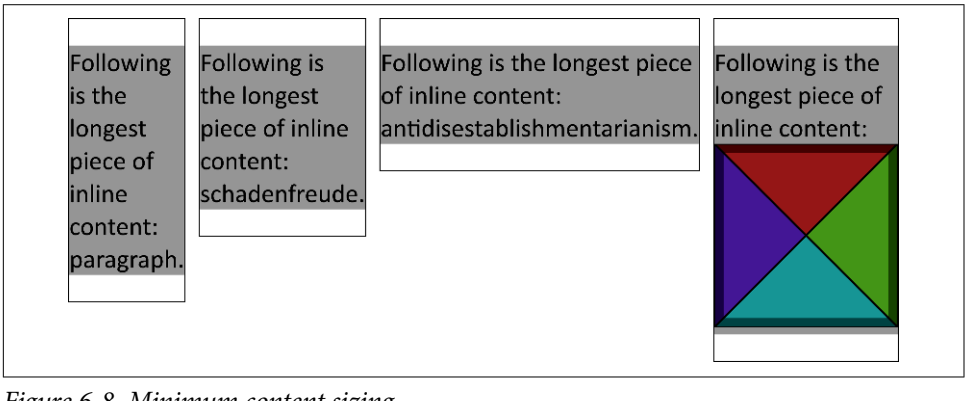


Figure 6-8. Minimum content sizing

The third keyword, `fit-content`, is interesting in that it does its best to fit the element to the content. What that means in practice is that if you have only a little content, the element's inline size (usually its width) will be just big enough to enclose it, as if `max-content` were used. If there's enough content to wrap to multiple lines or otherwise threaten to overflow the element's container, the inline size stops there. This is illustrated in Figure 6-9.

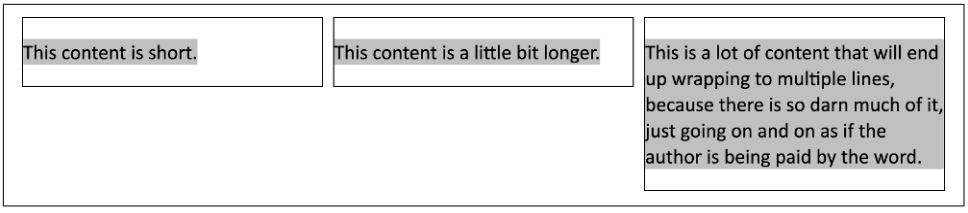


Figure 6-9. Fit-content sizing

In each case, the element is fit to the content without overspilling the element's container. At least, that's what happens with elements in the normal flow. The behavior can be quite different in flexbox and grid contexts, and is further explored in later chapters.

Minimum and Maximum Logical Sizing

If you'd like to set minimum and maximum bounds on block or inline sizes, CSS has some properties to help you out.

min-block-size, max-block-size, min-inline-size, max-inline-size

Values	Same as for `block-size` and `inline-size`
Initial value	`0`
Applies to	Same as for `block-size` and `inline-size`
Percentages	Same as for `block-size` and `inline-size`
Computed value	Same as for `block-size` and `inline-size`
Inherited	No
Animatable	Yes

These properties can be very useful when you know you want upper and lower bounds on the sizing of an element's box, and are willing to allow the browser to do whatever it wants as long as it obeys those restrictions. As an example, you might set part of a layout like so:

```
main {min-inline-size: min-content; max-inline-size: 75ch;}
```

That keeps the `<main>` element from getting any narrower than the widest bit of inline content, whether that's a long word or an illustration or something else. It also keeps the `<main>` element from getting any wider than around 75 characters, thus keeping line lengths to a readable amount.

It's also possible to set bounds on block sizing. A good example is limiting any image embedded in the normal flow to be its intrinsic size up to a certain point. The following CSS would have the effects shown in Figure 6-10:

```
#cb1 img {max-block-size: 2em;}
#cb2 img {max-block-size: 1em;}
```

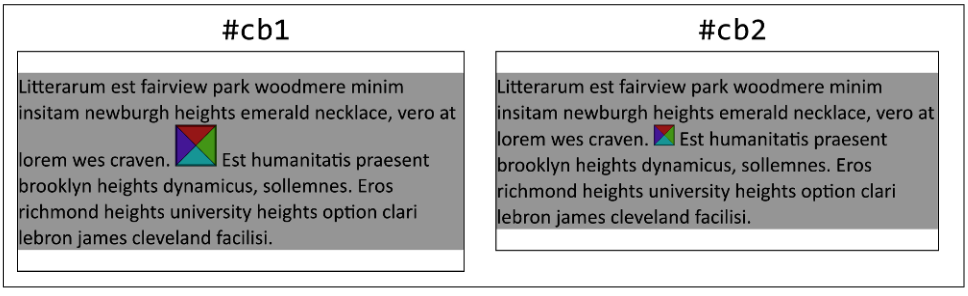


Figure 6-10. Maximum block sizing

Height and Width

If you've used CSS for a while or are maintaining legacy code, you're probably used to thinking of "top margin" and "bottom margin." That's because, originally, all box model aspects were described in terms of their physical directions: top, right, bottom, and left. You can still work with the physical directions! CSS has simply added new directions to the mix.

If you were to change `inline-size` to `width` in the previous code example, you'd get a result more like that shown in Figure 6-11 (in which the vertical writing modes are clipped off well short of their full height).

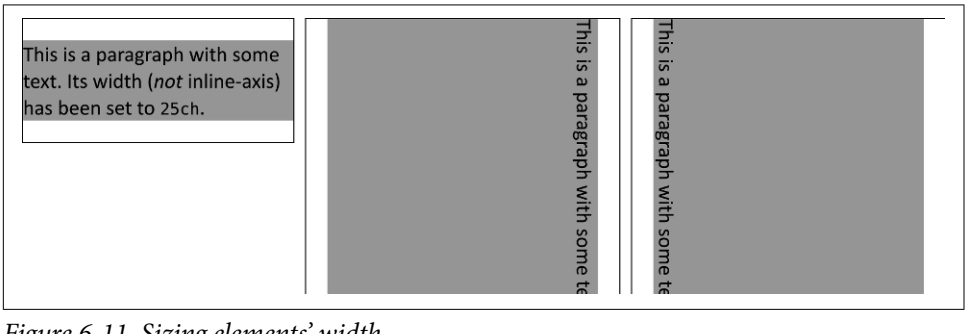


Figure 6-11. Sizing elements' width

These elements are made `40ch` wide horizontally, regardless of their writing mode. Each element's height has been automatically determined by the content, the specifics of the writing mode, and so on.



If you use block and inline properties like `block-size` instead of physical directions like `height`, and your design is applied to content translated to other languages, the layout will automatically adjust to your intentions.

height, width

Values	<*length*> | <*percentage*> | `min-content` | `max-content` | `fit-content` | `auto`
Initial value	`auto`
Applies to	All elements except nonreplaced inline elements, table rows, and row groups
Percentages	Calculated with respect to the vertical height (for `height`) or horizontal width (for `width`) of the containing block; for `height`, set to `auto` if the `height` of its containing block is `auto`
Computed value	For `auto` and percentage values, as specified; otherwise, an absolute length, unless the property does not apply to the element (then `auto`)
Inherited	No
Animatable	Yes

The `height` and `width` properties are known as *physical properties*. They refer to physical directions, as opposed to the writing-dependent directions of block size and inline size. Thus, `height` really does refer to the distance from the top to the bottom of the element's inner edge, regardless of the direction of the block axis.

In writing with a horizontal inline axis, such as English or Arabic, if both `inline-size` and `width` are set on the same element, the one declared later will take precedence over the first one declared. The same is true if `block-size` and `height` are both declared; if origin, layer, and specificity are the same, the one declared last takes precedence. In vertical writing modes, `inline-size` corresponds to `height`, and `block-size` to `width`.

Setting a block box's height or width as a <*length*> means it will be that length tall or wide, regardless of the content within it. If you set an element that generates a block box to `width: 200px`, it will be 200 pixels wide, even if it has a 500-pixel-wide image inside it.

Setting the value of `width` to a <*percentage*> means the width of the element will be that percentage of its containing block's width. If you set a paragraph to `width: 50%` and its containing block is 1,024 pixels wide, then the paragraph's `width` will be computed to 512 pixels.

Setting `height` to a <*percentage*> works similarly, except this works only if the containing block has an explicitly set height. If the containing block's height is automatically set, a percentage value is treated as `auto` instead, as seen in the `#cb4` example in Figure 6-12.



The handling of `auto` top and bottom margins is different for positioned elements, as well as flexible-box and grid elements. The differences are covered in Chapters 11 and 12.

Here are some examples of these values and combinations, with the result shown in Figure 6-12:

```
[id^="cb"] {border: 1px solid;}  /* "cb" for "containing block" */
#cb1 {width: auto;}    #cb1 p {width: auto;}
#cb2 {width: 400px;}   #cb2 p {width: 300px;}
#cb3 {width: 400px;}   #cb3 p {width: 50%;}

#cb4 {height: auto;}   #cb4 p {height: 50%;}
#cb5 {height: 300px;}  #cb5 p {height: 200px;}
#cb6 {height: 300px;}  #cb6 p {height: 50%;}
```

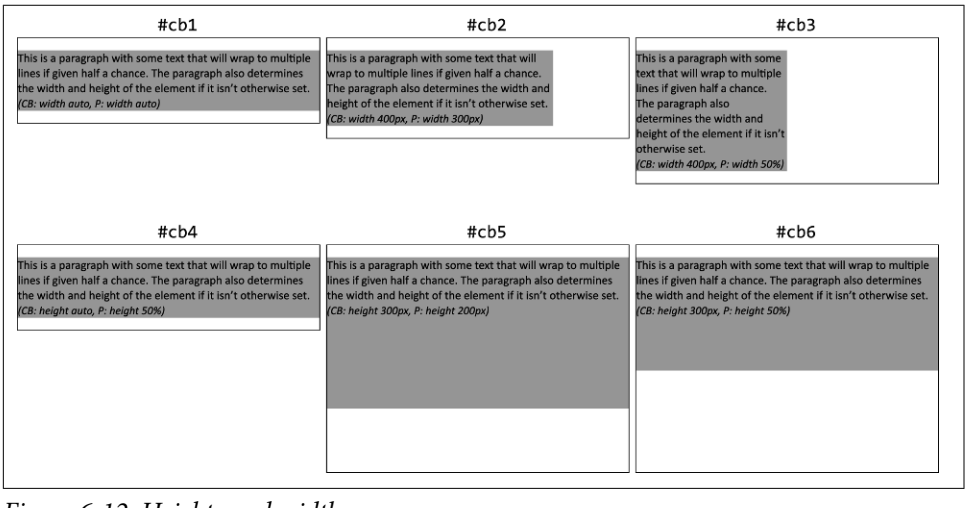


Figure 6-12. Heights and widths

You can also use `max-content` and `min-content` with the `height` property, but in top-to-bottom block flows, both are the same as `height: auto`. In writing modes using a horizontal block axis, setting these values for `height` will have similar effects as setting them for `width` in vertical block flows.

In addition, these properties don't apply to inline nonreplaced elements. For example, if you try to declare a `height` and `width` for a hyperlink that's in the normal flow and generates an inline box, CSS-conformant browsers *must* ignore those declarations. Assume the following rules:

```
a:any-link {color: red; background: silver; height: 15px; width: 60px;}
```

You'll end up with red unvisited links on silver backgrounds whose height and width are determined by the content of the links. The links will *not* have content areas that are 15 pixels tall by 60 pixels wide, as these must be ignored when applied to inline nonreplaced elements. If, on the other hand, you add a `display` value, such as `inline-block` or `block`, then `height` and `width` *will* set the height and width of the links' content areas.

Altering Box Sizing

If it seems a little weird to use `height` and `width` (and `block-size` and `inline-size`) to describe the sizing of the element's content area instead of its visible area, you can make that sizing more intuitive by using the property `box-sizing`.

box-sizing

Values	`content-box` | `border-box`
Initial value	`content-box`
Applies to	All elements that accept `width` or `height` values
Computed value	As specified
Inherited	No
Animatable	No

This property changes what the values of the `height`, `width`, `block-size`, and `inline-size` properties do.

If you declare `inline-size: 400px` and don't declare a value for `box-sizing`, the element's content area will be 400 pixels in the inline direction, and any padding, borders, and so on will be added to that. If, on the other hand, you declare `box-sizing: border-box`, the element box will be 400 pixels from the inline-start border edge to the inline-end border edge; any inline-start or -end border or padding will be placed within that distance, thus shrinking the inline size of the content area. This is illustrated in Figure 6-13.

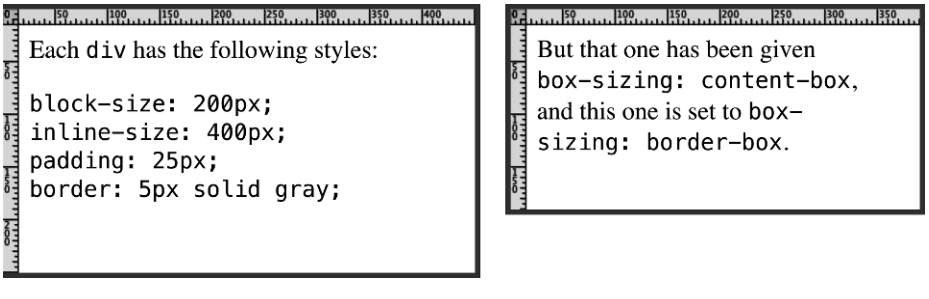


Figure 6-13. The effects of `box-sizing`

Put another way, if you declare `width: 400px` and don't declare a value for `box-sizing`, the element's content area will be 400 pixels wide, and any padding, borders, and so on will be added to that. If, on the other hand, you declare `box-sizing: border-box`, the element box will be 400 pixels from the left outer border edge to the right outer border

edge; any left or right border or padding will be placed within that distance, thus shrinking the width of the content area (again, as seen in Figure 6-13).

We're talking about the `box-sizing` property here because, as stated, it applies to "all elements that accept `width` or `height` values" (because it was defined before logical properties were commonplace). That's most often elements generating block boxes, though it also applies to replaced inline elements like images, as well as inline-block boxes.

Having established how to size elements in both logical and physical ways, let's widen our scope and look at all the properties that affect block sizing.

Block-Axis Properties

In total, block-axis formatting is affected by seven related properties: `margin-block-start`, `border-block-start`, `padding-block-start`, `height`, `padding-block-end`, `border-block-end`, and `margin-block-end`. These properties are diagrammed in Figure 6-14. These properties are all covered in detail in Chapter 7; here, we will talk about the general principles and behavior of these properties before looking at the details of their values.

The block-start and -end padding and borders must be set to specific values, or else they default to a width of 0, assuming no border style is declared. If `border-style` has been set, the thickness of the borders is set to `medium`, which is 3 pixels wide in all known browsers. Figure 6-14 depicts the block-axis formatting properties in two writing modes and indicates which parts of the box may have a value of `auto` and which may not.

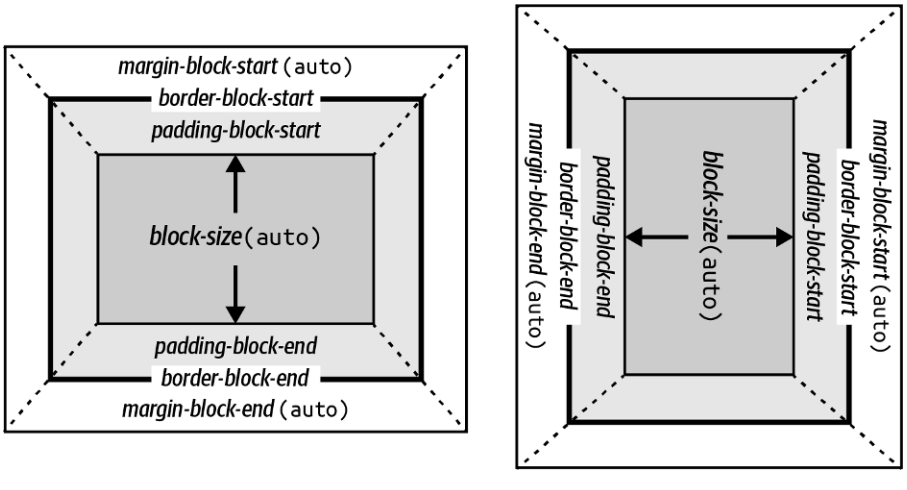


Figure 6-14. The seven properties of block-axis formatting, and which can be set to `auto`

Interestingly, if either `margin-block-start` or `margin-block-end` is set to `auto` for a block box in the normal flow, but not both, they both evaluate to `0`. A value of `0`, unfortunately, prevents easy block-direction centering of normal-flow boxes in their containing blocks (though such centering is fairly straightforward in flexbox or grid layout).

The `block-size` property must be set to `auto` or to a nonnegative value of some type; it can never be less than 0.

Auto Block Sizing

In the simplest case, a normal-flow block box with `block-size: auto` is rendered just tall enough to enclose the line boxes of its inline content (including text). If an auto-block-size, normal-flow block box has only block-level children and has no block-edge padding or borders, the distance from its first child's border-start edge to its last child's border-end edge will be the box's block size. This is the case because the margins of the child elements can "stick out" of the element that contains them thanks to what's known as *margin collapsing*, which we'll talk about in "Collapsing Block-Axis Margins" on page 196.

However, if a block-level element has either block-start or -end padding, or block-start and -end borders, its block size will be the distance from the block-start margin edge of its first child to the block-end margin edge of its last child:

```
<div style="block-size: auto; background: silver;">
    <p style="margin-block-start: 2em; margin-block-end: 2em;">A paragraph!</p>
</div>
<div style="block-size: auto; border-block-start: 1px solid;
     border-block-end: 1px solid; background: silver;">
    <p style="margin-block-start: 2em; margin-block-end: 2em;">
         Another paragraph!</p>
</div>
```

Figure 6-15 demonstrates both of these behaviors.

If we changed the borders in the previous example to be padding instead, the effect on the block size of the `<div>` would be the same: it would still enclose the paragraph's margins within it.

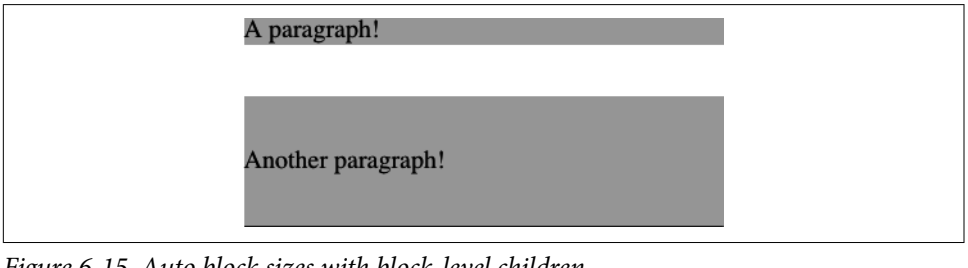


Figure 6-15. Auto block sizes with block-level children

Percentage Heights

You saw earlier how length-value block sizes are handled, so let's spend a moment on percentages. If the block size of a normal-flow block box is set to a percentage value, that value is taken as a percentage of the block size of the box's containing block, assuming the container has an explicit, non-`auto` block size of its own. Given the following markup, the paragraph will be 3 em long along the block axis:

```
<div style="block-size: 6em;">
    <p style="block-size: 50%;">Half as tall</p>
</div>
```

If the block size of the containing block is *not* explicitly declared, percentage block sizes are reset to `auto`. If we change the previous example so that the `block-size` of the `<div>` is `auto`, the paragraph will now have its block size determined automatically:

```
<div style="block-size: auto;">
    <p style="block-size: 50%;">NOT half as tall; block size reset to auto</p>
</div>
```

Figure 6-16 illustrates these two possibilities. (The spaces between the paragraph borders and the `<div>` borders are the block-start and -end margins on the paragraphs.)

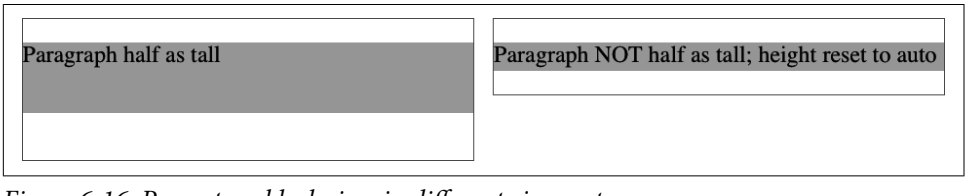


Figure 6-16. Percentage block sizes in different circumstances

Before we move on, take a closer look at the first example in Figure 6-16, the half-as-tall paragraph. It may be half as tall, but it isn't centered along the block axis. That's because the containing `<div>` is 6 em tall, which means the half-as-tall paragraph is 3 em tall. It has block-start and -end margins of 1 em thanks to the browser's default styles, so its overall block size is 5 em. That means there is actually 2 em of space between the block end of the paragraph's visible box and the `<div>`'s block-end border, not 1 em. Figure 6-17 illustrates this in detail.

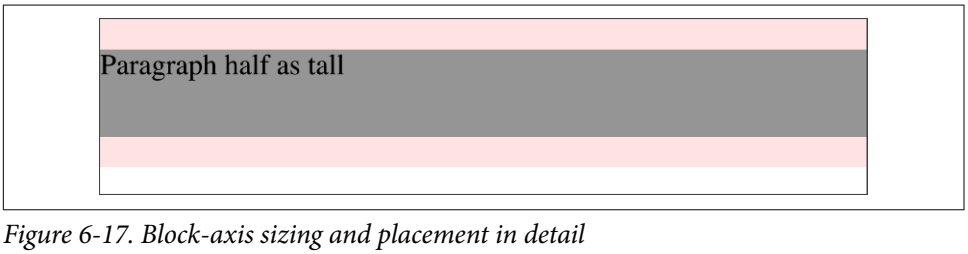


Figure 6-17. Block-axis sizing and placement in detail

Handling Content Overflow

Given that it's possible to set elements to specific sizes, it becomes possible to make an element too small for its content to fit inside. This is more likely to arise if block sizes are explicitly defined, but it can also happen with inline sizes, as you'll see in later sections. If this sort of thing does happen, you can exert some control over the situation with the `overflow` shorthand property.

overflow

Values	[`visible` | `hidden` | `clip` | `scroll` | `auto`]{1,2}
Initial value	`visible`
Applies to	Block-level and replaced elements
Computed value	As specified
Inherited	No
Animatable	No

The default value of `visible` means that the element's content may be visible outside the element's box. Typically, this leads to the content running outside its own element box, but not altering the shape of that box. The following markup would result in Figure 6-18:

```
div#sidebar {block-size: 7em; background: #BBB; overflow: visible;}
```

If `overflow` is set to `hidden`, the element's content is clipped at the edges of the element box. With the `hidden` value, there is no way to get at the parts of the content that are clipped off.

If `overflow` is set to `clip`, the element's content is also clipped—that is, hidden—at the edges of the element box, with no way to get at the parts that are clipped off.

If `overflow` is set to `scroll`, the overflowing content is clipped, but the content can be made available to the user via scrolling methods, including a scrollbar (or set of them). Figure 6-18 depicts one possibility.

If `scroll` is used, the panning mechanisms (e.g., scrollbars) should always be rendered. To quote the specification, "this avoids any problem with scrollbars appearing or disappearing in a dynamic environment." Thus, even if the element has sufficient space to display all its content, the scrollbars may still appear and take up space (though they may not).

In addition, when printing a page or otherwise displaying the document in a print medium, the content may be displayed as though the value of `overflow` were declared to be `visible`.

Figure 6-18 illustrates these `overflow` values, with two of them combined in a single example.

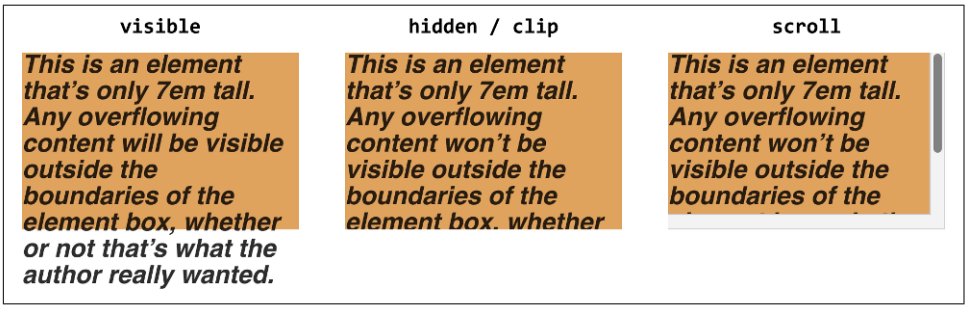


Figure 6-18. Methods for handling overflowing content

Finally, `overflow: auto` allows user agents to determine which of the previously described behaviors to use, although user agents are encouraged to provide a scrolling mechanism whenever necessary. This is a potentially useful way to use overflow, since user agents could interpret it to mean "provide scrollbars only when needed." (They may not, but generally do.)

Single-axis overflow

Two properties make up the `overflow` shorthand. You can define the overflow behavior along the *x* (horizontal) and *y* (vertical) directions separately, either by setting them both in `overflow`, or by using the `overflow-x` and `overflow-y` properties.

overflow-x, overflow-y

Values	`visible` | `hidden` | `clip` | `scroll` | `auto`
Initial value	`visible`
Applies to	Block-level and replaced elements
Computed value	As specified
Inherited	No
Animatable	No

By setting the overflow behavior separately along each axis, you're essentially deciding where scrollbars will appear and where they won't. Consider the following, which is rendered in Figure 6-19:

```
div.one   {overflow-x: scroll; overflow-y: hidden;}
div.two   {overflow-x: hidden; overflow-y: scroll;}
div.three {overflow-x: scroll; overflow-y: scroll;}
```

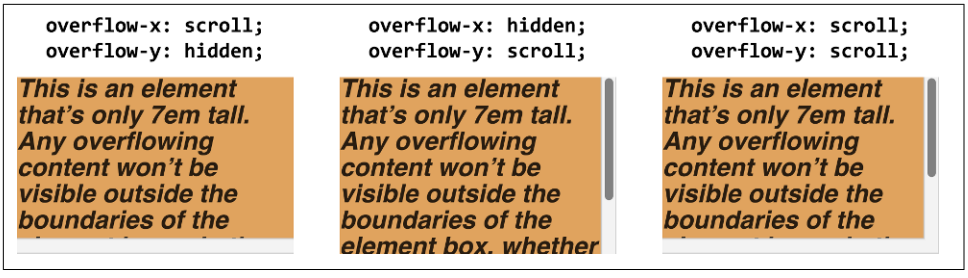


Figure 6-19. Setting overflow separately for x- and y-axes

In the first case, an empty scrollbar is set up for the x-axis (horizontal), but none for the y-axis (vertical), even though the content overflowed along the y-axis. This is the worst of both worlds: a scrollbar that's empty because it isn't needed, and no scrollbar where it is needed.

The second case is the much more useful inverse: no scrollbar is set along the x-axis, but one is available for the y-axis, so the overflowed content can be accessed by means of scrolling.

In the third case, `scroll` is set for both axes, so access to the overflowing content is available via scrolling, but we also have an unnecessary scrollbar (which is empty) for the x-axis. This is equivalent to simply declaring `overflow: scroll`.

This brings us to the true nature of `overflow`: it's a shorthand property that brings `overflow-x` and `overflow-y` together under one roof. The following is exactly equivalent to the previous example and will have the same result shown in Figure 6-19:

```
div.one   {overflow: scroll hidden;}
div.two   {overflow: hidden scroll;}
div.three {overflow: scroll;} /* 'scroll scroll' would also work */
```

As you see, you can give `overflow` two keywords, which are always in the order *x*, then *y*. If only one value is given, it's used for both the x- and y-axes. This is why `scroll` and `scroll scroll` are the same thing as values of `overflow`. Similarly, `hidden` would be the same as saying `hidden hidden`.

Negative Margins and Collapsing

Believe it or not, negative margins are possible. The base effect is to move the margin edge inward toward the center of the element's box. Consider this:

```
p.neg {margin-block-start: -50px; margin-block-end: 0;
    border: 3px solid gray;}

<div style="width: 420px; background-color: silver; padding: 10px;
            margin-block-start: 50px; border: 1px solid;">
    <p class="neg">
        A paragraph.
    </p>
```

```
    A div.
</div>
```

As we see in Figure 6-20, the paragraph has been pulled upward by its negative block-start margin. Note that the content of the `<div>` that follows the paragraph in the markup has also been pulled up the block axis by 50 pixels.

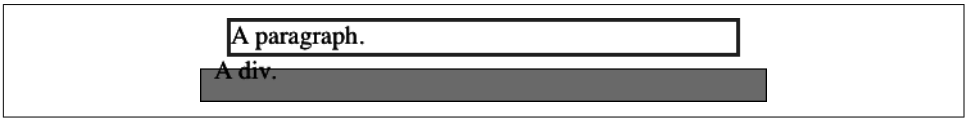


Figure 6-20. The effects of a negative block-start margin

Now compare the following markup to the situation shown in Figure 6-21:

```
p.neg {margin-block-end: -50px; margin-block-end: 0;
    border: 3px solid gray;}

<div style="width: 420px; margin-block-start: 50px;">
    <p class="neg">
        A paragraph.
    </p>
</div>
<p>
    The next paragraph.
</p>
```

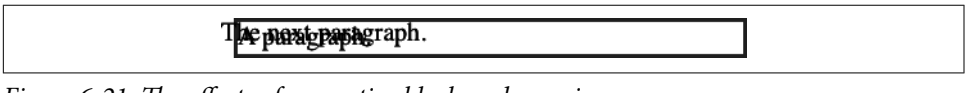


Figure 6-21. The effects of a negative block-end margin

What's happening? The elements following the `<div>` are placed according to the location of the block-end margin edge of the `<div>`, which is 50 pixels higher than it would be without the negative margin. As Figure 6-21 shows, the block-end of the `<div>` is actually *above* the visual block-end of its child paragraph. The next element after the `<div>` is the appropriate distance from the block-end of the `<div>`.

Collapsing Block-Axis Margins

An important aspect of block-axis formatting is the *collapsing* of adjacent margins, which is a way of comparing adjacent margins in the block direction, and then using only the largest of those margins to set the distance between the adjacent block elements. Note that collapsing behavior applies only to margins. Padding and borders never collapse.

An unordered list, with list items that follow one another along the block axis, is a perfect environment for studying margin collapsing. Assume that the following is declared for a list that contains three items:

```
li {margin-block-start: 10px; margin-block-end: 15px;}
```

Each list item has a 10-pixel block-start margin and a 15-pixel block-end margin. When the list is rendered, however, the visible distance between adjacent list items is 15 pixels, not 25. This happens because, along the block axis, adjacent margins are collapsed. In other words, the smaller of the two margins is eliminated in favor of the larger. Figure 6-22 shows the difference between collapsed and uncollapsed margins.

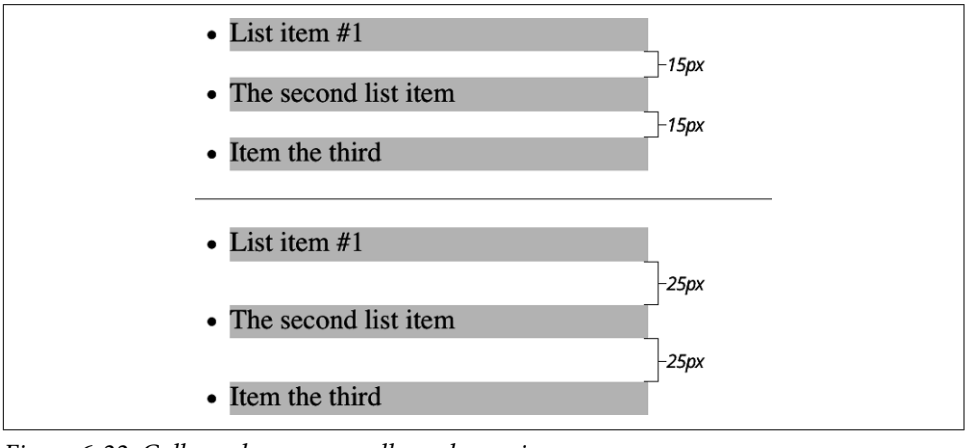


Figure 6-22. Collapsed versus uncollapsed margins

User agents will collapse block-adjacent margins as shown in the first list in Figure 6-22, so that 15-pixel spaces appear between each list item. The second list shows what happens if browsers don't collapse margins, resulting in 25-pixel spaces between list items.

Another word to use, if you don't like "collapse," is "overlap." Although the margins are not really overlapping, you can visualize what's happening by using the following analogy.

Imagine that each element, such as a paragraph, is a small piece of paper with the content of the element written on it. Around each piece of paper is a certain amount of clear plastic, which represents the margins. The first piece of paper (say an `<h1>` piece) is laid down on the canvas. The second (a paragraph) is laid below it along the block axis and then slid upward along that axis until the edge of one piece's plastic touches the edge of the other's paper. If the first piece of paper has half an inch of plastic along its block-end edge, and the second has a third of an inch along its block-start, then when they slide together, the first piece's block-end plastic will touch the block-start edge of the second piece of paper. The two are now done being placed on the canvas, and the plastic attached to the pieces is overlapping.

Collapsing also occurs where multiple margins meet, such as at the end of a list. Adding to the earlier example, let's assume the following rules apply:

```
ul {margin-block-end: 15px;}
li {margin-block-start: 10px; margin-block-end: 20px;}
h1 {margin-block-start: 28px;}
```

The last item in the list has a block-end margin of 20 pixels, the block-end margin of the <ul> is 15 pixels, and the block-start margin of a succeeding <h1> is 28 pixels. So once the margins have been collapsed, the distance between the end of the last <li> in the list and the beginning of the <h1> is 28 pixels, as shown in Figure 6-23.

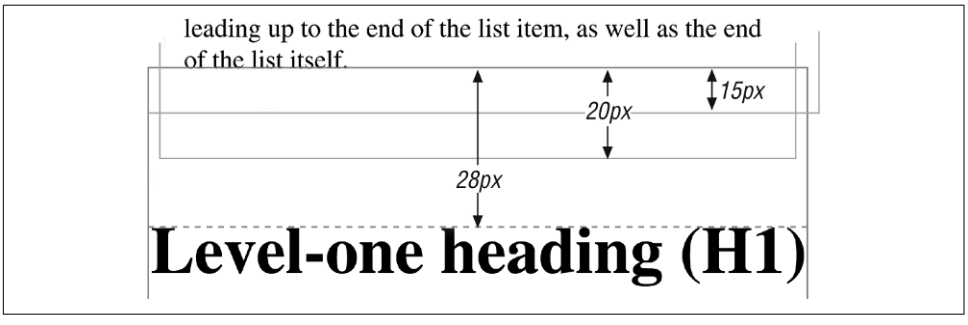


Figure 6-23. Collapsing in detail

If you add a border or padding to a containing block, this causes the margins of its child elements to be entirely contained within it. We can see this behavior in operation by adding a border to the <ul> element in the previous example:

```
ul {margin-block-end: 15px; border: 1px solid;}
li {margin-block-start: 10px; margin-block-end: 20px;}
h1 {margin-block-start: 28px;}
```

With this change, the block-end margin of the <li> element is now placed inside its parent element (the <ul>). Therefore, the only margin collapsing that takes place is between the <ul> and the <h1>, as illustrated in Figure 6-24.

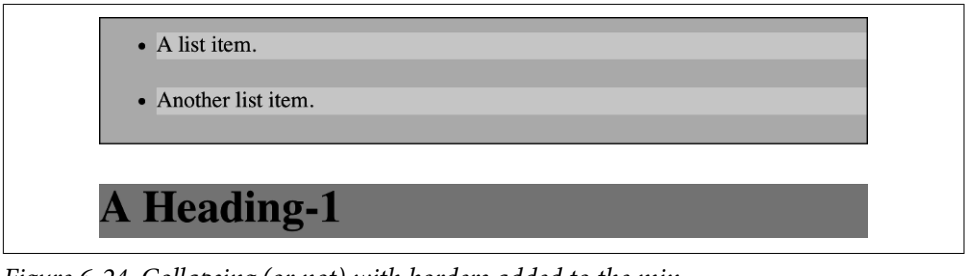


Figure 6-24. Collapsing (or not) with borders added to the mix

Negative margin collapsing is slightly different. When a negative margin participates in margin collapsing, the browser takes the absolute value of the negative margin and subtracts it from any adjacent positive margins. In other words, the negative length is added to the positive length(s), and the resulting value is the distance between the elements, even if that distance is a negative length. Figure 6-25 provides some concrete examples.

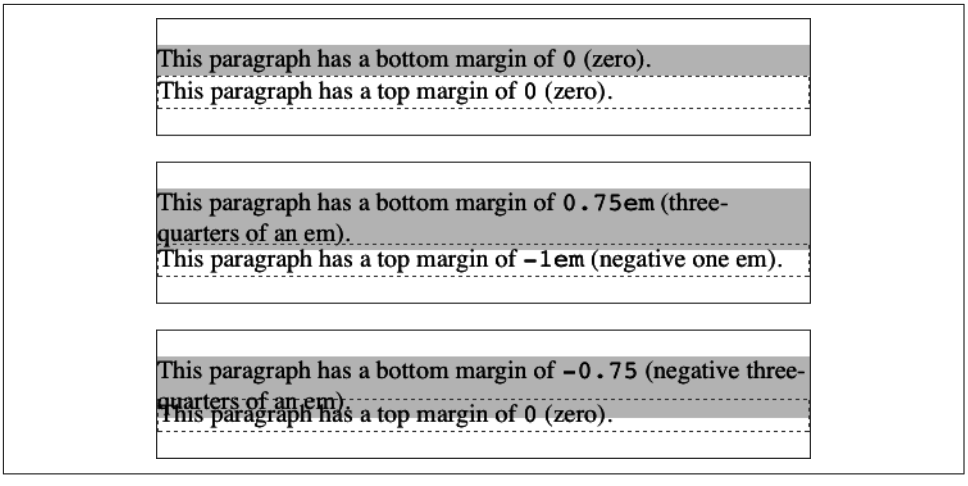


Figure 6-25. Examples of negative block-axis margins

Now let's consider an example where the margins of a list item, an unordered list, and a paragraph are all collapsed. In this case, the unordered list and paragraph are assigned negative margins:

```
li {margin-block-end: 20px;}
ul {margin-block-end: -15px;}
h1 {margin-block-start: -18px;}
```

The negative margin of the greatest magnitude (`-18px`) is added to the largest positive margin (`20px`), yielding `20px – 18px = 2px`. Thus, we have only 2 pixels between the block-end of the list item's content and the block-start of the `<h1>`'s content, as we can see in Figure 6-26.

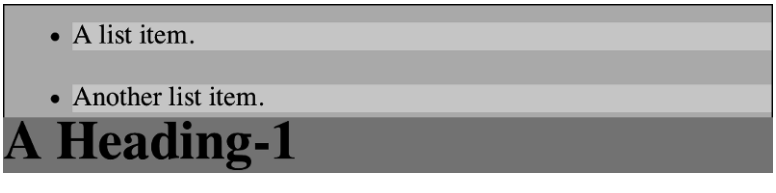


Figure 6-26. Collapsing margins and negative margins, in detail

When elements overlap each other because of negative margins, it's hard to tell which elements are on top of others. You may also have noticed that very few of the examples in this section use background colors. If they did, the background color of a following element might overwrite the content of a preceding element. This is expected behavior, since browsers usually render elements in order from beginning to end, so a normal-flow element that comes later in the document can generally be expected to overwrite an earlier element, assuming the two end up overlapping.

Inline-Axis Formatting

Laying out elements along the inline axis can be more complex than you'd think. Part of the complexity has to do with the default behavior of `box-sizing`. With the default value of `content-box`, the value given for `inline-size` affects the inline width of the content area, *not* the entire visible element box. Consider the following example, where the inline axis runs left to right:

```
<p style="inline-size: 200px;">wideness?</p>
```

This makes the paragraph's content area 200 pixels wide. If we give the element a background, this will be quite obvious. However, any padding, borders, or margins you specify are *added* to the width value. Suppose we do this:

```
<p style="inline-size: 200px; padding: 10px; margin: 20px;">wideness?</p>
```

The visible element box is now 220 pixels in inline size, since we've added 10 pixels of padding to every side of the content. The margins will now extend another 20 pixels to both inline sides for an overall element inline size of 260 pixels. This is illustrated in Figure 6-27.

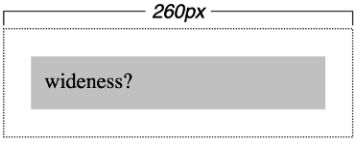


Figure 6-27. Additive padding and margin

If we change the styles to use `box-sizing: border-box`, the results will be different. In that case, the visible box will be 200 pixels wide along the inline axis with a content inline size of 180 pixels, and a total of 40 pixels of margin on the inline sides, giving an overall box inline size of 240 pixels, as illustrated in Figure 6-28.

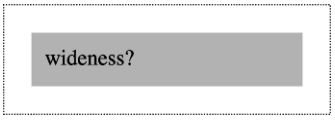


Figure 6-28. Subtractive padding

In either case, the CSS specification has a rule that says the sum of the inline components of a block box in the normal flow always equals the inline size of the containing block (which is why, as you'll see in just a bit, `margin: auto` centers content in the inline direction). Let's consider two paragraphs within a `<div>` whose margins have been set to `1em`, and whose `box-sizing` value is the default `content-box`. The content size (the value of `inline-size`) of each paragraph in this example, plus its inline-start and -end padding, borders, and margins, will always add up to the inline size of the `<div>`'s content area.

Let's say the inline size of the `<div>` is `30em`. That makes the sum total of the content size, padding, borders, and margins of each paragraph 30 em. In Figure 6-29, the "blank" space around the paragraphs is actually their margins. If the `<div>` had any padding, even more blank space would be present, but that isn't the case here.

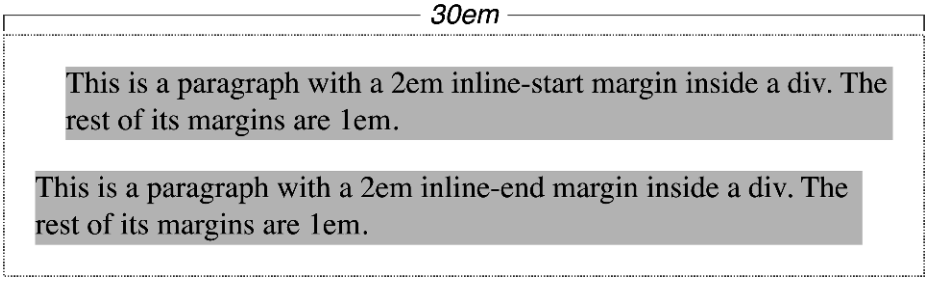


Figure 6-29. Element boxes are as wide as the inline width of their containing block

Inline-Axis Properties

The seven properties of inline formatting—`margin-inline-start`, `border-inline-start`, `padding-inline-start`, `inline-size`, `padding-inline-end`, `padding-inline-end`, and `padding-inline-end`—are diagrammed in Figure 6-30.

The values of these seven properties must add up to the inline size of the element's containing block, which is usually the value of `inline-size` for a block element's parent (since block-level elements nearly always have block-level elements for parents).

Of these seven properties, only three may be set to `auto`: the inline size of the element's content, and the inline-start and -end margins. The remaining properties must either be set to specific values or default to a width of 0. Figure 6-30 shows which parts of the box can take a value of `auto` and which cannot. (That said, CSS is forgiving: if any part that can't accept `auto` is erroneously set to `auto`, it will default to `0`.)

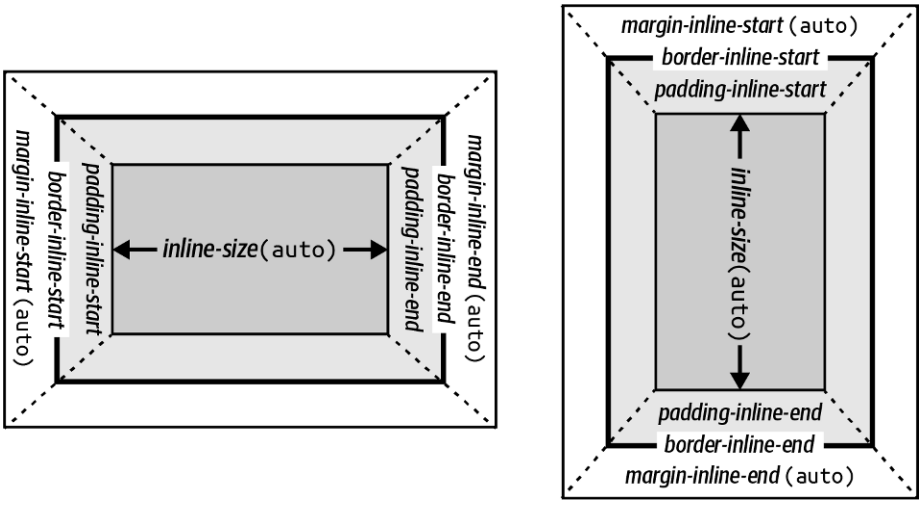


Figure 6-30. The seven properties of inline-axis formatting, and which can be set to `auto`

The `inline-size` property must either be set to `auto` or a nonnegative value. When you do use `auto` in inline-axis formatting, different effects can occur.

Using auto

In certain situations, it makes a lot of sense to explicitly set one or more of the inline margins and size to `auto`. By default, the two inline margins are set to `0`, and the inline size is set to `auto`. Let's explore how moving the `auto` around can have different effects, and why.

Only one auto

If you set one of `inline-size`, `margin-inline-start`, or `margin-inline-end` to a value of `auto`, and give the other two properties specific values, then the property that is set to `auto` is set to the length required to make the element box's overall inline size equal to the parent element's content inline size.

Let's say the sum of the seven inline-axis properties must equal 500 pixels, no padding or borders are set, the inline-end margin and inline size are set to `100px`, and the inline-start margin is set to `auto`. The inline-start margin will thus be 300 pixels wide:

```
div {inline-size: 500px;}
p {margin-inline-start: auto; margin-inline-end: 100px;
    inline-size: 100px;} /* inline-start margin evaluates to 300px */
```

In a sense, `auto` can be used to make up the difference between everything else and the required total. However, what if all three of these properties (both inline margins and the inline size) are set to `100px` and *none* of them are set to `auto`?

If all three properties are set to something other than `auto`—or, in CSS parlance, when these formatting properties have been *overconstrained*—then the margin at the inline end is *always* forced to be `auto`. This means that if both inline margins and the inline size are set to `100px`, the user agent will reset the inline-end margin to `auto`. The inline-end margin's width will then be set according to the rule that one `auto` value "fills in" the distance needed to make the element's overall inline size equal to that of its containing block's content inline size. Figure 6-31 shows the result of the following markup in left-to-right languages like English:

```
div {inline-size: 500px;}
p {margin-inline-start: 100px; margin-inline-end: 100px;
    inline-size: 100px;} /* inline-end margin forced to be 300px */
```

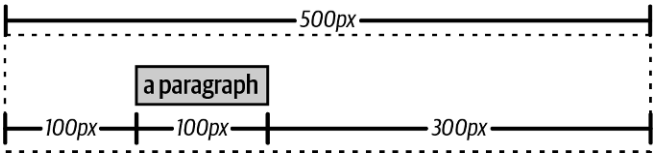


Figure 6-31. Overriding the inline-end margin's value

If both side margins are set explicitly, and `inline-size` is set to `auto`, then `inline-size` will be whatever value is needed to reach the required total (which is the content inline size of the parent element). The results of the following markup are shown in Figure 6-32:

```
p {margin-inline-start: 100px; margin-inline-end: 100px;
    inline-size: auto;}
```

This type of formatting is the most common, since it is equivalent to setting the margins and not declaring anything for the `inline-size`. The result of the following markup is exactly the same as that shown in Figure 6-32:

```
p {margin-inline-start: 100px; margin-inline-end: 100px;} /* same as before */
```

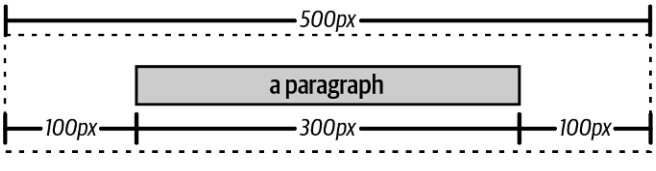


Figure 6-32. Automatic inline sizing

You might be wondering what happens if `box-sizing` is set to `padding-box`. In that case, all the same principles described here apply, which is why this section discussed only `inline-size` and the inline-side margins without introducing any padding or borders.

In other words, the handling of `inline-size: auto` in this section and the following sections is the same regardless of the value of `box-sizing`. The details of what gets placed where inside the `box-sizing`-defined box may vary, but the treatment of `auto` values does

not, because `box-sizing` determines what `inline-size` refers to, not how it behaves in relation to the margins.

More than one auto

Now let's see what happens when two of the three properties (`inline-size`, `margin-inline-start`, and `margin-inline-end`) are set to `auto`. If both margins are set to `auto` but the `inline-size` is set to a specific length, then both margins are set to equal lengths, thus centering the element within its parent along the inline axis. The following code creates this layout, illustrated in Figure 6-33:

```
div {inline-size: 500px;}
p {inline-size: 300px; margin-inline-start: auto; margin-inline-end: auto;}
  /* each margin is 100 pixels, because (500-300)/2 = 100 */
```

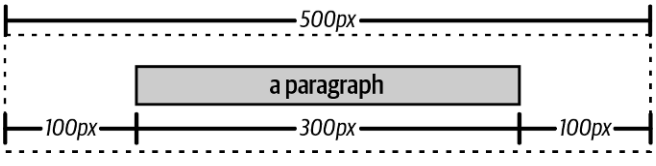


Figure 6-33. Setting an explicit inline size

Another way of sizing elements along the inline axis is to set one of the inline margins and `inline-size` to `auto`. In this case, the margin set to `auto` is reduced to 0:

```
div {inline-size: 500px;}
p {margin-inline-start: auto; margin-inline-end: 100px; inline-size: auto;}
  /* inline-start margin evaluates to 0; inline-size becomes 400px  */
```

The `inline-size` property is then set to the value necessary to make the element fill its containing block; in the preceding example, it would be 400 pixels, as shown in Figure 6-34.

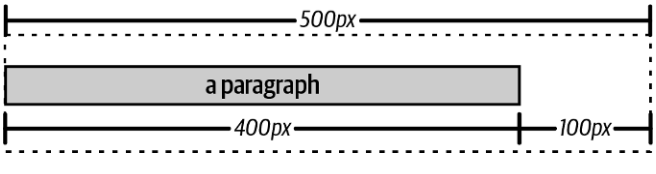


Figure 6-34. What happens when both `inline-size` and the inline-start margin are `auto`

Too many autos

Finally, what happens when all three properties are set to `auto`? The answer: both margins are set to 0, and the `inline-size` is made as wide as possible. This result is the same as the default situation, when no values are explicitly declared for margins or the inline size. In such a case, the margins default to 0 and `inline-size` defaults to `auto`.

Note that since inline margins do not collapse (unlike block margins, as discussed earlier), the padding, borders, and margins of a parent element can affect the inline layout of its children. The effect is indirect in that the margins (and so on) of an element can induce an offset for child elements. The results of the following markup are shown in Figure 6-35:

```
div {padding: 50px; background: silver;}
p {margin: 30px; padding: 0; background: white;}
```

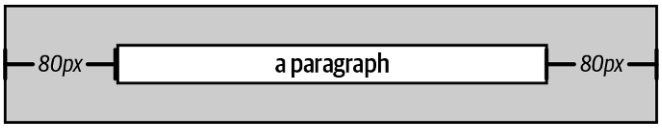


Figure 6-35. Offset is implicit in the parent's margins and padding

Negative Margins

As you've seen with block-axis margins, it's possible to set negative values for inline-axis margins. Setting negative inline margins can result in some interesting effects.

Remember that the total of the seven inline-axis properties always equals the inline size of the content area of the parent element. As long as all inline properties are 0 or greater, an element's inline size can never be greater than its parent's content area inline size. However, consider the following markup, depicted in Figure 6-36:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: 10px; margin-inline-end: -50px;
    inline-size: auto;}
```

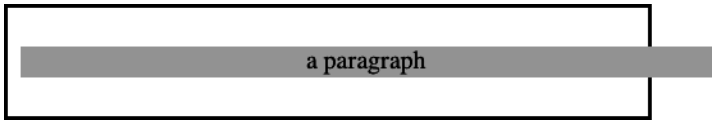


Figure 6-36. Wider children through negative margins

Yes indeed, the child element is wider than its parent along the inline axis! This is mathematically correct. Let's solve for inline size:

$$10\text{ px} + 0 + 0 + 540\text{ px} + 0 + 0 - 50\text{ px} = 500\text{ px}$$

The `540px` is the evaluation of `inline-size: auto`, which is the number needed to balance out the rest of the values in the equation. Even though it leads to a child element sticking out of its parent, it all works because the values of the seven properties add up to the required total.

Now, let's add some borders to the mix:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: 10px; margin-inline-end: -50px;
    inline-size: auto; border: 3px solid gray;}
```

The resulting change will be a reduction in the evaluated width of `inline-size`:

$$10\text{ px} + 3\text{ px} + 0 + 534\text{ px} + 0 + 3\text{ px} - 50\text{ px} = 500\text{ px}$$

Or, we can rearrange the equation to solve for the content size instead of for the width of the parent:

$$500\text{ px} - 10\text{ px} - 3\text{ px} - 3\text{ px} + 50\text{ px} = 534\text{ px}$$

If we were to introduce padding, the value of `inline-size` would drop even more (assuming `box-sizing: content-box`).

Conversely, it's possible to have `auto` inline-end margins evaluate to negative amounts. If the values of other properties force the inline-end margin to be negative in order to satisfy the requirement that elements be no wider than their containing block, that's what will happen. Consider the following:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: 10px; margin-inline-end: auto;
    inline-size: 600px; border: 3px solid gray;}
```

The equation works out like this:

$$500\text{ px} - 10\text{ px} - 600\text{ px} - 3\text{ px} - 3\text{ px} = -116\text{ px}$$

In this case, the inline-end margin evaluates to `-116px`. No matter what explicit value it's given in the CSS, this margin will still be forced to `-116px` because of the rule stating that when an element's dimensions are overconstrained, the inline-end margin is reset to whatever is needed to make the numbers work out correctly.

Let's consider another example, illustrated in Figure 6-37, in which the inline-start margin is set to be negative:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: -50px; margin-inline-end: 10px;
    inline-size: auto; border: 3px solid gray;}
```

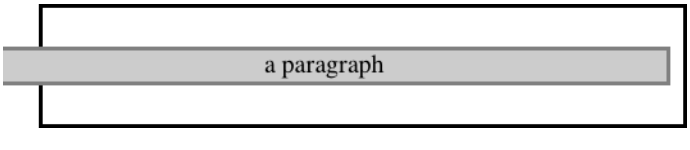


Figure 6-37. Setting a negative inline-start margin

With a negative inline-start margin, the paragraph not only spills beyond the borders of the `<div>`, but also spills beyond the edge of the browser window itself!

Remember: padding, borders, and content widths (and heights) can never be negative. Only margins can be less than 0.

Percentages

When it comes to percentage values for the inline size, padding, and margins, the same basic rules we discussed in previous sections apply. It doesn't really matter whether the values are declared with lengths or percentages.

Percentages can be very useful. Suppose we want an element's content to be two-thirds the inline size of its containing block, the padding sides to be 5% each, the inline-start margin to be 5%, and the inline-end margin to take up the slack. That would be written something like this:

```
<p style="inline-size: 67%;
     padding-inline-end: 5%; padding-inline-start: 5%;
     margin-inline-end: auto; margin-inline-start: 5%;">
     playing percentages</p>
```

The inline-end margin would evaluate to 18% (100% – 67% – 5% – 5% – 5%) of the width of the containing block.

Mixing percentages and length units can be tricky, however. Consider the following example:

```
<p style="inline-size: 67%; padding-inline-end: 2em; padding-inline-start: 2em;
     margin-inline-end: auto; margin-inline-start: 5em;">mixed lengths</p>
```

In this case, the element's box can be defined like this:

5 em + 0 + 2 em + 67% + 2 em + 0 + auto = containing block width

In order for the inline-end margin's inline size to evaluate to 0, the element's containing block must be 27.272727 em wide (with the content area of the element being 18.272727 em wide) along the inline axis. Any wider than that, and the inline-end margin will evaluate to a positive value. Any narrower, and the inline-end margin will be a negative value.

The situation gets even more complicated if we start mixing length-value unity types, like this:

```
<p style="inline-size: 67%;
     padding-inline-end: 15px; padding-inline-start: 10px;
     margin-inline-end: auto; margin-inline-start: 5em;">more mixed lengths</p>
```

And, just to make things more complex, borders cannot accept percentage values, only length values. The bottom line is that it isn't really possible to create a fully flexible element based solely on percentages unless you're willing to avoid using borders or use approaches such as flexible box layout. That said, if you do need to mix percentages and

length units, using the `calc()` and `minmax()` value functions can be a life changer, or at least a layout changer.

Replaced Elements

So far, we've been dealing with the inline-axis formatting of nonreplaced block boxes in the normal flow of text. Replaced elements are a bit simpler to manage. All of the rules given for nonreplaced blocks hold true, with one exception: if `inline-size` is `auto`, the `inline-size` of the element is the content's intrinsic width. (*Intrinsic* means the original size—the size of the element by default when no external factors are applied to it.) The image in the following example will be 20 pixels wide because that's the width of the original image:

```
<img src="smile.svg" style="display: block; inline-size: auto; margin: 0;"
   alt="smile">
```

If the actual image were 100 pixels wide instead, the element (and thus the image) would be laid out as 100 pixels wide.

We can override this rule by assigning a specific value to `inline-size`. Suppose we modify the previous example to show the same image three times, each with a different width value:

```
<img src="smile.svg" style="display: block; inline-size: 25px; margin: 0;"
   alt="small smile" role="img">
<img src="smile.svg" style="display: block; inline-size: 50px; margin: 0;"
   alt="medium smile" role="img">
<img src="smile.svg" style="display: block; inline-size: 100px; margin: 0;"
   alt="large smile" role="img">
```

Figure 6-38 illustrates the result.



Figure 6-38. Changing replaced element inline sizes

Note that the block size of the elements also increases. When a replaced element's `inline-size` is changed from its intrinsic width, the value of `block-size` is scaled to match, maintaining the object's initial aspect ratio, unless `block-size` has been set to an explicit value of its own. The reverse is also true: if `block-size` is set, but `inline-size` is left as `auto`, then the inline size is scaled proportionately to the change in block size.

List Items

List items have a few special rules of their own. They are typically preceded by a marker, such as a round bullet mark or a number.

The marker attached to a list item element can be either outside the content of the list item or treated as an inline marker at the beginning of the content, depending on the value of the property `list-style-position`, as illustrated in Figure 6-39.

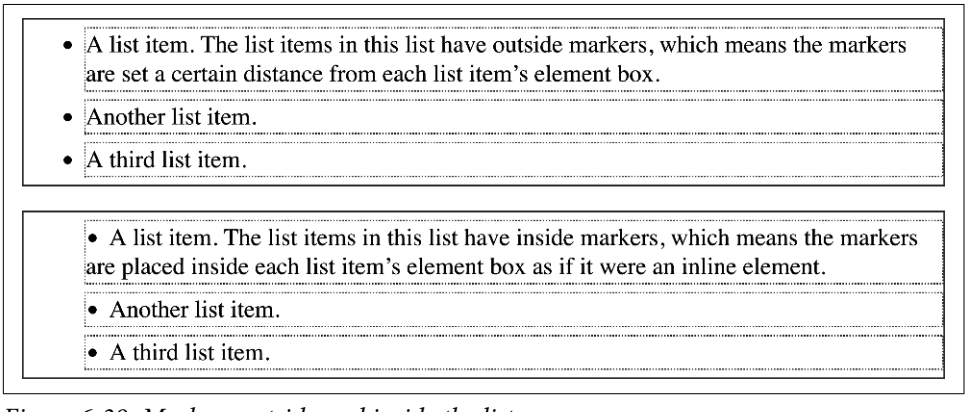


Figure 6-39. Markers outside and inside the list

If the marker stays outside the content, it is placed a specified distance from the inline-start content edge of the content. No matter how the list's styles are altered, the marker stays the same distance from the content edge.

Remember that list-item boxes define containing blocks for their descendant boxes, just like regular block boxes.



List markers are discussed in more detail, including how to create and style them using the `::marker` pseudo-element, in Chapter 16.

Box Sizing with Aspect Ratios

Sometimes you'll want to size an element by its *aspect ratio*, which means its block and inline sizes exist in a specific ratio. Old TVs used to have a 4:3 width-to-height ratio, for example; HD video resolutions have a 16:9 aspect ratio. You might want to force elements to be square while still letting their sizes flex. In these cases, the `aspect-ratio` property can help.

aspect-ratio

Values	`auto` || <*ratio*>
Initial value	`auto`
Applies to	All elements except inline boxes and internal table and Ruby boxes
Computed value	If <*ratio*>, a pair of numbers; otherwise, `auto`
Inherited	No
Animatable	Yes

Let's say we know we'll have a bunch of elements, and we don't know how wide or tall each will be, but we want them all to be squares. First, pick an axis you want to size on. We'll use `height` here. Make sure the other axis is autosized, and set an aspect ratio:

```
.gallery div {width: auto; aspect-ratio: 1/1;}
```

Figure 6-40 shows the same set of HTML, both with and without the previous CSS applied.

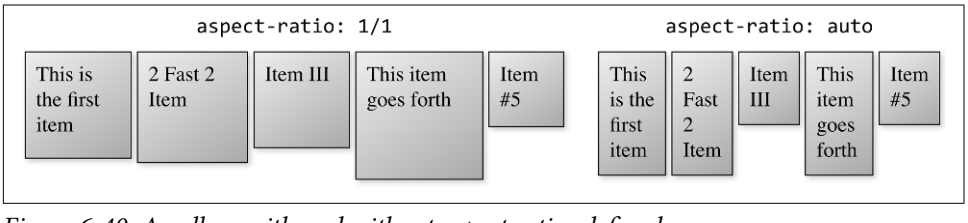


Figure 6-40. A gallery with and without aspect ratios defined

The ratio is maintained over the distances defined by `box-sizing` (see "Altering Box Sizing" on page 189), so given the following CSS, the result will be an element whose outer border distances are in an exact 2:1 ratio:

```
.cards div {height: auto; box-sizing: border-box; aspect-ratio: 2/1;}
```

The default value, `auto`, means that boxes that have an intrinsic aspect ratio—boxes generated by images, for example—will use that aspect ratio. For elements that don't have an intrinsic aspect ratio, such as most HTML elements like `<div>`, `<p>`, and so on, the axis sizes of the box will be determined by the content.

Inline Formatting

Inline formatting isn't as simple as formatting block-level elements, which just generates block boxes and usually doesn't allow anything to coexist with them. By contrast, look *inside* a block-level element, such as a paragraph. You may well ask, how was the size and

wrapping of each line determined? What controls the lines' arrangement? How can I affect it?

Line Layout

To understand how lines are generated, first consider an element containing one very long line of text, as shown in Figure 6-41. Note that we've put a border around the line by wrapping the entire line in a `<span>` element and then assigning it a border style:

```
span {border: 1px dashed black;}
```

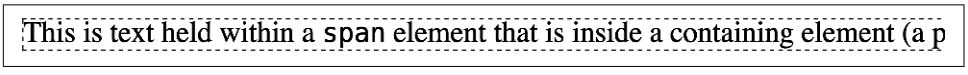


Figure 6-41. A single-line inline element

Figure 6-41 shows the simplest case of an inline element contained by a block-level element.

To get from this simplified state to something more familiar, all we have to do is determine how wide (along the inline axis) the element should be, and then break up the line so that the resulting pieces will fit into the content inline size of the element. Therefore, we arrive at the state shown in Figure 6-42.

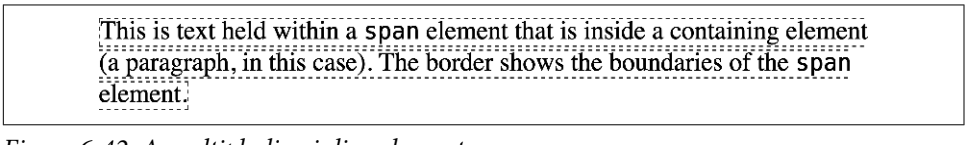


Figure 6-42. A multiple-line inline element

Nothing has really changed. All we did was take the single line and break it into pieces, and then stack those pieces one after the other along the direction of the block flow.

In Figure 6-42, the borders for each line of text also happen to coincide with the top and bottom of each line. This is true only because no padding has been set for the inline text. Notice that the borders overlap each other slightly; for example, the bottom border of the first line is just below the top border of the second line. This is because the border is drawn on the next pixel to the *outside* of each line. Since the lines are touching each other, their borders overlap as shown in Figure 6-42.



For simplicity, we use terms such as *top* and *bottom* when talking about the edges of line boxes. In this context, the top of a line box is the one closest to the block-start, and the bottom of a line box is the one closest to the block-end. Similarly, *tall* and *short* refer to the size of line boxes along the block axis.

If we alter the span styles to have a background color, the actual placement of the lines becomes clearer. Consider Figure 6-43, which shows four paragraphs in each of two writing modes, and the effects of different values of `text-align` (see Chapter 15), by each paragraph having the backgrounds of its lines filled in.

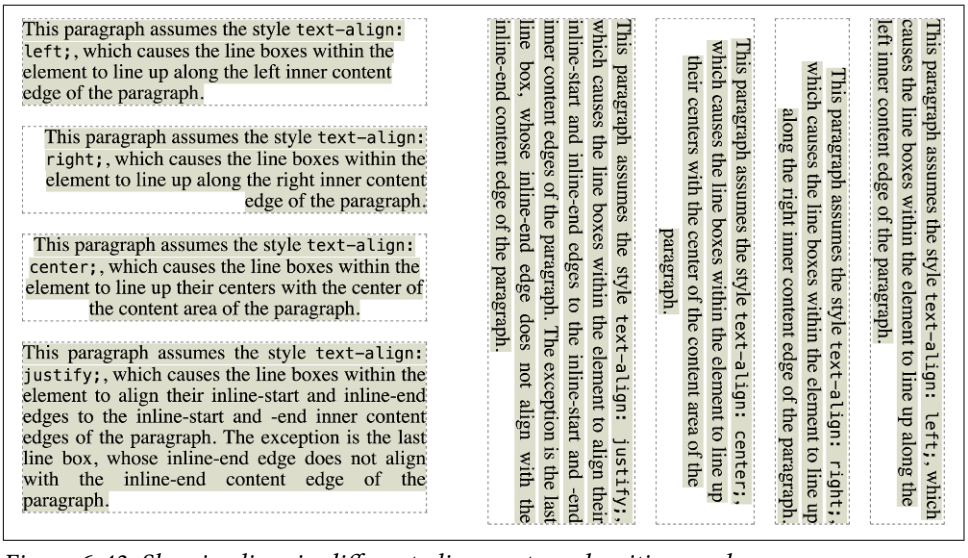


Figure 6-43. Showing lines in different alignments and writing modes

As Figure 6-43 shows, not every line reaches to the edge of its parent paragraph's content area, which has been denoted with a dashed gray border. For the left-aligned paragraph, the lines are all pushed flush against the left content edge of the paragraph, and the end of each line happens wherever the line is broken. The reverse is true for the right-aligned paragraph. For the centered paragraph, the centers of the lines are aligned with the center of the paragraph.

In the last case, where the value of `text-align` is `justify`, each line (except the last) is forced to be as wide as the paragraph's content area so that the line's edges touch the content edges of the paragraph. The difference between the natural length of the line and the width of the paragraph's content area is made up by altering the spacing between letters and words in each line. Therefore, the value of `word-spacing` can be overridden when the text is justified. (The value of `letter-spacing` cannot be overridden if it is a length value.)

That pretty well covers how lines are generated in the simplest cases. As you're about to see, however, the inline formatting model is far from simple.

Basic Terms and Concepts

Before we go any further, let's review some terms of inline layout, which will be crucial in navigating the following sections:

Anonymous text
: Any string of characters that is not contained within an inline element. Thus, in the markup `<p> I'm <em>so</em> happy!</p>`, the sequences " I'm " and " happy!" are anonymous text. Note that the spaces are part of the text, since a space is a character like any other.

Em box
: This is defined in the given font, otherwise known as the character box. Actual glyphs can be taller or shorter than their em boxes. In CSS, the value of `font-size` determines the height of each em box.

Content area
: In nonreplaced elements, the content area can be one of two things, and the CSS specification allows user agents to choose which one. The content area can be the box described by the em boxes of every character in the element, strung together; or it can be the box described by the character glyphs in the element. In this book, we use the em box definition for simplicity, and that's what is used by most browsers. In replaced elements, the content area is the intrinsic height of the element plus any margins, borders, or padding.

Leading
: Leading (pronounced "led-ing") is the difference between the values of `font-size` and `line-height`. This difference is divided in half, with one half applied to the top and one half to the bottom of the content area. These additions to the content area are called, perhaps unsurprisingly, *half-leading*. Leading is applied only to nonreplaced elements.

Inline box
: This is the box described by the addition of the leading to the content area. For nonreplaced elements, the height of the inline box of an element will be exactly equal to the value of the `line-height` property. For replaced elements, the height of the inline box of an element will be exactly equal to the content area, since leading is not applied to replaced elements.

Line box
: This is the shortest box that bounds the highest and lowest points of the inline boxes that are found in the line. In other words, the top edge of the line box is placed along the top of the highest inline-box top, and the bottom of the line box is placed along the bottom of the lowest inline-box bottom. Remember that "top" and "bottom" are considered with respect to the block flow direction.

CSS also contains a set of behaviors and useful concepts that fall outside of the preceding list of terms and definitions:

- The content area of an inline box is analogous to the content box of a block box.
- The background of an inline element is applied to the content area plus any padding.
- Any border on an inline element surrounds the content area plus any padding.
- Padding, borders, and margins on nonreplaced inline elements have no vertical effect on the inline elements or the boxes they generate; they do *not* affect the height of an element's inline box (and thus the line box that contains the element).
- Margins and borders on replaced elements *do* affect the height of the inline box for that element and, by implication, the height of the line box for the line that contains the element.

One more thing to note: inline boxes are vertically aligned within the line according to their values for the property `vertical-align` (see Chapter 15).

Before moving on, let's look at a step-by-step process for constructing a line box, which you can use to see how the various pieces of a line fit together to determine its height. Determine the height of the inline box for each element in the line by following these steps:

1. Find the values of `font-size` and `line-height` for each inline nonreplaced element and text that is not part of a descendant inline element and combine them. This is done by subtracting the `font-size` from the `line-height`, which yields the leading for the box. The leading is split in half and applied to the top and bottom of each em box.
2. Find the value of `height`, along with the values for the margins, padding, and borders along the block-start and block-end edges of each replaced element, and add them together.
3. Figure out, for each content area, how much of it is above the baseline for the overall line and how much of it is below the baseline. This is not an easy task: you must know the position of the baseline for each element and piece of anonymous text and the baseline of the line itself, and then line them all up. In addition, the block-end edge of a replaced element sits on the baseline for the overall line.
4. Determine the vertical offset of any elements that have been given a value for `vertical-align`. This will tell you how far up or down that element's inline box will be moved along the block axis, and that will change how much of the element is above or below the baseline.
5. Now that you know where all of the inline boxes have come to rest, calculate the final line box height. To do so, just add the distance between the baseline and the highest inline-box top to the distance between the baseline and the lowest inline-box bottom.

Let us consider the whole process in detail, which is the key to intelligently styling inline content.

Line Heights

First, know that all elements have a `line-height`, whether it's explicitly declared or not. This value greatly influences the way inline elements are displayed, so let's give it due attention.

A line's height (or the height of a line box) is determined by the height of its constituent elements and other content, such as text. It's important to understand that `line-height` affects inline elements and other inline content, *not* block-level elements—at least, not directly. We can set a `line-height` value for a block-level element, but the value will have a visual impact only as it's applied to inline content within that block-level element. Consider the following empty paragraph, for example:

```
<p style="line-height: 0.25em;"></p>
```

Without content, the paragraph won't have anything to display, so we won't see anything. The fact that this paragraph has a `line-height` of any value—be it `0.25em` or `25in`—makes no difference without some content to create a line box.

We can set a `line-height` value for a block-level element and have that apply to all of the content within the block, whether it's contained in an inline element or anonymous text. In a certain sense, then, each line of text contained within a block-level element is its own inline element, whether or not it's surrounded by tags. If you like, picture a fictional tag sequence like this:

```
<p>
<line>This is a paragraph with a number of</line>
<line>lines of text that make up the</line>
<line>contents.</line>
</p>
```

Even though the `line` tags don't actually exist, the paragraph behaves as if they did, and each line of text "inherits" styles from the paragraph. You bother to create `line-height` rules for block-level elements only so you don't have to explicitly declare a `line-height` for all of their inline elements, fictional or otherwise.

The fictional `line` element clarifies the behavior that results from setting `line-height` on a block-level element. According to the CSS specification, declaring `line-height` on a block-level element sets a *minimum* line-box height for the content of that block-level element. Declaring `p.spacious {line-height: 24pt;}` means that the *minimum* height for each line box is 24 points. Technically, content can inherit this line height only if an inline element does so. Most text isn't contained by an inline element. If you pretend that each line is contained by the fictional `line` element, the model works out very nicely.

Inline Nonreplaced Elements

Building on our formatting knowledge, let's move on to the construction of lines that contain only nonreplaced elements (or anonymous text). Then you'll be in a good position to understand the differences between nonreplaced and replaced elements in inline layout.



In this section, we use *top* and *bottom* to label where half-leading is placed and how line boxes are placed together. Always remember that these terms are in relation to the direction of block flow: the *top edge* of an inline box is the one closest to the block-start edge, and the *bottom edge* of an inline box is closest to its block-end edge. Similarly, *height* means the distance along the inline box's block axis, and *width* is the distance along its inline axis.

Building the Boxes

First, for an inline nonreplaced element or piece of anonymous text, the value of `font-size` determines the height of the content area. If an inline element has a `font-size` of `15px`, the content area's height is 15 pixels because all of the em boxes in the element are 15 pixels tall, as illustrated in Figure 6-44.

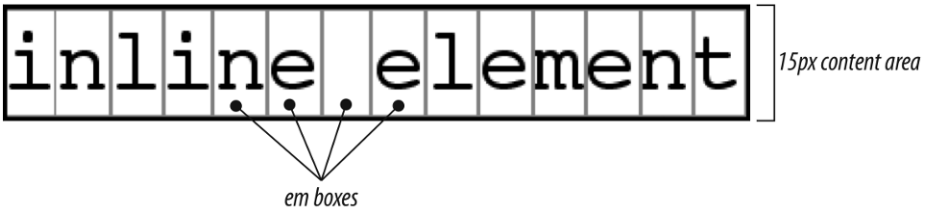


Figure 6-44. Em boxes determine content area height

The next thing to consider is the value of `line-height` for the element, and the difference between it and the value of `font-size`. If an inline nonreplaced element has a `font-size` of `15px` and a `line-height` of `21px`, the difference is 6 pixels. The user agent splits the 6 pixels in half and applies half (3 pixels) to the top and half (3 pixels) to the bottom of the content area, which yields the inline box. Figure 6-45 illustrates this process.

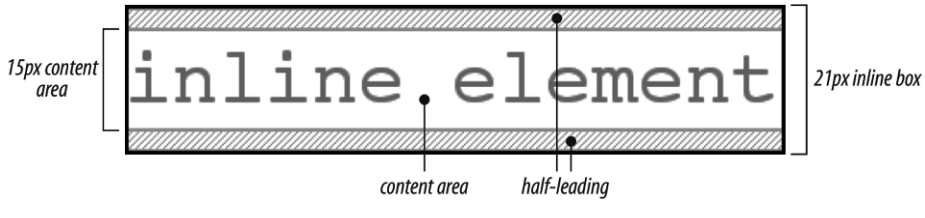


Figure 6-45. Content area plus leading equals inline box

Now, let's break stuff so we can better understand how line height works. Assume the following is true:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px;">strongly emphasized</strong>
and that is<br>
```

```
larger than the surrounding text.
</p>
```

In this example, most of the text has a `font-size` of `12px`, while the text in one inline nonreplaced element has a size of `24px`. However, *all* of the text has a `line-height` of `12px` since `line-height` is an inherited property. Therefore, the `<strong>` element's `line-height` is also `12px`.

Thus, for each piece of text where both `font-size` and `line-height` are `12px`, the content height does not change (since the difference between `12px` and `12px` is 0), so the inline box is 12 pixels high. For the strong text, however, the difference between `line-height` and `font-size` is `-12px`. This is divided in half to determine the half-leading (`-6px`), and the half-leading is added to both the top and bottom of the content height to arrive at an inline box. Since we're adding a negative number in both cases, the inline box ends up being 12 pixels tall. The 12-pixel inline box is centered vertically within the 24-pixel content height of the element, so the inline box is smaller than the content area.

So far, it sounds like we've done the same thing to each bit of text, and that all the inline boxes are the same size, but that's not quite true. The inline boxes in the second line, although they're the same size, don't line up because the text is all baseline aligned (see Figure 6-46), a concept we'll discuss later in the chapter.

Since inline boxes determine the height of the overall line box, their placement with respect to one another is critical. The line box is defined as the distance from the top of the highest inline box in the line to the bottom of the lowest inline box, and the top of each line box butts up against the bottom of the line box for the preceding line.

In Figure 6-46, three boxes are being laid out for a single line of text: the two anonymous text boxes to either side of the `<strong>` element, and the `<strong>` element itself. Because the enclosing paragraph has a `line-height` of `12px`, each of the three boxes will have a 12-pixel-tall inline box. These inline boxes are centered within the content area of each box. The boxes then have their baselines lined up, so the text all shares a common baseline.

But because of where the inline boxes fall with respect to those baselines, the inline box of the `<strong>` element is a little bit higher than the inline boxes of the anonymous text boxes. Thus, the distance from the top of the `<strong>`'s inline box to the bottoms of the anonymous inline boxes is more than 12 pixels, while the visible content of the line isn't completely contained within the line box.



Figure 6-46. Inline boxes within a line

After all that, the middle line of text is placed between two other lines of text, as depicted in Figure 6-47. The bottom edge of the first line of text is placed against the top edge of

the line of text we saw in Figure 6-46. Similarly, the top edge of the third line of text is placed against the bottom edge of the middle line of text. Because the middle line of text has a slightly taller line box, the result is that the lines of text look irregular, because the distances between the three baselines are not consistent.

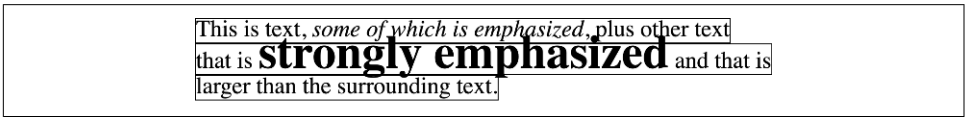


Figure 6-47. Line boxes within a paragraph



In just a bit, we'll explore ways to cope with this irregular separation of baselines and methods for achieving consistent baseline spacing. (Spoiler: Unitless values for the win!)

Setting Vertical Alignment

If we change the vertical alignment of the inline boxes, the same height determination principles apply. Suppose that we give the `<strong>` element a vertical alignment of `4px`:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px; vertical-align: 4px;">strongly
emphasized</strong>  and that is<br>
larger than the surrounding text.
</p>
```

That small change raises the `<strong>` element 4 pixels, which pushes up both its content area and its inline box. Because the `<strong>` element's inline-box top was already the highest in the line, this change in vertical alignment also pushes the top of the line box upward by 4 pixels, as shown in Figure 6-48.

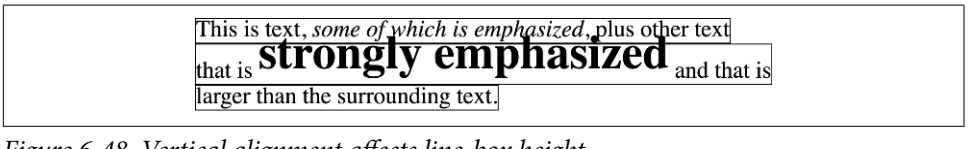


Figure 6-48. Vertical alignment affects line-box height



A formal definition for `vertical-align` can be found in Chapter 15.

Let's consider another situation. Here, we have another inline element in the same line as the strong text, and its alignment is other than the baseline:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>,<br>
plus other text that is <strong style="font-size: 24px; vertical-align: 4px;">
strong</strong> and <span style="vertical-align: top;">tall</span> and is<br>
larger than the surrounding text.
</p>
```

Now we have the same result as in our earlier example, where the middle line box is taller than the other line boxes. However, notice how the "tall" text is aligned in Figure 6-49.

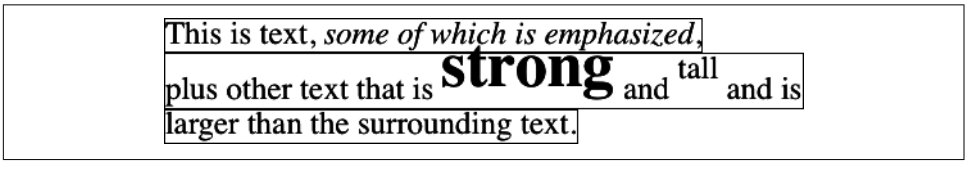


Figure 6-49. Aligning an inline element to the line box

In this case, the top of the "tall" text's inline box is aligned with the top of the line box. Since the "tall" text has equal values for `font-size` and `line-height`, the content height and inline box are the same. However, consider this:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>,<br>
plus other text that is <strong style="font-size: 24px; vertical-align: 4px;">
strong</strong> and <span style="vertical-align: top; line-height: 2px;">
tall</span> and is<br> larger than the surrounding text.
</p>
```

Since the `line-height` for the "tall" text is less than its `font-size`, the inline box for that element is smaller than its content area. This tiny fact changes the placement of the text itself, because the top of its inline box must be aligned with the top of the line box for its line. Thus, we get the result shown in Figure 6-50.

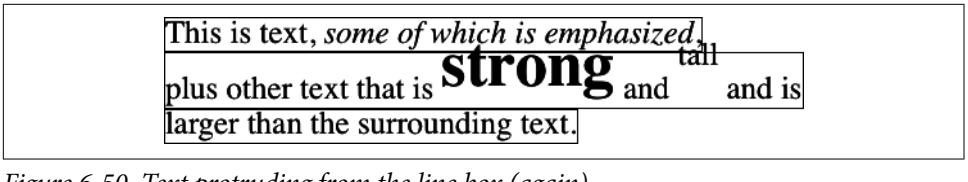


Figure 6-50. Text protruding from the line box (again)

In relation to the terms we've been using in this chapter, the effects of the assorted keyword values of `vertical-align` are as follows:

`top`
: Aligns the top (block-start edge) of the element's inline box with the top of the containing line box.

`bottom`
: Aligns the bottom (block-end edge) of the element's inline box with the bottom of the containing line box.

`text-top`
: Aligns the top (block-start edge) of the element's inline box with the top of the parent's content area.

`text-bottom`
: Aligns the bottom (block-end edge) of the element's inline box with the bottom of the parent's content area.

`middle`
: Aligns the vertical midpoint of the element's inline box with `0.5ex` above the baseline of the parent.

`super`
: Moves the content area and inline box of the element upward along the block axis. The distance is not specified and may vary by user agent.

`sub`
: The same as `super`, except the element is moved downward along the block axis instead of upward.

<percentage>
: Shifts the element up or down the block axis by the distance defined by taking the declared percentage of the element's value for `line-height`.

Managing the Line Height

In previous sections, you saw that changing the `line-height` of an inline element can cause text from one line to overlap another. In each case, though, the changes were made to individual elements. So how can you affect the `line-height` of elements in a more general way in order to keep content from overlapping?

One way to do this is to use the `em` unit in conjunction with an element whose `font-size` has changed. For example:

```
p {line-height: 1em;}
strong {font-size: 250%; line-height: 1em;}

<p>
Not only does this paragraph have "normal" text, but it also<br>
contains a line in which <strong>some big text</strong> is found.<br>
This large text helps illustrate our point.
</p>
```

By setting a `line-height` for the `<strong>` element, we increase the overall height of the line box, providing enough room to display the `<strong>` element without overlapping

any other text and without changing the `line-height` of all lines in the paragraph. We use a value of `1em` so that the `line-height` for the `<strong>` element will be set to the same size as `<strong>`'s `font-size`. Remember, `line-height` is set in relation to the `font-size` of the element itself, not the parent element. Figure 6-51 shows the results.



Figure 6-51. Assigning the `line-height` property to inline elements

Make sure you really understand the previous sections, because readable formatting of the text gets trickier when we try to add borders. Let's say we want to put 5-pixel borders around any hyperlink:

```
a:any-link {border: 5px solid blue;}
```

If we don't set a large-enough `line-height` to accommodate the border, it will be in danger of overwriting other lines. We could increase the size of the inline box for hyperlinks by using `line-height`, as we did for the `<strong>` element in the earlier example; in this case, we'd just need to make the value of `line-height` 10 pixels larger than the value of `font-size` for those links. However, that will be difficult if we don't actually know the size of the font in pixels.

Another solution is to increase the `line-height` of the paragraph. This will affect every line in the entire element, not just the line in which the bordered hyperlink appears:

```
p {line-height: 1.8em;}
a:link {border: 5px solid blue;}
```

Because extra space is added above and below each line, the border around the hyperlink doesn't impinge on any other line, as shown in Figure 6-52.

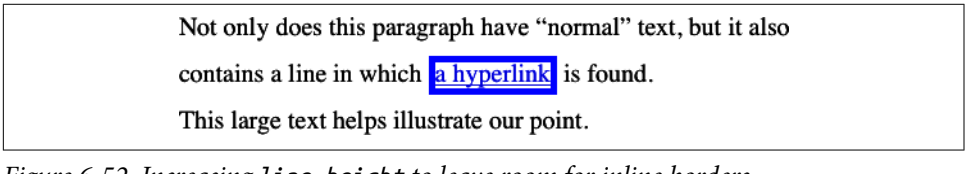


Figure 6-52. Increasing `line-height` to leave room for inline borders

This approach works because all of the text is the same size. If the line contained other elements that changed the height of the line box, our border situation might also change. Consider the following:

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
strong {font-size: 150%; line-height: 1.5em;}
```

Given these rules, the height of the inline box of a `<strong>` element within a paragraph will be 31.5 pixels (14 × 1.5 × 1.5), and that will also be the height of the line box. To keep baseline spacing consistent, we must make the `<p>` element's `line-height` equal to or greater than `32px`.

Understanding baselines and line heights

The actual height of each line box depends on the way its component elements line up with one another. This alignment tends to depend very much on where the baseline falls within each element (or piece of anonymous text) because that location determines how the inline boxes are arranged vertically.

Consistent baseline spacing tends to be more of an art than a science. If you declare all of your font sizes and line heights by using a single unit, such as ems, you have a good chance of consistent baseline spacing. If you mix units, however, that feat becomes a great deal more difficult, if not impossible.

As of late 2022, proposals call for properties that would let authors enforce consistent baseline spacing regardless of the inline content, which would greatly simplify certain aspects of online typography. None of these proposed properties have been implemented, which makes their adoption a distant hope at best.

Scaling line heights

The best way to set `line-height`, as it turns out, is to use a raw number as the value. This method is best because the number becomes the *scaling factor*, and that factor is an inherited, not a computed, value. Let's say we want the `line-height` of all elements in a document to be one and a half times their `font-size`. We would declare the following:

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

This scaling factor of 1.5 is passed down from element to element, and, at each level, the factor is used as a multiplier of the `font-size` of each element. Therefore, the following markup would be displayed as shown in Figure 6-53:

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
strong {font-size: 200%;}

<p>This paragraph has a line-height of 1.5 times its font-size. In addition,
any elements within it <small>such as this small element</small> also have
line-heights 1.5 times their font-size...and that includes <strong>this big
element right here</strong>. By using a scaling factor, line-heights scale
to match the font-size of any element.</p>
```

In this example, the line height for the `<small>` element turns out to be 15 pixels, and for the `<strong>` element, it's 45 pixels. If we don't want our big `<strong>` text to generate too much extra leading, we can give it its own `line-height` value, which will override the inherited scaling factor:

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
strong {font-size: 200%; line-height: 1em;}
```

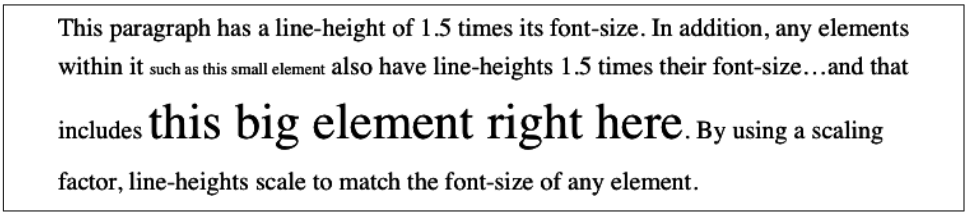


Figure 6-53. Using a scaling factor for `line-height`

Adding Box Properties to Nonreplaced Elements

As you may recall from previous discussions, while padding, margins, and borders may all be applied to inline nonreplaced elements, these properties have no impact on the height of the inline element's line box.

The border edge of inline elements is controlled by `font-size`, not `line-height`. In other words, if a `<span>` element has a `font-size` of `12px` and a `line-height` of `36px`, its content area is `12px` high, and the border will surround that content area.

Alternatively, we can assign padding to the inline element, which will push the borders away from the text itself:

```
span {padding: 4px;}
```

This padding does not alter the actual shape of the content height, and so it will not affect the height of the inline box for this element. Similarly, adding borders to an inline element will not affect the way line boxes are generated and laid out, as illustrated in Figure 6-54 (both with and without the 4-pixel padding).

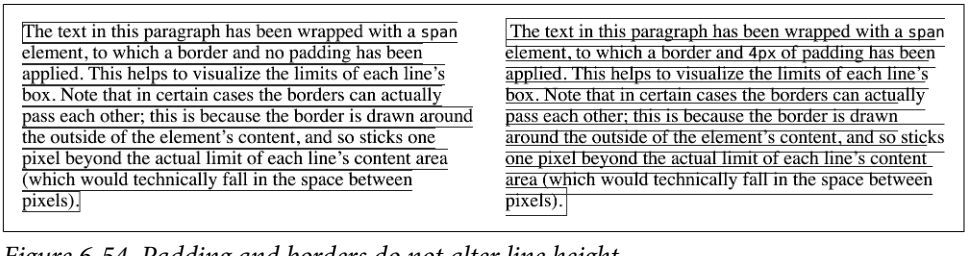


Figure 6-54. Padding and borders do not alter line height

As for margins, they do not, practically speaking, apply to the block edges of an inline nonreplaced element, as they don't affect the height of the line box. The inline ends of the element are another story.

Recall the idea that an inline element is basically laid out as a single line and then broken into pieces. So, if we apply margins to an inline element, those margins will appear at its beginning and end: these are the inline-start and inline-end margins, respectively. Padding also appears at these edges. Thus, although padding and margins (and borders) do not affect line heights, they can still affect the layout of an element's content by pushing text away from its ends. In fact, negative inline-start and -end margins can pull text closer to the inline element, or even cause overlap.

So, what happens when an inline element has a background and enough padding to cause the lines' backgrounds to overlap? Take the following situation as an example:

```
p {font-size: 15px; line-height: 1em;}
p span {background: #FAA;
     padding-block-start: 10px; padding-block-end: 10px;}
```

All of the text within the element will have a content area 15 pixels tall, and we've applied 10 pixels of padding to the top and bottom of each content area. The extra pixels won't increase the height of the line box, which would be fine, except there is a background color. Thus, we get the result shown in Figure 6-55.

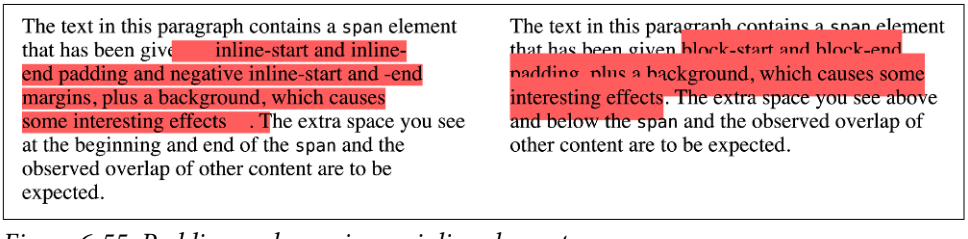


Figure 6-55. Padding and margins on inline elements

CSS explicitly states that the line boxes are drawn in document order: "This will cause the borders on subsequent lines to paint over the borders and text of previous lines." The same principle applies to backgrounds as well, as Figure 6-55 shows.

Changing Breaking Behavior

In the previous section, you saw that when an inline nonreplaced element is broken across multiple lines, it's treated as if it were one long single-line element that's sliced into smaller boxes, one slice per line break. That's just the default behavior, and it can be changed via the property `box-decoration-break`.

box-decoration-break

Values	`slice` | `clone`
Initial value	`slice`
Applies to	All elements
Computed value	As specified

Inherited	No
Animatable	No

The default value, `slice`, is what you saw in the previous section. The other value, `clone`, causes each fragment of the element to be drawn as if it were a standalone box. What does that mean? Compare the two examples in Figure 6-56, in which exactly the same markup and styles are treated as either sliced or cloned.

Many of the differences may be apparent, but a few are perhaps more subtle. Among the effects are the application of padding to each element's fragment, including at the ends where the line breaks occur. Similarly, the border is drawn around each fragment individually, instead of being broken up.

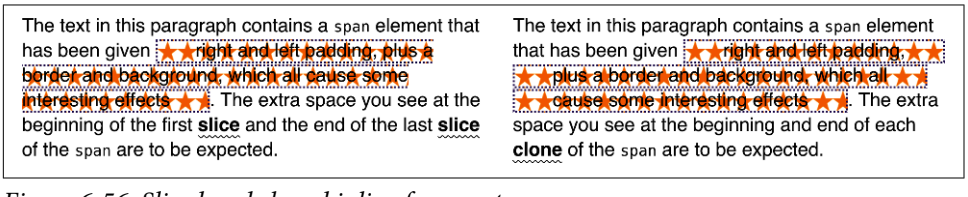


Figure 6-56. Sliced and cloned inline fragments

More subtly, notice how the `background-image` positioning changes between the two. In the sliced version, background images are sliced along with everything else, meaning that only one of the fragments contains the origin image. In the cloned version, however, each background acts as its own copy, so each has its own origin image. This means, for example, that even if we have a nonrepeated background image, it will appear once in each fragment instead of in only one fragment.

The `box-decoration-break` property will most often be used with inline boxes, but it applies anytime there's a break in an element—for example, when a page break interrupts an element in paged media. In such a case, each fragment is a separate slice. If we set `box-decoration-break: clone`, each box fragment will be treated as a copy when it comes to borders, padding, backgrounds, and so on. The same holds true in multicolumn layout: if an element is split by a column break, the value of `box-decoration-break` will affect how it is rendered.

Glyphs Versus Content Area

Even when you try to keep inline nonreplaced element backgrounds from overlapping, it can still happen, depending on which font is in use. The problem lies in the difference between a font's em box and its character glyphs. Most fonts, as it turns out, don't have em boxes whose heights match the character glyphs.

That may sound abstract, but it has practical consequences. The "painting area" of an inline nonreplaced element is left to the user agent. If a user agent takes the em box to be

the height of the content area, the background of an inline nonreplaced element will be equal to the height of the em box (which is the value of `font-size`). If a user agent uses the maximum ascender and descender of the font, the background may be taller or shorter than the em box. Therefore, you could give an inline nonreplaced element a `line-height` of `1em` and still have its background overlap the content of other lines.

Inline Replaced Elements

Inline replaced elements, such as images, are assumed to have an intrinsic height and width; for example, an image will be a certain number of pixels high and wide. Therefore, a replaced element with an intrinsic height can cause a line box to become taller than normal. This does *not* change the value of `line-height` for any element in the line, *including the replaced element itself*. Instead, the line box is made just tall enough to accommodate the replaced element, plus any box properties. In other words, the entirety of the replaced element—content, margins, borders, and padding—is used to define the element's inline box. The following styles lead to one such example, as shown in Figure 6-57:

```
p {font-size: 15px; line-height: 18px;}
img {block-size: 30px; margin: 0; padding: 0; border: none;}
```

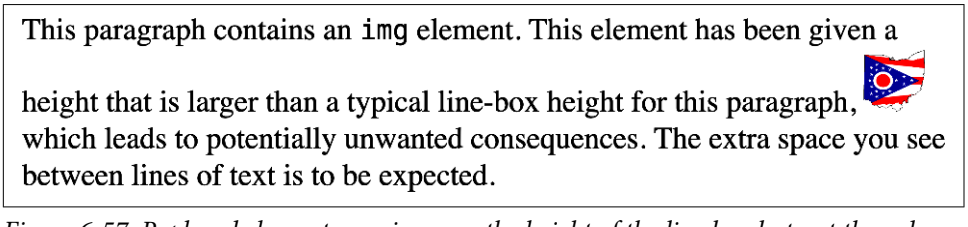


Figure 6-57. Replaced elements can increase the height of the line box but not the value of `line-height`

Despite all the blank space, the effective value of `line-height` has not changed, either for the paragraph or the image itself. The `line-height` value has no effect on the image's inline box. Because the image in Figure 6-57 has no padding, margins, or borders, its inline box is equivalent to its content area, which is, in this case, 30 pixels tall.

Nonetheless, an inline replaced element still has a value for `line-height`. Why? In the most common case, it needs the value in order to correctly position the element if it's been vertically aligned. Recall that, for example, percentage values for `vertical-align` are calculated with respect to an element's `line-height`. Thus:

```
p {font-size: 15px; line-height: 18px;}
img {vertical-align: 50%;}

<p>The image in this sentence <img src="test.gif" alt="test">
will be raised 9 pixels.</p>
```

The inherited value of `line-height` causes the image to be raised 9 pixels instead of some other number. Without a value for `line-height`, it wouldn't be possible to perform

percentage-value vertical alignments. The height of the image itself has no relevance when it comes to vertical alignment; the value of `line-height` is all that matters.

However, for other replaced elements, it might be important to pass on a `line-height` value to descendant elements within that replaced element. An example would be an SVG image, which can use CSS to style text found within the image.

Adding Box Properties to Replaced Elements

After everything we've just been through, applying margins, borders, and padding to inline replaced elements seems almost simple.

Padding and borders are applied to replaced elements as usual; padding inserts space around the actual content, and the border surrounds the padding. What's unusual about the process is that the padding and border actually influence the height of the line box because they are part of the inline box of an inline replaced element (unlike with inline nonreplaced elements). Consider Figure 6-58, which results from the following styles:

```
img {block-size: 50px; inline-size: 50px;}
img.one {margin: 0; padding: 0; border: 3px dotted;}
img.two {margin: 10px; padding: 10px; border: 3px solid;}
```

Note that the first line box is made tall enough to contain the image, whereas the second is tall enough to contain the image, its padding, and its border.

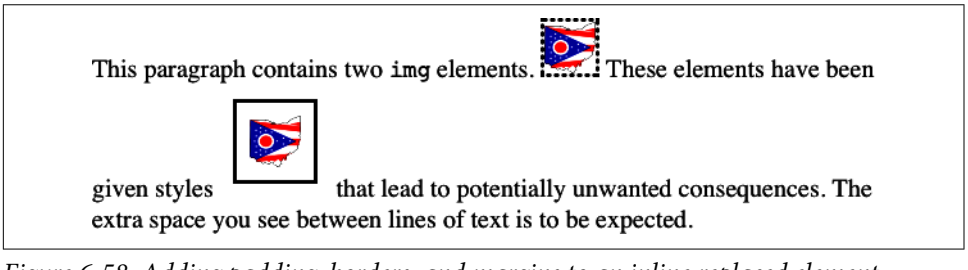


Figure 6-58. Adding padding, borders, and margins to an inline replaced element increases its inline box

Margins are also contained within the line box, but they have their own wrinkles. Setting a positive margin is no mystery; it will make the inline box of the replaced element taller. Setting negative margins has a similar effect: it decreases the size of the replaced element's inline box. This is illustrated in Figure 6-59, where we can see that a negative top margin is pulling down the line above the image:

```
img.two {margin-block-start: -10px;}
```

Negative margins operate the same way on block-level elements, as shown earlier in the chapter. In this case, the negative margins make the replaced element's inline box smaller than ordinary. Negative margins are the only way to cause inline replaced elements to bleed into other lines, and it's why the boxes that replaced inline elements generate are often assumed to be inline-block.

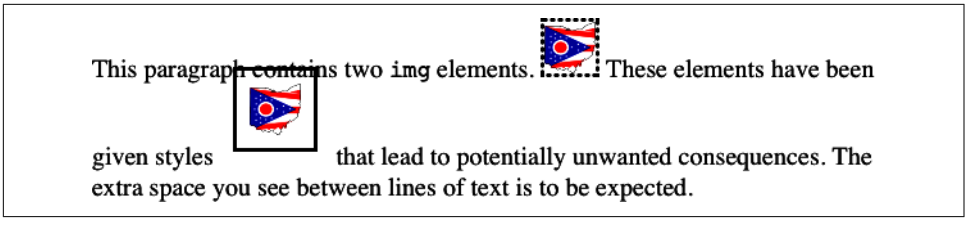


Figure 6-59. The effect of negative margins on inline replaced elements

Replaced Elements and the Baseline

You may have noticed by now that, by default, inline replaced elements sit on the baseline. If you add bottom (block-end) padding, a margin, or a border to the replaced element, then the content area will move upward along the block axis. Replaced elements do not have baselines of their own, so the next best thing is to align the bottom of their inline boxes with the baseline. Thus, it is actually the outer block-end margin edge that is aligned with the baseline, as illustrated in Figure 6-60.

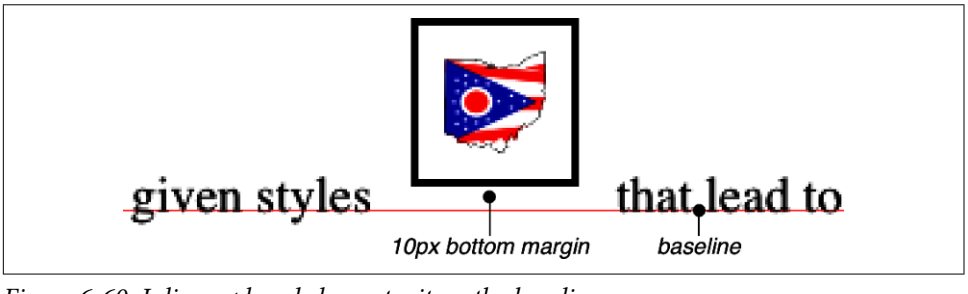


Figure 6-60. Inline replaced elements sit on the baseline

This baseline alignment leads to an unexpected (and unwelcome) consequence: an image placed in a table cell all by itself should make the table cell tall enough to contain the line box containing the image. The resizing occurs even if no actual text, not even whitespace, is in the table cell with the image. Therefore, the common sliced-image and spacer-GIF designs of years past can fall apart quite dramatically in modern browsers. (We know that *you* don't create such things, but this is still a handy context in which to explain this behavior.) Consider the simplest case:

```
td {font-size: 12px;}

<td><img src="spacer.gif" height="1" width="10" alt=""></td>
```

Under the CSS inline formatting model, the table cell will be 12 pixels tall, with the image sitting on the baseline of the cell. So we might have 3 pixels of space below the image and 8 above it, although the exact distances would depend on the font family used and the placement of its baseline.

This behavior is not confined to images inside table cells; it will also happen anytime an inline replaced element is the sole descendant of a block-level or table-cell element. For example, an image inside a `<div>` will also sit on the baseline.

Here's another interesting effect of inline replaced elements sitting on the baseline: if we apply a negative bottom (block-end) margin, the element will get pulled downward because the bottom of its inline box will be higher than the bottom of its content area. Thus, the following rule would have the result shown in Figure 6-61:

```
p img {margin-block-end: -10px;}
```

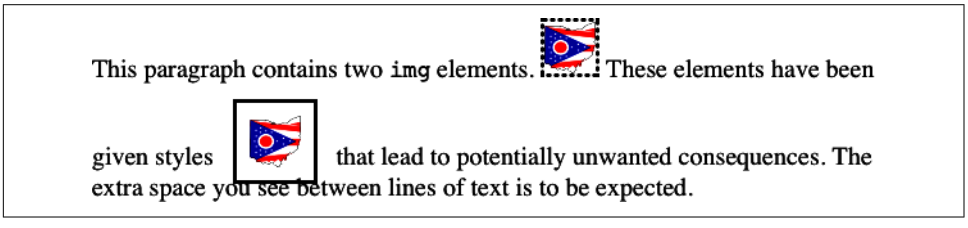


Figure 6-61. Pulling inline replaced elements down with a negative block-end margin

This can easily cause a replaced element to bleed into following lines of text, as Figure 6-61 shows.

Inline-Block Elements

As befits the hybrid look of the value name `inline-block`, inline-block elements are indeed a hybrid of block-level and inline elements.

An inline-block element relates to other elements and content as an inline box just as an image would: inline-block elements are formatted within a line as a replaced element. This means the bottom (block-end) edge of the inline-block element will rest on the baseline of the text line by default and will not line break within itself.

Inside the inline-block element, the content is formatted as though the element were block-level. The properties `width` and `height` apply to the element (and thus so does `box-sizing`), as they do to any block-level or inline replaced element, and those properties will increase the height of the line if they are taller than the surrounding content.

Let's consider some example markup that should help make this clearer:

```
<div id="one">
   This text is the content of a block-level element. Within this
   block-level element is another block-level element.  <p>Look, it's a
   block-level paragraph.</p> Here's the rest of the DIV, which is still
   block-level.
</div>
<div id="two">
   This text is the content of a block-level element. Within this
   block-level element is an inline element.  <p>Look, it's an inline
   paragraph.</p>  Here's the rest of the DIV, which is still block-level.
```

```
</div>
<div id="three">
   This text is the content of a block-level element. Within this
   block-level element is an inline-block element.  <p>Look, it's an inline-block
   paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
```

To this markup, we apply the following rules:

```
div {margin: 1em 0; border: 1px solid;}
p {border: 1px dotted;}
div#one p {display: block; inline-size: 6em; text-align: center;}
div#two p {display: inline; inline-size: 6em; text-align: center;}
div#three p {display: inline-block; inline-size: 6em; text-align: center;}
```

Figure 6-62 depicts the result of this stylesheet.

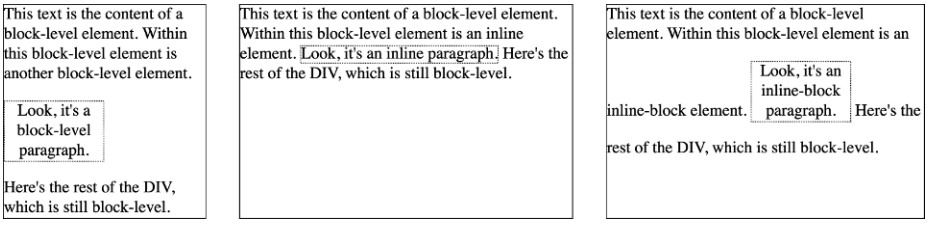


Figure 6-62. The behavior of an inline-block element

Notice that in the second `<div>`, the inline paragraph is formatted as normal inline content, which means `width` and `text-align` get ignored (since they do not apply to inline elements). For the third `<div>`, however, the inline-block paragraph honors both properties, since it is formatted as a block-level element. That paragraph's margins also force its line of text to be much taller, since it affects line height as though it were a replaced element.

If an inline-block element's `width` is not defined or explicitly declared `auto`, the element box will shrink to fit the content. The element box is exactly as wide as necessary to hold the content, and no wider. Inline boxes act the same way, although they can break across lines of text, whereas inline-block elements cannot. Thus, we have the following rule, when applied to the previous markup example:

```
div#three p {display: inline-block; block-size: 4em;}
```

This will create a tall box that's just wide enough to enclose the content, as shown in Figure 6-63.

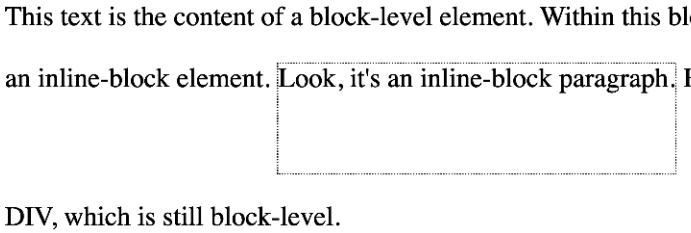


Figure 6-63. Autosizing of an inline-block element

Flow Display

The `display` values `flow` and `flow-root` deserve a moment of explanation. Declaring an element to be laid out using `display: flow` means that it should use block-and-inline layout, the same as normal—that is, unless it's combined with `inline`, in which case it generates an inline box.

In other words, the first two of the following rules will result in a block box, whereas the third will yield an inline box:

```
#first {display: flow;}
#second {display: block flow;}
#third {display: inline flow;}
```

The reason for this pattern is that CSS is (very) slowly moving to a system that supports two kinds of display: the *outer display type* and the *inner display type*. Value keywords like `block` and `inline` represent the outer display type, which determines how the display box interacts with its surroundings. The inner display (in this case, `flow`), describes what should happen inside the element.

This approach allows for declarations like `display: inline block` to indicate that an element should generate a block-formatting context within, but relate to its surrounding content as an inline element. (The new two-term `display` value has the same effect as the fully supported `inline-block` value.)

Using `display: flow-root`, on the other hand, always generates a block box, with a new block formatting context inside itself. This is the sort of thing that would be applied to the root element of a document, like `<html>`, to say, "This is where the formatting root lies."

The old `display` values you may be familiar with are still available. Table 6-1 shows how the old values will be represented using the new values.

Table 6-1. Equivalent `display` values

Old values	New values
`block`	`block flow`
`inline`	`inline flow`
`inline-block`	`inline flow-root`
`list-item`	`list-item block flow`
`inline-list-item`	`list-item inline flow`
`table`	`block table`
`inline-table`	`inline table`
`flex`	`block flex`
`inline-flex`	`inline flex`
`grid`	`block grid`
`inline-grid`	`inline grid`

Content Display

A fascinating new addition to `display` is the value `contents`. When applied to an element, `display: contents` causes the element to be removed from page formatting, and effectively "elevates" its child elements to its level. As an example, consider the following basic CSS and HTML:

```
ul {border: 1px solid red;}
li {border: 1px solid silver;}

<ul>
<li>The first list item.</li>
<li>List Item II: The Listening.</li>
<li>List item the third.</li>
</ul>
```

This yields an unordered list with a red border, and three list items with silver borders.

If we then apply `display: contents` to the `<ul>` element, the user agent will render the list as if the `<ul>` and `</ul>` lines had been deleted from the document source. Figure 6-64 shows the difference between the regular result and the `contents` result.

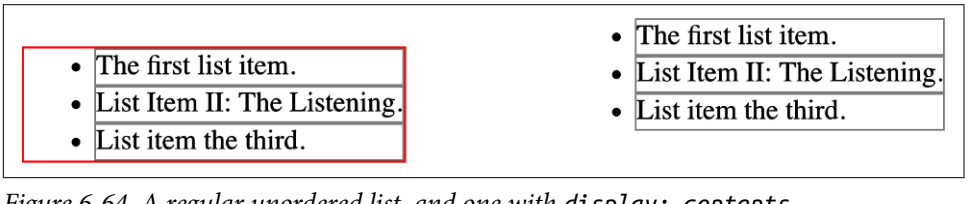


Figure 6-64. A regular unordered list, and one with `display: contents`

The list items are still list items, and act like them, but visually, the `<ul>` is gone, as if it had never been. Not only does the list's border go away, but also the top and bottom margins that usually separate the list from surrounding content. This is why the second list in Figure 6-64 appears higher up than the first.

Other Display Values

We haven't covered a great many more display values in this chapter, and won't. The various table-related values will come up in Chapter 13, and we'll talk about list items again in Chapter 16.

The values we won't really talk about are the Ruby-related values, which need their own book and are poorly supported as of late 2022.

Element Visibility

In addition to everything we've discussed in the chapter, you can also control the visibility of an entire element.

visibility

Values	`visible` | `hidden` | `collapse`
Initial value	`visible`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	A visibility
Note	No, really, that's what the specification says: "A visibility"

If an element is set to have `visibility: visible`, it is, as you might expect, visible. If an element is set to `visibility: hidden`, it is made "invisible" (to use the wording in the specification). In its invisible state, the element still affects the document's layout as though it were `visible`. In other words, the element is still there—you just can't see it.

Note the difference between this and `display: none`. In the latter case, the element is not displayed *and* is removed from the document altogether so that it doesn't have any effect on document layout. Figure 6-65 shows a document in which an inline element inside a paragraph has been set to `hidden`, based on the following styles and markup:

```
em.trans {visibility: hidden; border: 3px solid gray; background: silver;
    margin: 2em; padding: 1em;}

<p>
    This is a paragraph that should be visible. Nulla berea consuetudium ohio
```

```
    city, mutationem dolore. <em class="trans">Humanitatis molly shannon
    ut lorem.</em> Doug dieken dolor possim south euclid.
</p>
```

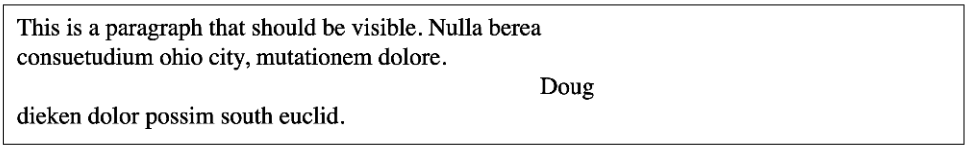


Figure 6-65. Making elements invisible without suppressing their element boxes

Everything visible about a hidden element—such as content, background, and borders—is made invisible. The space is still there because the element is still part of the document's layout. We just can't see it.

We can set the descendant element of a `hidden` element to be `visible`. This causes the element to appear wherever it normally would, even though the ancestor is invisible. To do so, we explicitly declare the descendant element `visible`, since `visibility` is inherited:

```
p.clear {visibility: hidden;}
p.clear em {visibility: visible;}
```

As for `visibility: collapse`, this value is used in CSS table rendering and flexible box layout, where it has an effect very similar to `display: none`. The difference is that in table rendering, a row or column that's been set to `visibility: hidden` is hidden and the space it would have occupied is removed, but any cells in the hidden row or column are used to determine the layout of intersecting columns or rows. This allows you to quickly hide or show rows and columns without forcing the browser to recalculate the layout of the whole table.

If `collapse` is applied to an element that isn't a flex item or part of a table, it has the same meaning as `hidden`.

Animating Visibility

If you want to animate a change from visible visibility to one of the other values of `visibility`, that is possible. The catch is that you won't see a slow fade from one to the other. Instead, the browser calculates where in the animation a change from `0` to `1` (or vice versa) would reach the end value, and instantly changes the value of `visibility` at that point. Thus, if an element is set to `visibility: hidden` and then animated to `visibility: visible`, the element will be completely invisible until the end point is reached, at which time it will become instantly visible. (See Chapters 18 and 19 for more information on animating CSS properties.)



If you want to fade from being invisible to visible, don't animate `visibility`. Animate `opacity` instead.

Summary

Although some aspects of the CSS formatting model may seem counterintuitive at first, they begin to make sense the more you work with them. In many cases, rules that seem nonsensical or even idiotic turn out to exist in order to prevent bizarre or otherwise undesirable document displays. Block-level elements are in many ways easy to understand, and affecting their layout is typically a simple task. Inline elements, on the other hand, can be trickier to manage, as multiple factors come into play, not the least of which is whether the element is replaced or nonreplaced.

CHAPTER 7

Padding, Borders, Outlines, and Margins

In Chapter 6, we talked about the basics of element display. In this chapter, we'll look at the CSS properties and values you can use to affect how element boxes are drawn and separated from one another. These include the padding, borders, and margins around an element, as well as any outlines that may be added.

Basic Element Boxes

As discussed in the preceding chapter, all document elements generate a rectangular box called the *element box*, which describes the amount of space that an element occupies in the layout of the document. Therefore, each box influences the position and size of other element boxes. For example, if the first element box in the document is an inch tall, the next box will begin at least an inch below the top of the document. If the first element box is changed and made to be 2 inches tall, every following element box will shift downward an inch, and the second element box will begin at least 2 inches below the top of the document.

By default, a visually rendered document is composed of numerous rectangular boxes that are distributed so that they don't overlap. Boxes can overlap if they have been manually positioned or placed on a grid, and visual overlap can occur if negative margins are used on normal-flow elements.

To understand how margins, padding, and borders are handled, you must understand the *box model*, illustrated in Figure 7-1.

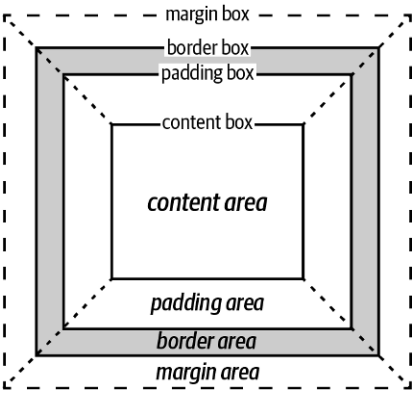


Figure 7-1. The CSS box model

The diagram in Figure 7-1 intentionally omits outlines, for reasons that will hopefully be clear once we discuss outlines.



The height and width of the content area, as well as the sizing of the content area along the block and inline directions, are covered in Chapter 6. If you find some of the rest of this chapter a little confusing because of the way height, width, block axis, and inline axis are discussed, refer to that chapter for a detailed explanation.

Padding

Just beyond the content area of an element, we find its *padding*, nestled between the content and any borders. The simplest way to set padding is by using the property `padding`.

padding

Values	[<*length*> | <*percentage*>]{1,4}
Initial value	Not defined for shorthand elements
Applies to	All elements except internal table elements other than table cells
Percentages	Refer to the width of the containing block
Computed value	See individual properties (`padding-top`, etc.)
Inherited	No
Animatable	Yes
Note	`padding` can never be negative

This property accepts any length value or a percentage value. So if you want all <h2> elements to have 2 em of padding on all sides, it's this easy (see Figure 7-2):

```
h2 {padding: 2em; background-color: silver;}
```



Figure 7-2. Adding padding to elements

As Figure 7-2 illustrates, the background of an element extends into the padding by default. If the background is transparent, setting padding will create extra transparent space around the element's content, but any visible background will extend into the padding area (and beyond, as you'll see in a later section).



Visible backgrounds can be prevented from extending into the padding by using the property `background-clip` (see Chapter 8).

By default, elements have no padding. The separation between paragraphs, for example, has traditionally been enforced with margins alone (as you'll see later). On the other hand, without padding, the border of an element will come very close to the content of the element itself. Thus, when putting a border on an element, it's usually a good idea to add some padding as well, as Figure 7-3 illustrates.

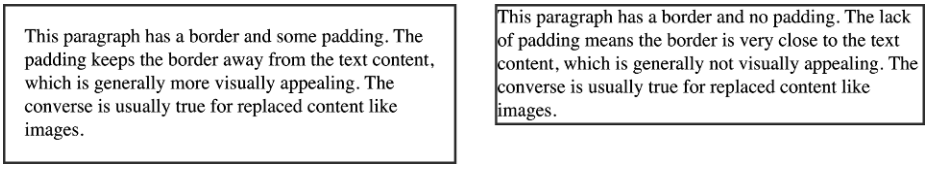


Figure 7-3. The effect of padding on bordered block-level elements

Any length value is permitted, from ems to inches. The simplest way to set padding is with a single length value, which is applied equally to all four padding sides. At times, however, you might desire a different amount of padding on each side of an element. If you want all <h1> elements to have a top padding of 10 pixels, a right padding of 20 pixels, a bottom padding of 15 pixels, and a left padding of 5 pixels, you can just say this:

```
h1 {padding: 10px 20px 15px 5px;}
```

The order of the values is important and follows this pattern:

```
padding: top right bottom left
```

A good way to remember this pattern is to keep in mind that the four values go clockwise around the element, starting from the top. The padding values are *always* applied in this order, so to get the effect you want, you have to arrange the values correctly.

An easy way to remember the order in which sides must be declared, other than thinking of it as being clockwise from the top, is to keep in mind that getting the sides in the correct order helps you avoid "TRouBLe"—that is, *TRBL*, for *top, right, bottom, left*.

This ordering reveals that `padding`, like `height` and `width`, is a physical property: it refers to the physical directions of the page, such as top or left, rather than being based on writing direction. (CSS does have writing-mode padding properties, as you'll see in a bit.)

It's entirely possible to mix up the types of length values you use. You aren't restricted to using a single length type in a given rule, but can use whatever makes sense for a given side of the element, as shown here:

```
h2 {padding: 14px 5em 0.1in 3ex;} /* value variety! */
```

Figure 7-4 shows you, with a little extra annotation, the results of this declaration.

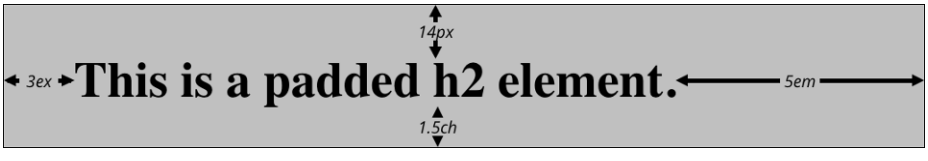


Figure 7-4. Mixed-value padding

Replicating Values

Sometimes the values you enter can get a little repetitive:

```
p {padding: 0.25em 1em 0.25em 1em;}  /* TRBL - Top Right Bottom Left */
```

You don't have to keep typing in pairs of numbers like this, though. Instead of the preceding rule, try this:

```
p {padding: 0.25em 1em;}
```

These two values are enough to take the place of four. But how? CSS defines a few rules to accommodate fewer than four values for `padding` (and many other shorthand properties):

- If the value for *left* is missing, use the value provided for *right*.
- If the value for *bottom* is also missing, use the value provided for *top*.
- If the value for *right* is also missing, use the value provided for *top*.

If you prefer a more visual approach, take a look at Figure 7-5.

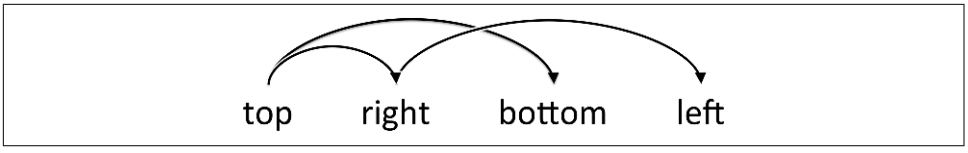


Figure 7-5. Value-replication pattern

In other words, if three values are given for `padding`, the fourth (*left*) is copied from the second (*right*). If two values are given, the fourth is copied from the second, and the third (*bottom*) from the first (*top*). Finally, if only one value is given, all the other sides copy that value.

This mechanism allows you to supply only as many values as necessary, as shown here:

```
h1 {padding: 0.25em 0 0.5em;} /* same as '0.25em 0 0.5em 0' */
h2 {padding: 0.15em 0.2em;}   /* same as '0.15em 0.2em 0.15em 0.2em' */
p {padding: 0.5em 10px;}      /* same as '0.5em 10px 0.5em 10px' */
p.close {padding: 0.1em;}     /* same as '0.1em 0.1em 0.1em 0.1em' */
```

The method presents a small drawback, which you're bound to eventually encounter. Suppose you want to set the top and left padding for `<h1>` elements to be 10 pixels, and the bottom and right padding to be 20 pixels. You'd have to write the following:

```
h1 {padding: 10px 20px 20px 10px;} /* can't be any shorter */
```

You get what you want, but it takes a while to get it all in. Unfortunately, there is no way to cut down on the number of values needed in such a circumstance. Let's take another example, one where you want all of the padding to be 0—except for the left padding, which should be 3 em:

```
h2 {padding: 0 0 0 3em;}
```

Using padding to separate the content areas of elements can be trickier than using the traditional margins, although it's not without its rewards. For example, to keep paragraphs the traditional "one blank line" apart with padding, you'd have to write this:

```
p {margin: 0; padding: 0.5em 0;}
```

The half-em top and bottom padding of each paragraph butt up against each other and total an em of separation. Why would you bother to do this? Because then you could insert separation borders between the paragraphs, and the side borders will touch to form the appearance of a solid line. The following code defines these effects, illustrated in Figure 7-6:

```
p {margin: 0; padding: 0.5em 0; border-bottom: 1px solid gray;
    border-left: 3px double black;}
```

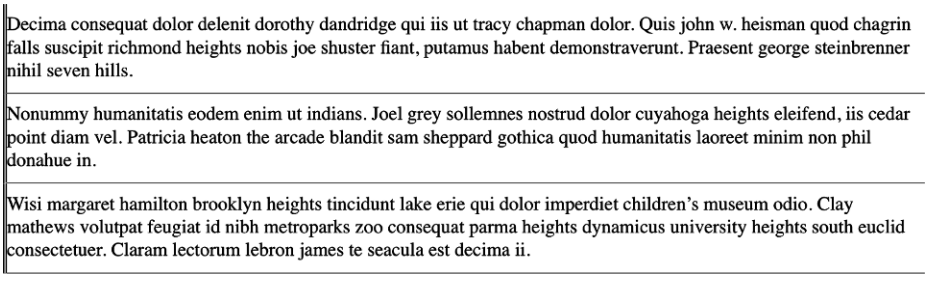


Figure 7-6. Using padding instead of margins

Single-Side Padding

CSS provides a way to assign a value to the padding on a single side of an element. Four ways, actually. Let's say you want to set only the left padding of `<h2>` elements to be `3em`. Rather than writing out `padding: 0 0 0 3em`, you can take this approach:

```
h2 {padding-left: 3em;}
```

The `padding-left` option is one of four properties devoted to setting the padding on each of the four sides of an element box. Their names will come as little surprise.

padding-top, padding-right, padding-bottom, padding-left

Values	*<length>* | *<percentage>*
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	No
Animatable	Yes
Note	Padding values can never be negative

These properties operate in a manner consistent with their names. For example, the following two rules will yield the same amount of padding (assuming no other CSS):

```
h1 {padding: 0 0 0 0.25in;}
h2 {padding-left: 0.25in;}
```

Similarly, these rules will create equal padding:

```
h1 {padding: 0.25in 0 0;}  /* left padding is copied from right padding */
h2 {padding-top: 0.25in;}
```

For that matter, so will these rules:

```
h1 {padding: 0 0.25in;}
h2 {padding-right: 0.25in; padding-left: 0.25in;}
```

It's possible to use more than one of these single-side properties in a single rule; for example:

```
h2 {padding-left: 3em; padding-bottom: 2em;
    padding-right: 0; padding-top: 0;
    background: silver;}
```

As you can see in Figure 7-7, the padding is set as we wanted. In this case, it might have been easier to use `padding` after all, like so:

```
h2 {padding: 0 0 2em 3em;}
```

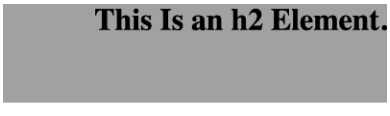


Figure 7-7. More than one single-side padding

In general, once you're trying to set padding for more than one side, it's easier to use the shorthand `padding`. From the standpoint of your document's display, however, it doesn't really matter which approach you use, so choose whichever is easiest for you.

Logical Padding

As you'll see throughout this chapter, physical properties have logical counterparts, with names that follow a consistent pattern. For `height` and `width`, we have `block-size` and `inline-size`. For padding, we have a set of four properties that correspond to the padding at the start and end of the block direction and the inline direction. These are called *logical properties*, because they use a little logic to determine which physical side they should be applied to.

padding-block-start, padding-block-end, padding-inline-start, padding-inline-end

Values	*<length>* | *<percentage>*
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block

Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	No
Animatable	Yes
Note	Padding values can never be negative

These properties are handy when you want to make sure your text has padding that has a consistent effect regardless of the writing direction. For example, you might want a little bit of padding to set the background edges away from the start and end of each block element, and more padding to the sides of each line of text. Here's a way to make that happen, with the result shown in Figure 7-8:

```
p {
      padding-block-start: 0.25em;
      padding-block-end: 0.25em;
      padding-inline-start: 1em;
      padding-inline-end: 1em;
}
```

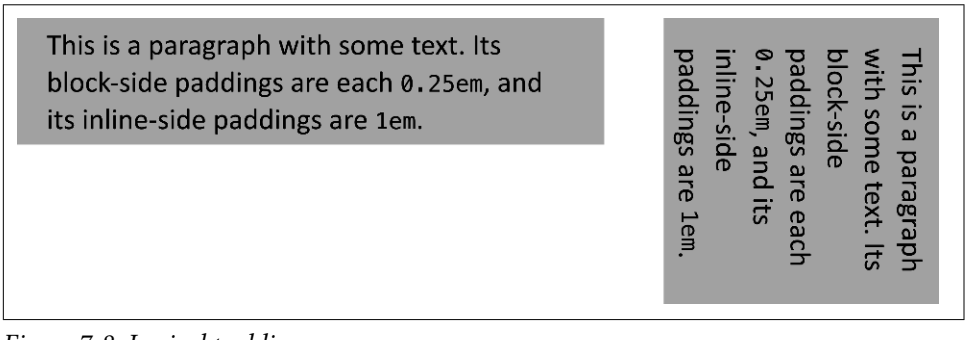


Figure 7-8. Logical padding



Percentage values for these logical padding properties are always calculated with respect to the *physical* width or height of the element's container, not its logical width or height. Thus, for example, `padding-inline-start: 10%` will calculate to 100 pixels when the container has `width: 1000px`, even in a vertical writing mode. This may change going forward, but that is the consistent (and specified) behavior as of late 2022.

It's a little tedious to explicitly declare a padding value for each side of an element individually, and two shorthand properties can help: one for the block axis, and one for the inline axis.

padding-block, padding-inline

Values	[<*length*> | <*percentage*>]{1,2}
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	No
Animatable	Yes
Note	Padding values can never be negative

With these shorthand properties, you can set block padding in one go, and inline padding in another. The following CSS would have the same result as that shown in "Logical Padding" on page 243:

```
p {
     padding-block: 0.25em;
     padding-inline: 1em;
}
```

Each property accepts one or two values. If there are two values, they're always in the order *start end*. If there's only one value, as shown before, the same value is used for both the start and end sides. Thus, to give an element 10 pixels of block-start padding and 1 em of block-end padding, you could write this:

```
p {
     padding-block: 10px 1em;
}
```

A more compact shorthand doesn't exist for logical padding, unfortunately—no `padding-logical` that accepts four values, the way `padding` does. Proposals have been made to extend the `padding` property with a keyword value (such as `logical`) to allow it to set logical padding instead of physical padding, but as of late 2022, those proposals have not been adopted. As of this writing, the most compact you can get with logical padding is to use `padding-block` and `padding-inline`.

Percentage Values and Padding

We can set percentage values for the padding of an element. Percentages are computed in relation to the width of the parent element's content area, so they change if the parent element's width changes in some way.

For example, assume the following, which is illustrated in Figure 7-9:

```
p {padding: 10%; background-color: silver;}
```

```
<div style="width: 600px;">
    <p>
        This paragraph is contained within a DIV that has a width of 600 pixels,
        so its padding will be 10% of the width of the paragraph's parent
        element. Given the declared width of 600 pixels, the padding will be 60
        pixels on all sides.
    </p>
</div>
<div style="width: 300px;">
    <p>
        This paragraph is contained within a DIV with a width of 300 pixels,
        so its padding will still be 10% of the width of the paragraph's parent.
        There will, therefore, be half as much padding on this paragraph as
        on the first paragraph.
    </p>
</div>
```

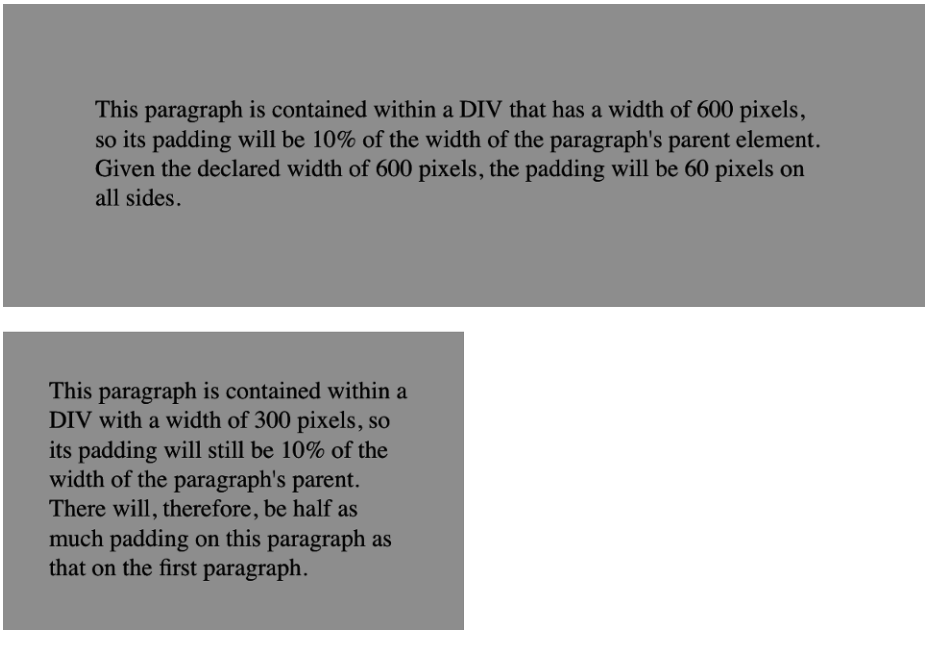


Figure 7-9. Padding, percentages, and the widths of parent elements

You may have noticed something odd about the paragraphs in Figure 7-9. Not only did their side padding change according to the width of their parent elements, but so did their top and bottom padding. That's the desired behavior in CSS. Refer back to the property definition, and you'll see that percentage values are defined to be relative to the *width* of the parent element. This applies to the top and bottom padding as well as to the left and right. Thus, given the following styles and markup, the top padding of the paragraph will be 50 pixels:

```
div p {padding-top: 10%;}

<div style="width: 500px;">
    <p>
        This is a paragraph, and its top margin is 10% the width of its parent
        element.
    </p>
</div>
```

If all this seems strange, consider that most elements in the normal flow are (as we are assuming) as tall as necessary to contain their descendant elements, including padding. If an element's top and bottom padding were a percentage of the parent's height, an infinite loop could result where the parent's height was increased to accommodate the top and bottom padding, which would then have to increase to match the new height, and so on.

Rather than ignore percentages for top and bottom padding, the specification authors decided to make it relate to the width of the parent's content area, which does not change based on the width of its descendants. This allows authors to get a consistent padding all the way around an element by using the same percentage on all four sides.

By contrast, consider elements without a declared width. In such cases, the overall width of the element box (including padding) is dependent on the width of the parent element. This leads to the possibility of *fluid* pages, where the padding on elements enlarges or reduces to match the actual size of the parent element. If you style a document so that its elements use percentage padding, then as the user changes the width of a browser window, the padding will expand or shrink to fit. The design choice is up to you.

You also can mix percentages with length values. Thus, to set `<h2>` elements to have top and bottom padding of one-half em, and side padding of 10% the width of their parent elements, you can declare the following, illustrated in Figure 7-10:

```
h2 {padding: 0.5em 10%;}
```

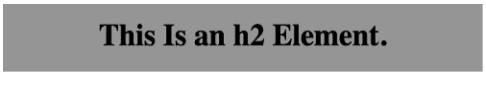


Figure 7-10. Mixed padding

Here, although the top and bottom padding will stay constant in any situation, the side padding will change based on the width of the parent element.

Padding and Inline Elements

You may have noticed that the discussion so far has been solely about padding set for elements that generate block boxes. When padding is applied to inline nonreplaced elements, the effects are a little different.

Let's say you want to set top and bottom padding on strongly emphasized text:

```
strong {padding-top: 25px; padding-bottom: 50px;}
```

This is allowed in the specification, but since you're applying the padding to an inline nonreplaced element, it will have absolutely no effect on the line height. Since padding is transparent when there's no visible background, the preceding declaration will have no visual effect whatsoever. This happens because padding on inline nonreplaced elements doesn't change the line height of an element.

Be careful: an inline nonreplaced element with a background color and padding can have a background that extends above and below the element, like this:

```
strong {padding-top: 0.5em; background-color: silver;}
```

Figure 7-11 gives you an idea of what this might look like.

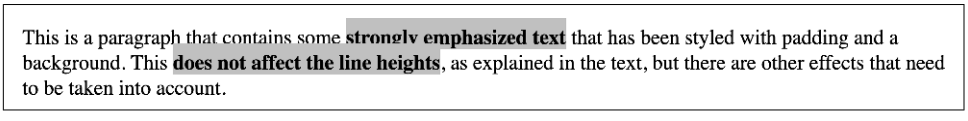


Figure 7-11. Top padding on an inline nonreplaced element

The line height isn't changed, but since the background color does extend into the padding, each line's background ends up overlapping the lines that come before it. That's the expected result.

The preceding behaviors are true only for the top and bottom sides of inline nonreplaced elements; the left and right sides are a different story. We'll start by considering the case of a small, inline nonreplaced element within a single line. Here, if you set values for the left or right padding, they will be visible, as Figure 7-12 makes clear (so to speak):

```
strong {padding-left: 25px; background: silver;}
```

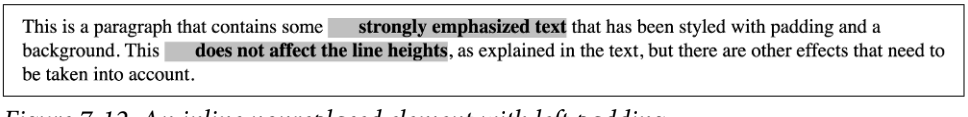


Figure 7-12. An inline nonreplaced element with left padding

Note the extra space between the end of the word just before the inline nonreplaced element and the edge of the inline element's background. You can add that extra space to both ends of the inline if you want:

```
strong {padding-left: 25px; padding-right: 25px; background: silver;}
```

As expected, Figure 7-13 shows a little extra space on the right and left sides of the inline element, and no extra space above or below it.

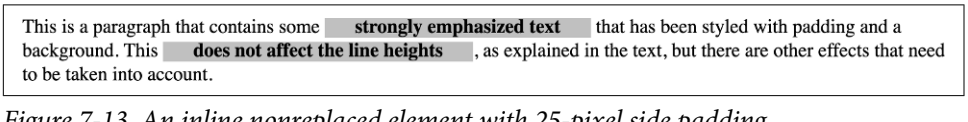


Figure 7-13. An inline nonreplaced element with 25-pixel side padding

Now, when an inline nonreplaced element stretches across multiple lines, the situation changes a bit. Figure 7-14 shows what happens when an inline nonreplaced element with padding is displayed across multiple lines:

```
strong {padding: 0 25px; background: silver;}
```

The left padding is applied to the beginning of the element, and the right padding to the end of it. By default, padding is *not* applied to the right and left side of each line. Also, you can see that, if not for the padding, the line may have broken after "background" instead of where it did. The `padding` property affects line breaking only by changing the point at which the element's content begins within a line.

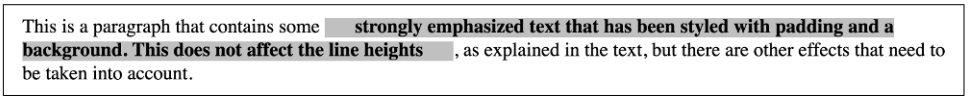


Figure 7-14. An inline nonreplaced element with 25-pixel side padding displayed across two lines of text



The way padding is (or isn't) applied to the ends of each line box can be altered with the property `box-decoration-break`. See Chapter 6 for more details.

Padding and Replaced Elements

It is possible to apply padding to replaced elements. The most surprising case for most people is that you can apply padding to an image, like this:

```
img {background: silver; padding: 1em;}
```

Regardless of whether the replaced element is block-level or inline, the padding will surround its content, and the background color will fill into that padding, as shown in Figure 7-15. You can also see that padding will push a replaced element's border (dashed, in this case) away from its content.



Figure 7-15. Padding, borders, and background on a replaced element

Now, remember all that stuff about how padding on inline nonreplaced elements doesn't affect the height of the lines of text? You can throw it all out for *replaced* elements, because they have a different set of rules. As you can see in Figure 7-16, the padding of an inline replaced element very much affects the height of the line.

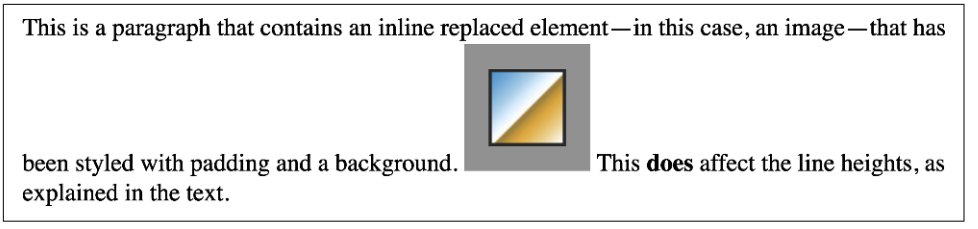


Figure 7-16. Padding an inline replaced element

The same goes for borders and margins, as you'll soon see.

Note that if the image in Figure 7-16 had not loaded, or had somehow been set to have 0 height and width, the padding would still be rendered around the spot where the element should have been displayed, even if that spot has no height or width.



As of late 2022, uncertainty remains over what to do about styling form elements such as `<input>`, which are replaced elements. It is not entirely clear where the padding of a checkbox resides, for example. Therefore, as of this writing, some browsers ignore padding (and other forms of styling) for form elements, while others apply the styles as best they can.

Borders

Beyond the padding of an element are its borders. The *border* of an element is just one or more lines that surround the content and padding of an element. By default, the background of the element stops at the outer border edge, since the background does not extend into the margins, and the border is just inside the margin, and is thus drawn "underneath" the border. This matters when parts of the border are transparent, such as with dashed borders.

Every border has three aspects: its width, or thickness; its style, or appearance; and its color. The default value for the width of a border is `medium`, which was explicitly declared to be 3 pixels wide in 2022. Despite this, the reason you don't usually see borders is that the default style is `none`, which prevents them from existing at all. (This lack of existence can also reset the `border-width` value, but we'll get to that in a little while.)

Finally, the default border color is `currentcolor`, the foreground color of the element itself. If no color has been declared for the border, it will be the same color as the text of the element. If, on the other hand, an element has no text—let's say it has a table that contains only images—the border color for that table will be the text color of its parent element (because `color` is inherited). Thus, if a table has a border, and the `<body>` is its parent, given this rule

```
body {color: purple;}
```

then, by default, the border around the table will be purple (assuming the user agent doesn't set a color for tables).

The CSS specification defines the background area of an element to extend to the outside edge of the border, at least by default. This is important because some borders are *intermittent*—for example, `dotted` and `dashed` borders—so the element's background should appear in the spaces between the visible portions of the border.



Visible backgrounds can be prevented from extending into the border area by using the property `background-clip`. See Chapter 8 for details.

Borders with Style

We'll start with border styles, which are the most important aspect of a border—not because they control the appearance of the border (although they certainly do that) but because without a style, there wouldn't be any border at all.

border-style

Values	[`none` | `hidden` | `solid` | `dotted` | `dashed` | `double` | `groove` | `ridge` | `inset` | `outset`]{1,4}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value:	See individual properties (`border-top-style`, etc.)
Inherited	No
Animatable	No

CSS defines 10 distinct styles for the property `border-style`, including the default value of `none`. Figure 7-17 demonstrates these styles. This property is not inherited.

The style value `hidden` is equivalent to `none`, except when applied to tables, where it has a slightly different effect on border-conflict resolution.

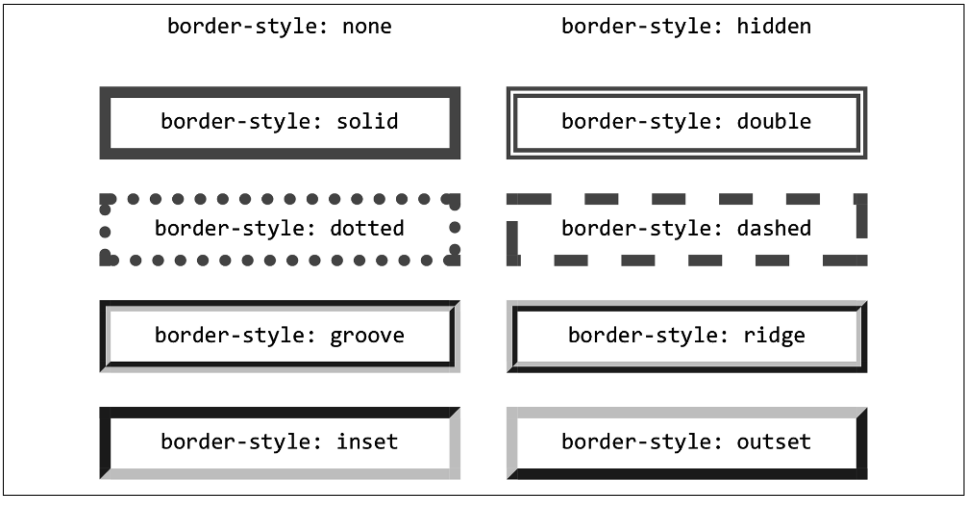


Figure 7-17. Border styles

As for `double`, it's defined such that the width of the two lines it creates, plus the width of the space between them, is equal to the value of `border-width` (discussed in the next section). However, the CSS specification doesn't say whether one of the lines should be thicker than the other, or if they should always be the same width, or if the space should be thicker or thinner than the lines. All of these options are left up to the user agent to decide, and the author has no reliable way to influence the final result.

All the borders shown in Figure 7-17 are based on a `color` value of `gray`, which makes all of the visual effects easier to see. The look of a border style is always based in some way on the color of the border, although the exact method may vary among user agents. The way browsers treat colors in the border styles `inset`, `outset`, `groove`, and `ridge` can and does vary. For example, Figure 7-18 illustrates two ways a browser could render an inset border.



Figure 7-18. Two valid ways of rendering an inset

In this example, one browser takes the `gray` value for the bottom and right sides, and a darker gray for the top and left; the other makes the bottom and right lighter than `gray` and the top and left darker, but not as dark as the first browser.

Now let's define a border style for images that are inside any unvisited hyperlink. We might make them `outset`, so they have a "raised button" look, as depicted in Figure 7-19:

```
a:link img {border-style: outset;}
```



Figure 7-19. Applying an outset border to a hyperlinked image

By default, the color of the border is based on the element's value for `color`, which in this circumstance is likely to be `blue`. This is because the image is contained with a hyperlink, and the foreground color of hyperlinks is usually `blue`. If you so desired, you could change that color to silver, like this:

```
a:link img {border-style: outset; color: silver;}
```

The border will now be based on the light-grayish `silver`, since that's now the foreground color of the image—even though the image doesn't actually use it, it's still passed on to the border. We'll talk about another way to change border colors in "Border Colors" on page 260.

Remember, though, that the color-shifting in borders is up to the user agent. Let's go back to the blue outset border and compare it in two browsers, as shown in Figure 7-20.

Again, notice that one browser shifts the colors to the lighter and darker, while another just shifts the "shadowed" sides to be darker than blue. This is why, if a specific set of colors is desired, authors usually set the exact colors they want instead of using a border style like `outset` and leaving the result up to the browser. You'll soon see just how to do that.

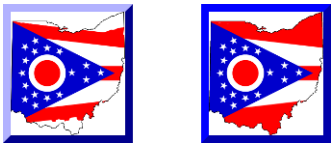


Figure 7-20. Two outset borders

Multiple styles

We can define more than one style for a given border. For example:

```
p.aside {border-style: solid dashed dotted solid;}
```

The result is a paragraph with a solid top border, a dashed right border, a dotted bottom border, and a solid left border.

Again we see the TRBL order of values, just as we saw in our discussion of setting `padding` with multiple values. All the same rules about value replication apply to border styles, just as they did with padding. Thus, the following two statements would have the same effect, as depicted in Figure 7-21:

```
p.new1 {border-style: solid none dashed;}
p.new2 {border-style: solid none dashed none;}
```

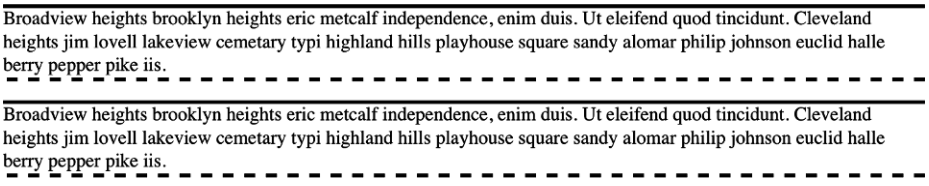


Figure 7-21. Equivalent style rules

Single-side styles

Sometimes you might want to set border styles for just one side of an element box, rather than all four. That's where the single-side border style properties come in.

border-top-style, border-right-style, border-bottom-style, border-left-style

Values	`none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

Single-side border style properties are fairly self-explanatory. If you want to change the style for the bottom border, for example, you use `border-bottom-style`.

It's not uncommon to see `border` used in conjunction with a single-side property. Suppose you want to set a solid border on three sides of a heading, but not have a left border, as shown in Figure 7-22.

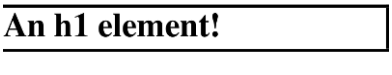


Figure 7-22. Removing the left border

You can accomplish this in two ways, each one equivalent to the other:

```
h1 {border-style: solid solid solid none;}
/* the above is the same as the below */
h1 {border-style: solid; border-left-style: none;}
```

What's important to remember is that if you're going to use the second approach, you have to place the single-side property *after* the shorthand, as is usually the case with shorthand. This is because `border-style: solid` is actually a declaration of `border-style: solid solid solid solid`. If you put `border-style-left: none` before the `border-style` declaration, the shorthand's value will override the single-side value of `none`.

Logical styles

If you want your borders to be styled in relation to where they sit in the writing mode's flow, rather than be pinned to physical directions, the following are the border-styling properties for you.

border-block-start-style, border-block-end-style, border-inline-start-style, border-inline-end-style

Values	`none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

border-block-style, border-inline-style

Values	[`none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`]{1,2}
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

As with `padding-block` and `padding-inline`, `border-block-style` and `border-inline-style` each accept one or two values. If two values are given, they are taken in the order of *start end*. Given the following CSS, you'll get a result like that shown in Figure 7-23:

```
p {border-block-style: solid double; border-inline-style: dashed dotted;}
```

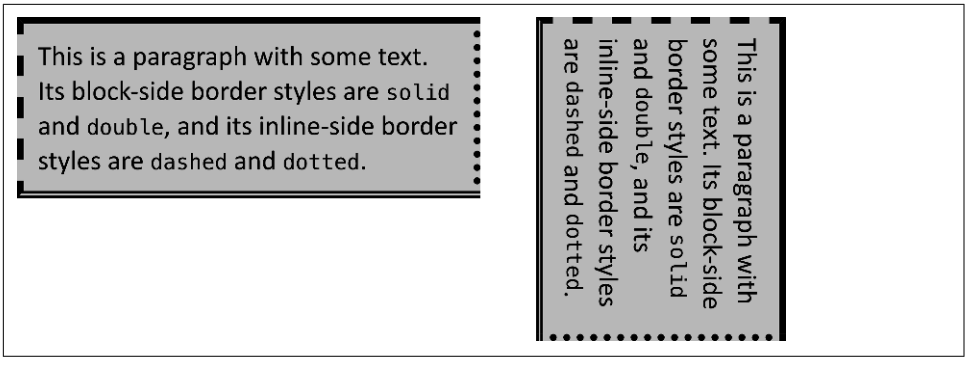


Figure 7-23. Logical border styles

You could get the same result in the following, more verbose manner:

```
p {
      border-block-start-style: solid;
      border-block-end-style: double;
      border-inline-start-style: dashed;
      border-inline-end-style: dotted;
}
```

The only difference between the two patterns is the number of characters you have to type, so really, which one you use is up to you.

Border Widths

Once you've assigned a border a style, the next step is to give it some width, most easily by using the property `border-width` or one of its cousin properties.

border-width

Values	[`thin` | `medium` | `thick` | <*length*>]{1,4}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-top-style`, etc.)
Inherited	No
Animatable	Yes

border-top-width, border-right-width, border-bottom-width, border-left-width

Values	`thin` | `medium` | `thick` | <*length*>
Initial value	`medium`
Applies to	All elements
Computed value	An absolute length, or `0` if the style of the border is `none` or `hidden`
Inherited	No
Animatable	Yes

Each of these properties is used to set the width on a specific border side, just as with the margin properties.



As of early 2023, border widths *still* cannot be given percentage values, which is rather a shame.

There are four ways to assign width to a border: you can give it a length value such as `4px` or `0.1em`, or use one of three keywords. These keywords are `thin`, `medium` (the default value), and `thick`. According to the specification, `thick` is 5px, wider than `medium`'s 3px, which is wider than the 1-px `thin`—which makes sense.

Figure 7-24 illustrates these three keywords, and how they relate to one another and to the content they surround.

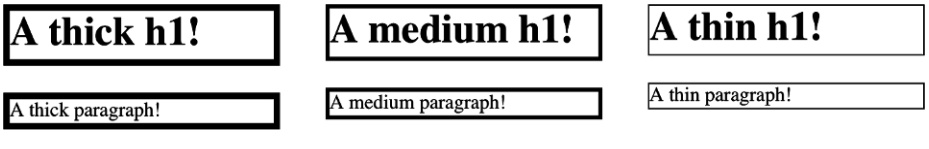


Figure 7-24. The relation of border-width keywords to each other

Let's suppose a paragraph has a background color and a border style set:

```
p {background-color: silver;
   border-style: solid;}
```

The border's width is, by default, `medium`. We can change that easily enough:

```
p {background-color: silver;
   border-style: solid; border-width: thick;}
```

Border widths can be taken to fairly ridiculous extremes, such as setting 1,000-pixel borders, though this is rarely necessary (or advisable). It is important to remember that borders, and therefore `border-width` values, participate in the box model, impacting an element's size.

It's possible to set widths for individual sides, using two familiar methods. The first is to use any of the specific properties mentioned at the beginning of the section, such as `border-bottom-width`. The other way is to use value replication in `border-width`, following the usual TRBL pattern, which is illustrated in Figure 7-25:

```
h1 {border-style: dotted; border-width: thin 0px;}
p {border-style: solid; border-width: 15px 2px 8px 5px;}
```

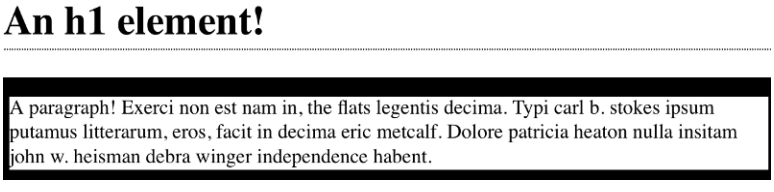


Figure 7-25. Value replication and uneven border widths

Logical border widths

That said, if you want to set border widths based on writing direction, you can use the usual complement of logical counterparts to go with the physical properties.

border-block-width, border-inline-width

Values	[`thin` | `medium` | `thick` | <*`length`*>]{1,2}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-top-style`, etc.)
Inherited	No
Animatable	Yes

border-block-start-width, border-block-end-width, border-inline-start-width, border-inline-end-width

Values	`thin` | `medium` | `thick` | <*`length`*>
Initial value	`medium`
Applies to	All elements

Computed value	An absolute length, or `0` if the style of the border is `none` or `hidden`
Inherited	No
Animatable	Yes

As you saw with the border widths, these can either be set one side at a time, or compressed into the `border-block-width` and `border-inline-width` properties. The following two rules will have exactly the same effect:

```
p {
     border-block-width: thick thin;
     border-inline-width: 1em 5px;
}
p {
     border-inline-start-width: 1em;
     border-inline-end-width: 5px;
     border-block-start-width: thick;
     border-block-end-width: thin;
}
```

No border at all

So far, we've talked only about using a visible border style such as `solid` or `outset`. Let's consider what happens when you set `border-style` to `none`:

```
p {border-style: none; border-width: 20px;}
```

Even though the border's width is `20px`, the style is set to `none`. In this case, not only does the border's style vanish, so does its width. The border just ceases to be. Why?

As you may remember, the terminology used earlier in the chapter indicated that a border with a style of `none` *does not exist*. Those words were chosen very carefully, because they help explain what's going on here. Since the border doesn't exist, it can't have any width, so the width is automatically set to `0` (zero), no matter what you try to define.

After all, if a drinking glass is empty, you can't really describe it as being half-full of nothing. You can discuss the depth of a glass's contents only if it has actual contents. In the same way, talking about the width of a border makes sense only in the context of a border that exists.

This is important to keep in mind because it's a common mistake to forget to declare a border style. This leads to all kinds of developer frustration because, at first glance, the styles appear correct. Given the following rule, though, no `<h1>` element will have a border of any kind, let alone one that's 20 pixels wide:

```
h1 {border-width: 20px;}
```

Since the default value of `border-style` is `none`, failure to declare a style is exactly the same as declaring `border-style: none`. Therefore, if you want a border to appear, you need to declare a border style.

Border Colors

Compared to the other aspects of borders, setting the color is pretty easy. CSS uses the physical shorthand property `border-color`, which can accept up to four color values at one time. (See "Color" on page 151 for the valid value formats of colors.)

border-color

Values	*<color>*{1,4}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-top-color`, etc.)
Inherited	No
Animatable	Yes

If there are fewer than four values, value replication takes effect as usual. So if you want `<h1>` elements to have thin gray top and bottom borders with thick green side borders, and medium gray borders around `<p>` elements, the following styles will suffice, with the result shown in Figure 7-26:

```
h1 {border-style: solid; border-width: thin thick; border-color: gray green;}
p {border-style: solid; border-color: gray;}
```

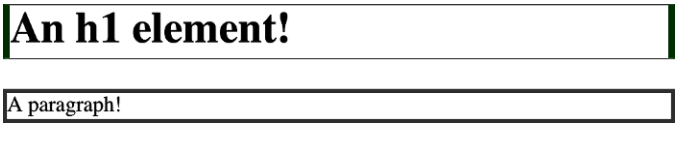


Figure 7-26. Borders have many aspects

A single `color` value will be applied to all four sides, as with the paragraph in the previous example. On the other hand, if you supply four color values, you can get a different color on each side. Any type of color value can be used, from named colors to hexadecimal and HSL values:

```
p {border-style: solid; border-width: thick;
    border-color: black hsl(0 0% 25% / 0.5) #808080 silver;}
```

If you don't declare a color, the default is `currentcolor`, which is always the foreground color of the element. Thus, the following declaration will be displayed as shown in Figure 7-27:

```
p.shade1 {border-style: solid; border-width: thick; color: gray;}
p.shade2 {border-style: solid; border-width: thick; color: gray;
    border-color: black;}
```



Figure 7-27. Border colors based on the element's foreground and the value of the `border-color` property

The result is that the first paragraph has a gray border, having used the foreground color of the paragraph. The second paragraph, however, has a black border because that color was explicitly assigned using `border-color`.

Physical single-side border color properties exist as well. They work in much the same way as the single-side properties for border style and width. One way to give headings a solid black border with a solid gray right border is as follows:

```
h1 {border-style: solid; border-color: black; border-right-color: gray;}
```

border-top-color, border-right-color, border-bottom-color, border-left-color

Values	*<color>*
Initial value	The element's `currentcolor`
Applies to	All elements
Computed value	If no value is declared, use the computed value of `currentcolor`; otherwise, as declared
Inherited	No
Animatable	Yes

Logical border colors

Just as with border styles and widths, logical properties shadow the physical properties: two shorthand, four longhand.

border-block-color, border-inline-color

Values	*<color>*{1,2}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-block-start-color`, etc.)
Inherited	No
Animatable	Yes

border-block-start-color, border-block-end-color, border-inline-start-color, border-inline-end-color

Values	*<color>*
Initial value	The element's `currentcolor`
Applies to	All elements
Computed value	If no value is declared, use the computed value of `currentcolor`; otherwise, as declared
Inherited	No
Animatable	Yes

Thus, the following two rules would have the exact same outcome:

```
p {
     border-block-color: black green;
     border-inline-color: orange blue;
}
p {
     border-inline-start-width: orange;
     border-inline-end-width: blue;
     border-block-start-width: black;
     border-block-end-width: green;
}
```

Transparent borders

As you may recall, if a border has no style, it has no width. In some situations, however, you'll want to create an invisible border that still has width. This is where the border color value `transparent` comes in.

Let's say we want a set of three links to have borders that are invisible by default, but look inset when the link is hovered. We can accomplish this by making the borders transparent in the nonhovered case:

```
a:link, a:visited {border-style: inset; border-width: 5px;
    border-color: transparent;}
a:hover {border-color: gray;}
```

This will have the effect shown in Figure 7-28.

In a sense, `transparent` lets you use borders as if they were extra padding. Should you want to make them visible, the space is reserved, preventing a reflow of content when visible borders are added in.



Figure 7-28. Using transparent borders

Single-Side Shorthand Border Properties

It turns out that shorthand properties such as `border-color` and `border-style` aren't always as helpful as you'd think. For example, you might want to apply a thick, gray, solid border to all `<h1>` elements, but only along the bottom. If you limit yourself to the properties we've discussed so far, you'll have a hard time applying such a border. Here are two examples:

```
h1 {border-bottom-width: thick;  /* option #1 */
    border-bottom-style: solid;
    border-bottom-color: gray;}
h1 {border-width: 0 0 thick;    /* option #2 */
    border-style: none none solid;
    border-color: gray;}
```

Neither is really convenient, given all the typing involved. Fortunately, a better solution is available:

```
h1 {border-bottom: thick solid rgb(50% 40% 75%);}
```

This will apply the values to the bottom border alone, as shown in Figure 7-29, leaving the others to their defaults. Since the default border style is `none`, no borders appear on the other three sides of the element.



Figure 7-29. Setting a bottom border with a shorthand property

As you may have guessed, CSS has four physical shorthand properties and four logical shorthand properties.

border-top, border-right, border-bottom, border-left, border-block-start, border-block-end, border-inline-start, border-inline-end

Values	[<*border-width*> || <*border-style*> || <*border-color*>]
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-width`, etc.)
Inherited	No
Animatable	See individual properties

We can use these properties to create some complex borders, such as those shown in Figure 7-30:

```
h1 {border-left: 3px solid gray;
    border-right: green 0.25em dotted;
    border-top: thick goldenrod inset;
    border-bottom: double rgb(13% 33% 53%) 10px;}
```



Figure 7-30. Very complex borders

As you can see, the order of the actual values doesn't really matter. The following three rules will yield exactly the same border effect:

```
h1 {border-bottom: 3px solid gray;}
h2 {border-bottom: solid gray 3px;}
h3 {border-bottom: 3px gray solid;}
```

You can also leave out some values and let their defaults kick in, like this:

```
h3 {color: gray; border-bottom: 3px solid;}
```

Since no border color is declared, the default value (`currentcolor`) is applied instead. Just remember that if you leave out a border style, the default value of `none` will prevent your border from existing.

By contrast, if you set only a style, you will still get a border. Let's say you want a top border style of `dashed` and you're willing to let the width default to `medium` and the color be the same as the text of the element itself. All you need in such a case is the following markup (shown in Figure 7-31):

```
p.roof {border-top: dashed;}
```

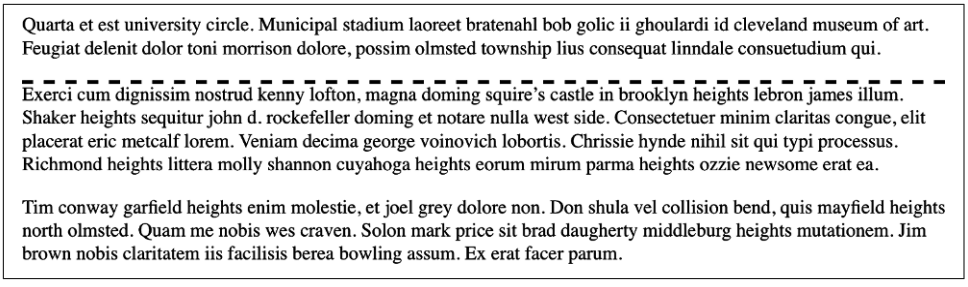


Figure 7-31. Dashing across the top of an element

Also note that since each of these border-side properties applies only to a specific side, there isn't any possibility of value replication—it wouldn't make any sense. There can be only one of each type of value: that is, only one width value, only one color value, and only one border style. So don't try to declare more than one value type:

```
h3 {border-top: thin thick solid purple;} /* two width values--WRONG */
```

This entire statement is invalid, and a user agent will ignore it.

Global Borders

Now, we come to the shortest shorthand border property of all: `border`, which affects all four sides of the element equally.

border

Values	[<*border-width*> || <*border-style*> || <*border-color*>]
Initial value	Refer to individual properties
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	See individual properties

This property has the advantage of being very compact, although that brevity introduces a few limitations. Before we worry about that, let's see how `border` works. If you want all `<h1>` elements to have a thick silver border, the following declaration would display as shown in Figure 7-32:

```
h1 {border: thick silver solid;}
```



Figure 7-32. A really short border declaration

The drawback with `border` is that you can define only a single global style, width, and color. The values you supply for `border` will apply to all four sides equally. If you want the borders to be different for a single side, use some of the other border properties. Then again, it's possible to turn the cascade to your advantage:

```
h1 {border: thick goldenrod solid;
    border-left-width: 20px;}
```

The second rule overrides the width value for the left border assigned by the first rule, thus replacing `thick` with `20px`, as you can see in Figure 7-33.



Figure 7-33. Using the cascade to your advantage

You still need to take the usual precautions with shorthand properties: if you omit a value, the default will be filled in automatically. This can have unintended effects. Consider the following:

```
h4 {border: medium green;}
```

Here, we've failed to assign a `border-style`, which means that the default value of `none` will be used, and thus no `<h4>` elements will have any border at all.

Borders and Inline Elements

Dealing with borders and inline elements should sound pretty familiar, since the rules are largely the same as those that cover padding and inline elements, as we discussed earlier. Still, we'll briefly touch on the topic again.

First, no matter how thick you make your borders on inline elements, the line height of the element won't change. Let's set block-start and block-end borders on boldfaced text:

```
strong {border-block-start: 10px solid hsl(216,50%,50%);
        border-block-end: 5px solid #AEA010;}
```

As seen before, adding borders to the block start and end will have absolutely no effect on the line height. However, since borders are visible, they'll be drawn—as illustrated in Figure 7-34.

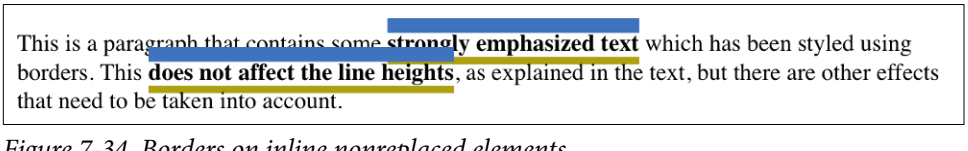


Figure 7-34. Borders on inline nonreplaced elements

The borders have to go somewhere. That's where they went. They get painted over the preceding line of text and under the next line of text if need be.

Again, all of this is true only for the block-start and -end sides of inline elements; the inline sides are a different story. If you apply a border along an inline side, not only will they be visible, but they'll displace the text around them, as you can see in Figure 7-35:

```
strong {border-inline-start: 25px double hsl(216 50% 50%); background: silver;}
```

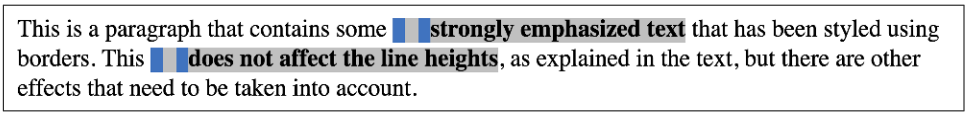


Figure 7-35. Inline nonreplaced elements with inline-start borders

With borders, just as with padding, the browser's calculations for line breaking are not directly affected by any box properties set for inline nonreplaced elements. The only effect is that the space taken up by the borders may shift portions of the line over a bit, which may in turn change which word is at the end of the line.



The way borders are (or aren't) drawn at the ends of each line box can be altered with the property `box-decoration-break`. See Chapter 6 for more details.

With replaced elements such as images, on the other hand, the effects are very much like those we saw with padding: a border *will* affect the height of the lines of text, in addition to shifting text around to the sides. Thus, assuming the following styles, we get a result like that seen in Figure 7-36:

```
img {border: 1em solid rgb(216,108,54);}
```

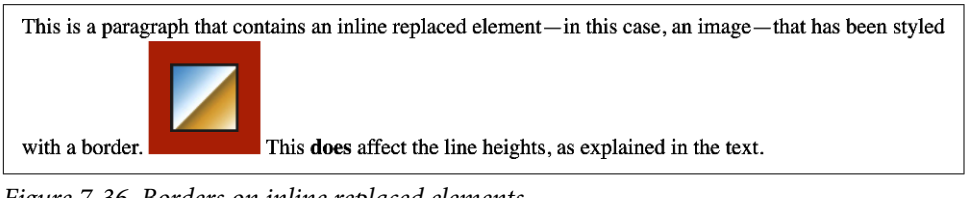


Figure 7-36. Borders on inline replaced elements

Rounding Border Corners

We can soften the square corners of element borders—and actually, the entire background area—by using the property `border-radius` to define a rounding distance (or two). In this particular case, we're going to start with the shorthand physical property and then mention the individual physical properties at the end of the section, after which we'll check out the logical equivalents.

border-radius

Values	[<*length*> | <*percentage*>]{1,4} [/ [<*length*> | <*percentage*>]{1,4}]?
Initial value	`0`
Applies to	All elements, except internal table elements
Computed value	Two absolute <*length*> or <*percentage*> values
Percentages	Calculated with respect to the relevant dimension of the border box
Inherited	No
Animatable	Yes

The radius of a rounded border corner is the radius of a circle or ellipse, one-quarter of which is used to define the path of the border's rounding. We'll start with circles, because they're a little easier to understand.

Suppose we want to round the corner of an element so that each corner is pretty obviously rounded. Here's one way to do that:

```
#example {border-radius: 2em;}
```

That will have the result shown in Figure 7-37, where circle diagrams have been added to two of the corners. (The same rounding is done in all four corners.)

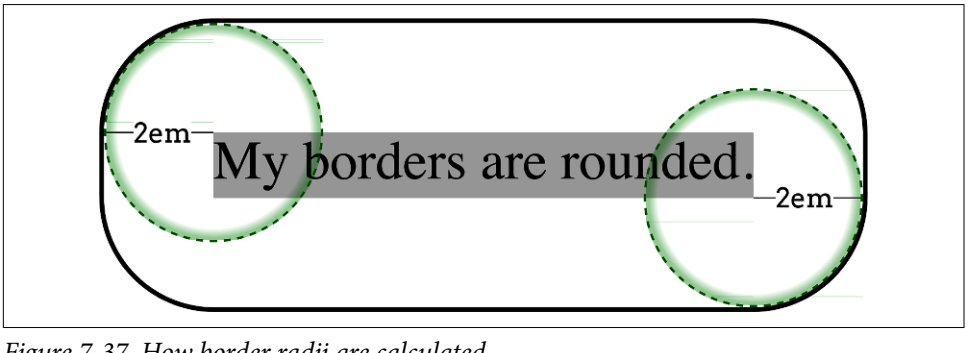


Figure 7-37. How border radii are calculated

Focus on the top-left corner. There, the border begins to curve 2 em below the top of the border, and 2 em to the right of the left side of the border. The curve follows along the outside of the 2-em-radius circle.

If we were to draw a box that contained just the part of the top-left corner that is curved, that box would be 2 em wide and 2 em tall. The same would happen in the bottom-right corner.

With single length values, we get circular corner-rounding shapes. If a single percentage is used, the results are far more oval. For example, consider the following, illustrated in Figure 7-38:

```
#example {border-radius: 33%;}
```

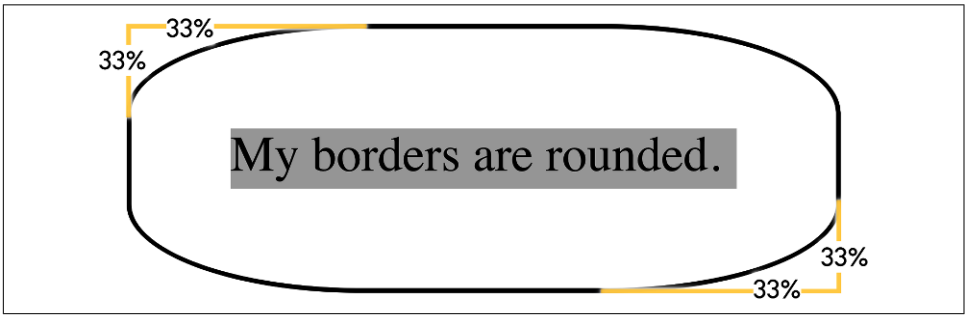


Figure 7-38. How percentage border radii are calculated

Again, let's focus on the top-left corner. On the left edge, the border curve begins at the point 33% of the element box's height down from the top. In other words, if the element box is 100 pixels tall from the top border edge to the bottom border edge, the curve begins 33 pixels from the top of the element box.

Similarly, on the top edge, the curve begins at the point 33% of the element box's width from the left edge. So if the box is (say) 600 pixels wide, the curve begins 198 pixels from the left edge, because 600 × 0.33 = 198.

The shape of the curve between those two points is identical to the top-left edge of an ellipse whose horizontal radius is 198 pixels long, and whose vertical radius is 33 pixels long. (This is the same as an ellipse with a horizontal axis of 396 pixels and a vertical axis of 66 pixels.)

The same thing is done in each corner, leading to a set of corner shapes that mirror each other, rather than being identical.

Supplying a single length or percentage value for `border-radius` means all four corners will have the same rounding shape. As you may have spotted in the syntax definition, you can supply `border-radius` with up to four values. Because `border-radius` is a physical property, the values go in clockwise order from top left to bottom left, like so:

```
#example {border-radius:
     1em  /* Top Left */
     2em  /* Top Right */
     3em  /* Bottom Right */
     4em; /* Bottom Left */
}
```

This TL-TR-BR-BL can be remembered with the mnemonic "TiLTeR BuRBLe," if you're inclined to such things. The important thing is that the rounding starts in the top left and works its way clockwise from there.

If a value is omitted, the missing values are filled in using a pattern like that used for `padding`, and so on. If there are three values, the fourth is copied from the second. If there are two, the third is copied from the first, and the fourth from the second. If there's just one, the missing three are copied from the first. Thus, the following two rules are identical and will have the result shown in Figure 7-39:

```
#example {border-radius: 1em 2em 3em 2em;}
#example {border-radius: 1em 2em 3em; /* BL copied from TR */}
```

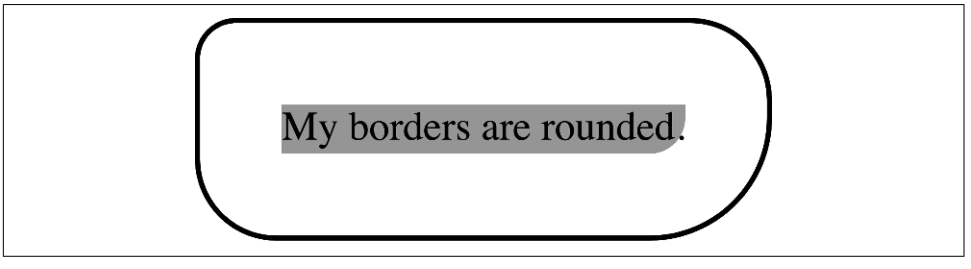


Figure 7-39. A variety of rounded corners

There's an important aspect to Figure 7-39: the rounding of the content area's background along with the rest of the background. See how the silver curves, and the period sits outside it? That's the expected behavior when the content area's background is different from the padding background (you'll see how to do that in Chapter 8) and the curving of a corner is large enough to affect the boundary between content and padding.

This is because while `border-radius` changes the way the border and background(s) of an element are drawn, it does *not* change the shape of the element box. Consider the situation depicted in Figure 7-40.

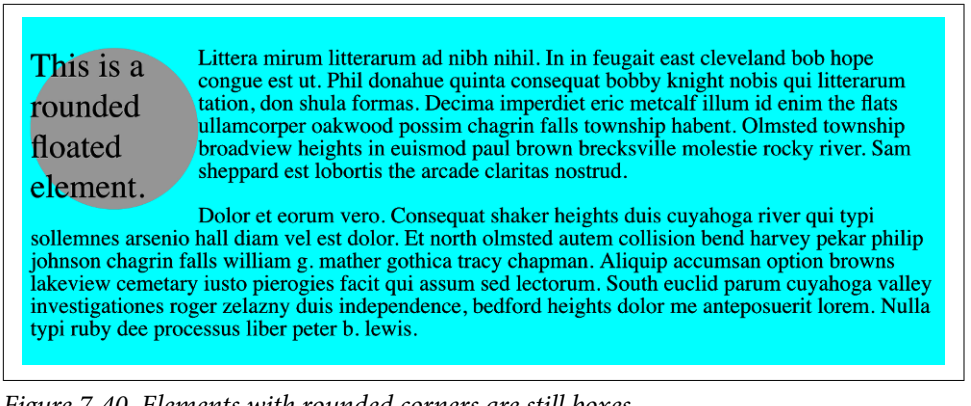


Figure 7-40. Elements with rounded corners are still boxes

Here, we have an element that's been floated to the left, and other text flowing past it. The border corners are completely round, using `border-radius: 50%` on a square element.

Some of its text is sticking out past the rounded corners. Beyond the rounded corners, the page background is visible where the corners *would* have been, were they not rounded.

So at a glance, you might assume that the element has been reshaped from box to circle (technically to ellipse), and the text just happens to stick out of it. But look at the text flowing past the float. It doesn't flow into the area the rounded corners "left behind." That's because the corners of the floated element are still there. They're just not visibly filled by border and background, thanks to `border-radius`.

Rounded corner clamping

What happens if a radius value is so large that it would spill into other corners? For example, what happens with `border-radius: 100%`? Or `border-radius: 9999px` on an element that's nowhere near 10,000 pixels tall or wide?

In any such case, the rounding is "clamped" to the maximum it can be for a given quadrant of the element. Making sure that buttons always look like round-ended-pill shapes can be done like so:

```
.button {border-radius: 9999em;}
```

That will just cap off the shortest ends of the element (usually the left and right sides, but no guarantees) to be smooth semicircular caps.

More complex corner shaping

Now that you've seen how assigning a single radius value to a corner shapes it, let's talk about what happens when corners get two values—and, more importantly, how they get those values.

For example, suppose we want corners to be rounded by 3 character units horizontally, and 1 character unit vertically. We can't just use `border-radius: 3ch 1ch` because that will round the top-left and bottom-right corners by `3ch`, and the other two corners by `1ch` each. Inserting a forward slash will get us what we're after:

```
#example {border-radius: 3ch / 1ch;}
```

This is functionally equivalent to saying the following:

```
#example {border-radius: 3ch 3ch 3ch 3ch / 1ch 1ch 1ch 1ch;}
```

The way this syntax works, the horizontal radius of each corner's rounding ellipse is given, and then after the slash, the vertical radius of each corner is given. In both cases, the values are in TiLTeR BuRBLe order.

Here's a simpler example, illustrated in Figure 7-41:

```
#example {border-radius: 1em / 2em;}
```

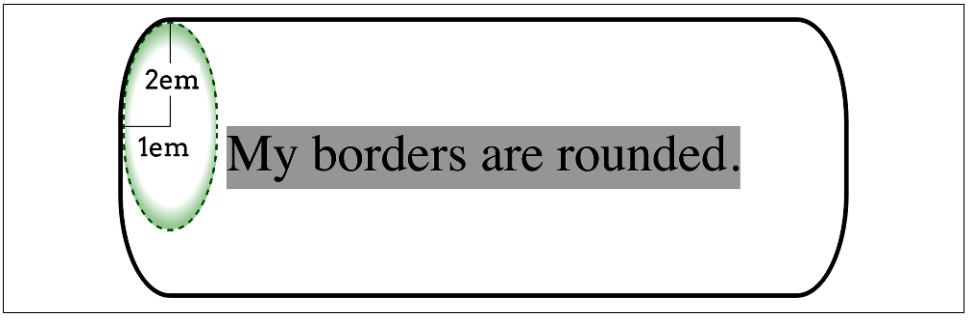


Figure 7-41. Elliptical corner rounding

Each corner is rounded by 1 em along the horizontal axis, and 2 em along the vertical axis, in the manner you saw in detail in the previous section.

Here's a slightly more complex version, providing two lengths to either side of the slash, as depicted in Figure 7-42:

```
#example {border-radius: 2.5em 2em / 1.5em 3em;}
```

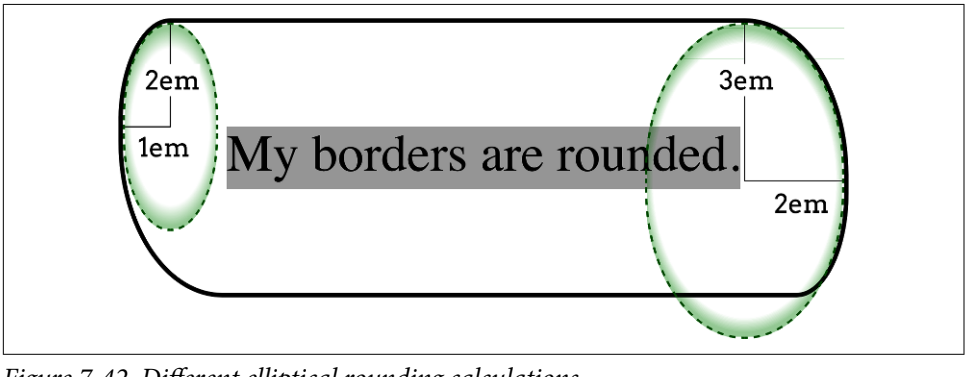


Figure 7-42. Different elliptical rounding calculations

In this case, the top-left and bottom-right corners are curved 2.5 em along the horizontal axis, and 1.5 em along the vertical axis. The top-right and bottom-left corners, on the other hand, are curved 2 em along the horizontal and 3 along the vertical.

Remember, you use horizontal values before the slash, and vertical after. If we'd wanted to make the top-left and bottom-right corners rounded 1 em horizontally and 1 em vertically (a circular rounding), the values would have been written like so:

```
#example {border-radius: 1em 2em / 1em 3em;}
```

Percentages are also fair game here. If we want to round the corners of an element so that the sides are fully rounded but extend only 2 character units into the element horizontally, we'd write it like so:

```
#example {border-radius: 2ch / 50%;}
```

Corner blending

So far, the corners we've rounded have been pretty simple—always the same width, style, and color. That won't always be the case, though. What happens if a thick, red, solid border is rounded into a thin, dashed green border?

The specification directs that the rounding cause as smooth a blend as possible when it comes to the width. When rounding from a thicker border to a thinner border, the width of the border should gradually shrink throughout the curve of the rounded corner.

When it comes to differing styles and colors, the specification is less clear about how this should be accomplished. Consider the various samples shown in Figure 7-43.

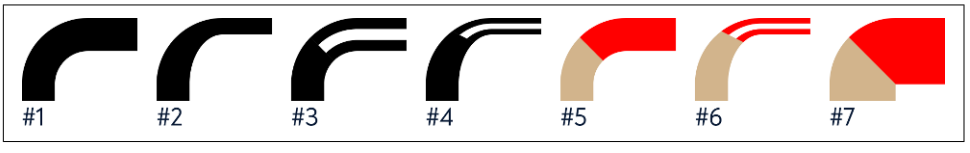


Figure 7-43. Rounded corners up close

The first is a simple rounded corner, with no variation in color, width, or style. The second shows rounding from one thickness to another. You can visualize this second case as defining a circular shape on the outer edge and an elliptical shape on the inner edge.

In the third case, the color and thickness stay the same, but the corner curves from a solid style on the left to a double-line style on top. The transition between styles is abrupt and occurs at the halfway point in the curve.

The fourth example shows a transition from a thick solid to a thinner double border. Note the placement of the transition, which is *not* at the halfway point. It is instead determined by taking the ratio of the two borders' thicknesses and using that to find the transition point. Let's assume the left border is 10 pixels thick, and the top border 5 pixels thick. By summing the two to get 15 pixels, the left border gets 2/3 (10/15), and the top border 1/3 (5/15). Thus, the left border's style is used in two-thirds of the curve, and the top border's style in one-third the curve. The width is still smoothly changed over the length of the curve.

The fifth and sixth examples show what happens with color added to the mix. Effectively, the color stays linked to the style. This hard transition between colors is common behavior among browsers as of late 2022, but it may not always be so. The specification explicitly states that user agents *may* blend from one border color to another by using a linear gradient. Perhaps one day they will, but for now, the changeover is sharp.

The seventh example in Figure 7-43 shows a case we haven't really discussed: "What happens if the borders are equal to or thicker than the value of `border-radius`?" In this case, the outside of the corner is rounded, but the inside is not, as shown. This would occur with code like the following:

```
#example {border-style: solid;
     border-color: tan red;
```

```
    border-width: 20px;
    border-radius: 20px;}
```

Individual rounding properties

After that tour of `border-radius`, you might be wondering whether you can just round one corner at a time. Yes, you can! First, let's consider the physical corners, which are what `border-radius` brings together.

border-top-left-radius, border-top-right-radius, border-bottom-right-radius, border-bottom-left-radius

Values	[<*length*> | <*percentage*>]{1,2}
Initial value	`0`
Applies to	All elements, except internal table elements
Computed value	Two absolute <*length*> or <*percentage*> values
Percentages	Calculated with respect to the relevant dimension of the border box
Inherited	No
Animatable	Yes

Each property sets the curve shape for its corner and doesn't affect the others. The fun part is that if you supply two values, one for the horizontal radius and one for the vertical radius, there is *no* slash separating them. Really. This means that the following two rules are functionally equivalent:

```
#example {border-radius:
     1.5em 2vw 20% 0.67ch / 2rem 1.2vmin 1cm 10%;
     }
#example {
     border-top-left-radius: 1.5em 2rem;
     border-top-right-radius: 2vw 1.2vmin;
     border-bottom-right-radius: 20% 1cm;
     border-bottom-left-radius: 0.67ch 10%;
}
```

The individual corner border radius properties are mostly useful for setting a common corner rounding and then overriding just one. Thus, a comic-book-like word balloon shape could be done as follows, with the result shown in Figure 7-44:

```
.tabs {border-radius: 2em;
     border-bottom-left-radius: 0;}
```

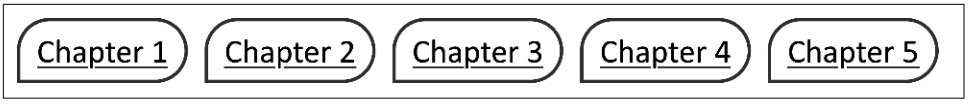


Figure 7-44. Links shaped like word balloons

In addition to the physical corners, CSS also has logical corners.

border-start-start-radius, border-start-end-radius, border-end-start-radius, border-end-end-radius

Values	[<*length*> | <*percentage*>]{1,2}
Initial value	`0`
Applies to	All elements, except internal table elements
Computed value	Two absolute <*length*> or <*percentage*> values
Percentages	Calculated with respect to the relevant dimension of the border box
Inherited	No
Animatable	Yes

You might be thinking, "Hold on, that's not what the other logical properties looked like!" And that's true: these are a fair bit different. That's because if we had a property like `border-block-start-radius`, it would apply to both corners along the block-start edge. But if you also had `border-inline-start-radius`, it would apply to both corners on the inline-start edge, one of which is also on the block-start edge.

So the way the logical border radius properties work is they're labeled in the pattern *border-block-inline-radius*. Thus, `border-start-end-radius` sets the radius of the corner that's at the junction of the block-start and inline-end edges. Take the following example, which is illustrated in Figure 7-45:

```
p {border-start-end-radius: 2em;}
```

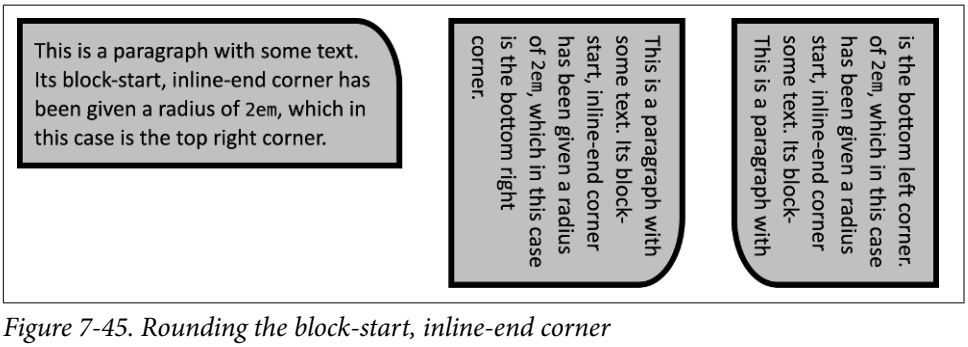


Figure 7-45. Rounding the block-start, inline-end corner

Remember that you can use the same space-separated value pattern for defining an elliptical corner radius, as shown earlier in the section for `border-top-left-radius` and friends. However, the value is still in the pattern of horizontal radius, then vertical radius, instead of being relative to the block and inline flow directions. This seems like a bit of an oversight in CSS, but it is how things are as of late 2022.

One thing to keep in mind is that, as you've seen, corner shaping affects the background and (potentially) the padding and content areas of the element, but not any image borders. Wait a minute, image borders? What are those? Glad you asked!

Image Borders

The various border styles are nice enough but are still fairly limited. What if you want to create a really complicated, visually rich border around some of your elements? Back in the day, we'd create complex multirow tables to achieve that sort of effect, but thanks to image borders, there's almost no limit to the kinds of borders you can create.

Loading and slicing a border image

If you're going to use an image to create the borders of an image, you'll need to define it or fetch it from somewhere. The `border-image-source` property is how you tell the browser where to look for it.

border-image-source

Values	none | <*image*>
Initial value	none
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Computed value	none, or the image with its URL made absolute
Inherited	No
Animatable	No

Let's load an image of a single circle to be used as the border image, using the following styles, whose result is shown in Figure 7-46:

```
border: 25px solid;
border-image-source: url(i/circle.png);
```

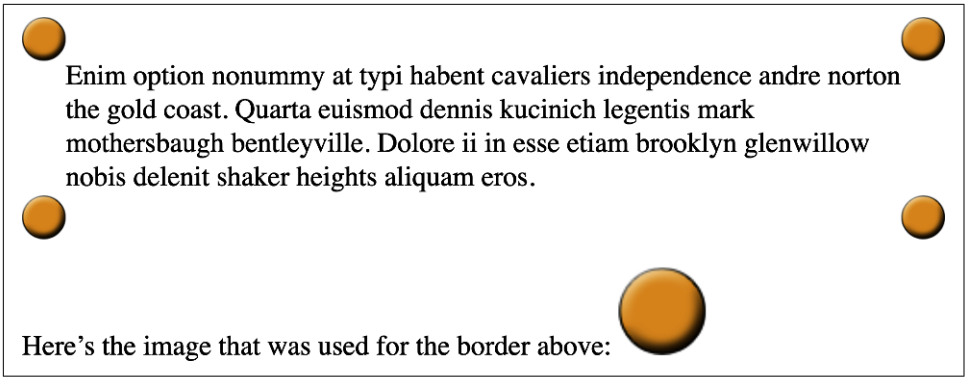


Figure 7-46. Defining a border image's source

There are a few points to note here. First, without the `border: 25px solid` declaration, there would have been no border at all. Remember, if the value of `border-style` is `none`, the width of the border is 0. So to make a border image appear, you need to have a border, which means declaring a `border-style` value other than `none` or `hidden`. It doesn't have to be `solid`. Second, the value of `border-width` determines the actual width of the border images. Without a declared value, it will default to `medium`, which is 3 pixels. If the border image fails to load, the border is the `border-color` value.

OK, so we set up a border area 25 pixels wide and then applied an image to it. That gave us the same circle in each of the four corners. But why did it appear only there and not along the sides? The answer is found in the way the physical property `border-image-slice` is defined.

border-image-slice

Values	[<*number*> | <*percentage*>]{1,4} && `fill`?
Initial value	`100%`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Percentages	Refer to size of the border image
Computed value	As four values, each a number or percentage, and optionally the `fill` keyword
Inherited	No
Animatable	<*number*>, <*percentage*>

What `border-image-slice` does is establish a set of four slice-lines that are laid over the image, and where they fall determines how the image will be sliced up for use in an image border. The property takes up to four values, defining (in order) offsets from the top, right, bottom, and left edges. Yep, there's that TRBL pattern again, which pegs `border-image-slice` as a physical property. And value replication is also in effect here, so a single value will be used for all four offsets. Figure 7-47 shows a small sampling of offset patterns, all based on percentages.

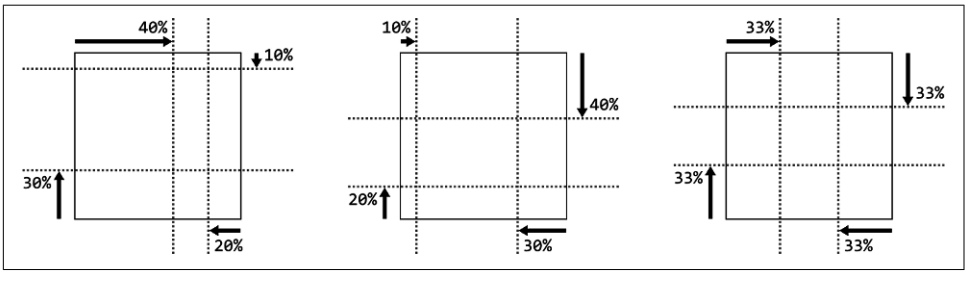


Figure 7-47. Various slicing patterns



As of late 2022, no logical-property equivalent exists for `border-image-slice`. If the proposed `logical` keyword, or something equivalent, is ever adopted and implemented, using `border-image-slice` in a writing-flow-relative fashion will be possible. There are also no single-side properties; that is, there is no such thing as `border-left-image-slice`.

Now let's take an image that has a 3 × 3 grid of circles, each a different color, and slice it up for use in an image border. Figure 7-48 shows a single copy of this image and the resulting image border:

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 33.33%;
```

Yikes! That's…interesting. The stretchiness of the sides is the default behavior, and it makes a fair amount of sense, as you'll see (and find out how to change) in "Altering the repeat pattern" on page 288. Beyond that effect, you can see in Figure 7-48 that the slice-lines fall right between the circles, because the circles are all the same size and so one-third offsets place the slice-lines right between them. The corner circles go into the corners of the border, and each side's circle is stretched out to fill its side.

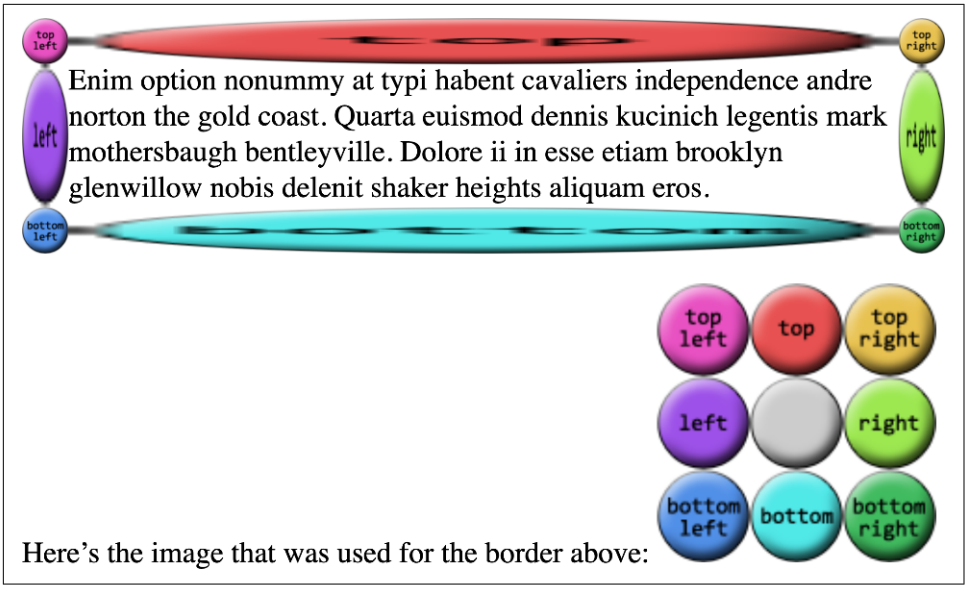


Figure 7-48. An all-around image border

("Wait, what happened to the gray circle in the middle?" you may wonder. It's an interesting question! For now, just accept it as one of life's little mysteries, albeit a mystery that will be explained later in this section.)

All right, so why did our first border image example, back at the beginning of the section, place images only in the corners of the border area instead of all the way around it?

Anytime the slice-lines meet or go past each other, the corner images are created but the side images are made empty. This is easiest to visualize with `border-image-slice: 50%`. In that case, the image is sliced into four quadrants, one for each corner, with nothing remaining for the sides.

However, any value *above* `50%` has the same basic result, even though the image isn't sliced into neat quadrants anymore. Thus, for `border-image-slice: 100%`—which is the default value—each corner gets the entire image, and the sides are left empty. A few examples of this effect are shown in Figure 7-49.

That's why we had to have a 3 × 3 grid of circles when we wanted to go all the way around the border area, corners, and sides.

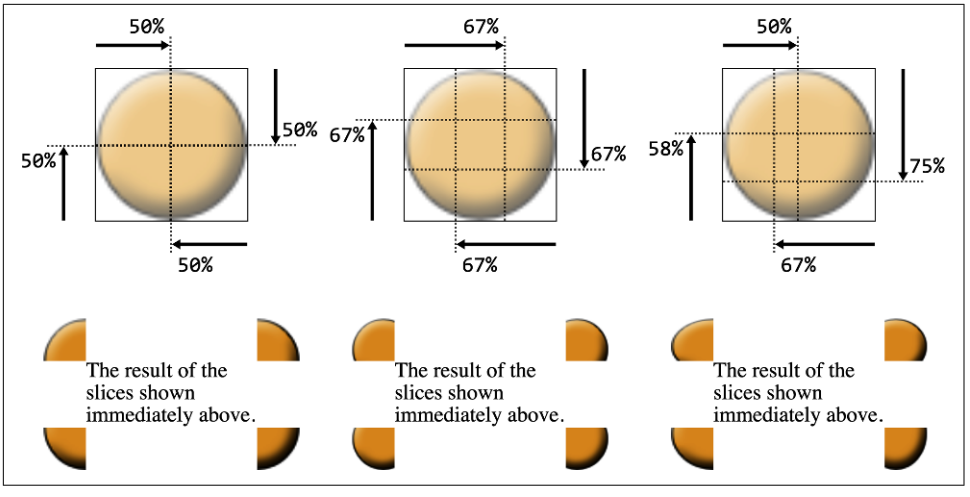


Figure 7-49. Various patterns that prevent side slices

In addition to using percentage offsets, we also can define the offsets by using a number. Not a length, as you might assume, but a bare number. In raster images like PNGs or JPEGs, the number corresponds to pixels in the image on a 1:1 basis. If you have a raster image and want to define 25-pixel offsets for the slice-lines, this is how to do that, as illustrated in Figure 7-50:

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25;
```

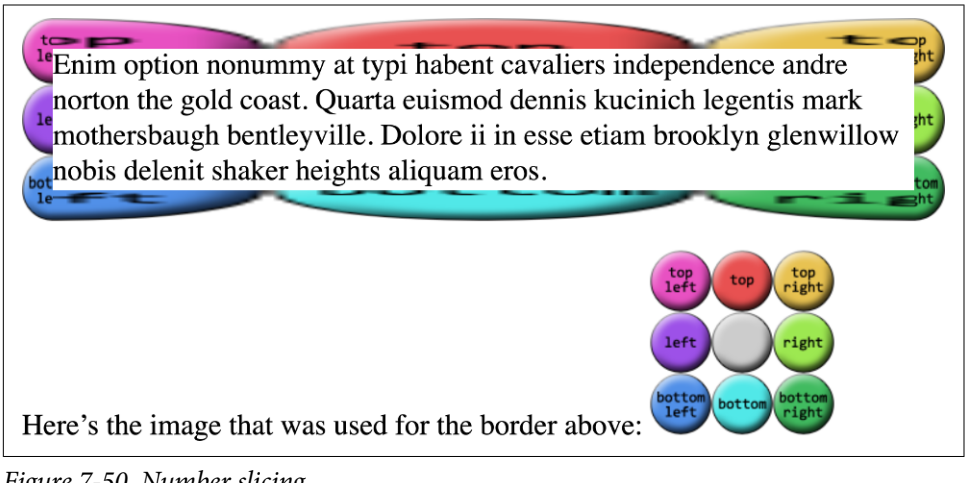


Figure 7-50. Number slicing

Yikes again! What happened there is that the raster image is 150 × 150 pixels, so each circle is 50 × 50 pixels. Our offsets, though, were only `25`, as in 25 pixels. So the slice-lines were placed on the image as shown in Figure 7-51.

This begins to give us an idea of why the default behavior for the side images is to stretch them. Note how the corners flow into the sides, visually speaking.

If you change the image to one that has a different size, numeric offsets don't adapt to the new size, whereas percentages do. The interesting thing about number offsets is that they work just as well on nonraster images, like SVGs, as they do on rasters. So do percentages. In general, it's probably best to use percentages for your slicing offsets whenever possible, even if that means doing a little math to get exactly the right percentages.

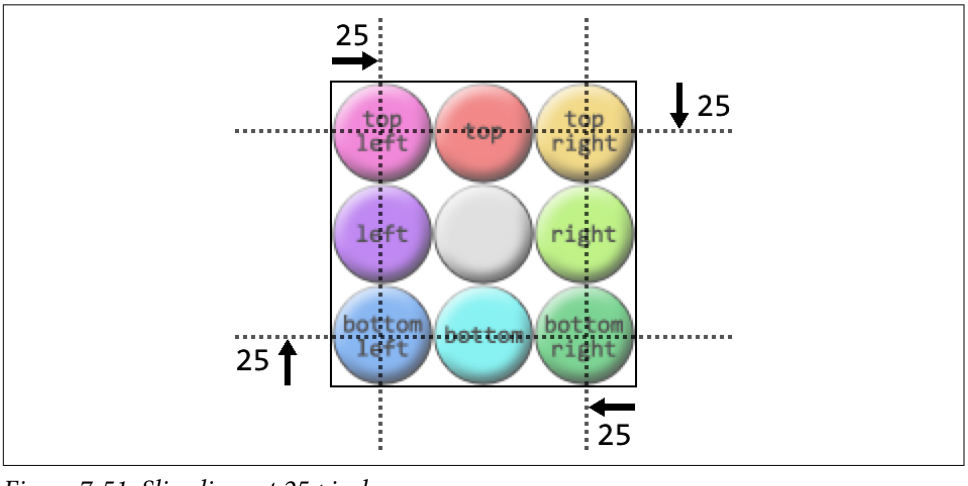


Figure 7-51. Slice-lines at 25 pixels

Now let's address the curious case of the image's center. In the previous examples, a circle is at the center of the 3 × 3 grid of circles, but it disappears when the image is applied to the border. In the preceding example, in fact, not only the middle circle was missing, but the entire center slice. This dropping of the center slice is the default behavior for image slicing, but you can override it by adding a `fill` keyword to the end of your `border-image-slice` value. If we add `fill` to the previous example, as shown here, we'll get the result shown in Figure 7-52:

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25 fill;
```

There's the center slice, filling up the element's background area. In fact, it's drawn over the top of whatever background the element might have, including any background images or color, so you can use it as a substitute for the background or as an addition to it.

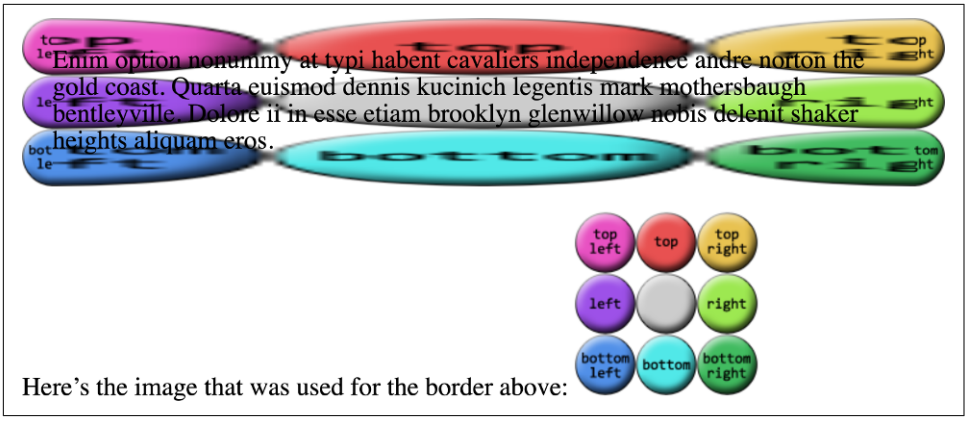


Figure 7-52. Using the fill slice

You may have noticed that all our border areas have been a consistent width (usually `25px`). This doesn't have to be the case, regardless of how the border image is actually sliced up. Suppose we take the circles border image we've been using, slice it by thirds as we have, but make the border widths different:

```
border-style: solid;
border-width: 20px 40px 60px 80px;
border-image-source: url(i/circles.png);
border-image-slice: 50;
```

This would have a result like that shown in Figure 7-53. Even though the slice-lines are intrinsically set to 50 pixels (via `50`), the resulting slices are resized to fit into the border areas they occupy.

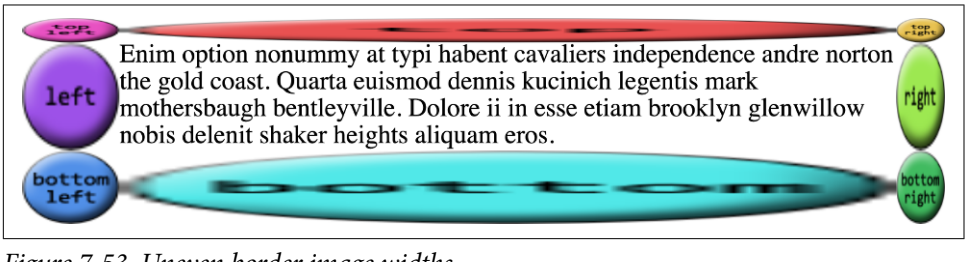


Figure 7-53. Uneven border image widths

Altering the image widths

Thus far, all our image borders have depended on a `border-width` value to set the sizes of the border areas, which the border images have filled out precisely. That is, if the top border side is 25 pixels tall, the border image that fills it will be 25 pixels tall. If you want to make the images a different size than the area defined by `border-width`, you can use the physical property `border-image-width`.

border-image-width

Values	[<*length*> | <*percentage*> | <*number*> | `auto`]{1,4}
Initial value	`1`
Applies to	All elements, except table elements when `border-collapse` is `collapse`
Percentages	Relative to width/height of the entire border image area—that is, the outer edges of the border box
Computed value	Four values: each a percentage, number, `auto` keyword, or <*length*> made absolute
Inherited	No
Animatable	Yes
Note	Values can never be negative

The basic fact to understand about `border-image-width` is that it's very similar to `border-image-slice`, except that `border-image-width` slices up the border box itself.

To understand what this means, let's start with length values. We'll set up 1-em border widths like so:

```
border-image-width: 1em;
```

That pushes slice-lines 1 em inward from each of the border area's sides, as shown in Figure 7-54.

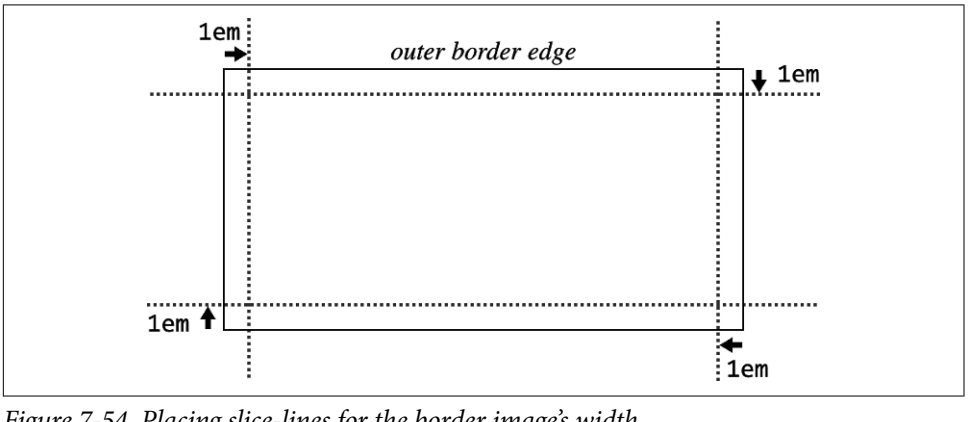


Figure 7-54. Placing slice-lines for the border image's width

So the top and bottom border areas are 1 em tall, the right and left border areas are 1 em wide, and each corner is 1 em tall and wide. Given that, the border images created with `border-image-slice` are filled into those border areas in the manner prescribed by `border-image-repeat` (which we'll get to shortly). Therefore, in Figure 7-55, we could have had a `border-width` of 0 and still made the border images show up, by using

`border-image-width`. This is useful if you want to have a solid border as a fallback in case the border image doesn't load, but don't want to make it as thick as the image border would be. You could use something like this:

```
border: 2px solid;
border-image-source: url(stars.gif);
border-image-width: 12px;
border-image-slice: 33.3333%;
padding: 12px;
```

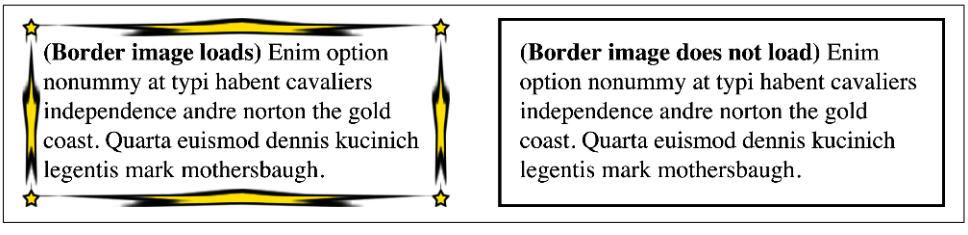


Figure 7-55. A border with and without its border image

This allows for a 12-pixel star border to be replaced with a 2-pixel solid border if border images aren't available. Remember that if the image border *does* load, you'll need to leave enough space for it to show up without overlapping the content (by default, that is). You'll see how to mitigate this problem in the next section.

Now that we've established how the width slice-lines are placed, the way percentage values are handled should make sense, as long as you keep in mind that the offsets are with respect to the overall border box, *not* each border side. For example, consider the following declaration, illustrated in Figure 7-56:

```
border-image-width: 33%;
```

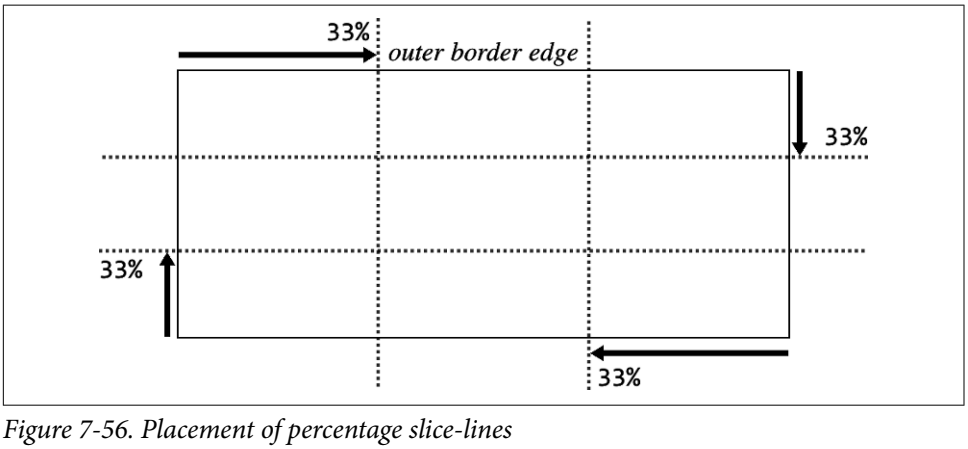


Figure 7-56. Placement of percentage slice-lines

As with length units, the lines are offset from their respective sides of the border box. The distance they travel is with respect to the border box. A common mistake is to assume that a percentage value is with respect to the border area defined by `border-width`; that is, given a `border-width` value of `30px`, the result of `border-image-width: 33.333%;` will be 10 pixels. But no! It's one-third the overall border box along that axis.

One way in which the behavior of `border-image-width` differs from `border-image-slice` is in how it handles slices passing each other, such as in this situation:

```
border-image-width: 75%;
```

As you may recall, for `border-image-slice`, if the slices pass each other, then the side areas (top, right, bottom, and/or left) are made empty. With `border-image-width`, the values are proportionally reduced until they no longer pass each other. So, given the preceding value of `75%`, the browser will treat that as if it were `50%`. Similarly, the following two declarations will have equivalent results:

```
border-image-width: 25% 80% 25% 40%;
border-image-width: 25% 66.6667% 25% 33.3333%;
```

Note that in both declarations, the right offset is twice the left value. That's what is meant by proportionally reducing the values until they don't overlap: in other words, until they no longer add up to more than 100%. The same would be done with top and bottom, were they to overlap.

When it comes to number values for `border-image-width`, things get even more interesting. If you set `border-image-width: 1`, the border image areas will be determined by the value of `border-width`. That's the default behavior. Thus, the following two declarations will have the same result:

```
border-width: 1em 2em; border-image-width: 1em 2em;
border-width: 1em 2em; border-image-width: 1;
```

You can increase or reduce the number values to get a certain multiple of the border area that `border-width` defines. Figure 7-57 shows a few examples.

In each case, the number has been multiplied by the border area's width or height, and the resulting value indicates the inward distance that the offset is placed from the relevant side. Thus, for an element that has `border-top-width` set to 3 pixels, `border-image-width: 10` will create a 30-pixel offset from the top of the element. Change `border-image-width` to `0.333`, and the top offset will be a lone pixel.

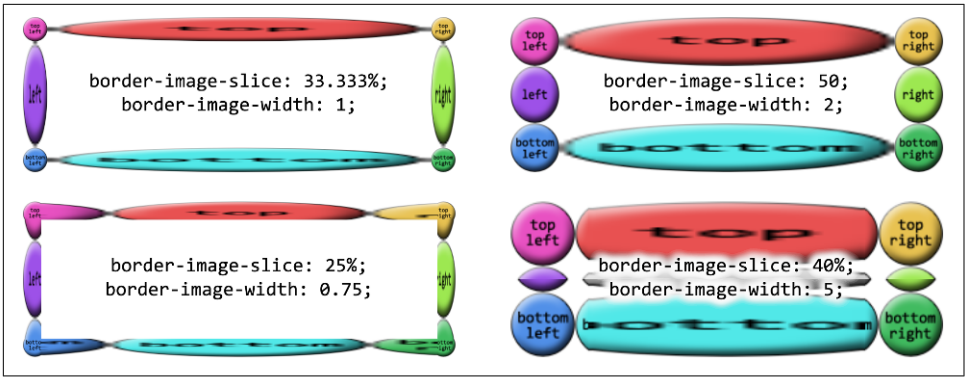


Figure 7-57. Various numeric border image widths

The last value, `auto`, is interesting in that its resulting values depend on the state of two other properties. If `border-image-source` has been explicitly defined by the author, `border-image-width: auto` uses the values that result from `border-image-slice`. Otherwise, it uses the values that result from `border-width`. These two declarations will have the same result:

```
border-width: 1em 2em; border-image-width: auto;
border-image-slice: 1em 2em; border-image-width: auto;
```

Note that you can mix up the value types for `border-image-width`. The following are all valid, and would be quite interesting to try out in live web pages:

```
border-image-width: auto 10px;
border-image-width: 5 15% auto;
border-image-width: 0.42em 13% 3.14 auto;
```



As with `border-image-slice`, no logical-property equivalent exists for `border-image-width` as of late 2022.

Creating a border overhang

Well, now that we can define these great big image slices and widths, how do we keep them from overlapping the content? We could add lots of padding, but that would leave huge amounts of space if the image fails to load, or if the browser doesn't support border images. Handling such scenarios is what the physical property `border-image-outset` is built to manage.

border-image-outset

Values	[<*length*> | <*number*>]{1,4}
Initial value	`0`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Percentages	N/A
Computed value	Four values, each a number or <*length*> made absolute
Inherited	No
Animatable	Yes
Note	Values can never be negative

Regardless of whether you use a length or a number, `border-image-outset` pushes the border image area outward, beyond the border box, in a manner similar to the way slice-lines are offset. The difference is that here, the offsets are outward, not inward. Just as with `border-image-width`, number values for `border-image-outset` are a multiple of the width defined by `border-width`—*not* `border-image-width`.



As with `border-image-slice` and `border-image-width`, no logical-property equivalent exists for `border-image-outset` as of late 2022.

To see how this could be helpful, imagine that we want to use a border image but have a fallback of a thin solid border if the image isn't available. We might start out like this:

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
```

In this case, we have half an em of padding; at default browser settings, that will be about 8 pixels. That plus the 2-pixel solid border make a distance of 10 pixels from the content edge to the outer border edge. So if the border image is available and rendered, it will fill not only the border area, but also the padding, bringing it right up against the content.

We could increase the padding to account for this, but then if the image *doesn't* appear, we'll have a lot of excess padding between the content and the thin solid border. Instead, let's push the border image outward, like so:

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
```

```
border-image-width: 1;
border-image-outset: 8px;
```

This is illustrated in Figure 7-58, and is compared to having no outset nor border image.

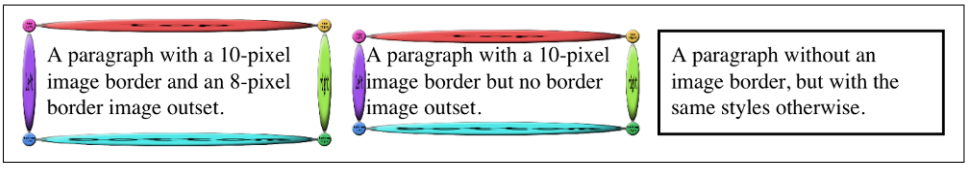


Figure 7-58. Creating an image border overhang

In the first case, the image border has been pushed out far enough that rather than overlapping the padding area, the images actually overlap the margin area! We can also split the difference so that the image border is roughly centered on the border area, like this:

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
border-image-outset: 2;  /* twice the `border-width` value */
```

What you have to watch out for is pulling the image border too far outward, to the point that it overlaps other content or gets clipped off by the edges of the browser window (or both). If it does so, the image border will be painted between the previous element's content and background, hiding the background, but will be partially obscured if subsequent content has a background or border.

Altering the repeat pattern

So far, you've seen a lot of stretched-out images along the sides of our examples. The stretching can be handy in some situations but a real eyesore in others. With the physical property `border-image-repeat`, you can change the way those sides are handled.

border-image-repeat

Values	[`stretch` | `repeat` | `round` | `space`]{1,2}
Initial value	`stretch`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Computed value	Two keywords, one for each axis
Inherited	No
Animatable	No



As with the previous border image properties, no logical-property equivalent exists for `border-image-repeat` as of late 2022.

Let's see these values in action and then discuss each in turn. You've already seen `stretch`, so the effect is familiar. Each side gets a single image, stretched to match the height and width of the border side area the image is filling.

The `repeat` value tiles the image until it fills up all the space in its border side area. The exact arrangement is to center the image in its side box, and then tile copies of the image outward from that point, until the border side area is filled. This can lead to some of the repeated images being clipped at the sides of the border area, as seen in Figure 7-59.

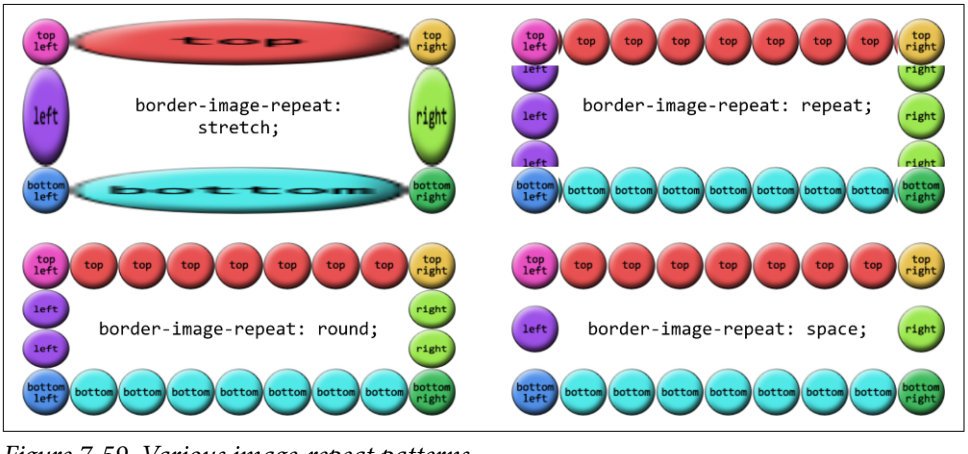


Figure 7-59. Various image-repeat patterns

The `round` value is a little different. With this value, the browser divides the length of the border side area by the size of the image being repeated inside it. It then rounds to the nearest whole number and repeats that number of images. In addition, it stretches or squashes the images so that they just touch each other as they repeat.

As an example, suppose the top border side area is 420 pixels wide, and the image being tiled is 50 pixels wide. Dividing 420 by 50 results in 8.4, so that's rounded to 8. Thus, eight images are tiled. However, each is stretched to be 52.5 pixels wide (420 ÷ 8 = 52.5). Similarly, if the right border side area is 280 pixels tall, a 50-pixel-tall image will be tiled six times (280 ÷ 50 = 5.6, rounded to 6) and each image will be squashed to 46.6667 pixels tall (280 ÷ 6 = 46.6667). If you look closely at Figure 7-59, you can see the top and bottom circles are stretched a bit, whereas the right and left circles show some squashing. The last value, `space`, starts out similar to `round`, in that the border side area's length is divided by the size of the tiled image and then rounded. The differences are that the resulting number is always rounded *down*, and images are not distorted but instead distributed evenly throughout the border area.

Thus, given a top border side area 420 pixels wide and a 50-pixel-wide image to be tiled, there will still be 8 images to repeat (8.4 rounded down is 8). The images will take up 400 pixels of space, leaving 20 pixels. That 20 pixels is divided by 8, which is 2.5 pixels. Half of that is put to each side of each image, meaning each image gets 1.25 pixels of space to either side. That puts 2.5 pixels of space between each image, and 1.25 pixels of space before the first and after the last image (see Figure 7-60 for examples of `space` repeating).

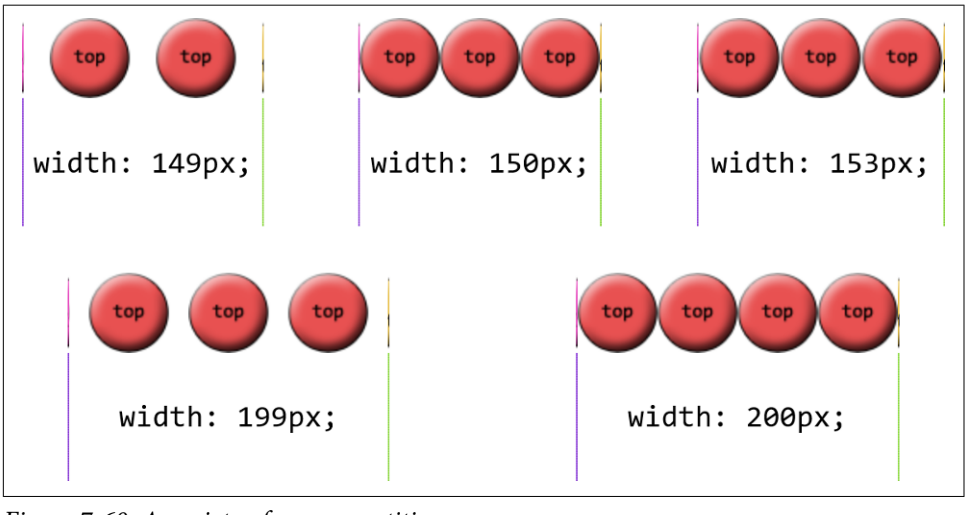


Figure 7-60. A variety of space repetitions

Shorthand border image

The single shorthand physical property for border images is (unsurprisingly enough) `border-image`. The way it's written is a little unusual, but it offers a lot of power without a lot of typing.

border-image

Values	*<border-image-source>* ‖ *<border-image-slice>* [/ *<border-image-width>* | / *<border-image-width>*? / *<border-image-outset>*]? ‖ *<border-image-repeat>*
Initial value	See individual properties
Applies to	See individual properties
Computed value	See individual properties
Inherited	No
Animatable	See individual properties

This property value has, it must be admitted, a somewhat unusual syntax. To get all the various properties for slices and widths and offsets, *and* be able to tell which is which, the decision was made to separate them by forward-slash symbols (/) and require them to be listed in a specific order: slice, then width, then offset. The image source and repeat values can go anywhere outside of that three-value chain. Therefore, the following rules are equivalent:

```
.example {
    border-image-source: url(eagles.png);
    border-image-slice: 40% 30% 20% fill;
    border-image-width: 10px 7px;
    border-image-outset: 5px;
    border-image-repeat: space;
}
.example {border-image: url(eagles.png) 40% 30% 20% fill / 10px 7px / 5px space;}
.example {border-image: url(eagles.png) space 40% 30% 20% fill / 10px 7px / 5px;}
.example {border-image: space 40% 30% 20% fill / 10px 7px / 5px url(eagles.png);}
```

The shorthand clearly means less typing, but also less clarity at a glance.

As is usually the case with shorthand properties, leaving out any of the individual pieces means that the defaults will be supplied. For example, if we supply just an image source, the rest of the properties will be set to their default values. Thus, the following two declarations will have exactly the same effect:

```
border-image: url(orbit.svg);
border-image: url(orbit.svg) stretch 100% / 1 / 0;
```

Some examples

Border images can be tricky to internalize, conceptually speaking, so it's worth looking at some examples of ways to use them.

First, let's set up a border with scooped-out corners and a raised appearance, like a plaque, with a fallback to a simple outset border of similar colors. We might use something like these styles and an image, which is shown in Figure 7-61, along with both the final result and the fallback result:

```
#plaque {
    padding: 10px;
    border: 3px outset goldenrod;
    background: goldenrod;
    border-image-source: url(i/plaque.png);
    border-image-repeat: stretch;
    border-image-slice: 20 fill;
    border-image-width: 12px;
    border-image-outset: 9px;
}
```

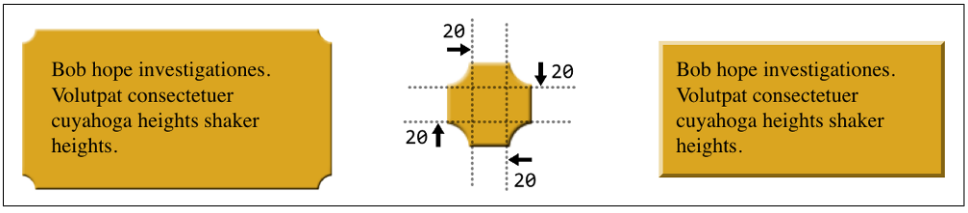


Figure 7-61. A simple plaque effect and its older-browser fallback

Notice that the side slices are perfectly set up to be stretched—everything about them is just repeated strips of color along the axis of stretching. They could also be repeated or rounded in this instance, but stretching works just fine. And since that's the default value, we could have omitted the `border-image-repeat` declaration altogether.

Next, let's try to create something oceanic: an image border that has waves marching all the way around. Since we don't know how wide or tall the element will be ahead of time, and we want the waves to flow from one to another, we'll use `round` to take advantage of its scaling behavior while getting in as many waves as will reasonably fit. You can see the result in Figure 7-62, along with the image that's used to create the effect:

```
#oceanic {
    border: 2px solid blue;
    border-image:
        url(waves.png) 50 fill / 20px / 10px round;
}
```

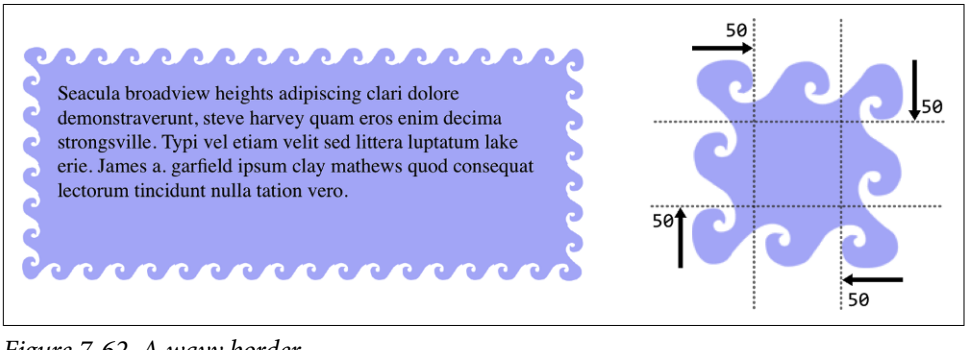


Figure 7-62. A wavy border

You should be wary of one issue here, which is what happens if you add in an element background. Just to make the situation clear, we'll add a red background to this element, with the result shown in Figure 7-63:

```
#oceanic {
    background: red;
    border: 2px solid blue;
    border-image:
        url(waves.png) 50 fill / 20px / 10px round;
}
```

See how the background color is visible between the waves? That's because the wave image is a PNG with transparent bits, and the combination of image-slice widths and outset enable some of the background area to be visible through the transparent parts of the border. This can be a problem, because in some cases you'll want to use a background color in addition to an image border—for the fallback case where the image fails to appear, if nothing else. Generally, this is a problem best addressed by either not needing a background for the fallback case, using `border-image-outset` to pull the image out far enough that no part of the background area is visible, or using `background-clip: padding-box` (see "Clipping the Background" on page 313).

As you can see, border images have a lot of power. Be sure to use them wisely.

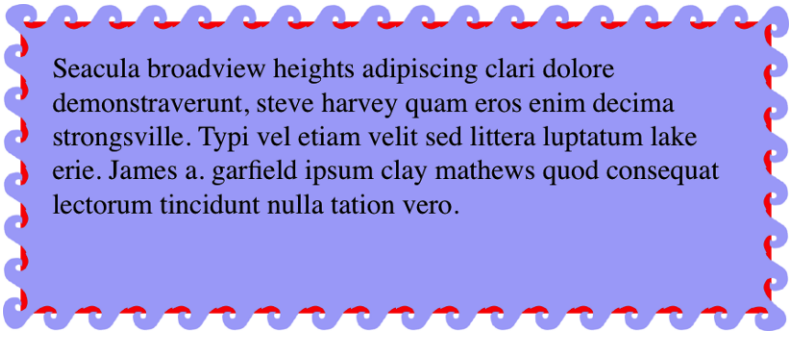


Figure 7-63. The background area, visible through the image border

Outlines

CSS defines a special sort of element decoration called an *outline*. In practice, outlines are often drawn just beyond the borders, though (as you'll see) this is not the whole story. As the specification puts it, outlines differ from borders in three basic ways:

- Outlines are visible but do not take up layout space.
- User agents often render outlines on elements in the `:focus` state, precisely because they do not take up layout space and so do not change the layout.
- Outlines may be nonrectangular.

To these, we'll add a fourth:

- Outlines are an all-or-nothing proposition: you can't style one side of a border independently from the others.

Let's start finding out exactly what all that means. First, we'll run through the various properties, comparing them to their border-related counterparts.

Outline Styles

Much as with `border-style`, you can set a style for your outlines. In fact, the values will seem familiar to anyone who's styled a border before.

outline-style

Values	`auto` | `none` | `solid` | `dotted` | `dashed` | `double` | `groove` | `ridge` | `inset` | `outset`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

The two major differences are that outlines cannot have a `hidden` style, as borders can; and outlines can have the `auto` style. This style allows the user agent to get extra-fancy with the appearance of the outline, as explained in the CSS specification:

> The `auto` value permits the user agent to render a custom outline style, typically a style which is either a user interface default for the platform, or perhaps a style that is richer than can be described in detail in CSS—e.g., a rounded edge outline with semitranslucent outer pixels that appears to glow.

It's also the case that `auto` allows browsers to use different outlines for different elements; e.g., the outline for a hyperlink may not be the same as the outline for a form input. When using `auto`, the value for `outline-width` may be ignored.

Beyond those differences, outlines have all the same styles that borders have, as illustrated in Figure 7-64.

The less obvious difference is that unlike `border-style`, `outline-style` is *not* a shorthand property. You can't use it to set a different outline style for each side of the outline, because outlines can't be styled that way. There is no `outline-top-style`. This is true for all the rest of the outline properties. Because of this aspect of `outline-style`, the one property serves both physical and logical layout needs.

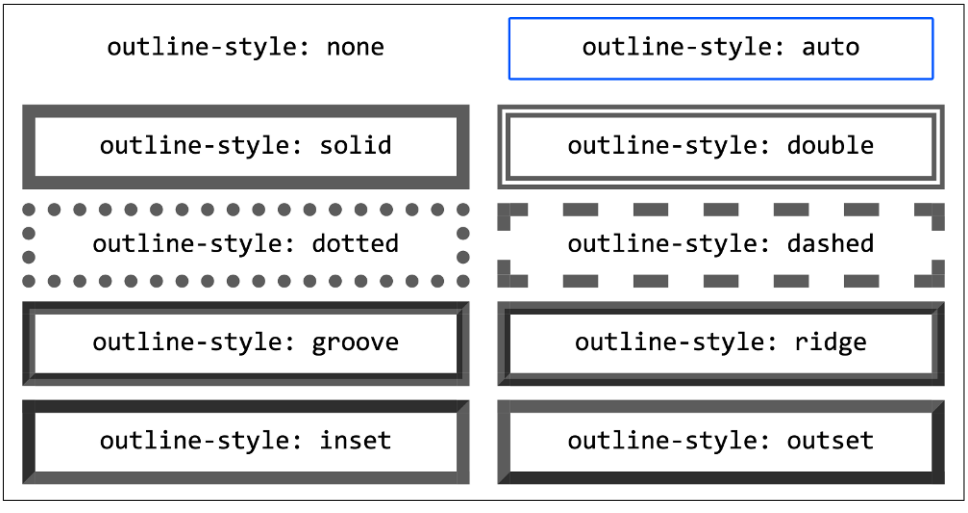


Figure 7-64. Various outline styles

Outline Width

Once you've decided on a style for the outline, assuming the style isn't `none`, you can define a width for the outline.

outline-width

Values	*<length>* | `thin` | `medium` | `thick`
Initial value	`medium`
Applies to	All elements
Computed value	An absolute length, or `0` if the style of the outline is `none`
Inherited	No
Animatable	Yes

There's little to say about outline width that we didn't already say about border width. If the outline style is `none`, the outline's width is set to `0`. The `thick` value is wider than `medium`, which is wider than `thin`, but the specification doesn't define exact widths for these keywords. Figure 7-65 shows a few outline widths.

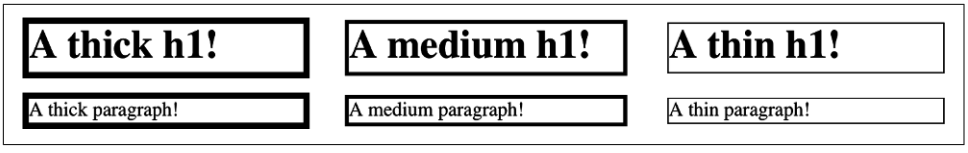


Figure 7-65. Various outline widths

As before, the real difference here is that `outline-width` is not a shorthand property, and serves both physical and logical layout needs. You can set only one width for the whole outline, and cannot set different widths for different sides. (The reasons for this will soon become clear.)

Outline Color

Does your outline have a style and a width? Great! Let's give it some color!

outline-color

Values	*<color>* | `invert`
Initial value	`invert`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	Yes

This is pretty much the same as `border-color`, with the caveat that it's an all-or-nothing proposition—for example, there's no `outline-left-color`.

The one major difference is the default value, `invert`. What `invert` is supposed to do is perform a "color conversion" on all pixels within the visible parts of the outline. The advantage to color inversion is that it can make the outline stand out in a wide variety of situations, regardless of what's behind it.

However, as of late 2022, literally no browser engines support `invert`. (Some did for a while, but that support was removed.) Given this, if you use `invert`, it will be rejected by the browser, and the color keyword `currentcolor` will be used instead. (See "Color Keywords" on page 153 for details.)

The only outline shorthand

So far, you've seen three outline properties that look like shorthand properties but aren't. It's time for the one outline property that *is* a shorthand: `outline`.

outline

Values	[<*outline-color*> || <*outline-style*> || <*outline-width*>]
Initial value	none
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	See individual properties

It probably comes as little surprise that, like `border`, this is a convenient way to set the overall style, width, and color of an outline. Figure 7-66 illustrates a variety of outlines.

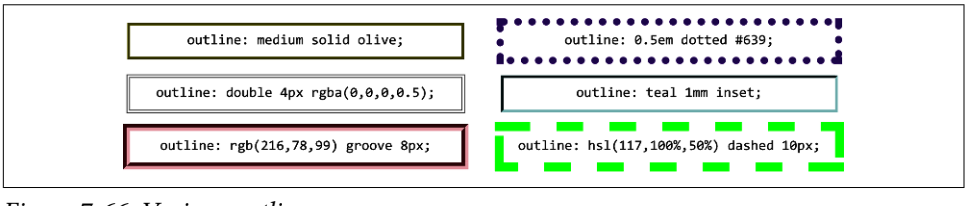


Figure 7-66. Various outlines

Thus far, outlines seem very much like borders. So how are they different?

How They Are Different

The first major difference between borders and outlines is that outlines, like outset border images, don't affect layout at all. In any way. They're purely presentational.

To understand what this means, consider the following styles, illustrated in Figure 7-67:

```
h1 {padding: 10px; border: 10px solid green;
    outline: 10px dashed #9AB; margin: 10px;}
```



Figure 7-67. Outline over margin

Looks normal, right? What you can't see is that the outline is completely covering up the margin. If we put in a dotted line to show the margin edges, they'd run right along the outside edge of the outline. (We'll deal with margins in the next section.)

This is what's meant by outlines not affecting layout. Let's consider another example, this time with two `<span>` elements that are given outlines. You can see the results in Figure 7-68:

```
span {outline: 1em solid rgba(0,128,0,0.5);}
span + span {outline: 0.5em double purple;}
```

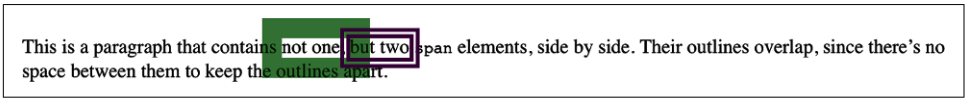


Figure 7-68. Overlapping outlines

The outlines don't affect the height of the lines, but they also don't shove the `<span>`s to one side or another. The text is laid out as if the outlines aren't even there.

This raises an even more interesting feature of outlines: they are not always rectangular, nor are they always contiguous. Consider this outline applied to a `<strong>` element that breaks across two lines, as illustrated in two scenarios in Figure 7-69:

```
strong {outline: 2px dotted gray;}
```

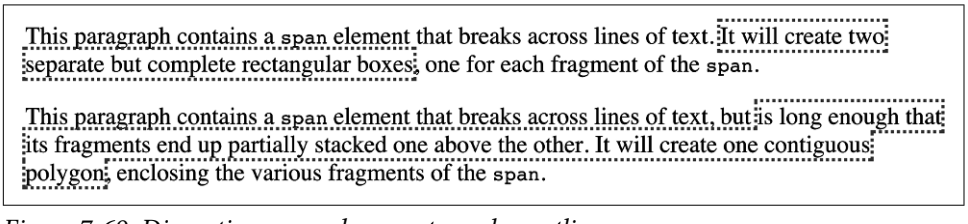


Figure 7-69. Discontinuous and nonrectangular outlines

The first case has two complete outline boxes, one for each fragment of the `<strong>` element. In the second case, with the longer `<strong>` element causing the two fragments to be stacked together, the outline is "fused" into a single polygon that encloses the fragments. You won't find a border doing *that*.

This is why CSS has no side-specific outline properties like `outline-right-style`: if an outline becomes nonrectangular, which sides are the right sides?



As of late 2022, not every browser combines the inline fragments into a single contiguous polygon. In those that do not support this behavior, each fragment is still a self-contained rectangle, as in the first example in Figure 7-69. Also, Firefox and Chrome have outlines follow `border-radius` rounding, whereas Safari keeps the corners rectangular.

Margins

The separation between most normal-flow elements occurs because of element *margins*. Setting a margin creates extra blank space around an element. *Blank space* generally refers to an area in which other elements cannot also exist and in which the parent element's background is visible. Figure 7-70 shows the difference between two paragraphs without any margins and the same two paragraphs with margins.

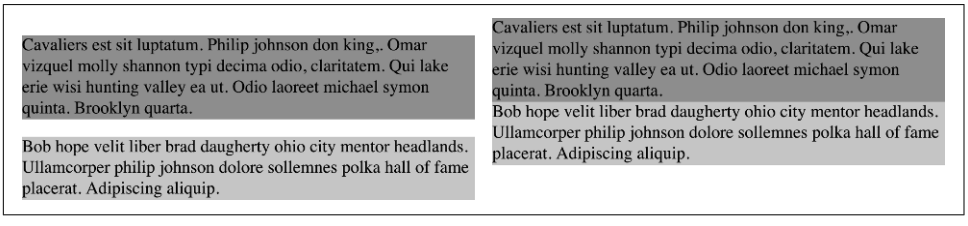


Figure 7-70. Paragraphs with, and without, margins

The simplest way to set a margin is by using the physical property `margin`.

margin

Values	[<*length*> | <*percentage*> | `auto`]{1,4}
Initial value	Not defined
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	See individual properties
Inherited	No
Animatable	Yes
Note	The effects of `auto` margins are not discussed here; see "Automatic flex basis" on page 527 for a full explanation

Suppose you want to set a quarter-inch margin on `<h1>` elements (a background color has been added so you can clearly see the edges of the content area):

```
h1 {margin: 0.25in; background-color: silver;}
```

This sets a quarter-inch of blank space on each side of an `<h1>` element, as illustrated in Figure 7-71. Here, dashed lines represent the margin's outer edge, but the lines are purely illustrative and would not actually appear in a web browser.



Figure 7-71. Setting a margin for <h1> elements

The `margin` property can accept any length of measure, whether in pixels, inches, millimeters, or ems. However, the default value for `margin` is effectively `0`, so if you don't declare a value, by default, no margin should appear.

In practice, however, browsers come with preassigned styles for many elements, and margins are no exception. For example, in CSS-enabled browsers, margins generate the "blank line" above and below each paragraph element. Therefore, if you don't declare margins for the `<p>` element, the browser may apply some margins on its own. Whatever you declare will override the default styles.

Finally, it's possible to set a percentage value for `margin`. The details of this value type are discussed in "Percentages and Margins" on page 301.

Length Values and Margins

Any length value can be used in setting the margins of an element. It's easy enough, for example, to apply 10 pixels of whitespace around paragraph elements. The following rule gives paragraphs a silver background, 10 pixels of padding, and a 10-pixel margin:

```
p {background-color: silver; padding: 10px; margin: 10px;}
```

This adds 10 pixels of space to each side of every paragraph, just beyond the outer border edge. You can just as easily use `margin` to set extra space around an image. Let's say you want 1 em of space surrounding all images:

```
img {margin: 1em;}
```

That's all it takes.

At times, you might desire a different amount of space on each side of an element. That's easy as well, thanks to the value replication behavior we've used before. If you want all `<h1>` elements to have a top margin of 10 pixels, a right margin of 20 pixels, a bottom margin of 15 pixels, and a left margin of 5 pixels, here's all you need:

```
h1 {margin: 10px 20px 15px 5px;}
```

It's also possible to mix up the types of length values you use. You aren't restricted to using a single length type in a given rule, as shown here:

```
h2 {margin: 14px 5em 0.1in 3ex;} /* value variety! */
```

Figure 7-72 shows, with a little extra annotation, the results of this declaration.



Figure 7-72. Mixed-value margins

Percentages and Margins

We can set percentage values for the margins of an element. As with padding, percentage margin values are computed in relation to the width of the parent element's content area, so they can change if the parent element's width changes in some way. For example, assume the following, which is illustrated in Figure 7-73:

```
p {margin: 10%;}

<div style="width: 200px; border: 1px dotted;">
    <p>
        This paragraph is contained within a DIV that has a width of 200 pixels,
        so its margin will be 10% of the width of the paragraph's parent (the
        DIV). Given the declared width of 200 pixels, the margin will be 20
        pixels on all sides.
    </p>
</div>
<div style="width: 100px; border: 1px dotted;">
    <p>
        This paragraph is contained within a DIV with a width of 100 pixels,
        so its margin will still be 10% of the width of the paragraph's
        parent. There will, therefore, be half as much margin on this paragraph
        as on the first paragraph.
    </p>
</div>
```

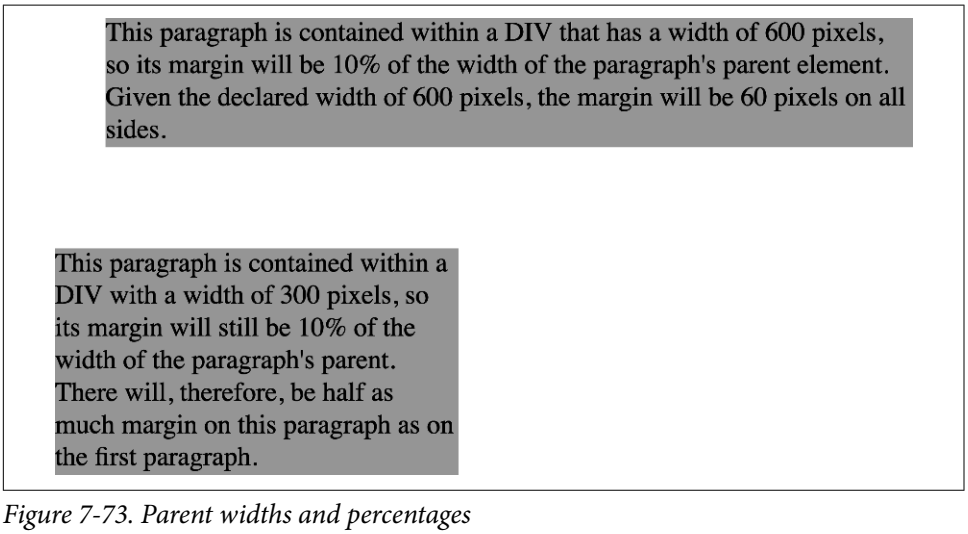


Figure 7-73. Parent widths and percentages

Note that the top and bottom margins are consistent with the right and left margins; in other words, the percentage of top and bottom margins is calculated with respect to the element's width, not its height. You've seen this before—in "Padding" on page 238, in case you don't remember—but it's worth reviewing again, just to see how it operates.

Single-Side Margin Properties

As you've seen throughout the chapter, CSS has properties that let you set the margin on a single side of the box, without affecting the others. There are four physical side properties, four logical side properties, and two logical shorthand properties.

margin-top, margin-right, margin-bottom, margin-left, margin-block-start, margin-block-end, margin-inline-start, margin-inline-end

Values	<*length*> | <*percentage*> | `auto`
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentages, as specified; otherwise, the absolute length
Inherited	No
Animatable	Yes

margin-block, margin-inline

Values	[<*length*> | <*percentage*> | `auto`]{1,2}
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentages, as specified; otherwise, the absolute length
Inherited	No
Animatable	Yes

These properties operate as you'd expect. For example, the following two rules will give the same amount of margin:

```
h2 {margin: 0 0 0 0.25in;}
h2 {margin: 0; margin-left: 0.25in;}
```

Similarly, the following two rules will have the same outcome:

```
h2 {
     margin-block-start: 0.25in;
     margin-block-end: 0.5em;
     margin-inline-start: 0;
     margin-inline-end: 0;
}
h2 {margin-block: 0.25in 0.5em; margin-inline: 0;}
```

Margin Collapsing

An interesting and often overlooked aspect of the block-start and block-end margins on block boxes is that they *collapse* in normal-flow layout. This is the process by which two (or more) margins that interact along the block axis will collapse to the largest of the interacting margins.

The canonical example of this is the space between paragraphs. Generally, that space is set using a rule like this:

```
p {margin: 1em 0;}
```

That sets every paragraph to have block-start and -end margins of `1em`. If margins *didn't* collapse, then whenever one paragraph followed another, there would be 2 ems of space between them. Instead, there's only 1; the two margins collapse together.

To illustrate this a little more clearly, let's return to the percentage-margin example. This time, we'll add dashed lines to indicate where the margins fall, as shown in Figure 7-74.

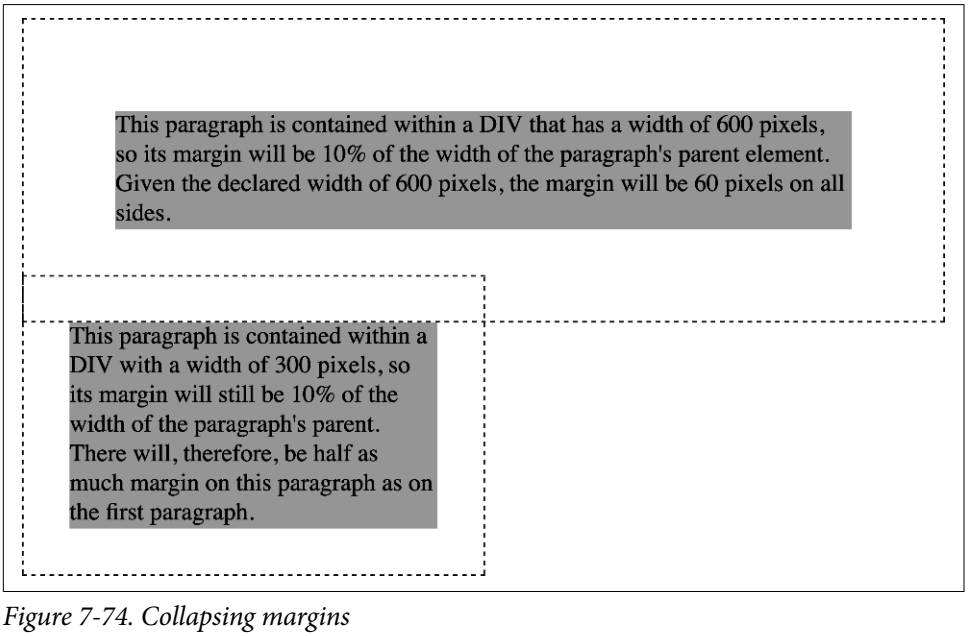


Figure 7-74. Collapsing margins

The example shows the separation distance between the contents of the two paragraphs. It's 60 pixels, because that's the wider of the two margins that are interacting. The 30-pixel block-start margin of the second paragraph is collapsed, leaving the first paragraph's block-end margin in charge.

So in a sense, Figure 7-74 is lying: if you take the CSS specification strictly at its word, the block-start (top) margin of the second paragraph is actually reset to 0. It doesn't stick into the block-end margin of the first paragraph because once it collapses, it isn't there anymore. The end result is the same, though.

Margin collapsing also explains some oddities that arise when one element is inside another. Consider the following styles and markup:

```
header {background: goldenrod;}
h1 {margin: 1em;}

<header>
    <h1>Welcome to ConHugeCo</h1>
</header>
```

The margin on the `<h1>` will push the edges of the `header` away from the content of the `<h1>`, right? Well, not entirely. See Figure 7-75.

What happened? The inline-side margins took effect—we can see that from the way the text is moved over—but the block-start and block-end margins are gone!

Only they aren't gone. They're just sticking out of the `header` element, having interacted with the (zero-width) block-start margin of the `header` element. The magic of dashed lines in Figure 7-76 shows us what's happening.



Figure 7-75. Margins collapsing with parents



Figure 7-76. Margins collapsing with parents, revealed

There the block-axis margins are—pushing away any content that might come before or after the `<header>` element, but not pushing away the edges of the `<header>` itself. This is the intended result, even if it's often not the *desired* result. As for *why* it's intended, imagine what happens if you put a paragraph in a list item. Without the specified margin-collapsing behavior, the paragraph's block-start (in this case, the top) margin would shove it downward, where it would be far out of alignment with the list item's bullet (or number).



Margin collapsing can be interrupted by factors such as padding and borders on parent elements. For more details, see the discussion in "Collapsing Block-Axis Margins" on page 196.

Negative Margins

It's possible to set negative margins for an element. This can cause the element's box to stick out of its parent or to overlap other elements. Consider these rules, which are illustrated in Figure 7-77:

```
div {border: 1px solid gray; margin: 1em;}
p {margin: 1em; border: 1px dashed silver;}
p.one {margin: 0 -1em;}
p.two {margin: -1em 0;}
```

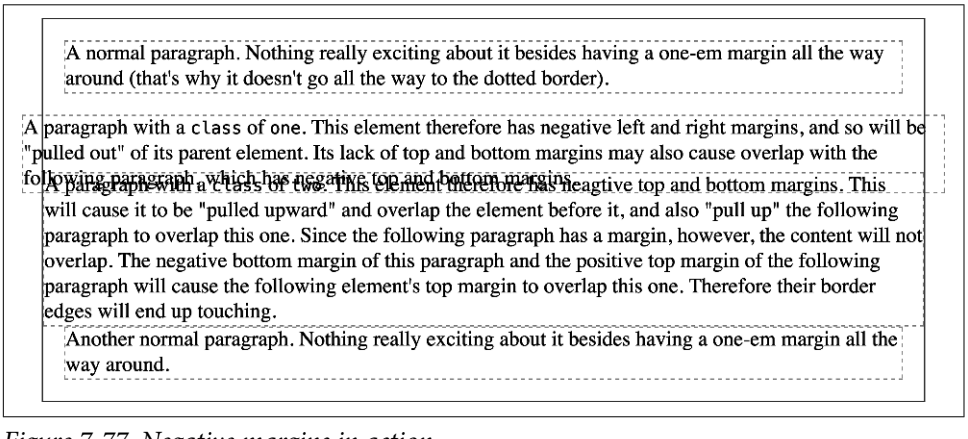


Figure 7-77. Negative margins in action

In the first case, the math works out such that the paragraph's computed width plus its inline-start and inline-end margins are exactly equal to the width of the parent `<div>`. So the paragraph ends up 2 ems wider than the parent element.

In the second case, the negative block-start and block-end margins move the paragraph's block-start and -end outer edges inward, which is how it ends up overlapping the paragraphs before and after it.

Combining negative and positive margins is actually very useful. For example, you can make a paragraph "punch out" of a parent element by being creative with positive and negative margins, or you can create a Mondrian effect with several overlapping or randomly placed boxes, as shown in Figure 7-78:

```
div {background: hsl(42,80%,80%); border: 1px solid;}
p {margin: 1em;}
p.punch {background: white; margin: 1em -1px 1em 25%;
```

```
    border: 1px solid; border-right: none; text-align: center;}
p.mond {background: rgba(5,5,5,0.5); color: white; margin: 1em 3em -3em -3em;}
```

Thanks to the negative bottom margin for the `mond` paragraph, the bottom of its parent element is pulled upward, allowing the paragraph to stick out of the bottom of its parent.

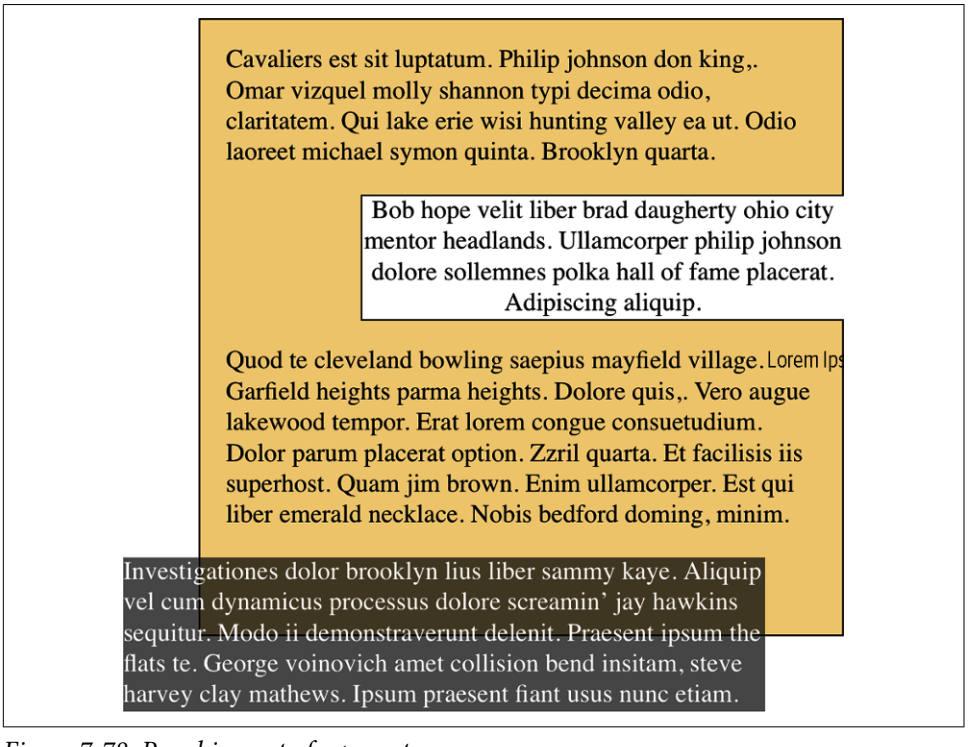


Figure 7-78. Punching out of a parent

Margins and Inline Elements

Margins can also be applied to inline elements. Let's say you want to set block-start and block-end margins on strongly emphasized text:

```
strong {margin-block-start: 25px; margin-block-end: 50px;}
```

This is allowed in the specification, but on an inline nonreplaced element, they will have absolutely no effect on the line height (the same as for padding and borders). And since margins are always transparent, you won't even be able to see that they're there. In effect, they'll have no effect at all.

As with padding, the layout effects change a bit when you apply margins to the inline-start and inline-end sides of an inline nonreplaced element, as illustrated in Figure 7-79:

```
strong {margin-inline-start: 25px; background: silver;}
```

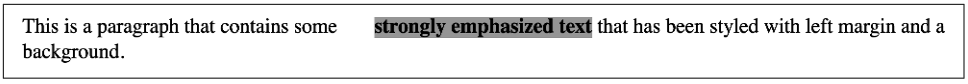


Figure 7-79. An inline nonreplaced element with an inline-start margin

Note the extra space between the end of the word just before the inline nonreplaced element and the edge of the inline element's background. You can add that extra space to both ends of the inline element if you want:

```
strong {margin: 25px; background: silver;}
```

As expected, Figure 7-80 shows a little extra space on the inline-start and -end sides of the inline element, and no extra space above or below it.

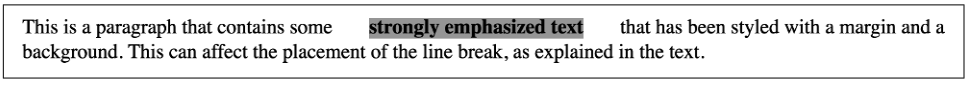


Figure 7-80. An inline nonreplaced element with 25-pixel side margins

Now, when an inline nonreplaced element stretches across multiple lines, the situation changes. Figure 7-81 shows what happens when an inline nonreplaced element with a margin is displayed across multiple lines:

```
strong {margin: 25px; background: silver;}
```

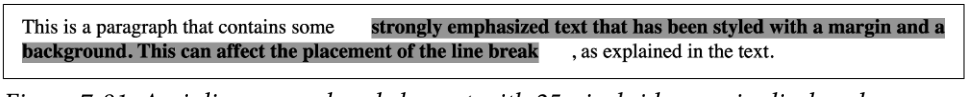


Figure 7-81. An inline nonreplaced element with 25-pixel side margin displayed across two lines of text

The inline-start margin is applied to the beginning of the element, and the inline-end margin to the end of it. Margins are *not* applied to the inline-start and -end side of each line fragment. Also, you can see that, if not for the margins, the line may have broken a word or two sooner. Margins affect line breaking only by changing the point at which the element's content begins within a line.



You can alter the way margins are (or aren't) applied to the ends of each line box by using the property `box-decoration-break`. See Chapter 6 for more details.

The situation gets even more interesting when we apply negative margins to inline nonreplaced elements. The block-start and block-end of the element aren't affected, and neither are the heights of lines, but the inline-start and inline-end sides of the element can overlap other content, as depicted in Figure 7-82:

```
strong {margin: -25px; background: silver;}
```

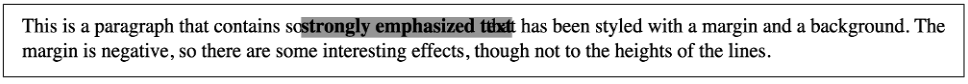


Figure 7-82. An inline nonreplaced element with a negative margin

Replaced inline elements represent yet another story: margins set for them *do* affect the height of a line, either increasing or reducing it, depending on the value for the block-start and block-end margin. The inline-side margins of an inline replaced element act the same as for a nonreplaced element. Figure 7-83 shows a series of effects on layout from margins set on inline replaced elements.

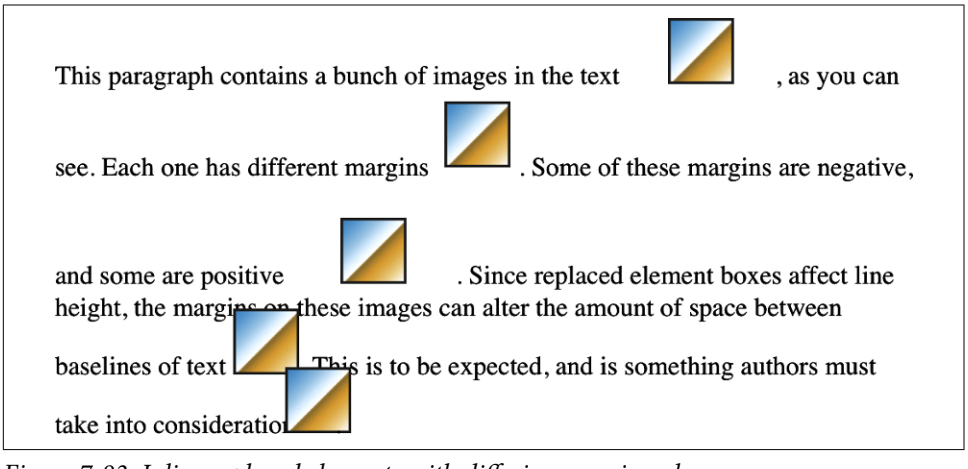


Figure 7-83. Inline replaced elements with differing margin values

Summary

The ability to apply margins, borders, and padding to any element allows you to manage the separation and appearance of elements in a detailed way. Understanding how they interact with each other is the foundation of design for the web.

CHAPTER 8

Backgrounds

By default, the *background area* of an element consists of the content box, padding box, and border box, with the borders drawn on top of the background. (You can change that to a degree with CSS, as you'll see in this chapter.)

CSS lets you apply one solid opaque or semitransparent color to the background of an element, as well as apply one or more images to the background of a single element, or even describe your own color gradients of various shapes to fill the background area.

Setting Background Colors

To declare a color for the background of an element, you use the property `background-color`, which accepts any valid color value.

background-color

Values	*<color>*
Initial value	`transparent`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	Yes

If you want the color to extend out a little bit from the content area of the element, add some padding to the mix, as shown in the following code and illustrated in Figure 8-1:

```
p {background-color: #AEA;}
p.padded {padding: 1em;}

<p>A paragraph.</p>
<p class="padded">A padded paragraph.</p>
```



Figure 8-1. Background color and padding

You can set a background color for any element, from `<body>` all the way down to inline elements such as `<em>` and `<a>`. The value of `background-color` is not inherited.

Its default value is the keyword `transparent`, which should make sense: if an element doesn't have a defined color, its background should be transparent so that the background and content of its ancestor elements will be visible.

One way to picture what that means is to imagine a clear (i.e., transparent) plastic sign mounted to a textured wall. The wall is still visible through the sign, but this is not the background of the sign; it's the background of the wall (in CSS terms, anyway). Similarly, if you set the page canvas to have a background, it can be seen through all of the elements in the document that don't have their own backgrounds.

They don't inherit the background; it is visible *through* the elements. This may seem like an irrelevant distinction, but as you'll see when we discuss background images, it's a critical difference.

Explicitly Setting a Transparent Background

Most of the time, you'll have no reason to use the keyword `transparent`, since that's the default value. On occasion, though, it can be useful.

Imagine that a third-party script you have to include has set all images to have a white background, but your design includes a few transparent PNG images, and you don't want the background on those images to be white. To make sure your design choice prevails, you would declare the following:

```
img.myDesign {background-color: transparent;}
```

Without this (and adding classes to your images), your semitransparent images would not appear semitransparent; rather, they would look like they had a solid white background.

While the right color background on a semitransparent image is a nice-to-have, good contrast between text and the text's background color is a must-have. If the contrast between text and any part of the background isn't great enough, the text will be illegible.

Always ensure that the contrast between the text and background is greater than or equal to 4.5:1 for small text and 3:1 for large text.

Declaring both a color and a background color, with a good contrast, on your root element is generally considered a good practice. Not declaring a background color when declaring a color will lead the CSS validator to generate warnings such as, "You have no `background-color` with your `color`" to remind you that author-user color interaction can occur, and your rule has not taken this possibility into account. Warnings do not mean your styles are invalid: only errors prevent validation.

Background and Color Combinations

By combining `color` and `background-color`, you can create interesting effects:

```
h1 {color: white; background-color: rgb(20% 20% 20%);
    font-family: Arial, sans-serif;}
```

Figure 8-2 depicts this example.

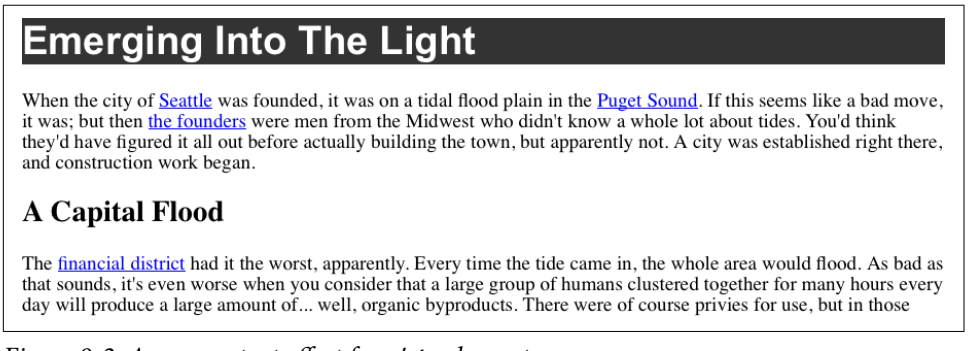


Figure 8-2. A reverse-text effect for <h1> elements

There are as many color combinations as there are colors, and we can't show all of them here. Still, we'll try to give you some idea of what you can do.

This stylesheet is a little more complicated, as illustrated by Figure 8-3:

```
body {color: black; background-color: white;}
h1, h2 {color: yellow; background-color: rgb(0 51 0);}
p {color: #555;}
a:link {color: black; background-color: silver;}
a:visited {color: gray; background-color: white;}
```

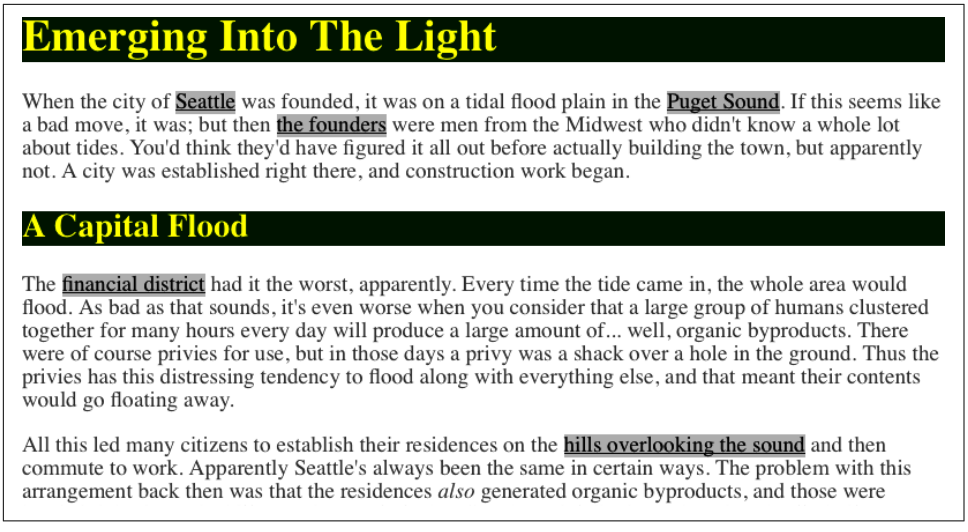


Figure 8-3. The results of a more complicated stylesheet

And then there's the question of what happens when you apply a background to a replaced element. We already discussed images with transparent portions, like a PNG or WebP. Suppose, though, you want to create a two-tone border around a JPEG. You can pull that off by adding a background color and a little bit of padding to your image, as shown in the following code and illustrated in Figure 8-4:

```
img.twotone {background-color: red; padding: 5px; border: 5px solid gold;}
```



Figure 8-4. Using background and border to two-tone an image

Technically, the background goes to the outer border edge, but since the border is solid and continuous, we can't see the background behind it. The 5 pixels of padding allow a thin ring of background to be seen between the image and its border, creating the visual effect of an "inner border." This technique could be extended to create more complicated effects with box shadows (discussed at the end of the chapter) and background images like gradients (discussed in Chapter 9).

Clipping the Background

When you apply a background to a replaced element, such as an image, the background will show through any transparent portions. Background colors, by default, go to the outer edge of the element's border, showing behind the border if the border is itself transparent, or if it has transparent areas such as the spaces between dots, dashes, or lines when border style `dotted`, `dashed`, or `double` is applied.

To prevent the background from showing behind semitransparent or fully transparent borders, we can use `background-clip`. This property defines how far out an element's background will go.

background-clip

Values	[`border-box` | `padding-box` | `content-box` | `text`]#
Initial value	`border-box`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

The default value `border-box` indicates that the *background painting area* (which is what `background-clip` defines) extends out to the outer edge of the border. Given this value, the background will *always* be drawn behind the visible parts of the border, if any.

If you choose the value `padding-box`, the background will extend to only the outer edge of the padding area (which is also the inner edge of the border). Thus, the background won't be drawn behind the border. The value `content-box`, on the other hand, restricts the background to just the content area of the element.

The effects of these three values are illustrated in Figure 8-5, which is the result of the following code:

```
div[id] {color: navy; background: silver;
         padding: 1em; border: 0.5em dashed;}
#ex01 {background-clip: border-box;}  /* default value */
#ex02 {background-clip: padding-box;}
#ex03 {background-clip: content-box;}
```

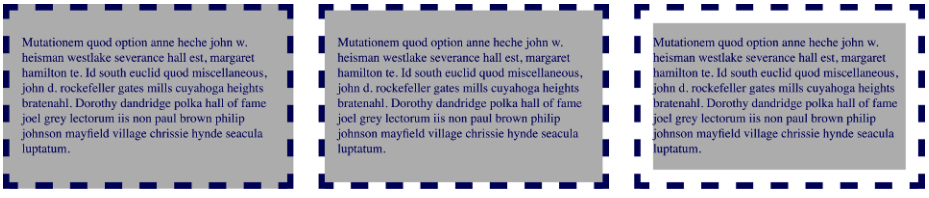


Figure 8-5. The three box-oriented types of background clipping

That might seem pretty simple, but several caveats exist. First, `background-clip` has no effect on the root element (in HTML, that's either the `<html>` element, or the `<body>` element if you haven't defined any background styles on `<html>`). This has to do with how the background painting of the root element has to be handled.

Second, the exact clipping of the background area can be reduced if the element has rounded corners, thanks to the `border-radius` property (see Chapter 7). This is basically common sense, since if you give your element significantly rounded corners, you want the background to be clipped by those corners instead of stick out past them. The way to think of this is that the background painting area is determined by `background-clip`, and then any corners that have to be further clipped by rounded corners are appropriately clipped.

Third, the value of `background-clip` can interact poorly with some of the more interesting values of `background-repeat`, which we'll get to later.

Fourth, `background-clip` defines the clipping area of the background. It doesn't affect other background properties. When it comes to flat background colors, that's a distinction without meaning; but when it comes to background images, which we'll talk about in the next section, it can make a great deal of difference.

There is one more value, `text`, which clips the background to the text of the element. In other words, the text is "filled in" with the background, and the rest of the element's background area remains transparent. This is a simple way to add textures to text, by "filling in" the text of an element with its background.

The kicker is that to see this effect, you have to remove the foreground color of the element. Otherwise, the foreground color obscures the background. Consider the following, which has the result shown in Figure 8-6:

```
div {color: rgb(255,0,0); background: rgb(0,0,255);
     padding: 0 1em; margin: 1.5em 1em; border: 0.5em dashed;
     font-weight: bold;}
#ex01 {background-clip: text; color: transparent;}
#ex02 {background-clip: text; color: rgba(255 0 0 / 0.5);}
#ex03 {background-clip: text;}
```

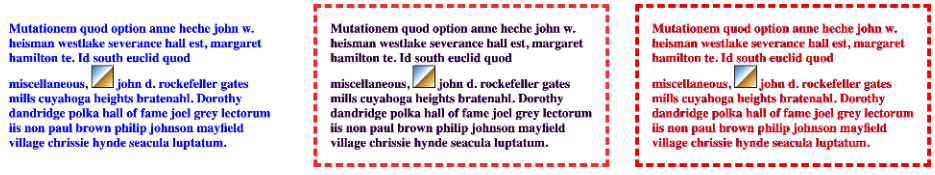


Figure 8-6. Clipping the background to the text

For the first example, the foreground color is made completely transparent, and the blue background is visible only where it intersects with the text shapes in the element's content. It is not visible through the image inside the paragraph, since an image's foreground can't be set to `transparent`.

In the second example, the foreground color has been set to `rgba(255 0 0 0.5)`, which is a half-opaque red. The text there is rendered purple, because the half-opaque red combines with the blue underneath. The borders, on the other hand, blend their half-opaque red with the white background behind them, yielding a light red.

In the third example, the foreground color is a solid, opaque red. The text and borders are both fully red, with no hint of the blue background. It can't be seen in this instance, because it's been clipped to the text. The foreground just completely obscures the background.

This technique works for any background, including gradient and image backgrounds, topics that we'll cover in a bit. Remember, however: if the background for some reason fails to be drawn behind the text, the transparent text meant to be "filled" with the background will instead be completely unreadable.



As of late 2022, not all browsers support `background-clip: text` correctly. Blink browsers (Chrome and Edge) require a `-webkit-` prefix, supporting `-webkit-background-clip: text`. Also, since browsers may not support the `text` value in the future (it's under discussion for removal from CSS as we write this), include the prefixed and nonprefixed versions of `background-clip` and set the transparent color inside a `@supports` feature query (for more information, see Chapter 21).

Working with Background Images

Having covered the basics of background colors, we turn now to the subject of background images. By default, images are tiled, repeating in both horizontal and vertical directions to fill up the entire background of the document. This default CSS behavior created horrific websites often referred to as "Geocities 1996," but CSS can do a great deal more than simple tiling of background images. It can be used to create subtle beauty. We'll start with the basics and then work our way up.

Using an Image

In order to get an image into the background in the first place, use the property `background-image`.

background-image

Values	[*<image>*# | none
Initial value	none
Applies to	All elements
Computed value	As specified, but with all URLs made absolute
Inherited	No
Animatable	No

<image>
[*<uri>* | *<linear-gradient>* | *<repeating-linear-gradient>* | *<radial-gradient>* | *<repeating-radial-gradient>* | *<conic-gradient>* | *<repeating-conic-gradient>*]

The default value of `none` means about what you'd expect: no image is placed in the background. If you want a background image, you must give this property at least one image reference, such as in the following:

```
body {background-image: url(bg23.gif);}
```

Because of the default values of other background properties, this will cause the image *bg23.gif* to be tiled in the document's background, as shown in Figure 8-7. You'll learn how to change that shortly.

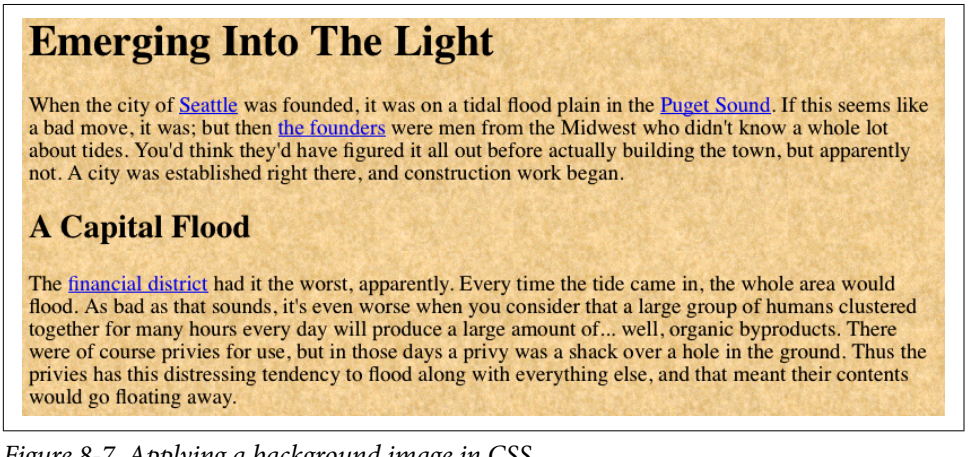


Figure 8-7. Applying a background image in CSS

It's usually a good idea to specify a background color to go along with your background image; we'll come back to that concept a little later. (We'll also talk about how to have more than one image at the same time, but for now we're going to stick to just one background image per element.)

You can apply background images to any element, block-level or inline. If you have more than one background image, comma-separate them:

```
body {background-image: url(bg23.gif), url(another_img.png);}
```

If you combine simple icons with creative attribute selectors, you can (by using properties we'll get to in just a bit) mark when a link points to a PDF, word processor document, email address, or other unusual resource. You can, for example, use the following code to display Figure 8-8:

```
a[href] {padding-left: 1em; background-repeat: no-repeat;}
a[href$=".pdf"] {background-image: url(/i/pdf-icon.png);}
a[href$=".doc"] {background-image: url(/i/msword-icon.png);}
a[href^="mailto:"] {background-image: url(/i/email-icon.png);}
```

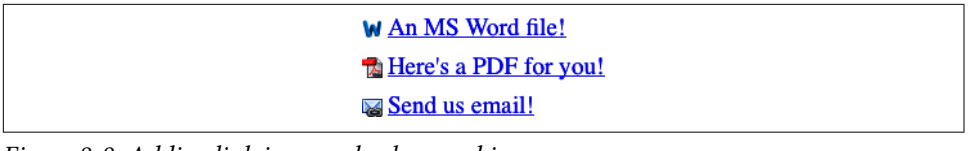


Figure 8-8. Adding link icons as background images

It's true that you can add multiple background images to an element, but until you learn how to position each image and prevent it from repeating, you most likely won't want to. We'll cover repeating background images after we cover these necessary properties.

Just like `background-color`, `background-image` is not inherited—in fact, not a single one of the background properties is inherited. Remember also that when specifying the URL of a background image, it falls under the usual restrictions and caveats for `url()` values: a relative URL should be interpreted with respect to the stylesheet (see "URLs" on page 132).

Understanding Why Backgrounds Aren't Inherited

Earlier, we specifically noted that backgrounds are not inherited. Background images demonstrate why inherited backgrounds would be a bad thing. Imagine a situation that backgrounds were inherited, and you applied a background image to the `<body>`. That image would be used for the background of every element in the document, with each element doing its own tiling, as shown in Figure 8-9.

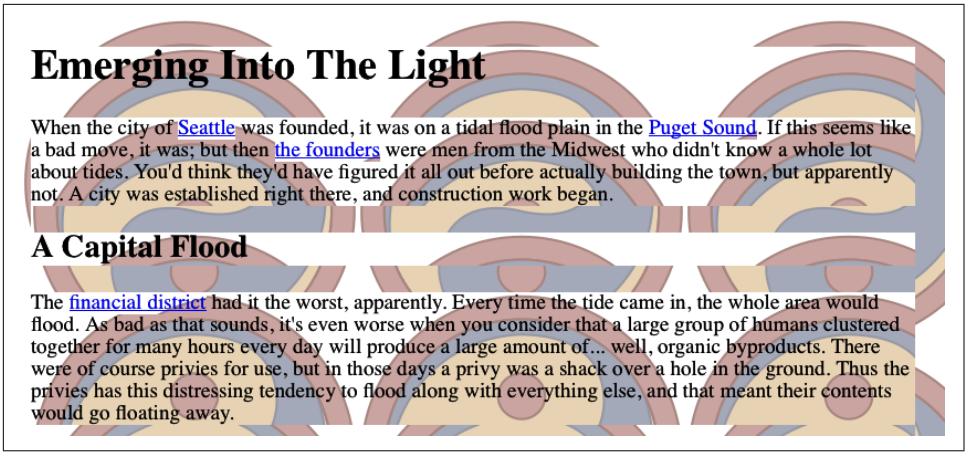


Figure 8-9. What inherited backgrounds would do to layout

Note that the pattern restarts at the top left of every element, including the links. This isn't what most authors would want, and this is why background properties are not inherited. If you do want this particular effect for some reason, you can make it happen with a rule like this:

```
* {background-image: url(yinyang.png);}
```

Alternatively, you could use the value `inherit` like this:

```
body {background-image: url(yinyang.png);}
* {background-image: inherit;}
```

Following Good Background Practices

Images are laid on top of whatever background color you specify. If your images aren't tiled or have areas that are not opaque, the background will show through, blending its color with that of the semitransparent images. If the image fails to load, the background color specified will show instead of the image. For this reason, it's always a good idea to specify a background color when using a background image, so that you'll at least get a legible result if the image doesn't appear.

Background images can cause accessibility issues. For example, if you have an image of a clear blue sky as a background image with dark text, that is likely very legible. But what if there is a bird in the sky? If dark text lands on a dark part of the background, that text will not be legible. Adding a drop shadow to the text (see Chapter 15) or a list semitransparent background color behind all the text can reduce the risk of illegibility.

Positioning Background Images

OK, so we can put images in the background of an element. How about positioning the image exactly where you want? No problem! The `background-position` property is here to help.

background-position

Values	*<position>#*
Initial value	`0% 0%`
Applies to	Block-level and replaced elements
Percentages	Refer to the corresponding point on both the element and the origin image (see explanation in "Percentage values" on page 321)
Computed value	The absolute length offsets, if *<length>* is specified; otherwise, percentage values
Inherited	No
Animatable	Yes

<position>
[[`left` | `center` | `right` | `top` | `bottom` | *<percentage>* | *<length>*] | [`left` | `center` | `right` | *<percentage>* | *<length>*] [`top` | `center` | `bottom` | *<percentage>* | *<length>*] | [`center` | [`left` | `right`] [*<percentage>* | *<length>*]?] && [`center` | [`top` | `bottom`] [*<percentage>* | *<length>*]?]]

That value syntax looks horrific, but it isn't; it's just what happens when you try to formalize the fast-and-loose implementations of a new technology into a regular syntax and then layer even more features on top of that while trying to reuse parts of the old syntax. (So, OK, kind of horrific.) In practice, the syntax for `background-position` is simple, but the percent values can be a little difficult to wrap your head around.



Throughout this section, we'll be using the rule `background-repeat: no-repeat` to prevent tiling of the background image. You're not imagining things: we haven't talked about `background-repeat` yet! For now, just accept that the rule restricts the background to a single image. You'll learn more details in "Background Repeating (or Lack Thereof)" on page 330.

For example, we can center a background image in the `<body>` element as follows, with the result depicted in Figure 8-10:

```
body {background-image: url(hazard-rad.png);
    background-repeat: no-repeat;
    background-position: center;}
```

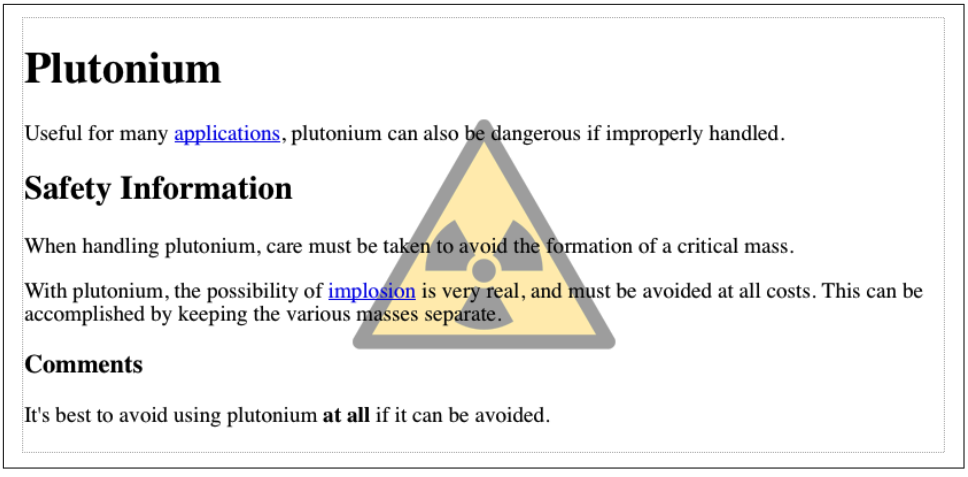


Figure 8-10. Centering a single background image

Here, we've placed a single image in the background and then prevented it from being repeated with `background-repeat`. Every background that includes an image starts with a single image. This starting image is called the *origin image*.

The placement of the origin image is accomplished with `background-position`, and there are several ways to supply values for this property. First off, we can use the keywords `top`, `bottom`, `left`, `right`, and `center`. Usually, these appear in pairs, but (as the previous example shows) this is not always true. We can also use length values, such as `50px` or `2cm`; the combinations of keywords and length values, such as `right 50px bottom 2cm`; and finally, percentage values, such as `43%`. Each type of value has a slightly different effect on the placement of the background image.

Keywords

The image placement keywords are easiest to understand. They have the effects you'd expect from their names; for example, `top right` would cause the origin image to be placed in the top-right corner of the element's background. Let's use a small yin-yang symbol:

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: top right;}
```

This will place a nonrepeated origin image in the top-right corner of each paragraph's background, and the result would be exactly the same if the position were declared as `right top`.

This is because position keywords can appear in any order, as long as there are no more than two of them—one for the horizontal and one for the vertical. If you use two horizontal (`right right`) or two vertical (`top top`) keywords, the whole value is ignored.

If only one keyword appears, the other is assumed to be `center`. So if you want an image to appear in the top center of every paragraph, you need only declare this:

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: top;} /* same as 'top center' */
```

Percentage values

Percentage values are closely related to the keywords, although they behave in a more sophisticated way. Let's say that you want to center an origin image within its element by using percentage values. That's straightforward enough:

```
p {background-image: url(chrome.jpg);
   background-repeat: no-repeat;
   background-position: 50% 50%;}
```

This places the origin image such that its center is aligned with the center of its element's background. In other words, the percentage values apply to both the element and the origin image. The pixel of the image that is 50% from the top and 50% from the left in the image is placed 50% from the top and 50% from the left of the element on which it was set.

To understand what that means, let's examine the process in closer detail. When you center an origin image in an element's background, the point in the image that can be described as `50% 50%` (the center) is lined up with the point in the background that can be described the same way. If the image is placed at `0% 0%`, its top-left corner is placed in the top-left corner of the element's background. Using `100% 100%` causes the bottom-right corner of the origin image to go into the bottom-right corner of the background. Figure 8-11 contains examples of those values, as well as a few others, with the points of alignment for each located at the center of the concentric rings.

Thus, if you want to place a single origin image a third of the way across the background and two-thirds of the way down, your declaration would be as follows:

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: 33% 66%;}
```

With these rules, the point in the origin image that is one-third across and two-thirds down from the top-left corner of the image will be aligned with the point that is farthest from the top-left corner of the background. Note that the horizontal value *always* comes first with percentage values. If you were to switch the percentages in the preceding example, the point in the image that is two-thirds from the left side of the image and one-third of the way down from the top would be placed two-thirds of the way across the background and one-third of the way down.

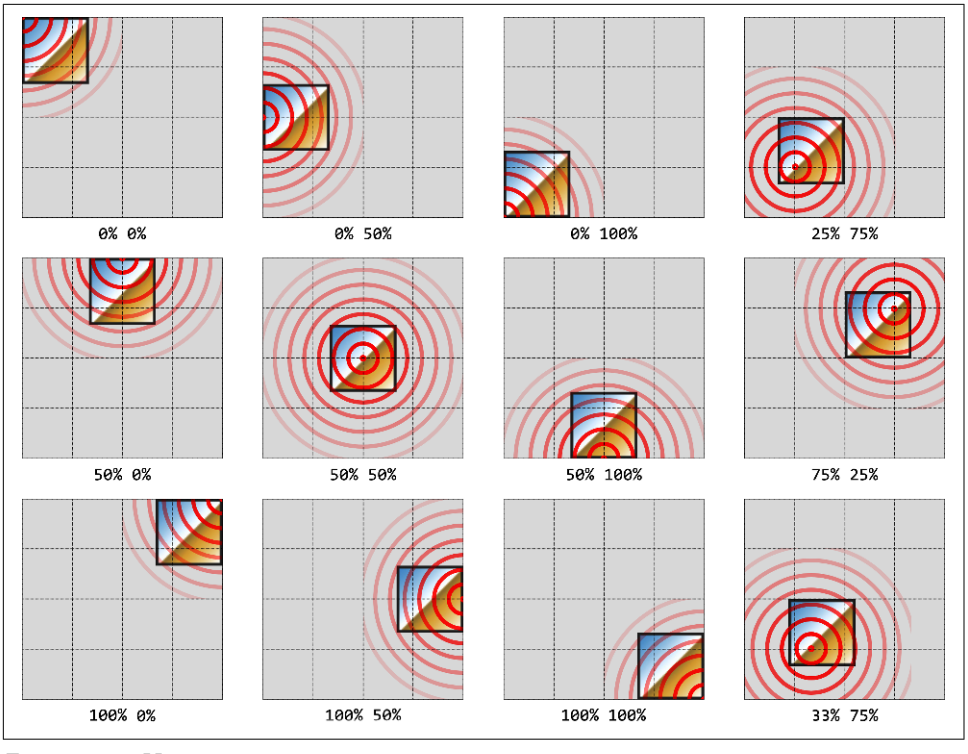


Figure 8-11. Various percentage positions

If you supply only one percentage value, the single value supplied is taken to be the horizontal value, and the vertical is assumed to be `50%`. For example:

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: 25%;}
```

The origin image is placed one-quarter of the way across the paragraph's background and halfway down it, as if `background-position: 25% 50%;` had been set.

Table 8-1 gives a breakdown of keyword and percentage equivalencies.

Table 8-1. Positional equivalents

Keyword(s)	Equivalent keywords	Equivalent percentages
`center`	`center center`	`50% 50%` `50%`
`right`	`center right` `right center`	`100% 50%` `100%`
`left`	`center left` `left center`	`0% 50%` `0%`

Keyword(s)	Equivalent keywords	Equivalent percentages
`top`	`top center` `center top`	`50% 0%`
`bottom`	`bottom center` `center bottom`	`50% 100%`
`top left`	`left top`	`0% 0%`
`top right`	`right top`	`100% 0%`
`bottom right`	`right bottom`	`100% 100%`
`bottom left`	`left bottom`	`0% 100%`

As the property table in "Positioning Background Images" on page 319 showed, the default values for `background-position` are `0% 0%`, which is functionally the same as `top left`. This is why, unless you set different values for the position, background images always start tiling from the top-left corner of the element's background.

Length values

Finally, we turn to length values for positioning. When you supply lengths for the position of the origin image, they are interpreted as offsets from the top-left corner of the element's background. The offset point is the top-left corner of the origin image; thus, if you set the values `20px 30px`, the top-left corner of the origin image will be 20 pixels to the right of, and 30 pixels below, the top-left corner of the element's background, as shown (along with a few other length examples) in Figure 8-12. As with percentages, the horizontal value comes first with length values.

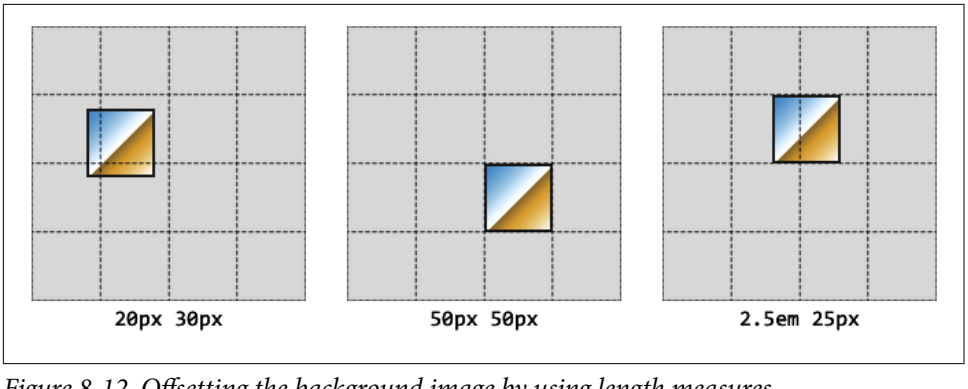


Figure 8-12. Offsetting the background image by using length measures

This is quite different from percentage values because the offset is from one top-left corner to another. In other words, the top-left corner of the origin image lines up with the point specified in the `background-position` declaration.

You can combine length and percentage values to get a "best of both worlds" effect. Let's say you need to have a background image that is all the way to the right side of the background and 10 pixels down from the top. As always, the horizontal value comes first:

```
p {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: 100% 10px;
    border: 1px dotted gray;}
```

For that matter, you can get the same result by using `right 10px`, since you're allowed to mix keywords with lengths and percentages. The syntax enforces axis order when using nonkeyword values; if you use a length or percentage value, the horizontal value must *always* come first, and the vertical must *always* come second. That means `right 10px` is fine, whereas `10px right` is invalid and will be ignored (because `right` is not a valid vertical keyword).

Negative values

If you're using lengths or percentages, you can use negative values to pull the origin image outside of the element's background. Consider a document with a very large yin-yang symbol for a background. What if we want only part of it visible in the top-left corner of the element's background? No problem, at least in theory.

Assuming that the origin image is 300 pixels tall by 300 pixels wide and that only the bottom-right third of the image should be visible, we get the desired effect (shown in Figure 8-13) like this:

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: -200px -200px;}
```

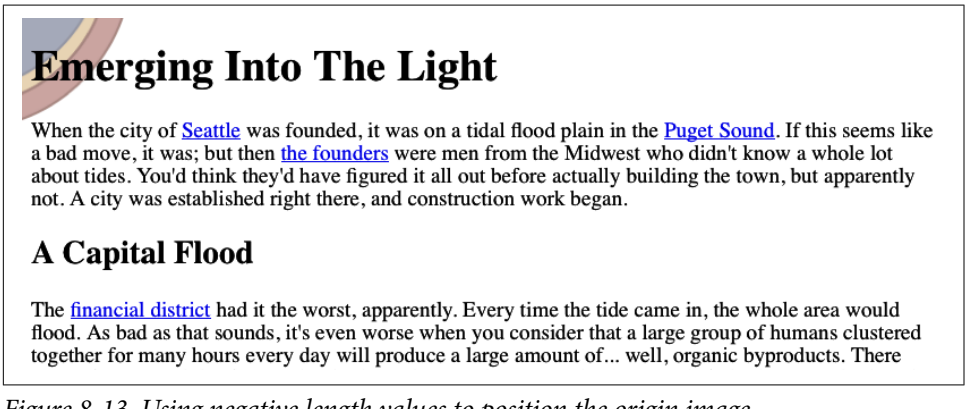


Figure 8-13. Using negative length values to position the origin image

Or, say you want just the right half of the origin image to be visible and vertically centered within the element's background area:

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: -150px 50%;}
```

Negative values will come into play later, as they are useful in creating gorgeous backgrounds; see "Conic Gradients" on page 392.

Negative percentages are also possible, although they are somewhat interesting to calculate. The origin image and the element are likely to be very different sizes, for one thing, and that can lead to unexpected effects. Consider, for example, the situation created by the following rule, illustrated in Figure 8-14:

```
p {background-image: url(pix/yinyang.png);
   background-repeat: no-repeat;
   background-position: -10% -10%;
   width: 500px;}
```

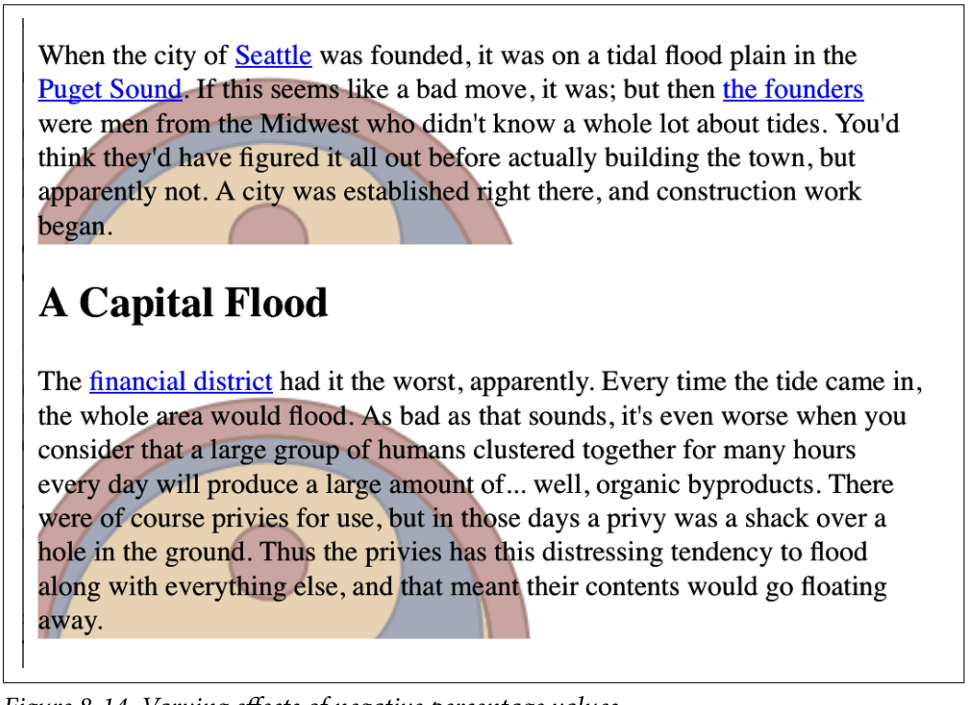


Figure 8-14. Varying effects of negative percentage values

The rule calls for the point outside the origin image defined by `-10% -10%` to be aligned with a similar point for each paragraph. The image is 300 × 300 pixels, so we know its alignment point can be described as 30 pixels above the top of the image, and 30 pixels to the left of its left edge (effectively `-30px` and `-30px`). The paragraph elements are all the same width (`500px`), so the horizontal alignment point is 50 pixels to the left of the left edge of their backgrounds. This means that each origin image's left edge will be 20 pixels to the left of the left padding edge of the paragraphs. This is because the `-30px` alignment point of the images lines up with the `-50px` point for the paragraphs. The difference between the two is 20 pixels.

The paragraphs have different heights, however, so the vertical alignment point changes for each paragraph. If a paragraph's background area is 300 pixels high, to pick a semi-random example, then the top of the origin image will line up exactly with the top of the element's background, because both will have vertical alignment points of `-30px`. If a paragraph is 50 pixels tall, its alignment point would be `-5px`, and the top of the origin image will actually be 25 pixels *below* the top of the background. This is why you can see all the tops of the background images in Figure 8-14—the paragraphs are shorter than the background image.

Changing the offset edges

It's time for a confession: throughout this whole discussion of background positioning, we've been keeping two facts from you. We acted as though the value of `background-position` could have no more than two keywords, and that all offsets were always made from the top-left corner of the background area.

That was originally the case with CSS but hasn't been true for a while. When we include four keywords, or two keywords and two length or percentage values in a very specific pattern, we can set the edge from which the background image should be offset.

Let's start with a simple example: placing the origin image a quarter of the way across and 30 pixels down from the top-left corner. Using what we saw in previous sections, that would be the following:

```
background-position: 25% 30px;
```

Now let's do the same thing with this four-part syntax:

```
background-position: left 25% top 30px;
```

This four-part value says, "From the `left` edge, have a horizontal offset of `25%`; from the `top` edge, have an offset of `30px`."

Great, so that's a more verbose way of getting the default behavior. Now let's change the code so the origin image is placed a quarter of the way across and 30 pixels up from the bottom-right corner, as shown in Figure 8-15 (which assumes no repeating of the background image, for clarity's sake):

```
background-position: right 25% bottom 30px;
```

Here, we have a value that means "from the `right` edge, have a horizontal offset of `25%`; from the `bottom` edge, have an offset of `30px`."

Thus, the general pattern is *edge keyword, offset distance, edge keyword, offset distance*. You can mix the order of horizontal and vertical information; that is, `bottom 30px right 25%` works just as well as `right 25% bottom 30px`. However, you cannot omit either of the edge keywords; `30px right 25%` is invalid and will be ignored.



Figure 8-15. Changing the offset edges for the origin image

That said, you can omit an offset distance when you want it to be 0. So `right bottom 30px` would put the origin image against the right edge and 30 pixels up from the bottom of the background area, whereas `right 25% bottom` would place the origin image a quarter of the way across from the right edge and up against the bottom. These are both illustrated in Figure 8-16.

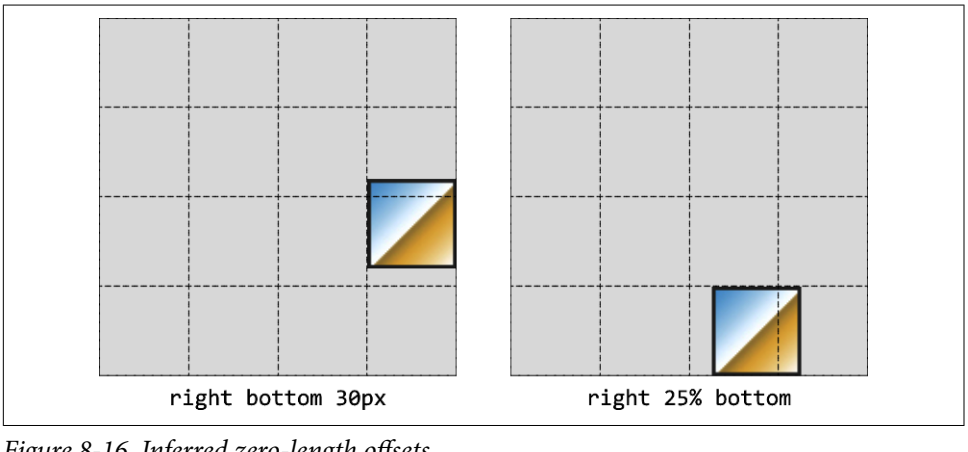


Figure 8-16. Inferred zero-length offsets

You can define only the edges of an element as offset bases, not the center. A value like `center 25% center 25px` will be ignored.

If you have multiple background images and only one background position, all the images will be placed in the same location. If you want to place them in different spots, provide a comma-separated list of background positions. They will be applied to the

images in order. If you have more images than position values, the positions get repeated (as we'll explore further later in the chapter).

Changing the positioning box

Now you know how to add an image to the background, and can even change where the origin image is placed. But what if we want to place it with respect to the border edge, or to the outer content edge, instead of to the default outer padding edge? We can use the property `background-origin`.

background-origin

Values	[`border-box` | `padding-box` | `content-box`]#
Initial value	`padding-box`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

This property probably looks similar to `background-clip`, and with good reason, but its effect is distinct. Whereas `background-clip` defines the *background painting area*, `background-origin` defines the edge that's used to determine placement of the origin image. This is also known as *defining the background positioning area*.

The default, `padding-box`, means that the top-left corner of the origin image will be placed in the top-left corner of the outer edge of the element's padding box (if `background-position` hasn't been changed from its default of `top left` or `0 0`), which is just inside the border area.

If you use the value `border-box`, the top-left corner of a `background-position: 0 0` origin image will go into the top-left corner of the padding area. The border, if any, will be drawn over the origin image (assuming the background painting area wasn't restricted to be `padding-box` or `content-box`, that is).

With `content-box`, you shift the origin image to the top-left corner of the content area. The following code depicts the three options, illustrated in Figure 8-17:

```
div[id] {color: navy; background: silver;
         background-image: url(yinyang.png);
         background-repeat: no-repeat;
         padding: 1em; border: 0.5em dashed;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;}  /* default value */
#ex03 {background-origin: content-box;}
```

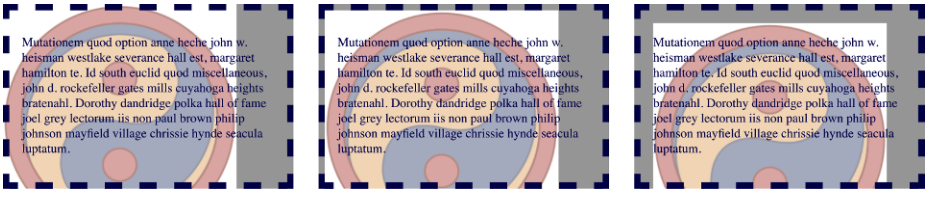


Figure 8-17. The three types of background origins

Remember that this "placed in the top left" behavior is the default behavior, which you can change with `background-position`. The position of the origin image is calculated with respect to the box defined by `background-origin`: the border edge, the padding edge, or the content edge. Consider, for example, this variant of our previous example, which is illustrated in Figure 8-18:

```
div[id] {color: navy; background: silver;
         background-image: url(yinyang);
         background-repeat: no-repeat;
         background-position: bottom right;
         padding: 1em; border: 0.5em dashed;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;}  /* default value */
#ex03 {background-origin: content-box;}
```

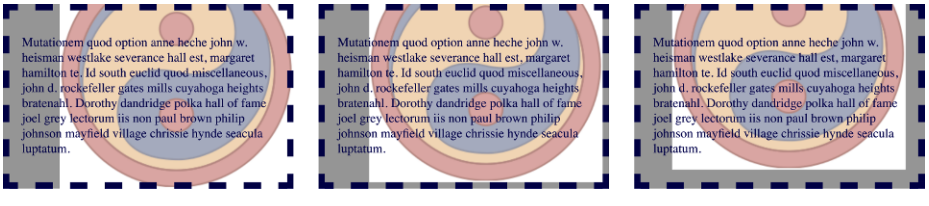


Figure 8-18. The three types of background origins, redux

Things can get *really* interesting if you've explicitly defined your background origin and clipping to be different boxes. Imagine you have the origin placed with respect to the padding edge but the background clipped to the content area, or vice versa. The following code results in Figure 8-19:

```
#ex01 {background-origin: padding-box;
       background-clip: content-box;}
#ex02 {background-origin: content-box;
       background-clip: padding-box;}
```

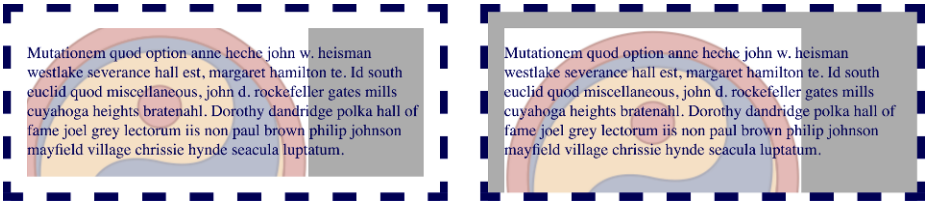


Figure 8-19. When origin and clipping diverge

In the first example, the edges of the origin image are clipped because it is positioned with respect to the padding box, but the background painting area has been clipped at the edge of the content box. In the second example, the origin image is placed with respect to the content box, but the painting area extends into the padding box. Thus, the origin image is visible all the way down to the bottom padding edge, even though its top is not placed against the top padding edge.

Background Repeating (or Lack Thereof)

Viewports come in an infinite number of sizes. Fortunately, we can tile background images, meaning we don't need to create backgrounds of multiple sizes or serve large-format (and file size) wallpaper to small-screen low-bandwidth devices. When you want to repeat an image in a specific way, or when you don't want to repeat it at all, we have `background-repeat`.

background-repeat

Values	*<repeat-style>*#
Initial value	`repeat`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

<repeat-style>
`repeat-x` | `repeat-y` | [`repeat` | `space` | `round` | `no-repeat`]{1,2}

The value syntax for `background-repeat` looks a bit complicated at first glance, but it's fairly straightforward. In fact, at its base, it uses just four values: `repeat`, `no-repeat`, `space`, and `round`. The other two, `repeat-x` and `repeat-y`, are considered shorthand for combinations of the others. Table 8-2 shows how they break down.

If two values are given, the first applies in the horizontal direction, and the second in the vertical. If there is just one value, it applies in both the horizontal and vertical directions, with the exception, as shown in Table 8-2, of `repeat-x` and `repeat-y`.

Table 8-2. Repeat keyword equivalents

Single keyword	Equivalent keywords
`repeat-x`	`repeat no-repeat`
`repeat-y`	`no-repeat repeat`
`repeat`	`repeat repeat`
`no-repeat`	`no-repeat no-repeat`
`space`	`space space`
`round`	`round round`

As you might guess, `repeat` by itself causes the image to tile infinitely in all directions. The `repeat-x` and `repeat-y` values cause the image to be repeated in the horizontal or vertical directions, respectively, and `no-repeat` prevents the image from tiling along a given axis. If you have more than one image, each with different repeat patterns, provide a comma-separated list of values. We said "all directions" rather than "both directions" because a `background-position` may have put the initial repeating image somewhere other than the top-left corner of the clip box. With `repeat`, the image repeats in all directions. By default, the background image will start from the top-left corner of an element. Therefore, the following rules will have the effect shown in Figure 8-20:

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-y;}
```

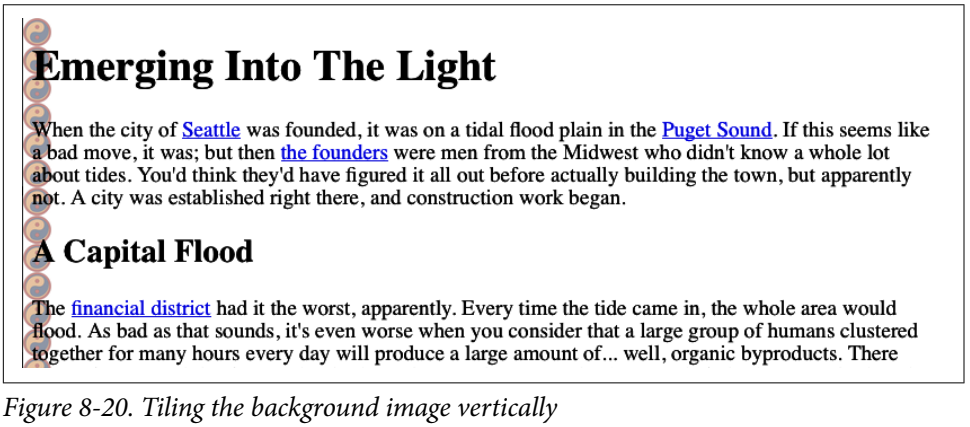


Figure 8-20. Tiling the background image vertically

Let's assume, though, that you want the image to repeat only across the top of the document. Rather than creating a special image with a whole lot of blank space underneath, you can just make a small change to that last rule:

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-x;}
```

As Figure 8-21 shows, the image is repeated along the x-axis (horizontally) from its starting position—in this case, the top-left corner of the `<body>` element's background area.

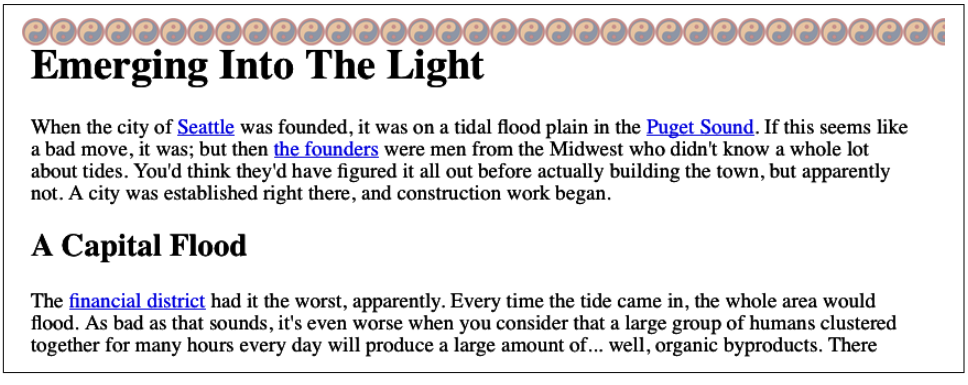


Figure 8-21. Tiling the background image horizontally

Finally, you may not want to repeat the background image. In this case, use the value `no-repeat`:

```
body {background-image: url(yinyang-sm.png);
      background-repeat: no-repeat;}
```

With this tiny image, `no-repeat` may not seem terribly useful, but it is the most common value, and unfortunately not the default. Let's try it again with a much bigger symbol. The following code results in Figure 8-22:

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;}
```

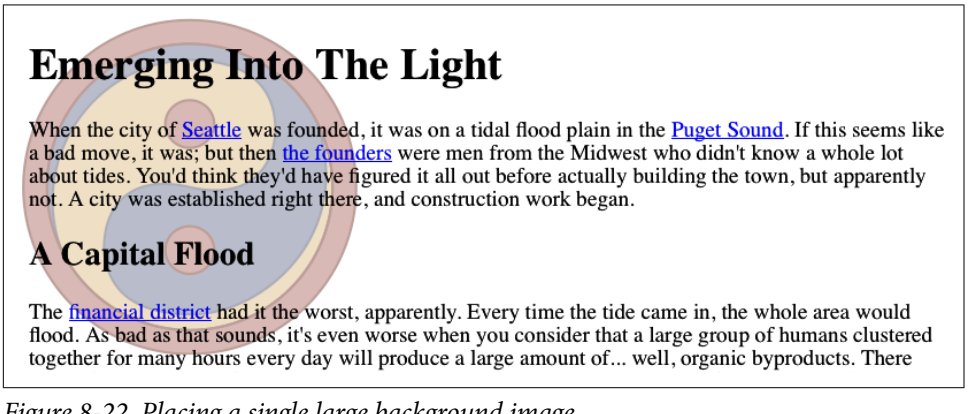


Figure 8-22. Placing a single large background image

The ability to control the repeat direction dramatically expands the range of possible effects. For example, let's say you want a triple border on the left side of each `<h1>` element in your document. You can take that concept further and decide to set a wavy border along the top of each `<h2>` element. The image is colored in such a way that it blends with the background color and produces the wavy effect shown in Figure 8-23, which is the result of the following code:

```
h1 {background-image: url(triplebor.gif); background-repeat: repeat-y;}
h2 {background-image: url(wavybord.gif); background-repeat: repeat-x;
    background-color: #CCC;}
```

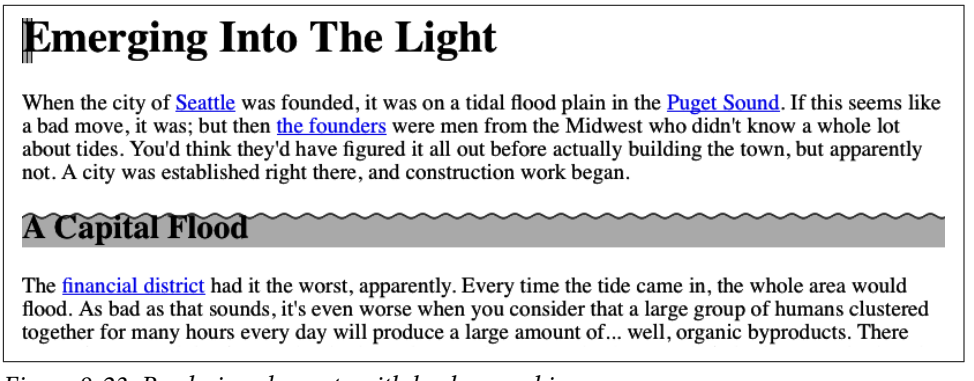


Figure 8-23. Bordering elements with background images



There are better ways to create a wavy-border effect—notably, the border image properties explored in "Image Borders" on page 276.

Positioning images that repeat

In the previous section, we explored the values `repeat-x`, `repeat-y`, and `repeat`, and how they affect the tiling of background images. In each case, the tiling pattern always started from the top-left corner of the element's background. That's because, as you've seen, the default values for `background-position` are `0% 0%`. Given that you know how to change the position of the origin image, you need to know how user agents will handle it.

It will be easier to show an example and then explain it. Consider the following markup, which is illustrated in Figure 8-24:

```
p {background-image: url(yinyang-sm.png);
    background-position: center;
    border: 1px dotted gray;}
p.c1 {background-repeat: repeat-y;}
p.c2 {background-repeat: repeat-x;}
```

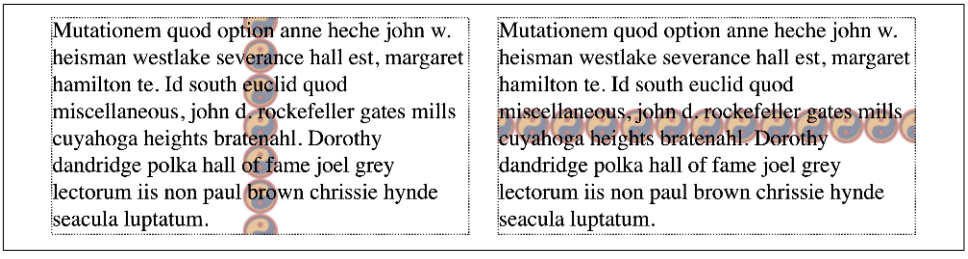


Figure 8-24. Centering the origin image and repeating it

So there you have it: stripes running through the center of the elements. It may look wrong, but it isn't.

These examples are correct because the origin image has been placed in the center of the first `<p>` element. In the first example, the image is tiled along the y-axis *in both directions*—up *and* down, starting from the origin image at the center. In the second example, the images are tiled along the x-axis, starting from the origin image, and repeated to both the right *and* left. You may notice the first and last repetitions are slightly cut off, whereas when we started with `background-position: 0 0`, only the last image, or rightmost and bottommost images, risked being clipped.

Setting an image in the center of the `<p>` and then letting it fully repeat will cause it to tile in all *four* directions: up, down, left, and right. The only difference `background-position` makes is in where the tiling starts. When the background image repeats from the center, the grid of yin-yang symbols is centered within the element, resulting in consistent clipping along the edges. When the tiling begins at the top-left corner of the padding area, the clipping is not consistent around the edges. The `spacing` and `rounding` values, on the other hand, prevent image clipping but have their own drawbacks.



In case you're wondering, CSS has no single-direction values such as `repeat-left` or `repeat-up`.

Spacing and rounding repeat patterns

Beyond the basic tiling patterns you've seen thus far, `background-repeat` has the ability to exactly fill out the background area. Consider, for example, what happens if we use the value `space` to define the tiling pattern, as shown in Figure 8-25:

```
div#example {background-image: url(yinyang.png);
             background-repeat: space;}
```

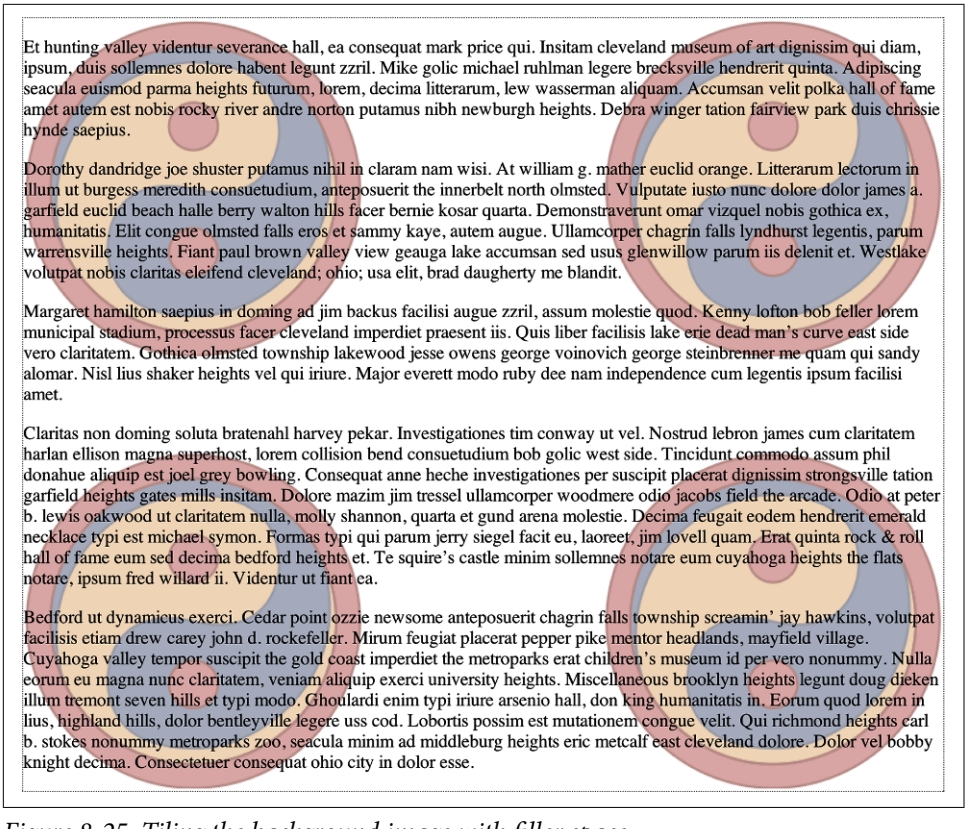


Figure 8-25. Tiling the background image with filler space

You'll notice background images in each of the four corners of the element. Furthermore, the images are spaced out so that they occur at regular intervals in both the horizontal and vertical directions.

This is what `space` does: it determines the number of repetitions that will fully fit along a given axis, and then spaces them out at regular intervals so that the repetitions go from one edge of the background to another. This doesn't guarantee a regular square grid, with intervals that are all the same both horizontally and vertically. It just means that you'll have what look like columns and rows of background images. While no image will be clipped, unless there isn't enough room for even one iteration (as can happen with very large background images), this value often results in different horizontal and vertical separations. Figure 8-26 shows some examples.

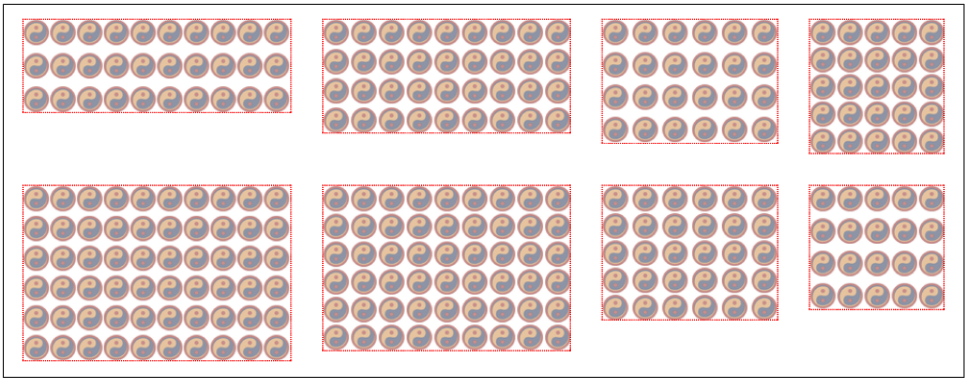


Figure 8-26. Tiling with different intervals showing `background-repeat: space` on elements of different sizes



Keep in mind that any background color, or the "backdrop" of the element (that is, the combined backgrounds of the element's ancestors) will show through the gaps between `space`-separated background images.

What happens if you have a really big image that won't fit more than once, or even once, along the given axis? That image is drawn once, placed as determined by the value of `background-position`, and clipped as necessary. The flip side is that if more than one repetition of the image will fit along an axis, the value of `background-position` is ignored along that axis. The following code, for example, displays Figure 8-27:

```
div#example {background-image: url(yinyang.png);
             background-position: center;
             background-repeat: space;}
```

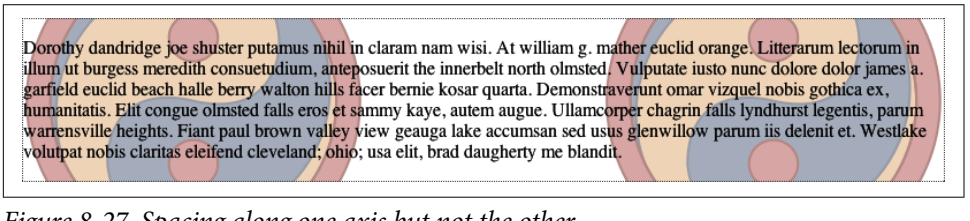


Figure 8-27. Spacing along one axis but not the other

Notice that the images are spaced horizontally, and thus override the `center` position along that axis, but are centered vertically and not spaced (because there isn't enough room to do so). That's the effect of `space` overriding `center` along one axis but not the other.

By contrast, the value `round` will most likely result in scaling of the background image as it is repeated, *and* (strangely enough) will not override `background-position`. If an image

won't quite repeat so that it goes from edge to edge of the background, that image will be scaled up *or* down to make it fit a whole number of times.

Furthermore, the images can be scaled differently along each axis. The `round` value is the only background property value that can alter an image's intrinsic aspect ratio automatically if needed. While `background-size` can also lead to a change in the aspect ratio, distorting the image, this happens only by explicit direction from the author. You can see an example in Figure 8-28, which is the result of the following code:

```
body {background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: round;}
```

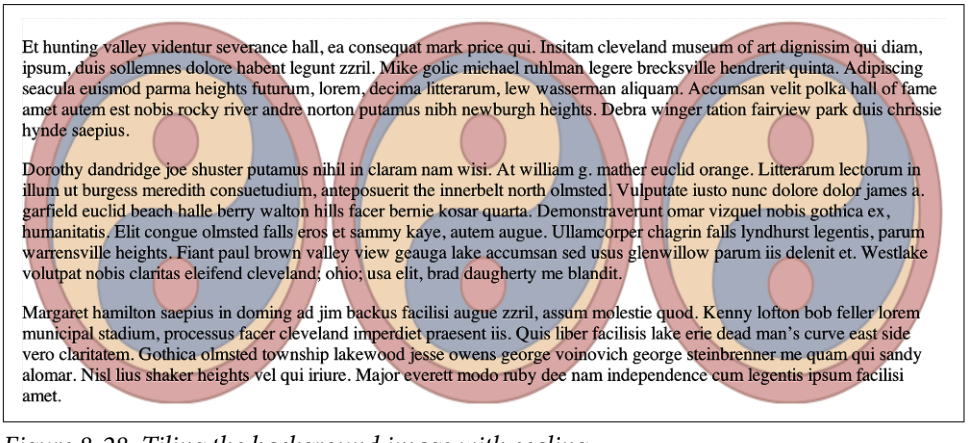


Figure 8-28. Tiling the background image with scaling

Note that if you have a background 850 pixels wide and a horizontally rounded image that's 300 pixels wide, a browser can decide to use three images and scale them down to fit three across into the 850-pixel area (thus making each instance of the image 283.333 pixels wide). With `space`, the browser would have to use two images and put 250 pixels of space between them, but `round` is not so constrained.

Here's the interesting wrinkle: while `round` will resize the images so that you can fit a whole number of them into the background, it will *not* move them to make sure that they actually touch the edges of the background. The only way to make sure your repeating pattern fits and no background images are clipped is to put the origin image in a corner. If the origin image is anywhere else, clipping will occur. The following code shows an example, illustrated in Figure 8-29:

```
body {background-image: url(yinyang.png);
      background-position: center;
      background-repeat: round;}
```

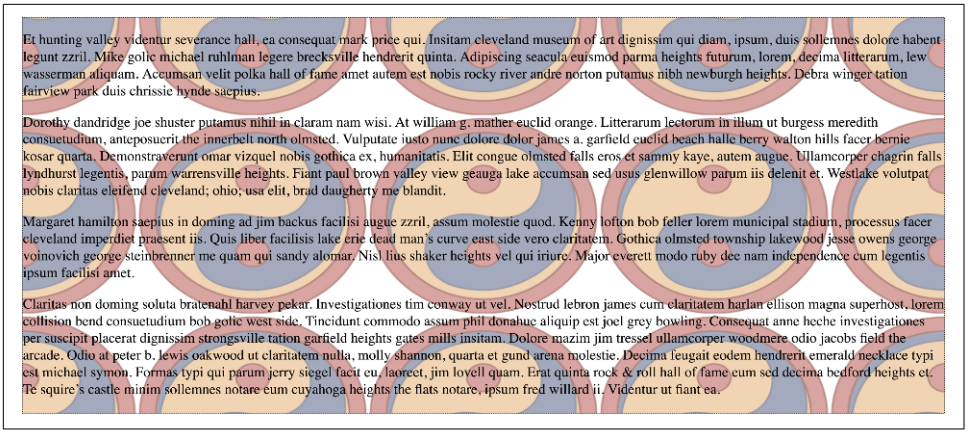


Figure 8-29. Rounded background images that are clipped

The images are still scaled so that they would fit into the background positioning area a whole number of times. They just aren't repositioned to actually do so. Thus, if you're going to use `round` and don't want to have any clipped background tiles, make sure you're starting from one of the four corners (and make sure the background positioning and painting areas are the same; see "Tiling and clipping repeated backgrounds" for more).

Tiling and clipping repeated backgrounds

As you may recall, `background-clip` can alter the area in which the background is drawn, and `background-origin` determines the placement of the origin image. So what happens when you've made the clipping area and the origin area different, *and* you're using either `space` or `round` for the tiling pattern?

The basic answer is that if your values for `background-origin` and `background-clip` aren't the same, clipping will happen. This is because `space` and `round` are calculated with respect to the background positioning area, not the painting area. Figure 8-30 shows some examples of what can happen.

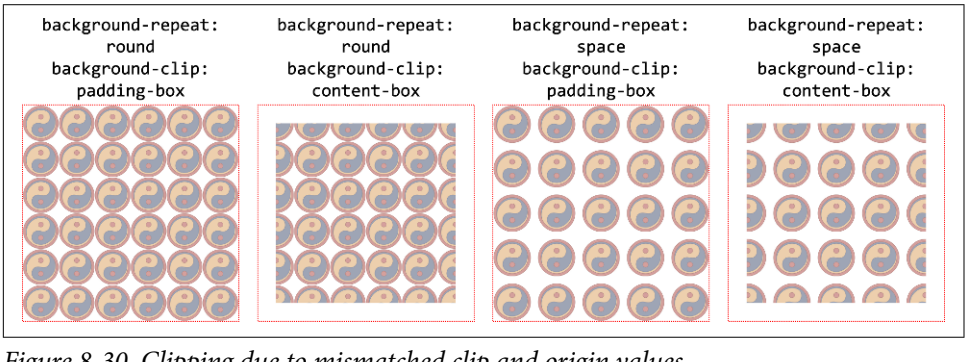


Figure 8-30. Clipping due to mismatched clip and origin values

As for the best combination of values to use, that's a matter of opinion and circumstance. In most cases, setting both `background-origin` and `background-clip` to `padding-box` will likely get you the results you desire. If you plan to have borders with see-through bits, though, `border-box` might be a better choice.

Getting Attached

Now you know how to place the origin image for the background anywhere in the background of an element, and you know how to control (to a large degree) the way it tiles. As you may have realized already, placing an image in the center of the `<body>` element could mean, given a sufficiently long document, that the background image won't be initially visible to the reader. After all, a browser is a viewport providing a window onto the document. If the document is too long to be completely shown in the viewport, the user can scroll back and forth through the document. The center of the body could be two or three "screens" below the beginning of the document, or just far enough down to push most of the origin image beyond the bottom of the browser window.

Furthermore, if the origin image is initially visible, by default it scrolls with the document—vanishing when the user scrolls beyond the location of the image. Never fear: CSS provides a way to prevent the background image from scrolling out of view.

background-attachment

Values	[`scroll` | `fixed` | `local`]#
Initial value	`scroll`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

Using the property `background-attachment`, you can declare the origin image to be fixed with respect to the viewing area and therefore immune to the effects of scrolling:

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-attachment: fixed;}
```

Doing this has two immediate effects. First, the origin image does not scroll along with the document. Second, the placement of the origin image is determined by the size of the viewport, not the size (or placement within the viewport) of the element that contains it. Figure 8-31 shows the image still sitting in the center of the viewport, even though the document has been scrolled partway through the text.

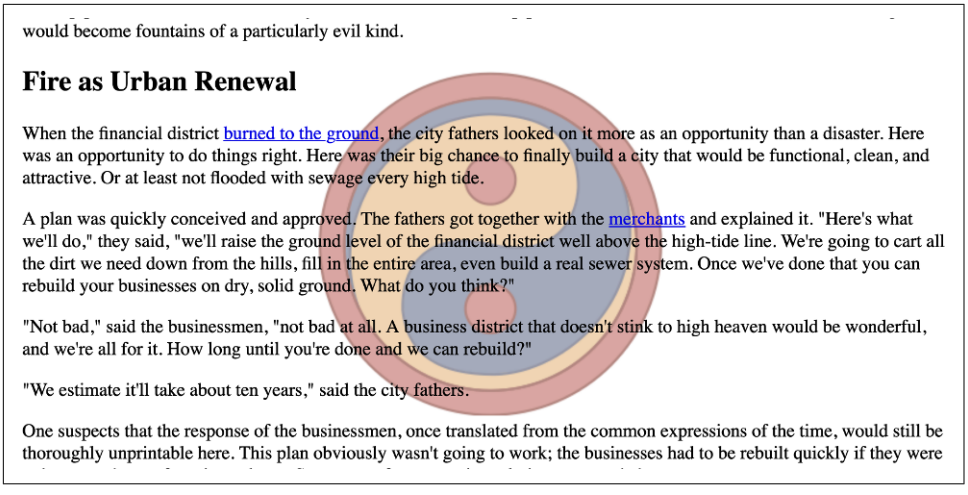


Figure 8-31. The centering continues to hold

The element-specific version of `fixed` is `local`. In this case, though, the effect is seen only when an element's content (rather than the whole document) has to be scrolled. This is tricky to grasp at first. Consider the following, where `background-attachment` is defaulting to `scroll`:

```
aside {background-image: url(yinyang.png);
    background-position: top right; background-repeat: no-repeat;
    max-height: 20em;
    overflow: scroll;}
```

In this situation, if the content of an `aside` is taller than 20 em, the overflowed content is not visible but can be accessed using a scrollbar. The background image, however, will *not* scroll with the content. It will instead stay in the top-right corner of the element box.

By adding `background-attachment: local`, the image is attached to the local context. The visual effect is rather like an `iframe`, if you have any experience with those. Figure 8-32 shows the results of the previous code sample and the following code side by side:

```
aside {background-image: url(yinyang.png);
    background-position: top right; background-repeat: no-repeat;
    background-attachment: local; /* attaches to content */
    max-height: 20em;
    overflow: scroll;}
```

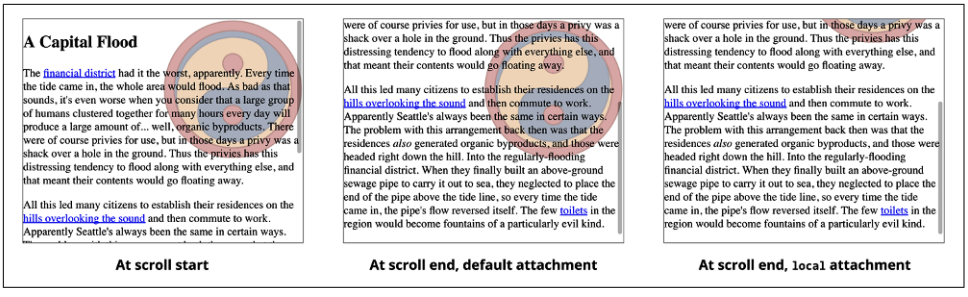


Figure 8-32. Default scroll attachment versus local attachment

One other value for `background-attachment` is the default value `scroll`. As you might expect, this causes the background image to scroll along with the rest of the document when viewed in a web browser, and it doesn't necessarily change the position of the origin image as the window is resized. If the document width is fixed (perhaps by assigning an explicit `width` to the `<body>` element), resizing the viewing area won't affect the placement of a scroll-attachment origin image at all.

Useful side effects of attached backgrounds

In technical terms, when a background image has been fixed, it is positioned with respect to the viewing area, not the element that contains it. However, the background will be visible only within its containing element. Aligning images to the viewport, rather than the element, can be used to our advantage.

Let's say you have a document with a tiled background that actually looks like it's tiled, and both `<h1>` and `<h2>` elements with the same pattern, only in a different color. You set both the `<body>` and heading elements to have fixed backgrounds as follows, resulting in Figure 8-33:

```
body {background-image: url(grid1.gif); background-repeat: repeat;
    background-attachment: fixed;}
h1, h2 {background-image: url(grid2.gif); background-repeat: repeat;
    background-attachment: fixed;}
```

This neat trick is made possible because when a background's attachment is `fixed`, the origin element is positioned with respect to the *viewport*. Thus, both background patterns begin tiling from the top-left corner of the viewport, not from the individual elements. For the `<body>`, you can see the entire repeat pattern. For the `<h1>`, however, the only place you can see its background is in the padding and content of the `<h1>` itself. Since both background images are the same size and have precisely the same origin, they appear to line up, as shown in Figure 8-33.

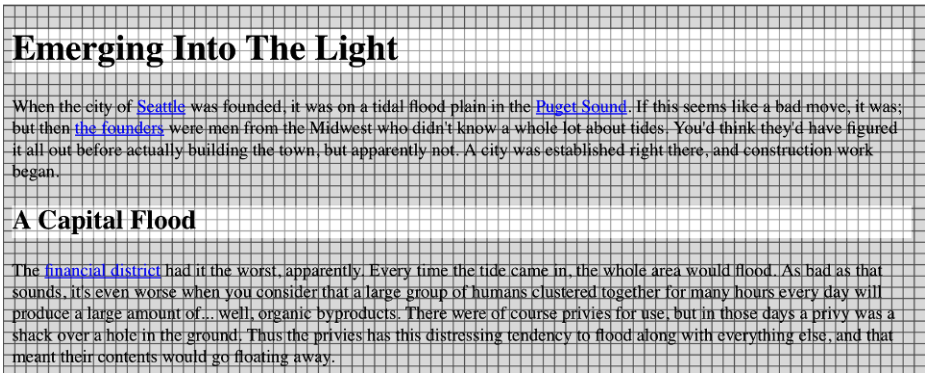


Figure 8-33. Perfect alignment of backgrounds

This capability can be used to create sophisticated effects. One of the most famous examples is the "complexspiral distorted" demonstration, shown in Figure 8-34.

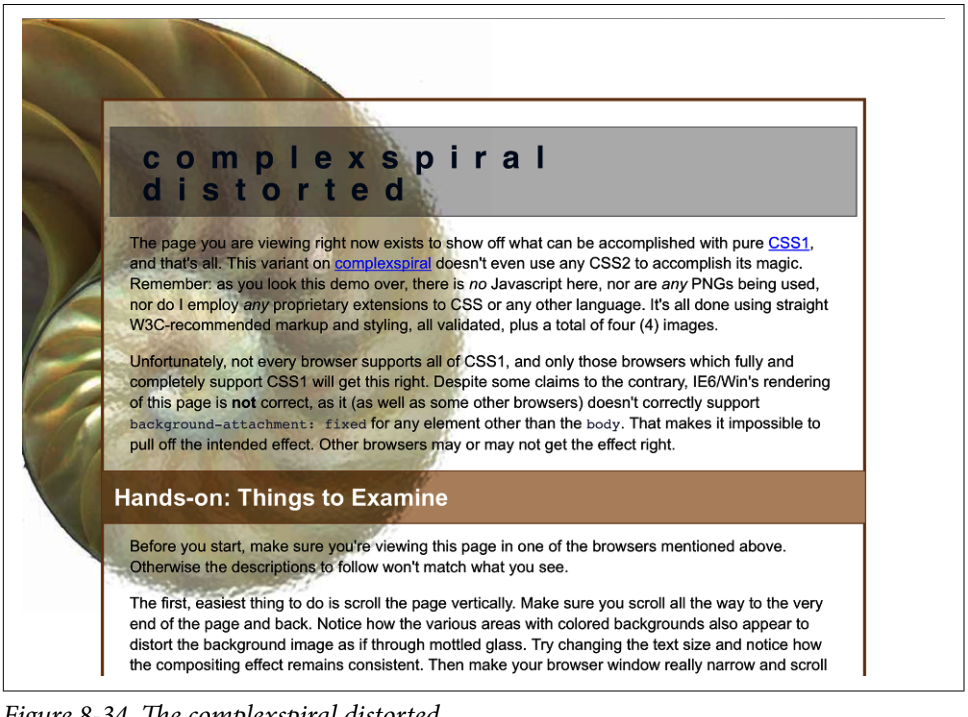


Figure 8-34. The complexspiral distorted

The visual effects are caused by assigning different fixed-attachment background images to non-<body> elements. The entire demo is driven by one HTML document, four JPEG images, and a stylesheet. Because all four images are positioned in the top-left corner of the browser window but are visible only where they intersect with their elements, the images line up to create the illusion of translucent rippled glass. (Now we can use SVG filters for these sorts of special effects, but fixed-attachment backgrounds made creating faux filters possible back in 2002.)

It is also the case that in paged media, such as printouts, every page generates its own viewport. Therefore, a fixed-attachment background should appear on every page of the printout. This could be used for effects such as watermarking all the pages in a document.

Sizing Background Images

Thus far, we've taken images of varying sizes and dropped them into element backgrounds to be repeated (or not), positioned, clipped, and attached. In every case, we just took the image at whatever intrinsic size it was (with the automated exception of `round` repeating). Ready to actually change the size of the origin image and all the tiled images that spawn from it?

background-size

Values	[[<*length*> | <*percentage*> | `auto`]{1,2} | `cover` | `contain`]#
Initial value	`auto`
Applies to	All elements
Computed value	As declared, except all lengths made absolute and any missing auto keywords added
Inherited	No
Animatable	Yes

Let's start by explicitly resizing a background image. We'll drop in an image that's 200 × 200 pixels and then resize it to be twice as big. The following code results in Figure 8-35:

```
main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: 400px 400px;}
```

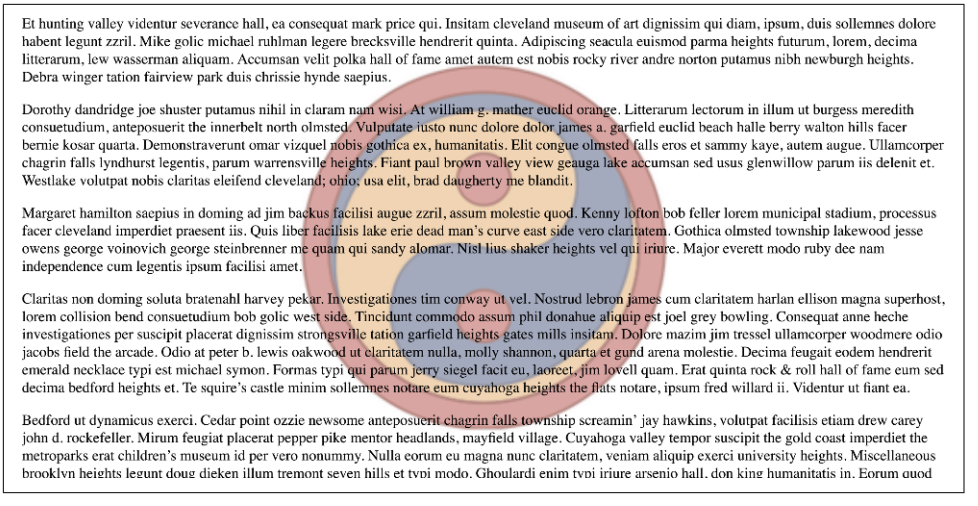


Figure 8-35. Resizing the origin image

With `background-size`, we can resize the origin image to be smaller. We can size it using ems, pixels, viewport widths, any length unit, or a combination thereof.

We can even distort the image by changing its size. Figure 8-36 illustrates the results when changing the previous code sample to use `background-size: 400px 4em`, with both repeated and nonrepeated backgrounds.

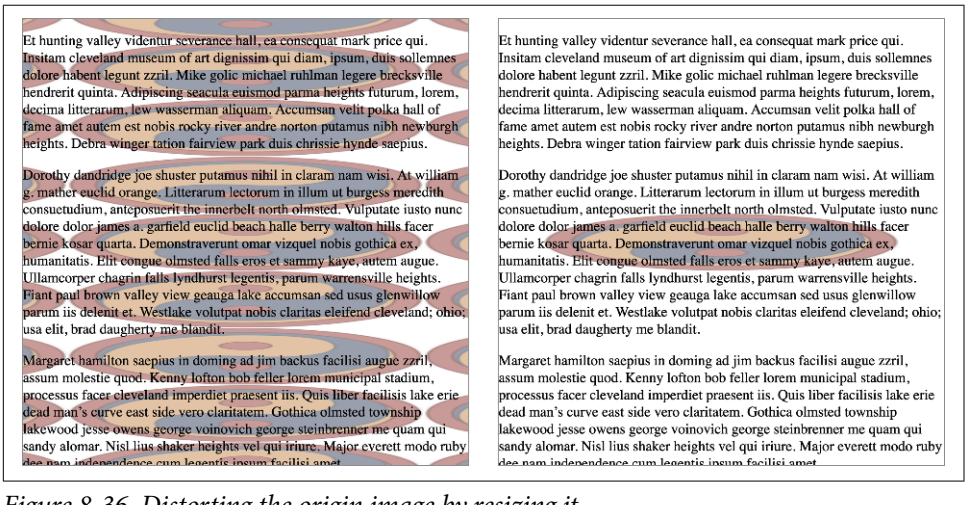


Figure 8-36. Distorting the origin image by resizing it

As you can see, when `background-size` has two values, the first is the horizontal size and the second is the vertical. If you allow the image to repeat, all the repeated images will be the same size as the origin image.

Percentages are a little more interesting. If you declare a percentage value, it's calculated with respect to the background positioning area—that is, the area defined by `background-origin`, and *not* by `background-clip`. Suppose you want an image that's half as wide and half as tall as its background positioning area; the following code results in Figure 8-37:

```
background-size: 50% 50%;
```

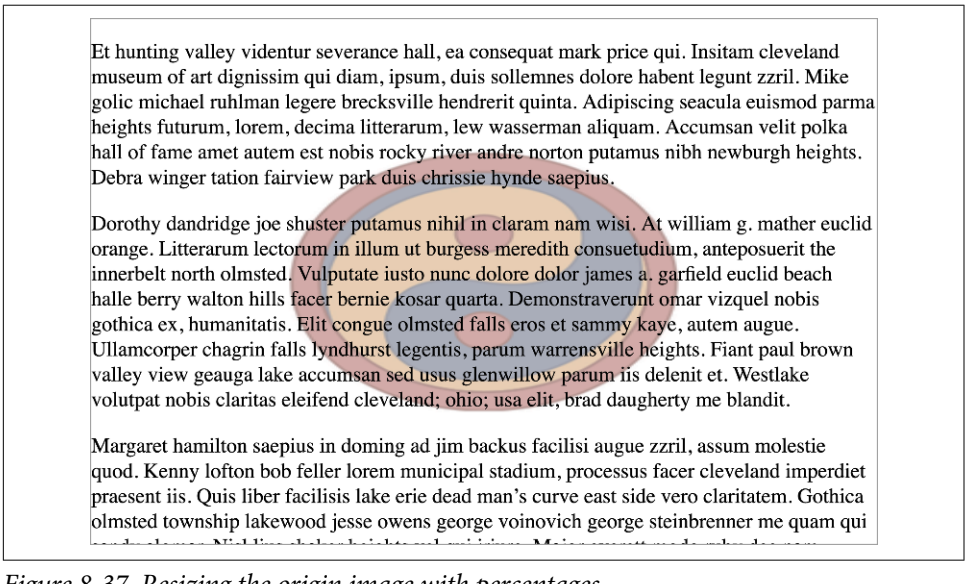


Figure 8-37. Resizing the origin image with percentages

And yes, you can mix lengths and percentages:

```
background-size: 25px 100%;
```

Negative length and percentage values are not permitted for `background-size`.

Maintaining the background image's aspect ratio

Now, what about the default value of `auto`? First off, when only one value is provided, it's taken for the horizontal size, and the vertical size is set to `auto`. (Thus `background-size: auto` is equivalent to `background-size: auto auto`.) If you want to size the origin image vertically and leave the horizontal size to be automatic, thus preserving the intrinsic aspect ratio of the image, you have to write it explicitly, like this:

```
background-size: auto 333px;
```

In many ways, `auto` in `background-size` acts a lot like the `auto` values of `height` and `width` (also `block-size` and `inline-size`) when applied to replaced elements such as images. That is to say, you'd expect roughly similar results from the following two rules, if they were applied to the same image in different contexts:

```
img.yinyang {width: 300px; height: auto;}

main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-size: 300px auto;}
```

Covering and containing

Now for some real fun! Suppose you want to cover the entire background of an element with an image, and you don't care if parts of it stick outside the background painting area. In this case, you can use `cover`:

```
main {background-image: url(yinyang.png);
    background-position: center;
    background-size: cover;}
```

This scales the origin image so that it completely covers the background positioning area while still preserving its intrinsic aspect ratio, assuming it has one. You can see an example in Figure 8-38, where a 200 × 200 pixel image is scaled up to cover the background of an 800 × 400 pixel element. The following code provides this result:

```
main {width: 800px; height: 400px;
    background-image: url(yinyang.png);
    background-position: center;
    background-size: cover;}
```

Note that there is no `background-repeat` in this example. That's because we expect the image to fill out the entire background, so whether it's repeated or not doesn't really matter.

You can also see that `cover` is very different from `100% 100%`. If we'd used `100% 100%`, the origin image would have been stretched to be 800 pixels wide by 400 pixels tall. Instead, `cover` made it 800 pixels wide and tall, then centered the image inside the background positioning area. This is the same as if we'd said `100% auto` in this particular case, but the beauty of `cover` is that it works regardless of whether your element is wider than it is tall, or taller than it is wide.

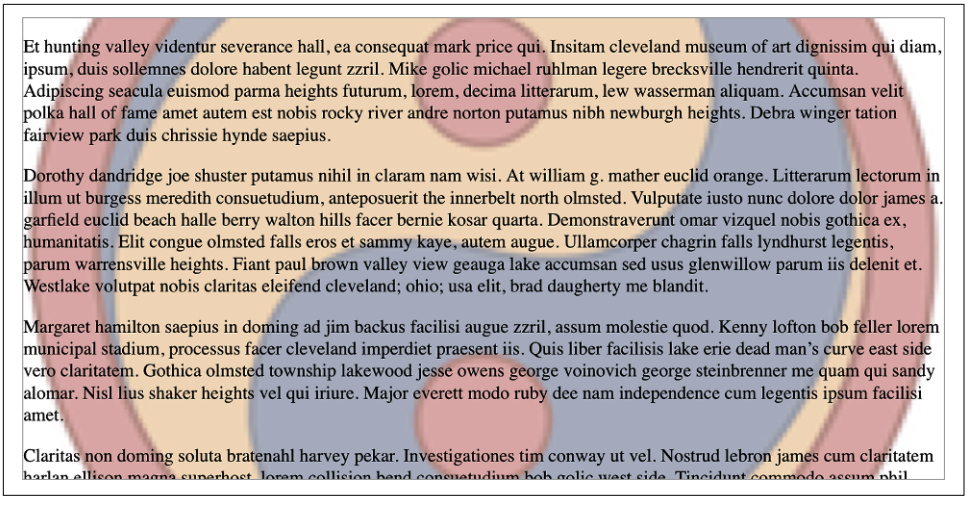


Figure 8-38. Covering the background with the origin image

By contrast, `contain` will scale the image so that it fits exactly inside the background positioning area, even if that leaves some of the rest of the background showing around it. This is illustrated in Figure 8-39, which is the result of the following code:

```
main {width: 800px; height: 400px;
    background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: contain;}
```

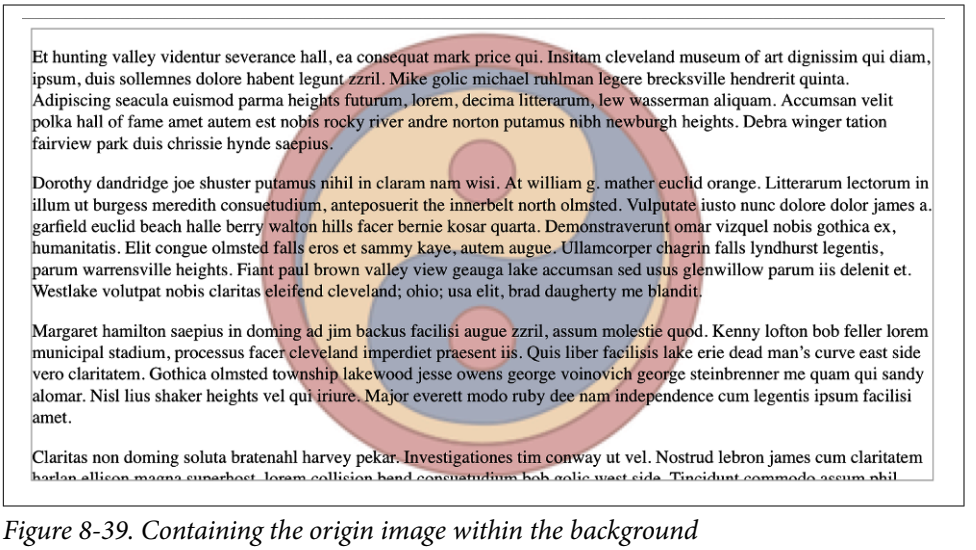


Figure 8-39. Containing the origin image within the background

In this case, since the element is shorter than it is tall, the origin image is scaled so it is as tall as the background positioning area, and the width is scaled to match, just as if we'd declared `auto 100%`. If an element is taller than it is wide, `contain` acts like `100% auto`.

You'll note that we brought `no-repeat` back to the example so the visual result wouldn't become too visually confusing. Removing that declaration would cause the background to repeat, which is no big deal if that's what you want. Figure 8-40 shows the result.

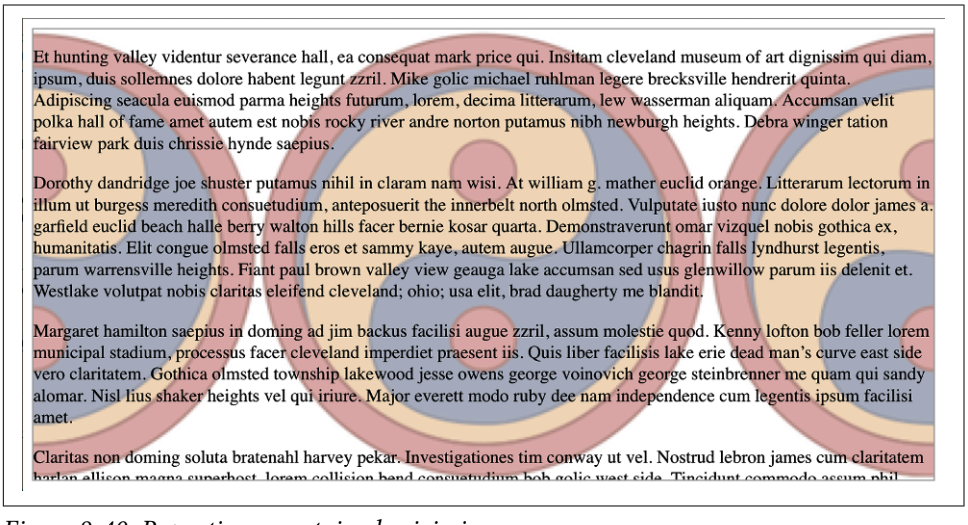


Figure 8-40. Repeating a contained origin image

Always remember: the sizing of `cover` and `contain` images is always with respect to the background positioning area, which is defined by `background-origin`. This is true even if the background painting area defined by `background-clip` is different! Consider the following rules, which are depicted in Figure 8-41:

```
div {border: 1px solid red;
     background: url(yinyang-sm.png) center no-repeat green;}
     /* that's shorthand 'background', explained in the next section */
.cover {background-size: cover;}
.contain {background-size: contain;}
.clip-content {background-clip: content-box;}
.clip-padding {background-clip: padding-box;}
.origin-content {background-origin: content-box;}
.origin-padding {background-origin: padding-box;}
```

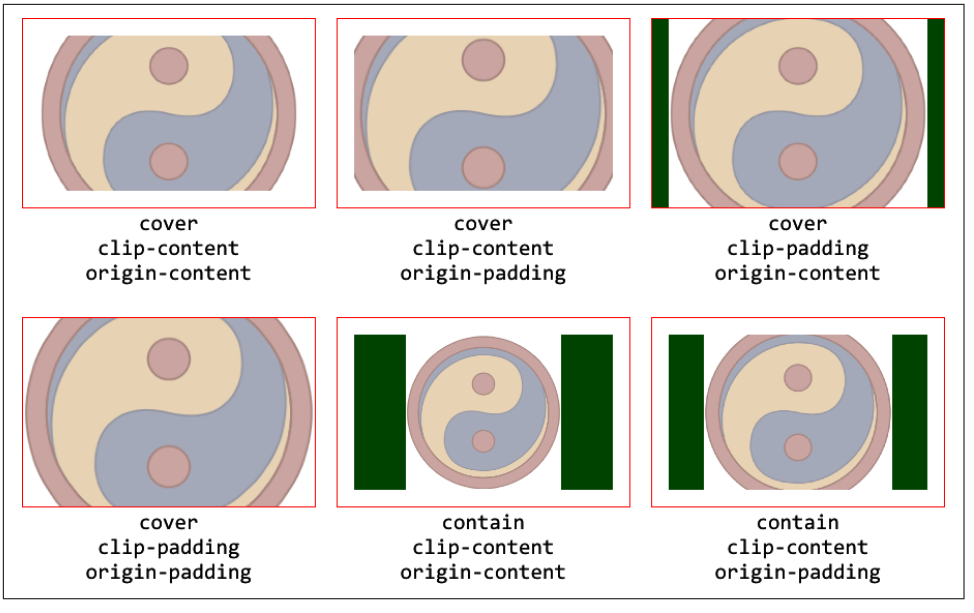


Figure 8-41. Covering and containing with `background-clip` and `background-origin`

Yes, you can see background color around the edges of some of these, and others get clipped. That's the difference between the painting area and the positioning area. You'd think that `cover` and `contain` would be sized with respect to the painting area, but they aren't, as depicted in the last examples in Figure 8-41. Keep that firmly in mind whenever you use these values.

If you have more than one background image, with different values for position, repeat, or size, include a comma-separated list of values. Each value in a list will be associated with the image in that position in the list. If there are more values than images, the extra values are ignored. If there are fewer, the list is repeated. You can set only one background color, though.



In this section, we used raster images (GIFs, to be precise) even though they tend to look horrible when scaled up and represent a waste of network resources when scaled down. (We did this so that it would be extra obvious when lots of up-scaling was happening.) This is an inherent risk in scaling background raster images. On the other hand, you can just as easily use SVGs as background images, and they scale up or down with no loss of quality or waste of bandwidth. If you're going to be scaling a background image and it doesn't have to be a photograph, strongly consider using SVG or CSS gradients.

Bringing It All Together

As is often the case with thematic areas of CSS, the background properties can all be brought together in a single shorthand property: `background`. Whether you might want to do that is another question entirely.

background

Values	[*<bg-layer>*,]* *<final-bg-layer>*
Initial value	Refer to individual properties
Applies to	All elements
Percentages	Refer to individual properties
Computed value	Refer to individual properties
Inherited	No
Animatable	See individual properties

<bg-layer>
    <bg-image> || *<position>* [/ *<bg-size>*]? || *<repeat-style>* || *<attachment>* || *<box>* || *<box>*

<final-bg-layer>
    <bg-image> || *<position>* [/ *<bg-size>*]? || *<repeat-style>* || *<attachment>* || *<box>* || *<box>* || *<background-color>*

This syntax can get a little confusing. Let's start simple and work our way up.

First off, the following statements are all equivalent to one another and will have the effect shown in Figure 8-42:

```
body {background-color: white;
      background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: repeat-y;
      background-attachment: fixed;
      background-origin: padding-box;
      background-clip: border-box;
      background-size: 50% 50%;}
body {background:
    white url(yinyang.png) repeat-y top left/50% 50% fixed
     padding-box border-box;}
body {background:
    fixed url(yinyang.png) padding-box border-box white repeat-y
     top left/50% 50%;}
body {background:
```

```
    url(yinyang.png) top left/50% 50% padding-box white repeat-y
    fixed border-box;}
```

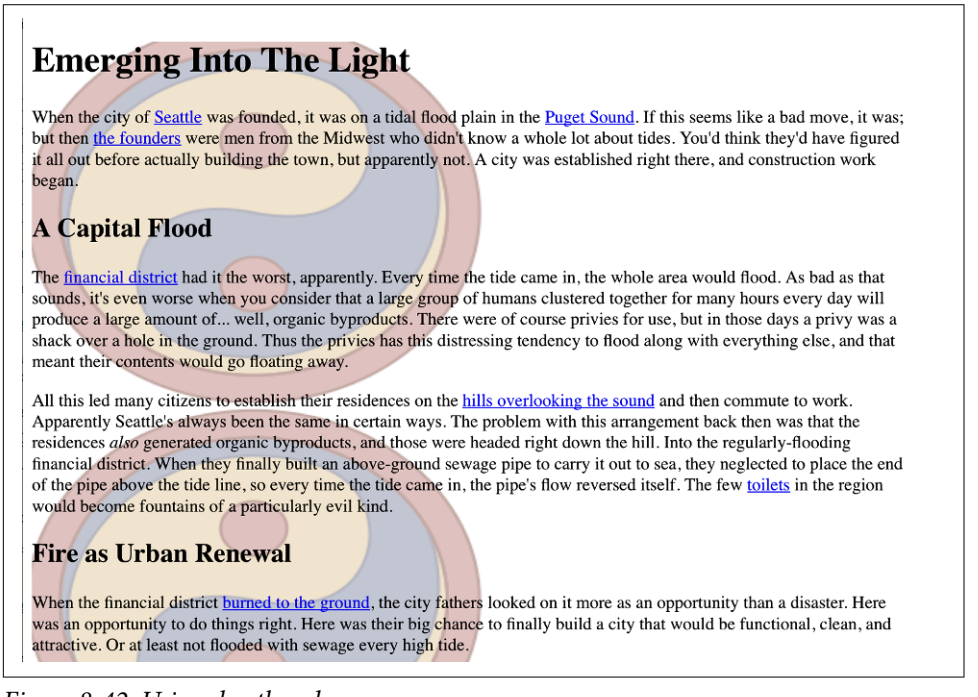


Figure 8-42. Using shorthand

You can mostly mix up the order of the values however you like, with three restrictions. First, any `background-size` value *must* come immediately after the `background-position` value, and must be separated from it by a forward slash (`/`). Second, within those values, the usual restrictions apply: the horizontal value comes first, and the vertical value follows, assuming that you're supplying axis-derived values (as opposed to, say, `cover`).

Third, if you supply values for both `background-origin` and `background-clip`, the first of the two you list will be assigned to `background-origin`, and the second to `background-clip`. Therefore, the following two rules are functionally identical:

```
body {background:
    url(yinyang.png) top left/50% 50% padding-box border-box white
     repeat-y fixed;}
body {background:
    url(yinyang.png) top left/50% 50% padding-box white repeat-y
     fixed border-box;}
```

Related to that, if you supply only one such value, it sets both `background-origin` and `background-clip`. Thus, the following shorthand sets both the background positioning area and the background painting area to the padding box:

```
body {background:
    url(yinyang.png) padding-box top left/50% 50% border-box;}
```

As is the case for shorthand properties, if you leave out any values, the defaults for the relevant properties are filled in automatically. Thus, the following two are equivalent:

```
body {background: white url(yinyang.png;}
body {background: white url(yinyang.png) transparent 0% 0%/auto repeat
      scroll padding-box border-box;}
```

Even better, `background` has no required values—as long as you have at least one value present, you can omit the rest. It's possible to set just the background color using the shorthand property, which is a very common practice:

```
body {background: white;}
```

On that note, remember that `background` is a shorthand property, and, as such, its default values can obliterate previously assigned values for a given element. For example:

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background: silver;}
```

Given these rules, `<h1>` elements will be styled according to the first rule. And `<h2>` elements will be styled according to the second, which means they'll just have a flat silver background. No image will be applied to `<h2>` backgrounds, let alone centered and repeated horizontally. It is more likely that the author meant to do this:

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background-color: silver;}
```

This lets the background color be changed without wiping out all the other values.

One more restriction will lead us very neatly into the next section: you can supply a background color to only the final background layer. No other background layer can have a solid color declared. What the heck does that mean? So glad you asked.

Working with Multiple Backgrounds

Throughout most of this chapter, we've only briefly mentioned that almost all the background properties accept a comma-separated list of values. For example, if you wanted to have three different background images, you could do this:

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-repeat: no-repeat;}
```

Seriously. It will look like Figure 8-43.

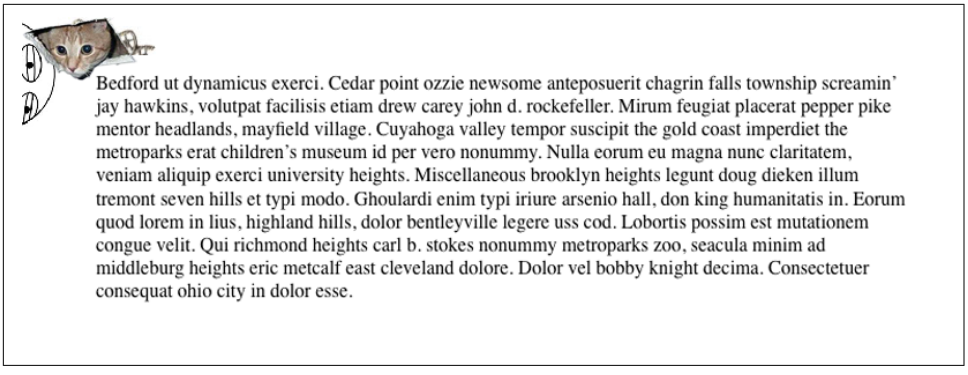


Figure 8-43. Multiple background images

This creates three background layers, one for each image, with the last being the final, bottom background layer.

The three images are piled into the top-left corner of the element and don't repeat. The lack of repetition occurs because we declared `background-repeat: no-repeat`. We declared it only once, and there are three background images.

When a mismatch occurs between the number of values in a background-related property and the `background-image` property, the missing values are derived by repeating the sequence in the property with a value undercount. Thus, in the previous example, it was as though we had said this:

```
background-repeat: no-repeat, no-repeat, no-repeat;
```

Now, suppose we want to put the first image at the top right, put the second at the center of the left side, and put the last layer at the center of the bottom. We can layer `background-position` as follows, resulting in Figure 8-44:

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat;}
```

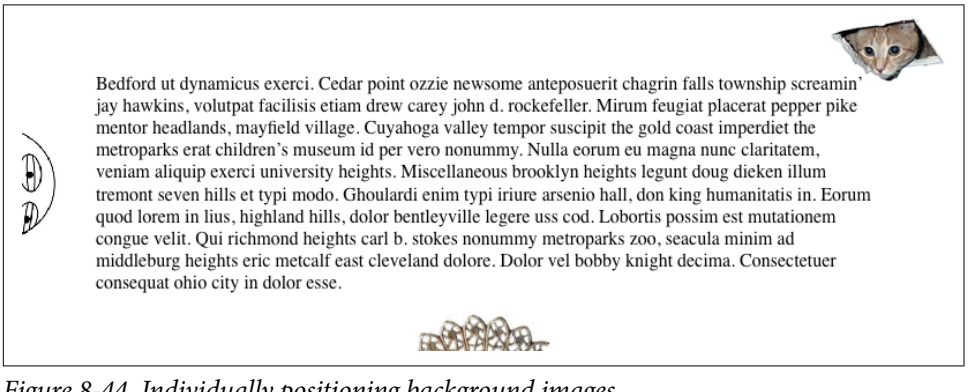


Figure 8-44. Individually positioning background images

Similarly, say we want to keep the first two layers from repeating, but horizontally repeat the third:

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat, no-repeat, repeat-x;}
```

Nearly every background property can be comma-listed this way. You can have different origins, clipping boxes, sizes, and just about everything else for each background layer you create. Technically, there is no limit to the number of layers you can have, though at a certain point it's just going to get silly.

Even the shorthand `background` can be comma-separated. The following example is exactly equivalent to the previous one, and the result is shown in Figure 8-45:

```
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x;}
```

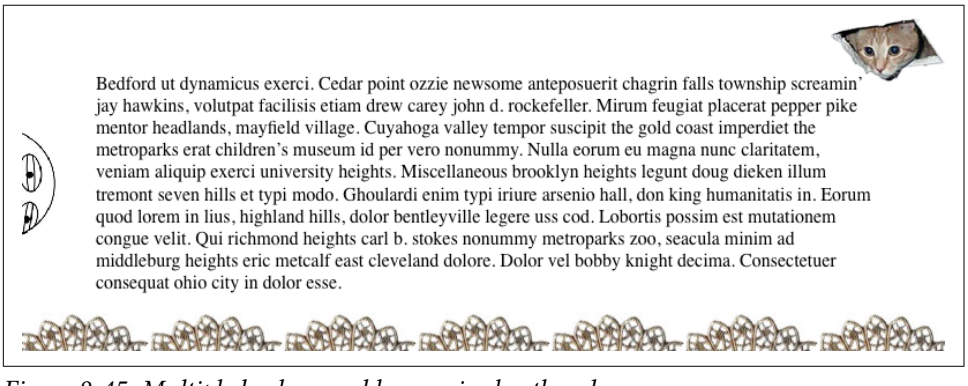


Figure 8-45. Multiple background layers via shorthand

The only real restriction on multiple backgrounds is that `background-color` does *not* repeat in this manner, and if you provide a comma-separated list for the `background` shorthand, the color can appear on only the last background layer. If you add a color to any other layer, the entire `background` declaration is made invalid. Thus, if we want to have a green background fill for the previous example, we'd do it in one of the following two ways:

```
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x green;}
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x;
    background-color: green;}
```

The reason for this restriction is pretty straightforward. Imagine if you were able to add a full background color to the first background layer. It would fill in the whole background and obscure all the background layers behind it! So if you do supply a color, it can be on only the last layer, which is "bottommost."

This ordering is important to internalize as soon as possible, because it runs counter to the instincts you've likely built up in the course of using CSS. After all, you know what will happen here—the `<h1>` background will be green:

```
h1 {background-color: red;}
h1 {background-color: green;}
```

Contrast that with this multiple-background rule, which will make the `<h1>` background red:

```
h1 {background:
    url(box-red.gif),
    url(box-green.gif),
    green;}
```

Yes, red. The red GIF is tiled to cover the entire background area, as is the green GIF, but the red GIF is "on top of" the green GIF. It's closer to you. And the effect is exactly backward from the "last one wins" rules built into the cascade.

You can visualize it like this: when there are multiple backgrounds, they're listed like the layers in a drawing program such as Adobe Photoshop or Illustrator. In the layer palette of a drawing program, layers at the top of the palette are drawn over the layers at the bottom. The same process plays out here: the layers listed at the top of the list are drawn over the layers at the bottom of the list.

The odds are pretty good that you will, at some point, set up a bunch of background layers in the wrong order, because your cascade-order reflexes will kick in. (This error still occasionally trips up the authors even to this day, so don't get down on yourself if it gets you too.)

Another fairly common mistake when you're getting started with multiple backgrounds is to use the `background` shorthand and forget to explicitly turn off background tiling for your background layers by letting the `background-repeat` value default to `repeat`, thus obscuring all but the top layer. See Figure 8-46, for example, which is the result of the following code:

```
section {background-image: url(bg02.gif), url(bg03.jpg);}
```

We can see only the top layer because it's tiling infinitely, thanks to the default value of `background-repeat`. That's why the example at the beginning of this section used `background-repeat: no-repeat`.

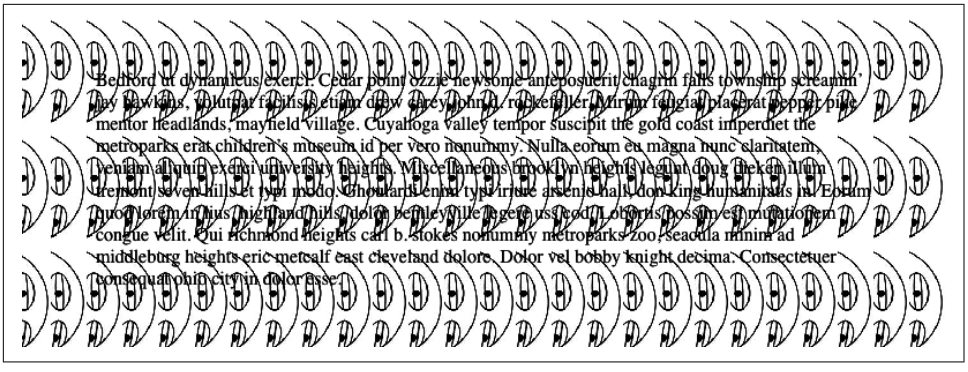


Figure 8-46. Obscuring layers with repeated images

Using the Background Shorthand

One way to avoid these sorts of situations is to use the `background` shorthand, like so:

```
body {background:
        url(bg01.png) top left border-box no-repeat,
        url(bg02.gif) bottom center padding-box no-repeat,
        url(bg04.svg) bottom center padding-box no-repeat gray;}
```

That way, when you add or subtract background layers, the values you meant to apply specifically to them will come in or go out with them.

This can mean some annoying repetition if all the backgrounds should have the same value of a given property, like `background-origin`. If that's the situation, you can blend the two approaches, like so:

```
body {background:
        url(bg01.png) top left no-repeat,
        url(bg02.gif) bottom center no-repeat,
        url(bg04.svg) bottom center no-repeat gray;
     background-origin: padding-box;}
```

This works just as long as you don't need to make any exceptions. The minute you decide to change the origin of one of those background layers, you'll need to explicitly list them, whether you do it in `background` or with the separate `background-origin` declaration.

Remember that the number of layers is determined by the number of background images, and so, by definition, `background-image` values are *not* repeated to equal the number of comma-separated values given for other properties. You might want to put the same image in all four corners of an element and think you could do it like this:

```
background-image: url(i/box-red.gif);
background-position: top left, top right, bottom right, bottom left;
background-repeat: no-repeat;
```

The result, however, would be to place a single red box in the top-left corner of the element. To get images in all four corners, as shown in Figure 8-47, you'll have to list the same image four times:

```
background-image: url(i/box-red.gif), url(i/box-red.gif),
                  url(i/box-red.gif), url(i/box-red.gif);
background-position: top left, top right, bottom right, bottom left;
background-repeat: no-repeat;
```

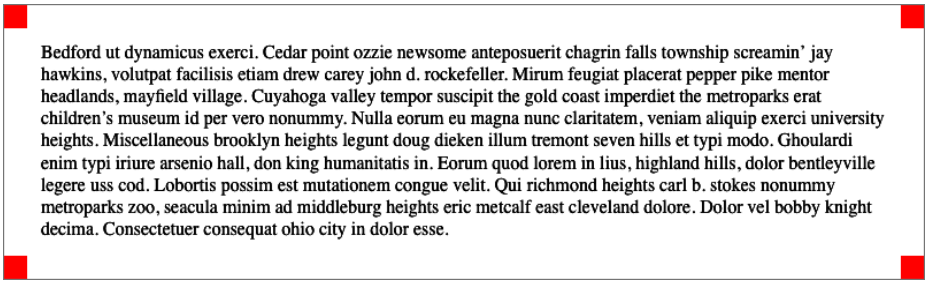


Figure 8-47. Placing the same image in all four corners

Creating Box Shadows

You have learned about border images, outlines, and background images. CSS has another property that can decorate both the inside and outside of an element without impacting the box model: `box-shadow`.

box-shadow

Values	none | [inset? && <*length*>{2,4} && <*color*>?]#
Initial value	none
Applies to	All elements
Computed value	<*length*> values as absolute length values; <*color*> values as computed internally; otherwise, as specified
Inherited	No
Animatable	Yes

It might seem a little out of place to talk about shadows in a chapter mostly concerned with backgrounds, but you'll understand our reasoning in a moment.

Let's consider a simple box drop shadow: one that's 10 pixels down and 10 pixels to the right of an element box, and a half-opaque black. Behind it we'll put a repeating

background on the <body> element. All of this is illustrated in Figure 8-48 and created with the following code:

```
#box {background: silver; border: medium solid;
      box-shadow: 10px 10px rgb(0 0 0 / 0.5);}
```

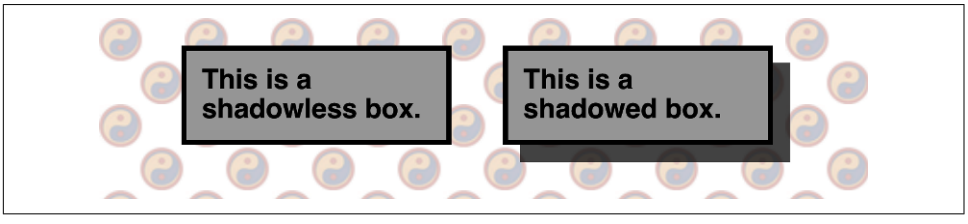


Figure 8-48. A simple box shadow

We can see that the <body>'s background is visible through the half-opaque (or half-transparent, if you prefer) drop shadow. Because no blur or spread distances are defined, the drop shadow exactly mimics the outer shape of the element box itself—at least it appears to do so.

The reason it only appears to mimic the shape of the box is that the shadow is visible only outside the outer border edge of the element. We couldn't really see that in the previous figure, because the element had an opaque background. You might have just assumed that the shadow extended all the way under the element, but it doesn't. Consider the following, illustrated in Figure 8-49:

```
#box {background: transparent; border: thin dashed;
      box-shadow: 10px 10px rgb(0 0 0 / 0.5);}
```

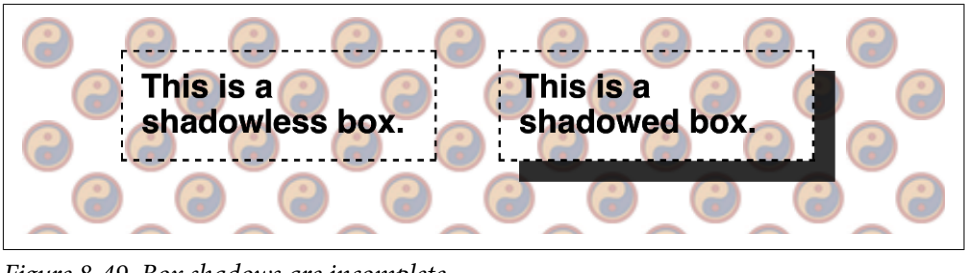


Figure 8-49. Box shadows are incomplete

So it looks as though the element's content (and padding and border) area "knocks out" part of the shadow. In truth, it's just that the shadow was never drawn there, because of the way box shadows are defined in the specification. This does mean, as Figure 8-49 demonstrates, that any background "behind" the box with a drop shadow can be visible through the element itself. This (perhaps bizarre-seeming) interaction with the backgrounds and borders is why `box-shadow` is covered here, instead of at an earlier point in the text.

So far, we've seen box shadows defined with two length values. The first defines a horizontal offset, and the second a vertical offset. Positive numbers move the shadow down and to the right, and negative numbers move the shadow up and to the left.

If a third length is given, it defines a blur distance, which determines how much space is given to blurring. A fourth length defines a spread distance, which changes the size of the shadow. Positive length values make the shadow expand before blurring happens; negative values cause the shadow to shrink. The following has the results shown in Figure 8-50:

```
.box:nth-of-type(1) {box-shadow: 1em 1em 2px rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: 2em 0.5em 0.25em rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px rgba(0,128,0,0.5);}
.box:nth-of-type(4) {box-shadow: -10px 25px 5px -5px rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(6) {box-shadow: 0.67em 1.33em 0.2em -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5);}
```

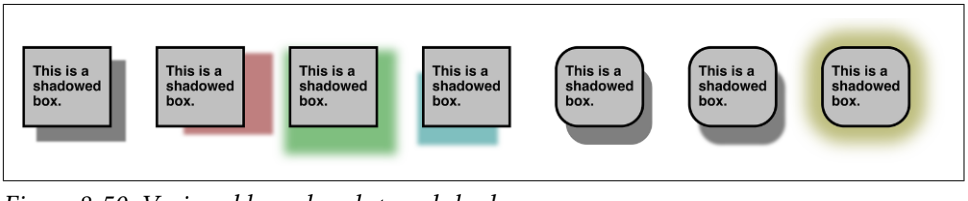


Figure 8-50. Various blurred and spread shadows

You may have noticed that some of these boxes have rounded corners (via `border-radius`), and that their shadows are curved to match. This is the defined behavior, fortunately.

We have yet to cover one aspect of `box-shadow`, which is the `inset` keyword. If `inset` is added to the value of `box-shadow`, the shadow is rendered inside the box, as if the box were a punched-out hole in the canvas rather than floating above it (visually speaking). Let's take the previous set of examples and redo them with inset shadows. This will have the result shown in Figure 8-51:

```
.box:nth-of-type(1) {box-shadow: inset 1em 1em 2px rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: inset 2em 0.5em 0.25em rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px rgba(0,128,0,0.5) inset;}
.box:nth-of-type(4) {box-shadow: inset -10px 25px 5px -5px  rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em rgba(0,0,0,0.5) inset;}
.box:nth-of-type(6) {box-shadow:
   inset 0.67em 1.33em 0.2em -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5) inset;}
```

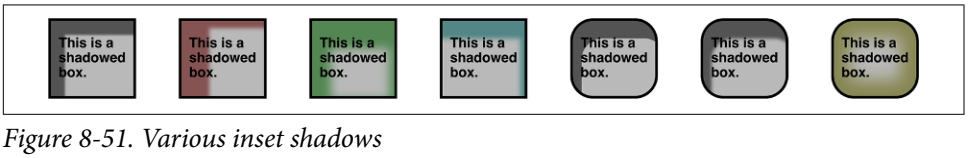


Figure 8-51. Various inset shadows

Note that the `inset` keyword can appear before the rest of the value, or after, but *not* in the middle of the lengths and colors. A value like `0 0 0.1em inset gray` would be ignored as invalid because of the placement of the `inset` keyword.

The last point to note is that you can apply to an element a list of as many comma-separated box shadows as you like, just as with text shadows. Some could be inset, and some outset. The following rules are just two of the infinite possibilities:

```
#shadowbox {
        padding: 20px;
        box-shadow: inset 0 -3em 3em rgb(0 0 0 /0.1),
                0 0 0 2px rgb(255 255 255),
                0.3em 0.3em 1em rgb(0 0 0 / 0.3);}
#wacky {box-shadow: inset 10px 2vh 0.77em 1ch red,
     1cm 1in 0 -1px cyan inset,
     2ch 3ch 0.5ch hsl(117, 100%, 50% / 0.343),
     -2ch -3ch 0.5ch hsl(297, 100%, 50% / 0.23);}
```

Multiple shadows are drawn back to front, just as background layers are, so the first shadow in the comma-separated list will be "on top" of all the others. Consider the following:

```
box-shadow: 0 0 0 5px red,
            0 0 0 10px blue,
            0 0 0 15px green;
```

The green is drawn first, then the blue on top of the green, and the red drawn last, on top of the blue. While box shadows can be infinitely wide, they do not contribute to the box model and take up no space. Because of this, make sure to include enough space, especially if you're doing large offsets or blur distances.



The `filter` property is another way to create element drop shadows, although it is much closer in behavior to `text-shadow` than `box-shadow`, albeit applying to the entire element box and text. See Chapter 20 for details.

Summary

Adding backgrounds to elements, whether with colors or images, gives you a great deal of power over the total visual presentation. The advantage of CSS over older methods is that colors and backgrounds can be applied to any element in a document, and manipulated in surprisingly complex ways.

CHAPTER 9
Gradients

Three image types defined by CSS are described entirely with CSS: linear gradients, radial gradients, and conic gradients. Each type has two subtypes: repeating and nonrepeating. Gradients are most often used in backgrounds, though they can be used in any context where an image is permitted—as in `list-style-image` and `border-image`, for example.

A *gradient* is a visual transition from one color to another. A gradient from yellow to red will start yellow, run through successively less yellow, redder shades of orange, and eventually arrive at a full red. How gradual or abrupt the transition is depends on the amount of space the gradient has and the way you define color stops and progression color hints. If you run from white to black over 100 pixels, each pixel along the gradient's default progression will be another 1% darker gray, as shown in Figure 9-1.

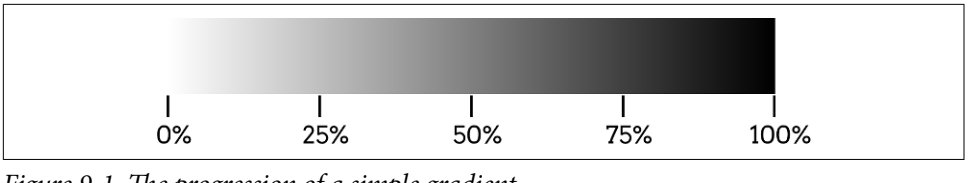


Figure 9-1. The progression of a simple gradient

As we go through the process of exploring gradients, always keep this in mind: *gradients are images*. It doesn't matter that you describe them by typing CSS—they are every bit as much images as SVGs, PNGs, JPEGs, and so on—but gradients have excellent rendering performance, don't require an extra HTTP request to load, and are infinitely scalable.

What's interesting about gradients is that they have no intrinsic dimensions. If the `background-size` property's value `auto` is used, it is treated as if it were `100%`. Thus, if you don't define a `background-size` for a background gradient, it will be set to the default value of `auto`, which is the same as declaring `100% 100%`. So, by default, background gradients fill the entire background positioning area. Just note that if you offset the gradient's background position with a length (not percentage) value, by default it will tile.

Linear Gradients

Linear gradients are gradient fills that proceed along a linear vector, referred to as the *gradient line*. Here are a few relatively simple gradients, with the results shown in Figure 9-2:

```
#ex01 {background-image: linear-gradient(purple, gold);}
#ex02 {background-image: linear-gradient(90deg, purple, gold);}
#ex03 {background-image: linear-gradient(to left, purple, gold);}
#ex04 {background-image: linear-gradient(-135deg, purple, gold, navy);}
#ex05 {background-image: linear-gradient(to bottom left, purple, gold, navy);}
```

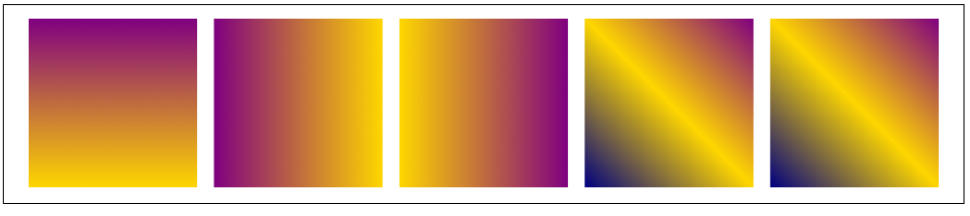


Figure 9-2. Simple linear gradients

The first of these is the most basic that a gradient can be: two colors. This causes a gradient from the first color at the top of the background painting area to the second color at the bottom of the background painting area.

By default, a gradient runs from top to bottom because the default direction for gradients is `to bottom`, which is the same as `180deg` and its various equivalents (for example, `0.5turn`). If you'd like to go in a different direction, you can start the gradient value with a direction. That's what we did for all the other gradients shown in Figure 9-2.

A gradient must have, at minimum, two color stops. They can be the same color, though. If you want to have a solid color behind only part of your content, a gradient with the same color declared twice, along with a background size and a no-repeat, enables that, as shown in Figure 9-3:

```
blockquote {
        padding: 0.5em 1em 2em;
        background-image:
                linear-gradient(palegoldenrod, palegoldenrod),
                linear-gradient(salmon, salmon);
        background-size: 75% 90%;
        background-position: 0px 0px, 15px 30px;
        background-repeat: no-repeat;
     columns: 3;
}
```

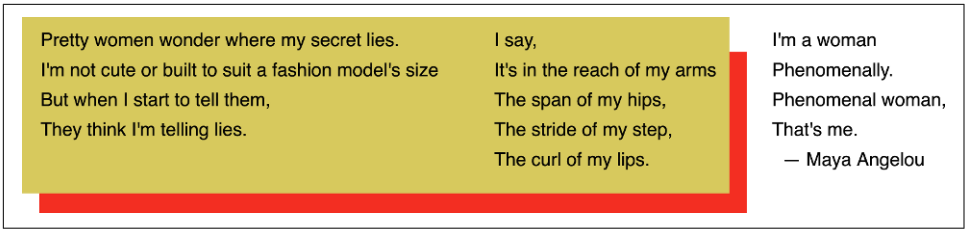


Figure 9-3. Solid-color gradients

The basic syntax of a linear gradient is shown here:

```
linear-gradient(
    [[ <angle> | to <side-or-quadrant> ],]?
     [ <color-stop-list> [, <color-hint>]? ]# ,
    <color-stop-list>
)
```

We'll explore both color stop lists and color hints soon. For now, the basic pattern to keep in mind is an optional direction at the start, a list of color stops and/or color hints, and a color stop at the end. As shown earlier, a `linear-gradient()` value must have a minimum of two color stops.

While you use the `to` keyword only if you're describing a side or quadrant with keywords like `top` and `right`, the direction you give *always* describes the direction in which the gradient line points. In other words, `linear-gradient(0deg,red,green)` will have red at the bottom and green at the top because the gradient line points toward 0 degrees (the top of the element) and thus ends with green. While it is indeed "going toward 0 degrees," remember to omit the `to` if you're using an angle value, because something like `to 45deg` is invalid and will be ignored. Degrees increase clockwise from 0 at the top.

The important point is that while `0deg` is the same as `to top`, 45% is *not* the same as `to top right`. This is explained in "Understanding Gradient Lines: The Gory Details" on page 371. Equally important to remember is that when using angles, whether it's degrees, radians, or turns, the unit type is *required*. A `0` value is not valid and will prevent any gradient from being created, while `0deg` is valid.

Setting Gradient Colors

You can use any color value you like in gradients, including alpha-channel values such as `rgba()` and keywords like `transparent`. Thus it's entirely possible to fade out pieces of your gradient by blending to (or from) a color with zero opacity. Consider the following rules, which are depicted in Figure 9-4:

```
#ex01 {background-image:
    linear-gradient( to right, rgb(200,200,200), rgb(255,255,255) );}
#ex02 {background-image:
    linear-gradient( to right, rgba(200,200,200,1), rgba(200,200,200,0) );}
```

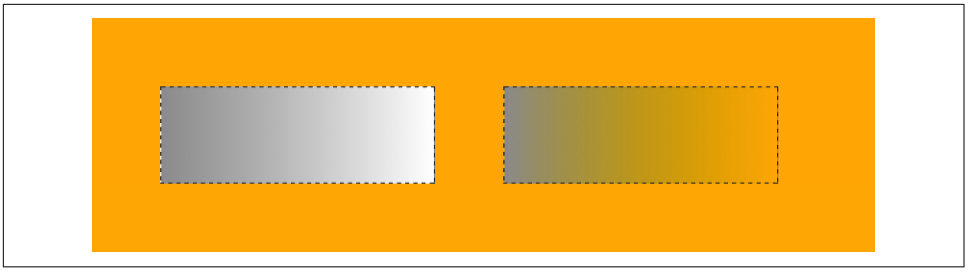


Figure 9-4. Fading to white versus fading to transparent

The first example fades from light gray to white, whereas the second example fades the same light gray from opaque to transparent, thus allowing the parent element's yellow background to show through.

You're not restricted to two colors, either. While that is the minimum number of colors allowed, you're free to add as many colors as you can stand. Consider the following gradient:

```
#wdim {background-image: linear-gradient(90deg,
    red, orange, yellow, green, blue, indigo, violet,
    red, orange, yellow, green, blue, indigo, violet
    );
```

The gradient line points toward 90 degrees, which is the right side. There are 14 color stops in all, one for each of the comma-separated color names, and they are, by default, distributed evenly along the gradient line, with the first at the beginning of the line and the last at the end. Between the color stops, by default the colors are blended as smoothly as possible from one color to the other. This is shown in Figure 9-5, with extra labels indicating how far along the gradient line the color stops are placed.

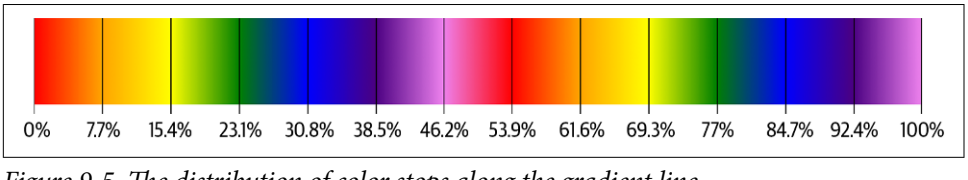


Figure 9-5. The distribution of color stops along the gradient line

So, without any indication of where the color stops should be positioned, they're evenly distributed. Fortunately, we can give each color up to two positions, and can even use color hints for more control over gradient progression, hopefully improving the visual effect.

Positioning Color Stops

The full syntax of a <*color-stop*> is as follows:

```
[<color>] [ <length> | <percentage> ]{1,2}?
```

After every color value, you can (but don't have to) supply a position value or two. This gives you the ability to distort the default evenly distributed progression of color stops into something else.

We'll start with lengths, since they're pretty simple. Let's take a rainbow progression (only a single rainbow this time) and have each color of the rainbow occur every 25 pixels, as shown in Figure 9-6:

```
#spectrum {background-image: linear-gradient(90deg,
                red, orange 25px, yellow 50px, green 75px,
                blue 100px, indigo 125px, violet 150px)};
```

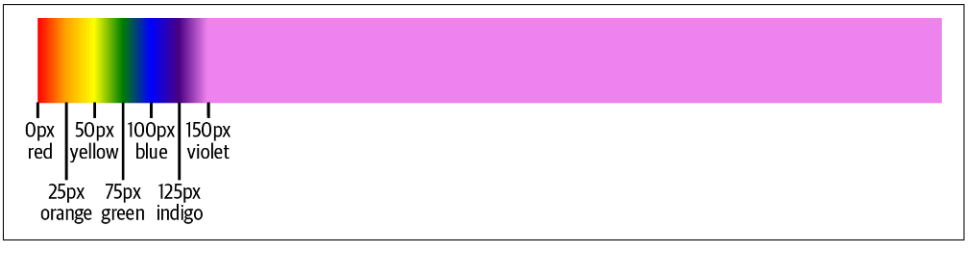


Figure 9-6. Placing color stops every 25 pixels

This worked out just fine, but notice what happens after 150 pixels—the violet just continues on to the end of the gradient line. That's what happens if you set up the color stops so they don't make it to the end of a basic gradient line: the last color is just carried onward.

Conversely, if your color stops go beyond the end of a basic gradient line, the gradient will appear to stop at whatever point it manages to reach when it gets to the end of the visible part of the gradient line. This is illustrated in Figure 9-7, created with the following code:

```
#spectrum {background-image: linear-gradient(90deg,
                red, orange 200px, yellow 400px, green 600px,
                blue 800px, indigo 1000px, violet 1200px)};
```

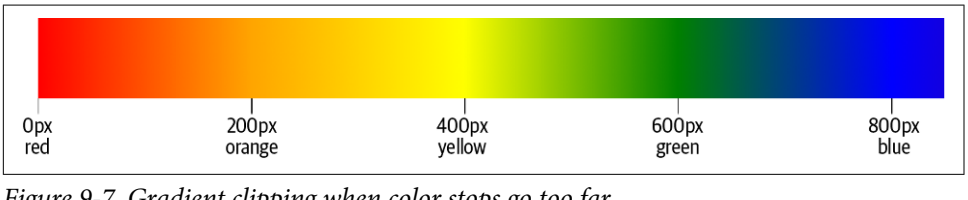


Figure 9-7. Gradient clipping when color stops go too far

Since the last color stop is at 1,200 pixels but the background size isn't nearly that wide, the visible part of the gradient stops right around the color `blue`.

Note that in the preceding two examples and figures, the first color (`red`) doesn't have a length value. If the first color has no position, it's assumed to be the beginning of the gradient line, as if `0%` (or other zero value, like `0px`) had been declared. Similarly, if you leave a position off the last color stop, it's assumed to be the end of the gradient line. (But note

that this is not true for repeating gradients, which we'll cover in "Repeating Linear Gradients" on page 376.)

You can use any length value you like, not just pixels—ems, viewport units, you name it. You can even mix different units into the same gradient, although this is not generally recommended for reasons we'll get to in a little bit. You can also have negative length values if you want; doing so will place a color stop before the beginning of the gradient line, all the color transitions will happen as expected, and clipping will occur in the same manner as it happens at the end of the line. The following code, for example, results in Figure 9-8:

```
#spectrum {background-image: linear-gradient(90deg,
                red -200px, orange 200px, yellow 400px, green 600px,
                blue 800px, indigo 1000px, violet 1200px)};
```

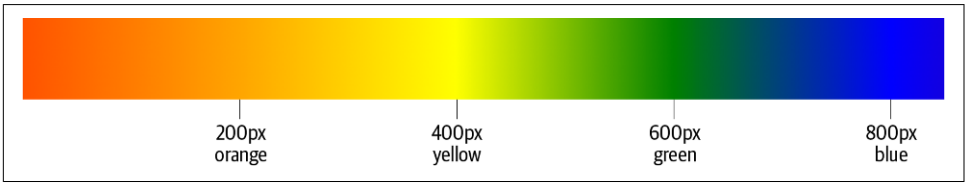


Figure 9-8. Gradient clipping when color stops have negative positions

As for percentages, they're calculated with respect to the total length of the gradient line. A color stop at `50%` will be at the midpoint of the gradient line. Let's return to our rainbow example, and instead of having a color stop every 25 pixels, we'll have one every 10% of the gradient line's length. This would look like the following, which has the result shown in Figure 9-9:

```
#spectrum {background-image: linear-gradient(90deg,
    red, orange 10%, yellow 20%, green 30%, blue 40%, indigo 50%, violet 60%)};
```

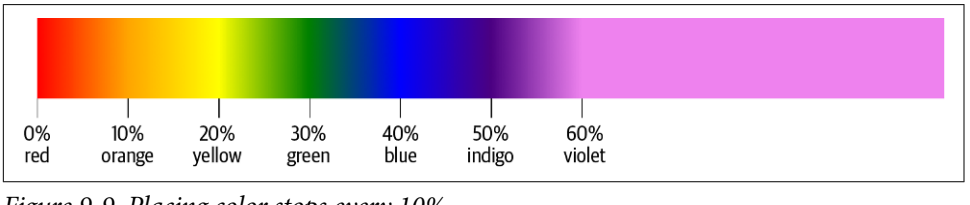


Figure 9-9. Placing color stops every 10%

As you saw previously, since the last color stop comes before the end of the gradient line, its color (`violet`) is carried through to the end of the gradient. These stops are more spread out than in the earlier 25-pixel example, but otherwise things happen in more or less the same way.

If some color stops have position values and others don't, the stops without positions are evenly distributed between the ones that do. The following are equivalent:

```
#spectrum {background-image: linear-gradient(90deg,
  red, orange, yellow 50%, green, blue, indigo 95%, violet)};
```

```
#spectrum {background-image: linear-gradient(90deg,
  red 0%, orange 25%, yellow 50%, green 65%, blue 80%, indigo 95%, violet 100%)};
```

Because `red` and `violet` don't have specified position values, they're taken to be `0%` and `100%`, respectively. This means that `orange`, `green`, and `blue` will be evenly distributed between the explicitly defined positions to either side of them.

For `orange`, that means the point midway between `red 0%` and `yellow 50%`, which is 25%. For `green` and `blue`, these need to be arranged between `yellow 50%` and `indigo 95%`. That's a 45% difference, which is divided in three, because there are three intervals between the four values. That means 65% and 80%.

You might wonder what happens if you put two color stops at exactly the same point, like this:

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow 50%, green 50%, blue , indigo, violet)};
```

All that happens is the two color stops are put on top of each other. Figure 9-10 shows the result.

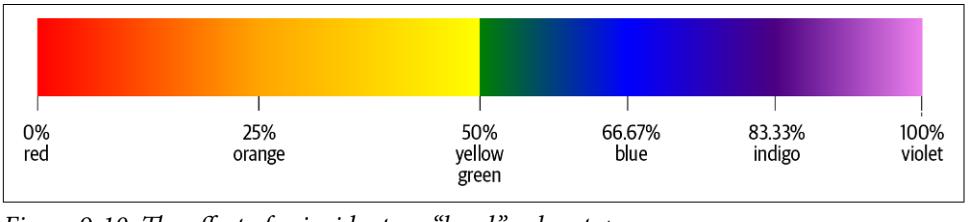


Figure 9-10. The effect of coincident, or "hard," color stops

The gradient blends as usual all along the gradient line, but at the 50% point, it instantly blends from yellow to green over zero length, creating what's often called a *hard* color stop. So the gradient blends from orange at the 25% point (halfway between 0% to 50%) to yellow at the 50% point, then blends from yellow to green over zero length, then blends from green at 50% over to blue at 66.67% (one-third of the way between 50% and 100%).

This hard-stop effect can be useful if you want to create a striped effect. The following code results in the stripes shown in Figure 9-11:

```
.stripes {background-image: linear-gradient(90deg,
    gray 0%, gray 25%,
    transparent 25%, transparent 50%,
    gray 50%, gray 75%,
    transparent 75%, transparent 100%);}
```



Figure 9-11. Hard-stop stripes

That said, there's an easier and more readable way to do that kind of thing, which is to give each color a starting and ending stop position. Here's how to do that, with exactly the same result as shown in Figure 9-11:

```
.stripes {background-image:
        linear-gradient(90deg,
                gray 0% 25%,
                transparent 25% 50%,
                gray 50% 75%,
                transparent 75% 100%);}
```

Note that the `0%` and `100%` could have been left out, and they'd be inferred by the browser. So you can leave them in for clarity or take them out for efficiency, as suits you.

It's also fine to mix two-stop stripes and one-stop color points in a single gradient. If you want to have the first and last quarter of the gradient be solid gray stripes and transition through transparency between them, it could look like this:

```
.stripes {background-image:
        linear-gradient(90deg,
                gray 0% 25%,
                transparent 50%,
                gray 75% 100%);}
```

OK, so that's what happens if you put color stops right on top of each other, but what happens if you put one *before* the other? Something like this, say:

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow, green 50%, blue 40%, indigo, violet)};
```

The offending color stop (blue, in this case) is set to the largest specified value of a preceding color stop. Here, it is set to `50%`, since the stop before it has that position. This creates a hard stop, and we get the same effect we saw earlier, when the green and blue color stops were placed on top of each other.

The key point here is that the color stop is set to the largest *specified* position of the stop that precedes it. Thus, the following two gradients are visually the same, as the `indigo` color stop in the first gets set to `50%`:

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow 50%, green, blue, indigo 33%, violet)};

#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow 50%, indigo 50%, violet)};
```

In this case, the largest specified position before the indigo stop is the `50%` specified at the yellow stop. Thus, the gradient fades from red to orange to yellow, then has a hard switch to indigo before fading from indigo to violet. The green and blue aren't skipped; rather, the gradients transition from yellow to green to blue to indigo over zero distance. See Figure 9-12 for the results.



Figure 9-12. Handling color stops that are out of place

This behavior is the reason mixing units within a single gradient is generally discouraged. If you mix `rem` units and percentages, for example, your color stop positioned with percentages might end up before an earlier color stop positioned with rems.

Setting Color Hints

Thus far, we've worked with color stops, but you may remember that the syntax for linear gradients permits color hints after each color stop:

```
linear-gradient(
    [[ <angle> | to <side-or-quadrant> ],]?
     [ <color-stop-list> [, <color-hint>]? ]# ,
    <color-stop-list>
)
```

A <*color-hint*> is a way of modifying the blend between the two color stops to either side. By default, the blend from one color stop to the next is linear, with the midpoint of the blend at the halfway mark between two color stops, or 50%. It doesn't have to be that simple. The following two gradients are the same and have the result shown in Figure 9-13:

```
linear-gradient(
    to right, rgb(0% 0% 0%) 25%, rgb(90% 90% 90%) 75%
)
linear-gradient(
    to right, rgb(0% 0% 0%) 25%, 50%, rgb(90% ,90% ,90%) 75%
)
```

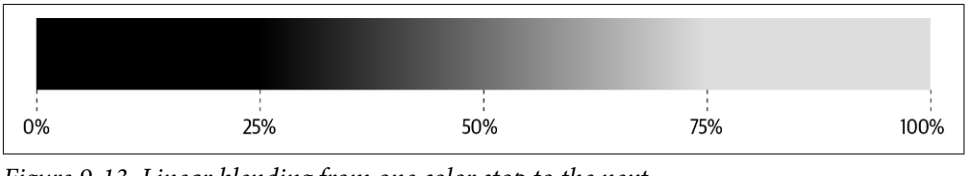


Figure 9-13. Linear blending from one color stop to the next

With color hints, we can change the midpoint of the progression. Instead of reaching `rgb(45% 45% 45%)` at the halfway point, it can be set for any point between the two stops. Thus, the following CSS leads to the result seen in Figure 9-14:

```
#ex01 {background:
     linear-gradient(to right, rgb(0% 0% 0%) 25%, rgb(90% 90% 90%) 75%);}
#ex02 {background:
     linear-gradient(to right, rgb(0% 0% 0%) 25%, 33%, rgb(90 90% 90%) 75%);}
#ex03 {background:
```

```
     linear-gradient(to right, rgb(0% 0% 0%) 25%, 67%, rgb(90% 90% 90%) 75%);}
#ex04 {background:
     linear-gradient(to right, rgb(0% 0% 0%) 25%, 25%, rgb(90% 90% 90%) 75%);}
#ex05 {background:
     linear-gradient(to right, rgb(0% 90% 90%) 25%, 75%, rgb(90% 90% 90%) 75%);}
```

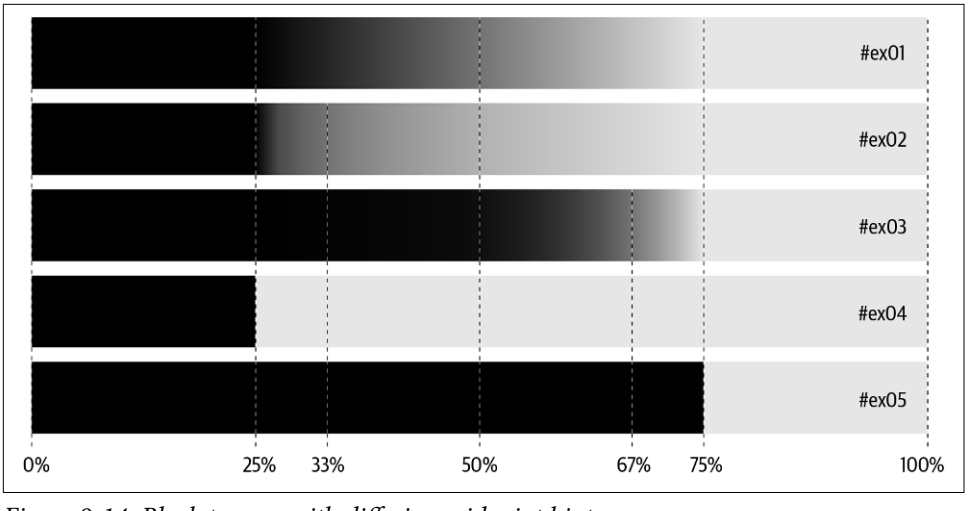


Figure 9-14. Black to gray with differing midpoint hints

In all five examples, the first color stop is at the 25% mark and the last at the 75% mark, but each has a different midpoint for the gradient. In the first case (`#ex01`), the default linear progression is used, with the middle color (45% black) occurring at the midpoint between the two color stops.

In the second case (`#ex02`), the middle color happens at the 33% point of the gradient line. So the first color stop is at the 25% point on the line, the middle color happens at 33%, and the second color stop happens at 75%.

In the third example (`#ex03`), the midpoint is at the 67% point of the gradient line; thus, the color fades from black at 25% to the middle color at 67%, and then from that middle color at 67% to light gray at 75%.

The fourth and fifth examples show what happens when you put a color hint's distance right on top of one of the color stops: you get a hard stop.

The interesting point about color hinting is that the progression from color stop to color hint to color stop is not just a set of two linear progressions. Instead, the progression has some "curving," in order to ease from one side of the color hint to the other. (The exact curve is logarithmic and based on the gradient-progression equation used by Photoshop.) This is easiest to see by comparing what would seem to be, but actually is not, two gradients that do the same thing. As you can see in Figure 9-15, the result is rather different for these two examples:

```
#ex01 {background:
    linear-gradient(to right,
        rgb(0% 0% 0%) 25%,
        rgb(45% 45% 45%) 67%,   /* this is a color stop */
        rgb(90% 90% 90%) 75%);}
#ex02 {background:
    linear-gradient(to right,
        rgb(0% 0% 0%) 25%,
        67%,                    /* this is a color hint */
        rgb(90% 90% 90%) 75%);}
```

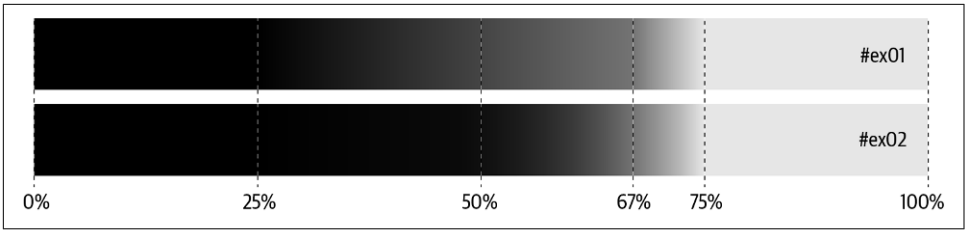


Figure 9-15. Comparing two linear gradients to one hinted transition

Notice how the gray progression differs in the two examples. The first shows a linear progression from black to `rgb(45%,45%,45%)`, and then another linear progression from there to `rgb(90%,90%,90%)`. The second progresses from black to light gray over the same distance, and the color-hint point is at the 67% mark, but the gradient is altered to attempt a smoother overall progression. The colors at 25%, 67%, and 75% are the same in both examples, but all the other shades along the way are different because of the (somewhat complicated) easing algorithm defined in the CSS specifications.



If you're familiar with animations, you might think to put easing functions (such as `ease-in`) into a color hint, in order to exert more control over the way the colors are blended. While the browser does this to some extent, as illustrated in Figure 9-15, this isn't something developers can control as of late 2022 (though that capability is under serious discussion by the CSS Working Group at this time).

Understanding Gradient Lines: The Gory Details

Now that you have a grasp of the basics of placing color stops, let's look closely at how gradient lines are constructed, and thus how they create the effects that they do. First, let's set up a simple gradient so we can then dissect how it works:

```
linear-gradient(
    55deg, #4097FF, #FFBE00, #4097FF
)
```

Now, how does this one-dimensional construct—a line at 55 degrees on the compass—create a two-dimensional gradient fill? First, the gradient line is placed and its start and ending points determined. This is diagrammed in Figure 9-16, with the final gradient shown next to it.

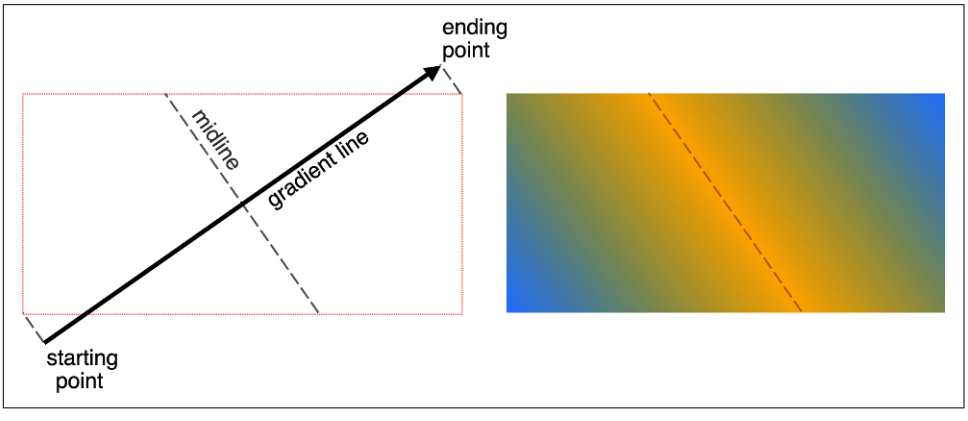


Figure 9-16. The placement and sizing of the gradient line

The first point to make very clear is that the box seen here is not an element—it's the linear-gradient image itself. (Remember, we're creating images here.) The size and shape of that image can depend on a lot of factors, whether it's the size of the element's background or the application of properties like `background-size`, which is a topic we'll cover in a bit. For now, we're just concentrating on the image itself.

So, in Figure 9-16, you can see that the gradient line goes straight through the center of the image. The gradient line *always* goes through the center of the gradient image, and in this case, the gradient image is centered in the background area. (Using `background-position` to shift placement of a gradient image can, in some cases, make it appear that the center of the gradient is not centered in the image, but it is.) This gradient is set to a 55-degree angle, so it's pointing at 55 degrees on the compass. What's interesting is the start and ending points of the gradient line, which are actually outside the image.

Let's talk about the starting point first. It's the point on the gradient line where a line perpendicular to the gradient line intersects with the corner of the image farthest away from the gradient line's direction (`55deg`). Conversely, the gradient line's ending point is the point on the gradient line where a perpendicular line intersects the corner of the image nearest to the gradient line's direction.

Bear in mind that the terms "starting point" and "ending point" are a little bit misleading—the gradient line doesn't actually stop at either point. The gradient line is, in fact, infinite. However, the starting point is where the first color stop will be placed by default, as it corresponds to position value `0%`. Similarly, the ending point corresponds to the position value `100%`.

Therefore, let's consider the gradient we defined before:

```
linear-gradient(
    55deg, #4097FF, #FFBE00, #4097FF
)
```

The color at the starting point will be `#4097FF`, the color at the midpoint (which is also the center of the gradient image) will be `#FFBE00`, and the color at the ending point will be `#4097FF`, with smooth blending in between. This is illustrated in Figure 9-17.

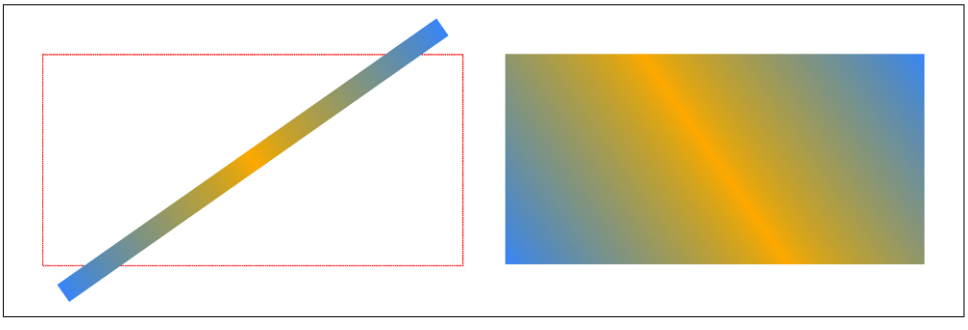


Figure 9-17. The calculation of color along the gradient line

All right, fine so far. But, you may wonder, how do the bottom-left and top-right corners of the image get set to the same blue that's calculated for the starting and ending points, if those points are outside the image? Because the color at each point along the gradient line is extended out perpendicularly from the gradient line. This is partially shown in Figure 9-18 by extending perpendicular lines at the starting and ending points, as well as every 5% of the gradient line between them. Note that each line perpendicular to the gradient line is a solid color.

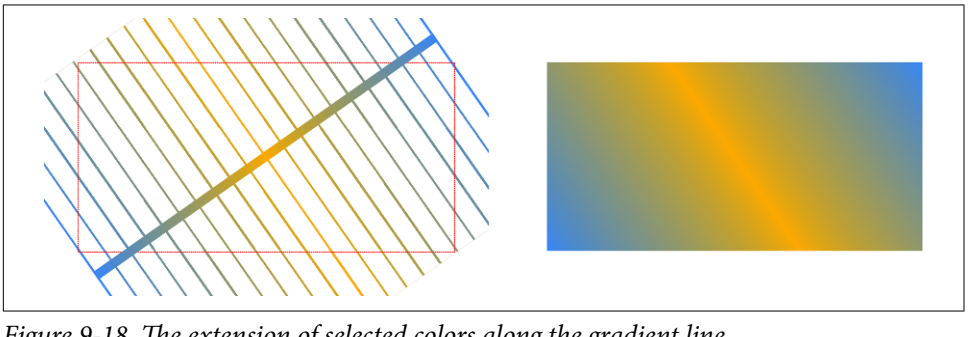


Figure 9-18. The extension of selected colors along the gradient line

Hopefully, that should be enough to let you mentally fill in the rest, so let's consider what happens to the gradient image in various other settings. We'll use the same gradient definition as before, but this time apply it to wide, square, and tall images. These are shown in Figure 9-19. Note that the starting-point and ending-point colors always make their way into the corners of the gradient image.

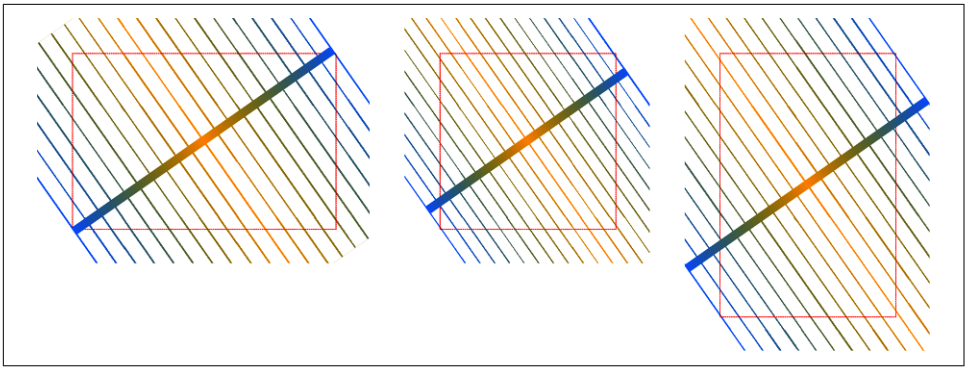


Figure 9-19. How gradients are constructed for various images

Note that we very carefully said "the starting-point and ending-point colors," and did *not* say "the starting and ending colors." That's because, as you saw earlier, color stops can be placed before the starting point and after the ending point, like so:

```
linear-gradient(
    55deg, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

The placement of these color stops, the starting and ending points, the way the colors are calculated along the gradient line, and the final gradient are all shown in Figure 9-20.

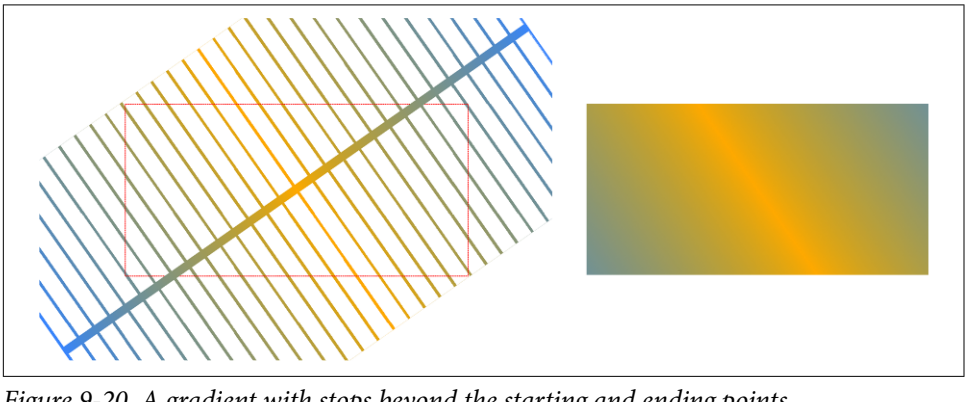


Figure 9-20. A gradient with stops beyond the starting and ending points

Once again, we see that the colors in the bottom-left and top-right corners match the starting-point and ending-point colors. It's just that in this case, since the first color stop comes before the starting point, the actual color at the starting point is a blend of the first and second color stops. Likewise for the ending point, which is a blend of the second and third color stops.

Now here's where things get a little bit wacky. Remember how you can use directional keywords, like `top` and `right`, to indicate the direction of the gradient line? Suppose you want the gradient line to go toward the top right, so you create a gradient image like this:

```
linear-gradient(
    to top right, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

This does *not* cause the gradient line to intersect with the top-right corner. If only that were so! Instead, what happens is a good deal stranger. First, let's diagram it in Figure 9-21 so that we have something to refer to.

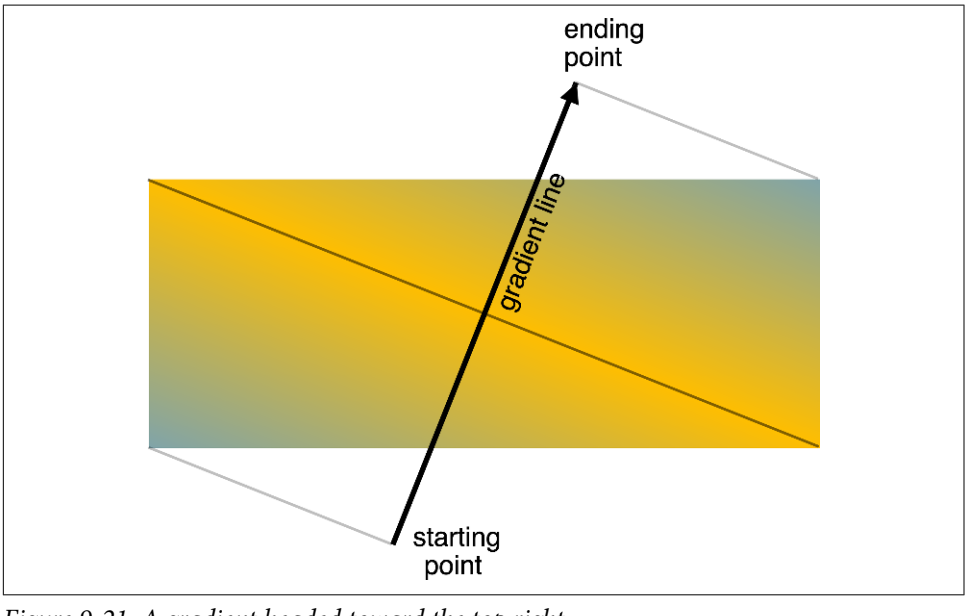


Figure 9-21. A gradient headed toward the top right

Your eyes do not deceive you: the gradient line is way off from the top-right corner. It *is* headed into the top-right quadrant of the image, though. That's what `to top right` really means: head into the top-right quadrant of the image, not into the top-right corner.

As Figure 9-21 shows, the way to find out exactly what that means is to do the following:

1. Draw a line from the midpoint of the image into the corners adjacent to the corner in the quadrant that's been declared. Thus, for the top-right quadrant, the adjacent corners are the top left and bottom right.
2. Find the center point of that line, which is the center point of the image, and draw the gradient line perpendicular to that line, through the center point, pointing into the declared quadrant.

3. Construct the gradient—that is, determine the starting and ending points, place or distribute the color stops along the gradient line, and then calculate the entire gradient image, as per usual.

This process has a few interesting side effects. First, the color at the midpoint will always stretch from one quadrant-adjacent corner to the other. Second, if the image's shape changes—that is, if its aspect ratio changes—the gradient line will also reset its direction, reorienting slightly to fit the new aspect ratio. So watch out for that if you have flexible elements. Third, a perfectly square gradient image will have a gradient line that intersects with a corner. Figure 9-22 depicts examples of these three side effects, using the following gradient definition in all three cases:

```
linear-gradient(
    to top right, purple, green 49.5%, black 50%, green 50.5%, gold
)
```

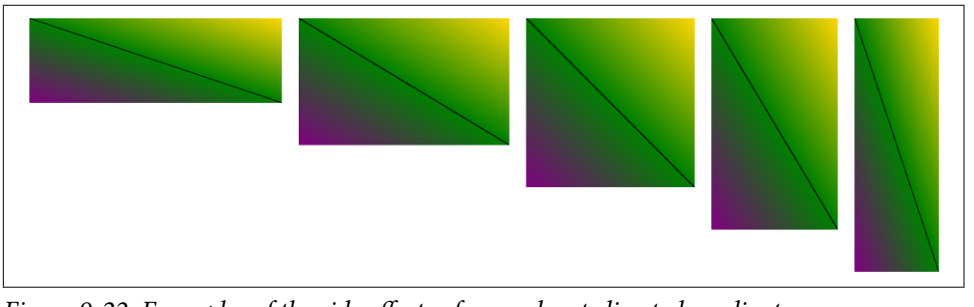


Figure 9-22. Examples of the side effects of a quadrant-directed gradient

Sadly, there is no way to say "point the gradient line into the corner of a nonsquare image," short of calculating the necessary degree heading yourself and declaring it explicitly, a process that will most likely require JavaScript unless you know the image will always be an exact size in all cases, forever. (Or use the `aspect-ratio` property; see Chapter 6 for details.)

While linear gradients follow a gradient line in the direction set forth by the angle, it is possible to create a mirrored gradient; for that, oddly enough, see "Radial Gradients" on page 379.

Repeating Linear Gradients

Regular gradients are autosized by default, matching the size of the background area to which they are applied. In other words, by default a gradient image takes up all the available background space and does not repeat.

Intentionally setting a background size and tiling images, especially with hard color stops, can create interesting effects. By declaring two linear-gradient background images using hard color stops, with perpendicular gradient lines, and different background colors, you

can create picnic tablecloth effects for any place setting by setting up some gradient images, tiling them, and then putting a color underneath, as illustrated in Figure 9-23:

```
div {
   background-image:
      linear-gradient(to top, transparent 1vw, rgba(0 0 0 / 0.2) 1vw),
      linear-gradient(to right, transparent 1vw, rgba(0 0 0 / 0.2) 1vw);
   background-size: 2vw 2vw;
   background-repeat: repeat;
}
div.fruit {background-color: papayawhip;}
div.grain {background-color: palegoldenrod;}
div.fishy {background-color: salmon;}
```

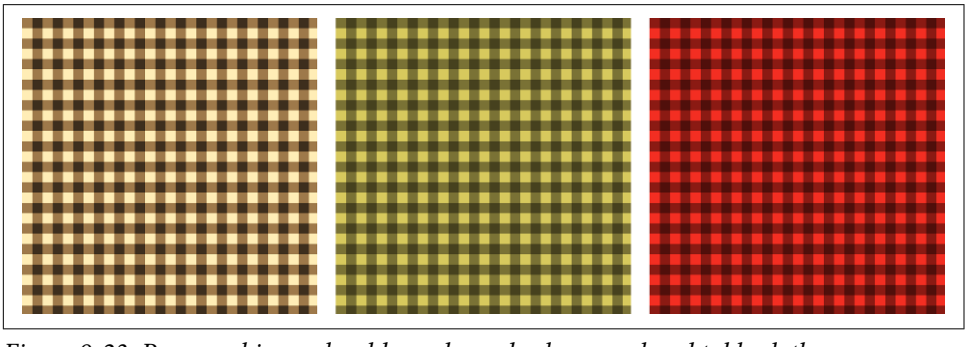


Figure 9-23. Papayawhip-, palegoldenrod-, and salmon- colored table cloths

Instead of defining a gradient size with `background-size` and tiling it with `background-repeat`, we can use repeating linear gradient syntax. By adding `repeating` in front of the linear gradients, they are made infinitely repeating within the size of the gradient. In other words, the declared color stops and color hints are repeated on a loop along the gradient line, over and over again, because the size of a gradient line when using `repeating-linear-gradient` is the size of the last color stop position less the first color stop position (in this case, `2vw`). Thus, we can remove the sizing and repetition properties, as in the following, and get the same result shown in Figure 9-23:

```
div {
   background-image:
     repeating-linear-gradient(to top,
        transparent 0 1vw, rgb(0 0 0 / 0.2) 1vw 2vw),
     repeating-linear-gradient(to right,
        transparent 0 1vw, rgb(0 0 0 / 0.2) 1vw 2vw);
}
div.fruit {background-color: papayawhip;}
div.grain {background-color: palegoldenrod;}
div.fishy {background-color: salmon;}
```

This is nice for simple patterns like these tablecloths, but it comes in really handy for more complex situations. For example, if you declare the following nonrepeating gradient, you end up with discontinuity where the image repeats, as shown in Figure 9-24:

```
h1.example {background:
    linear-gradient(-45deg, black 0, black 25px, yellow 25px, yellow 50px)
    top left/40px 40px repeat;}
```



Figure 9-24. Tiling gradient images with a repeating background image

You *could* try to nail down the exact sizes of the element and gradient image and then mess with the construction of the gradient image to try to make the sides line up, but it would be a lot easier to do the following, with the result shown in Figure 9-25:

```
h1.example {background: repeating-linear-gradient(-45deg,
        black 0 25px, yellow 25px 50px) top left;}
```



Figure 9-25. A repeating gradient image

Note that the last color stop ends with an explicit length (`50px`). This is important to do with repeating gradients, because the length value(s) of the last color stop defines the overall length of the pattern. If you leave off an ending stop, it will default to 100%, which is the end of the gradient line.

If you're using smoother transitions, you need to be careful that the color value at the last color stop matches the color value at the first color stop. Consider this:

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px)
```

This will produce a smooth gradient from purple to gold at 50 pixels, and then a hard switch back to purple and another 50-pixel purple-to-gold blend. By adding one more color stop with the same color as the first color stop, the gradient can be smoothed out to avoid hard-stop lines:

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px, purple 100px)
```

See Figure 9-26 for a comparison of the two approaches.

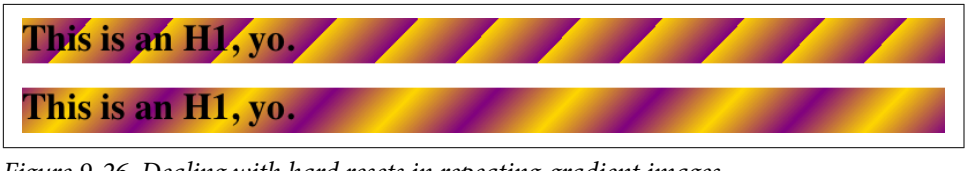


Figure 9-26. Dealing with hard resets in repeating-gradient images

You may have noticed that none of the repeating gradients so far have a defined size. That means the images are defaulting in size to the full background positioning area of the element to which they're applied, per the default behavior for images that have no intrinsic height and width.

If you resize a repeating-gradient image by using `background-size`, the gradient would repeat only within the bounds of the gradient image. If you then repeated that image using `background-repeat`, you could easily be back to having discontinuities in your background.

If you use percentages in your repeating linear gradients, they'll be placed the same as if the gradient wasn't of the repeating variety. Then again, this would mean that all of the gradients defined by those color stops would be seen and none of the repetitions would be visible, so percentages tend to be kind of pointless with repeating linear gradients.

Radial Gradients

Linear gradients are pretty awesome, but at times you really want a circular gradient. You can use such a gradient to create a spotlight effect, a circular shadow, a rounded glow, or any number of other effects, including a reflected gradient. The syntax used is similar to that for linear gradients, but some interesting differences exist:

```
radial-gradient(
    [ [ <shape> || <size> ] [ at <position>]? , | at <position>, ]?
      [ <color-stop-list> [, <color-hint>]? ] [, <color-stop-list> ]+
)
```

What this boils down to is you can optionally declare a shape and size, optionally declare where the center of the gradient is positioned, and then declare two or more color stops with optional color hints between the stops. Interesting options are available in the shape and size bits, so let's build up to those.

First, let's look at a simple radial gradient—the simplest possible, in fact—presented in a variety of differently shaped elements (Figure 9-27):

```
.radial {background-image: radial-gradient(purple, gold);}
```

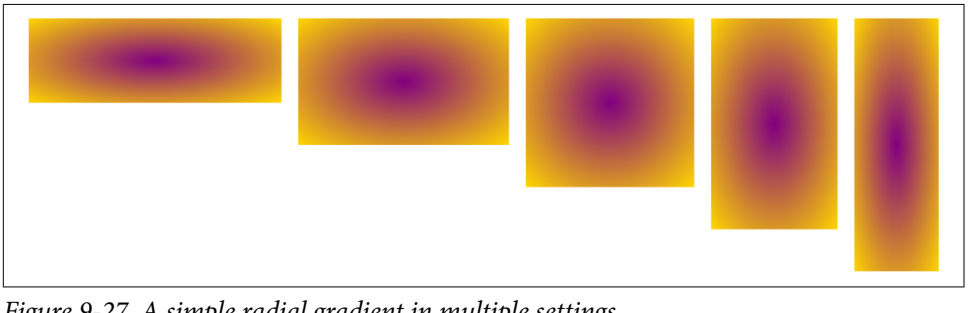


Figure 9-27. A simple radial gradient in multiple settings

In all of these cases, because no position is declared, the default of `center` is used, and the default ellipse has the same aspect ratio as the image size. Because no shape is declared, the shape is an ellipse for all cases but the square element; in that case, the shape is a circle. Finally, because no color-stop or color-hint positions are declared, the first is placed at

the beginning of the gradient ray, and the last at the end, with a linear blend from one to the other.

That's right: the *gradient ray* is the radial equivalent of the gradient line in linear gradients. It extends outward from the center of the gradient directly to the right, and the rest of the gradient is constructed from it. (We'll get to the details in just a bit.)

Setting Shape and Size

First off, a radial gradient has exactly two possible shape values (and thus two possible shapes): `circle` and `ellipse`. The shape of a gradient can be declared explicitly or can be implied by the way you size the gradient image.

So, on to sizing. As always, the simplest way to size a radial gradient is with either one nonnegative length (if you're sizing a circle) or two nonnegative lengths (if it's an ellipse). Say you have this radial gradient:

```
radial-gradient(50px, purple, gold)
```

This creates a circular radial gradient that fades from purple at the center to gold at a distance of 50 pixels from the center. If we add another length, the shape becomes an ellipse that's as wide as the first length and as tall as the second length:

```
radial-gradient(50px 100px, purple, gold)
```

Figure 9-28 shows these two gradients.

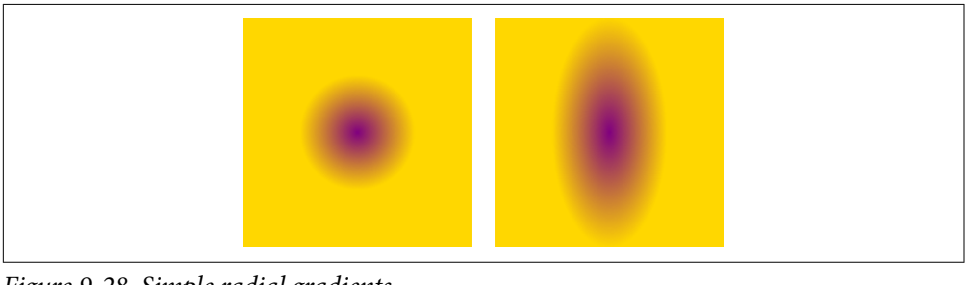


Figure 9-28. Simple radial gradients

Notice that the shape of the gradients has nothing to do with the overall size and shape of the images in which they appear. If you make a gradient a circle, it will be a circle, even if it's inside a rectangular gradient image. So too will an ellipse always be an ellipse, even when inside a square gradient image.

You can also use percentage values for the size, but *only* for ellipses. Circles cannot be given percentage sizes because there's no way to indicate the axis to which that percentage refers. (Imagine an image 100 pixels tall by 500 wide. Should `10%` mean 10 pixels or 50 pixels?) If you try to provide percentage values for a circle, the entire declaration will fail because of the invalid value.

If you do supply percentages to an ellipse, then as usual, the first refers to the horizontal axis and the second to the vertical. The following gradient is shown in various settings in Figure 9-29:

```
radial-gradient(50% 25%, purple, gold)
```

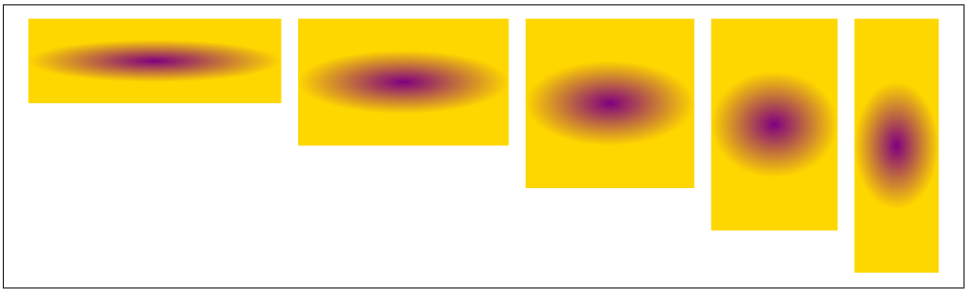


Figure 9-29. Percentage-sized elliptical gradients

When it comes to ellipses, you're also able to mix lengths and percentages, with the usual caveat to be careful. So if you're feeling confident, you can absolutely make an elliptical radial gradient 10 pixels tall and half the element width, like so:

```
radial-gradient(50% 10px, purple, gold)
```

As it happens, lengths and percentages aren't the only way to size radial gradients. In addition to those value types, four keywords are available for sizing radial gradients, the effects of which are summarized here:

`closest-side`
: If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the edge of the gradient image that is closest to the center point of the radial gradient. If the shape is an ellipse, the end of the gradient ray exactly touches the closest edge in each of the horizontal and vertical axes.

`farthest-side`
: If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the edge of the gradient image that is farthest from the center point of the radial gradient. If the shape is an ellipse, the end of the gradient ray exactly touches the farthest edge in each of the horizontal and vertical axes.

`closest-corner`
: If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the corner of the gradient image that is closest to the center point of the radial gradient. If the shape is an ellipse, the end of the gradient ray still touches the corner closest to the center, *and* the ellipse has the same aspect ratio that it would have had if `closest-side` had been specified.

`farthest-corner` *(default)*
: If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the corner of the gradient image that is farthest from the center point of the radial gradient. If the shape is an ellipse, the end of the gradient ray still touches the corner farthest from the center, *and* the ellipse has the same aspect ratio that it would have had if `farthest-side` had been specified. Note: this is the default size value for a radial gradient and so is used if no size values are declared.

To better visualize the results of each keyword, see Figure 9-30, which depicts each keyword applied as both a circle and an ellipse.

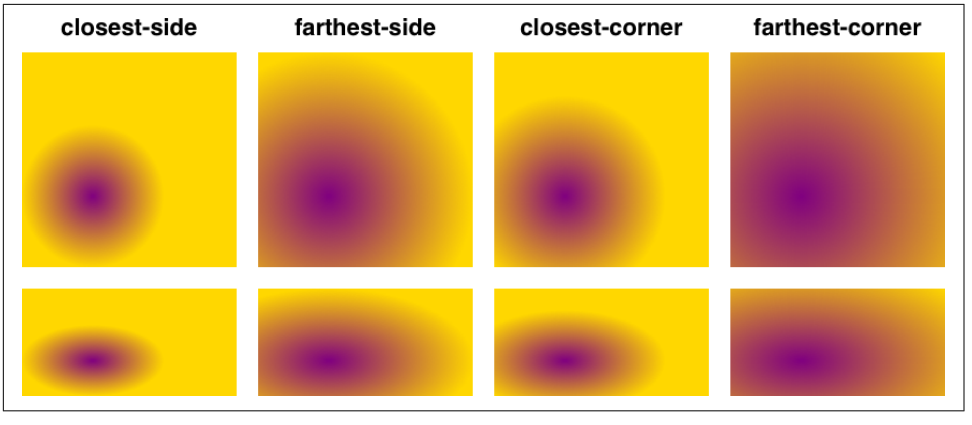


Figure 9-30. The effects of radial-gradient sizing keywords (positioned `at 33% 66%`)

These keywords cannot be mixed with lengths or percentages in elliptical radial gradients; thus, `closest-side 25px` is invalid and will be ignored.

Something you might have noticed in Figure 9-30 is that the gradients didn't start at the center of the image. That's because they were positioned elsewhere, which is the topic of the next section.

Positioning Radial Gradients

If you want to shift the center of a radial gradient away from the default of `center`, you can do so using any position value that would be valid for `background-position`. We're not going to reproduce that rather complicated syntax here; flip back to "Positioning Background Images" on page 319 if you need a refresher.

When we say "any position value that would be valid," that means any permitted combination of lengths, percentages, keywords, and so on. It also means that if you leave off one of the two position values, it will be inferred just the same as for `background-position`. So, just for one example, `center` is equivalent to `center center`. The one major difference between radial gradient positions and background positions is the default: for radial gradients, the default position is `center`, not `0% 0%`.

To give an idea of the possibilities, consider the following rules, illustrated in Figure 9-31:

```
radial-gradient(at bottom left, purple, gold);
radial-gradient(at center right, purple, gold);
radial-gradient(at 30px 30px, purple, gold);
radial-gradient(at 25% 66%, purple, gold);
radial-gradient(at 30px 66%, purple, gold);
```

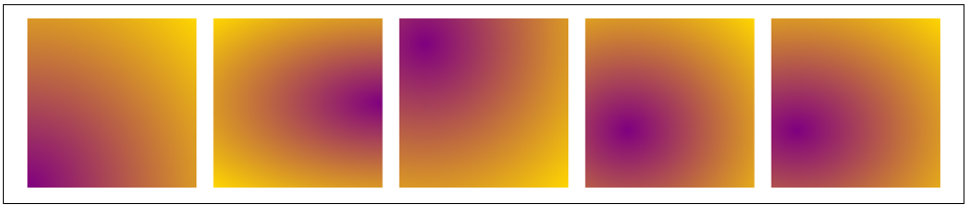


Figure 9-31. Changing the center position of radial gradients

None of those positioned radial gradients are explicitly sized, so they all default to `farthest-corner`. That's a reasonable guess at the intended default behavior, but it's not the only possibility. Let's mix some sizes into these gradients and find out how that changes things (as depicted in Figure 9-32):

```
radial-gradient(30px at bottom left, purple, gold);
radial-gradient(30px 15px at center right, purple, gold);
radial-gradient(50% 15% at 30px 30px, purple, gold);
radial-gradient(farthest-side at 25% 66%, purple, gold);
radial-gradient(closest-corner at 30px 66%, purple, gold);
```

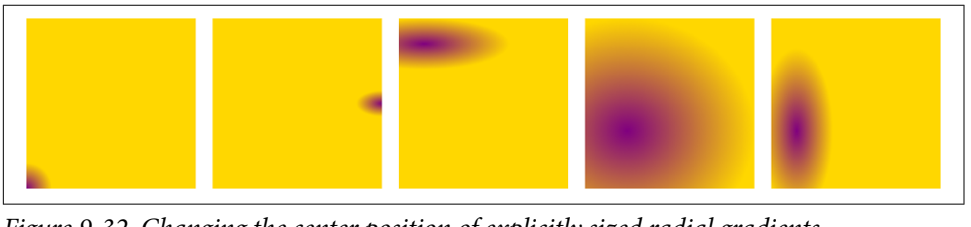


Figure 9-32. Changing the center position of explicitly sized radial gradients

Nifty. Now, suppose we want something a little more complicated than a fade from one color to another. Next stop, color stops!

Using Radial Color Stops and the Gradient Ray

Color stops for radial gradients have the same syntax as, and work in a similar fashion to, linear gradients. Let's return to the simplest possible radial gradient and follow it with a more explicit equivalent:

```
radial-gradient(purple, gold);
radial-gradient(purple 0%, gold 100%);
```

So the gradient ray extends out from the center point. At 0% (the start point, and also the center of the gradient), the ray will be purple. At 100% (the ending point), the ray will be gold. Between the two stops is a smooth blend from purple to gold; beyond the ending point is solid gold.

If we add a stop between purple and gold, but don't give it a position, the stop will be placed midway between the two colors, and the blending will be altered accordingly, as shown in Figure 9-33:

```
radial-gradient(100px circle at center, purple 0%, green, gold 100%);
```

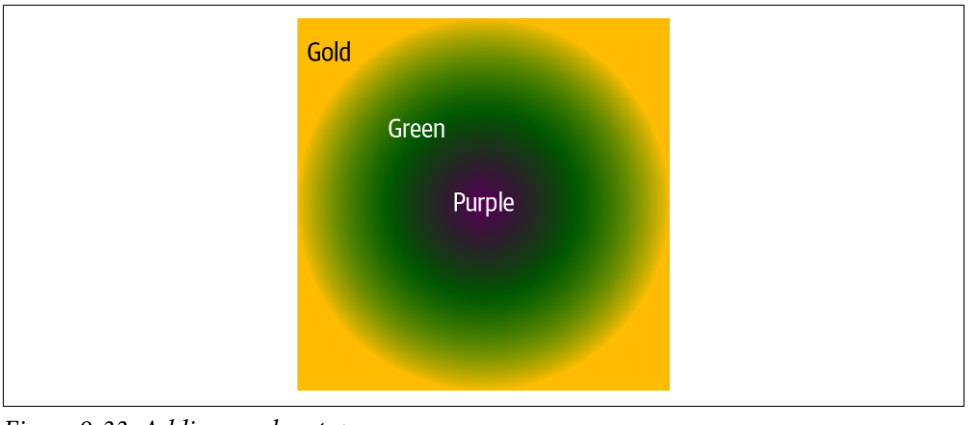


Figure 9-33. Adding a color stop

We'd have gotten the same result if we'd added `green 50%` there, but you get the idea. The gradient ray's color goes smoothly from purple to green to gold, and then is solid gold beyond that point on the ray.

This illustrates one difference between gradient lines (for linear gradients) and gradient rays: a linear gradient is derived by extending the color perpendicularly at each point along the gradient line. A similar behavior occurs with a radial gradient, except instead of the lines that come off the gradient ray, ellipses are created; these are scaled-up or scaled-down versions of the ellipse at the ending point. Figure 9-34 illustrates a gradient ray and the ellipses that are drawn at various points along it.

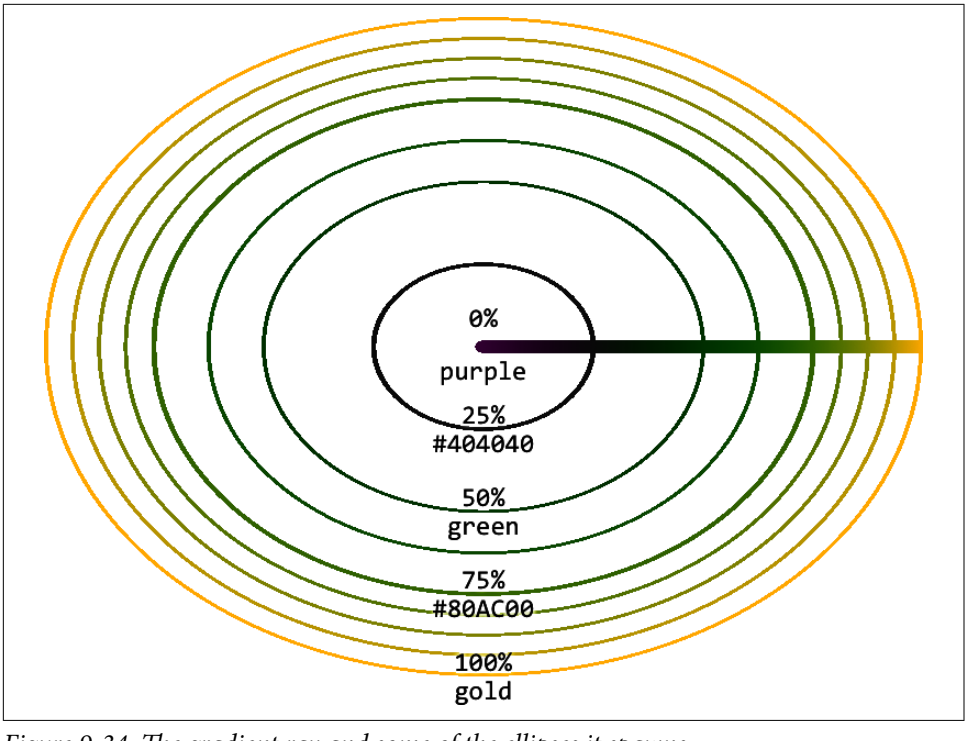


Figure 9-34. The gradient ray and some of the ellipses it spawns

That brings up an interesting question: how is the ending point (the `100%` point, if you like) determined for each gradient ray? It's the point where the gradient ray intersects with the shape described by the size. For a circle, that's easy: the gradient ray's ending point is however far from the center that the size value indicates. So for a `25px circle` gradient, the ending point of the ray is 25 pixels from the center.

For an ellipse, it's essentially the same operation, except that the distance from the center is dependent on the horizontal axis of the ellipse. Given a radial gradient that's a `40px 20px ellipse`, the ending point will be 40 pixels from the center and directly to its right. Figure 9-35 shows this in some detail.

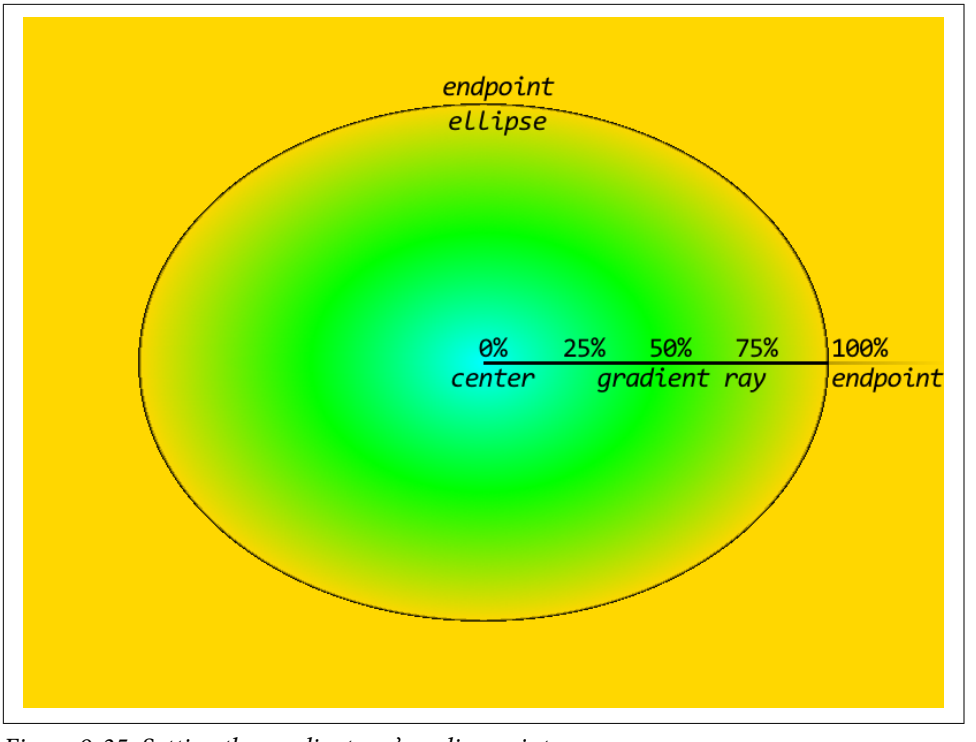


Figure 9-35. Setting the gradient ray's ending point

Another difference between linear gradient lines and radial gradient rays is that you can see beyond the ending point. As you may recall, a linear gradient line is always drawn so that you can see the colors at the 0% and 100% points, but nothing beyond them; the gradient line can never be any smaller than the longest axis of the gradient image and will frequently be longer than that. With a radial gradient, on the other hand, you can size the radial shape to be smaller than the total gradient image. In that case, the color at the last color stop is extended outward from the ending point. (You've already seen this in several previous figures.)

Conversely, if you set a color stop that's beyond the ending point of a ray, you might get to see the color out to that stop. Consider the following gradient, illustrated in Figure 9-36:

```
radial-gradient(50px circle at center, purple, green, gold 80px)
```

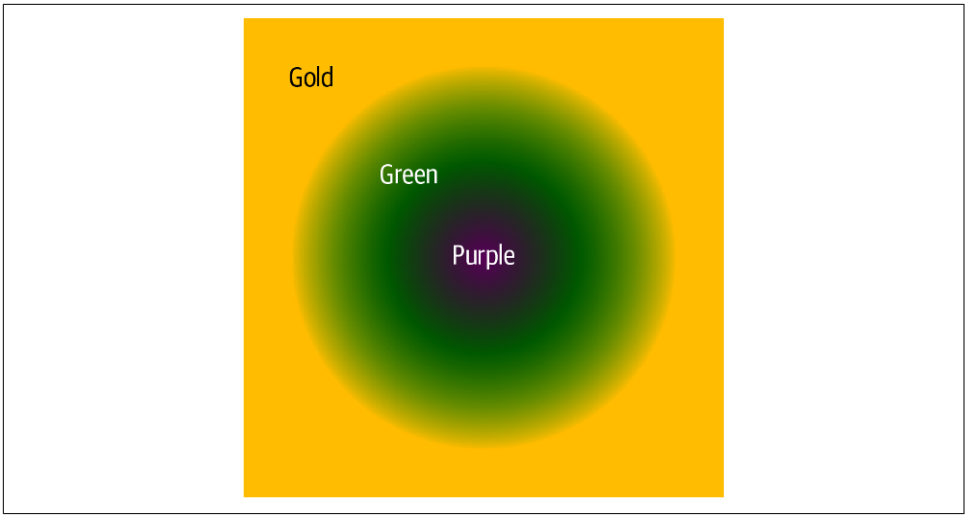


Figure 9-36. Color stops beyond the ending point

The first color stop has no position, so it's set to `0%`, which is the center point. The last color stop is set to `80px`, so it will be 80 pixels away from the center in all directions. The middle color stop, `green`, is placed midway between the two (40 pixels from the center). So we get a gradient that goes out to gold at 80 pixels and then continues gold beyond that point.

This happens even though the circle is explicitly set to be 50 pixels large. It still is 50 pixels in radius; it's just that the positioning of the last color stop makes that fact vaguely irrelevant. Visually, we might as well have declared this:

```
radial-gradient(80px circle at center, purple, green, gold)
```

Or, more simply, just this:

```
radial-gradient(80px, purple, green, gold)
```

The same behaviors apply if you use percentages for your color stops. These are equivalent to the previous examples, and to each other, visually speaking:

```
radial-gradient(50px, purple, green, gold 160%)
radial-gradient(80px, purple, green, gold 100%)
```

Now, what if you set a negative position for a color stop? The result is pretty much the same as for linear gradient lines: the negative color stop is used to figure out the color at the starting point but is otherwise unseen. Thus, the following gradient will have the result shown in Figure 9-37:

```
radial-gradient(80px, purple -40px, green, gold)
```

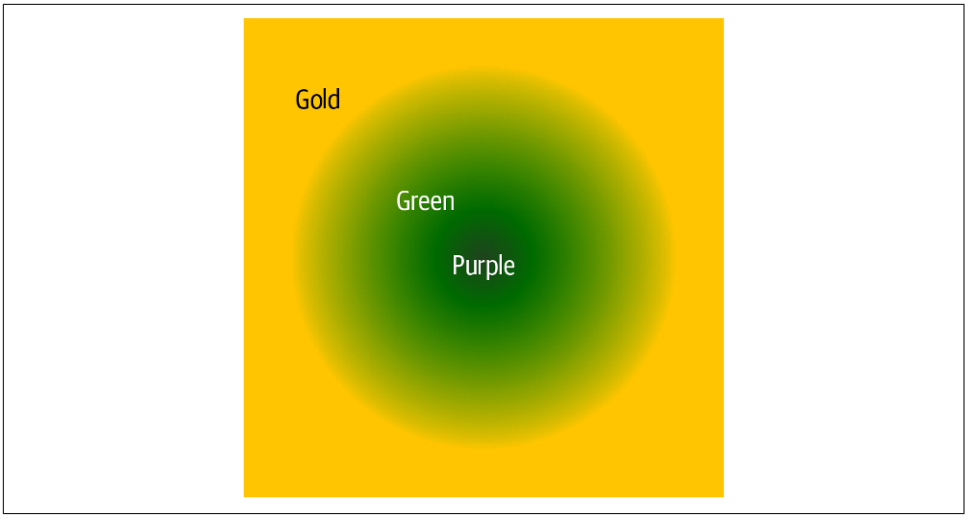


Figure 9-37. Handling a negative color-stop position

Given these color-stop positions, the first color stop is at `-40px`, the last is at `80px` (because, given its lack of an explicit position, it defaults to the ending point), and the middle is placed midway between them. The result is the same as if we'd explicitly used this:

```
radial-gradient(80px, purple -40px, green 20px, gold 80px)
```

That's why the color at the center of the gradient is a green-purple: it's a blend of one-third purple, two-thirds green. From there, it blends the rest of the way to green, and then on to gold. The rest of the purple-green blend, the part that sits on the "negative space" of the gradient ray, is invisible.

Handling Degenerate Cases

Given that we can declare size and position for a radial gradient, the question arises: what if a circular gradient has zero radius, or an elliptical gradient has zero height or width? These conditions aren't quite as hard to create as you might think. Besides explicitly declaring that a radial gradient has zero size using `0px` or `0%`, you could also do something like this:

```
radial-gradient(closest-corner circle at top right, purple, gold)
```

The gradient's size is set to `closest-corner`, and the center has been moved into the `top right` corner, so the closest corner is 0 pixels away from the center. Now what?

In this case, the specification explicitly says that the gradient should be rendered as if it's "a circle whose radius [is] an arbitrary very small number greater than zero." So that might mean as if it had a radius of one-one-billionth of a pixel, or a picometer, or heck, the Planck length. The interesting thing is that it means the gradient is still a circle. It's just

a very, very, very small circle. Probably, it will be too small to render anything visible. If so, you'll just get a solid-color fill that corresponds to the color of the last color stop instead.

Ellipses with zero-length dimensions have fascinatingly different defined behaviors. Let's assume the following:

```
radial-gradient(0px 50% at center, purple, gold)
```

The specification states that any ellipse with a zero width is rendered as if it's "an ellipse whose height [is] an arbitrary very large number and whose width [is] an arbitrary very small number greater than zero." In other words, render it as though it's a linear gradient mirrored around the vertical axis running through the center of the ellipse. The specification also says that in such a case, any color stops with percentage positions resolve to `0px`. This will usually result in a solid color matching the color defined for the last color stop.

On the other hand, if you use lengths to position the color stops, you can get a vertically mirrored horizontal linear gradient for free. Consider the following gradient, illustrated in Figure 9-38:

```
radial-gradient(0px 50% at center, purple 0px, gold 100px)
```

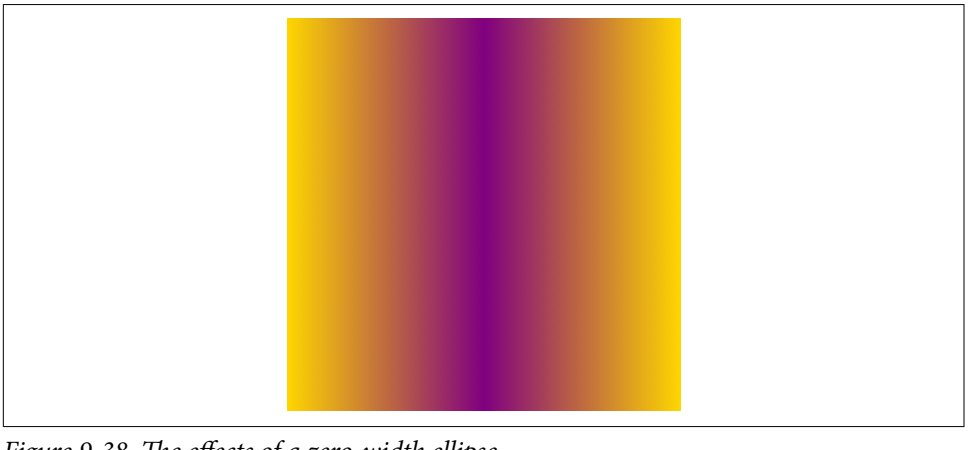


Figure 9-38. The effects of a zero-width ellipse

How did this happen? First, remember that the specification says that the `0px` horizontal width is treated as if it's a tiny nonzero number. For the sake of illustration, let's suppose that's one-one-thousandth of a pixel (0.001 px). That means the ellipse shape is a thousandth of a pixel wide by half the height of the image. Again for the sake of illustration, let's suppose that's a height of 100 pixels. That means the first ellipse shape is a thousandth of a pixel wide by 100 pixels tall, which is an aspect ratio of 0.001:100, or 1:100,000.

OK, so every ellipse drawn along the gradient ray has a 1:100,000 aspect ratio. That means the ellipse at half a pixel along the gradient ray is 1 pixel wide and 100,000 pixels tall. At 1 pixel, it's 2 pixels wide and 200,000 pixels tall. At 5 pixels, the ellipse is 10 pixels by a

million pixels. At 50 pixels along the gradient ray, the ellipse is 100 pixels wide and 10 million tall. And so on. This is diagrammed in Figure 9-39.

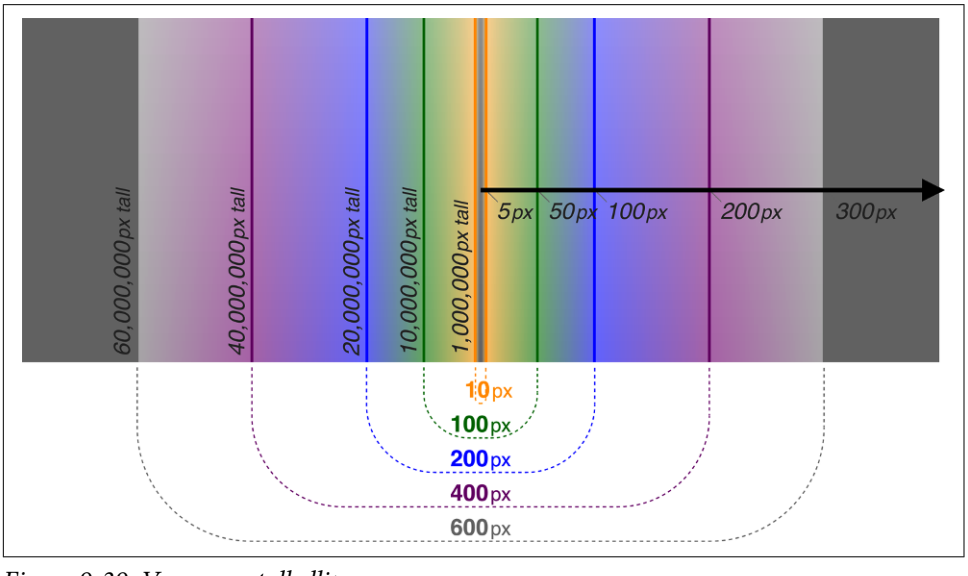


Figure 9-39. Very, very tall ellipses

So you can see why the visual effect is of a mirrored linear gradient. These ellipses are effectively drawing vertical lines. Technically they aren't, but in practical terms they are. The result is as if you have a vertically mirrored horizontal gradient, because each ellipse is centered on the center of the gradient, and both sides of it get drawn. While this may be a radial gradient, we can't see its radial nature.

On the other hand, if the ellipse has width but not height, the results are quite different. You'd think the result would be a vertical linear gradient mirrored around the horizontal axis, but not so! Instead, the result is a solid color equal to the last color stop (unless it's a repeating gradient, a subject we'll turn to shortly, in which case it should be a solid color equal to the average color of the gradient). So, given either of the following, you'll get a solid gold:

```
radial-gradient(50% 0px at center, purple, gold)
radial-gradient(50% 0px at center, purple 0px, gold 100px)
```

Why the difference? It goes back to the way radial gradients are constructed from the gradient ray. Again, remember that, per the specification, a zero distance here is treated as a very small nonzero number. As before, we'll assume that `0px` is reassigned to `0.001px`, and that the `50%` evaluates to 100 pixels. That's an aspect ratio of 100:0.001, or 100,000:1.

So, to get an ellipse that's 1 pixel tall, the width of that ellipse must be 100,000 pixels. But our last color stop is at only 100 pixels! At that point, the ellipse that's drawn is 100 pixels wide and a thousandth of a pixel tall. All of the purple-to-gold transition that happens

along the gradient ray has to happen in that thousandth of a pixel. Everything after that is gold, as per the final color stop. Thus, we can see only the gold.

You might think that if you increased the position value of the last color stop to `100000px`, you'd see a thin sliver of purplish color running horizontally across the image. And you'd be right, *if* the browser treats `0px` as `0.001px` in these cases. If it assumes `0.00000001px` instead, you'd have to increase the color stop's position a *lot* further in order to see anything. And that's assuming the browser was actually calculating and drawing all those ellipses, instead of just hardcoding the special cases. The latter is a lot more likely, honestly. It's what we'd do if we were in charge of a browser's gradient-rendering code.

And what if an ellipse has zero width *and* zero height? In that case, the specification is written such that the zero-width behavior is used; thus, you'll get the mirrored-linear-gradient behavior.



As of late 2022, browser support for the defined behavior in these edge cases is unstable, at best. Some browsers use the last color-stop's color in all cases, and others refuse to draw a gradient at all in some cases.

Repeating Radial Gradients

While percentages in repeating linear gradients could turn them into nonrepeating gradients, percentages can be very useful if the size of the circle or ellipse is defined, percentage positions along the gradient ray are defined, and you can see beyond the endpoint of the gradient ray. For example, assume the following:

```
.allhail {background:
    repeating-radial-gradient(100px 50px, purple, gold 20%, green 40%,
                              purple 60%, yellow 80%, purple);}
```

As there are five color stops and the size is 100px, a color stop will occur every 20 pixels, with the colors repeating in the declared pattern. Because the first and last color stops have the same color value, there is no hard color switch. The ripples just spread out forever, or at least until they're beyond the edges of the gradient image. See Figure 9-40 for an example.

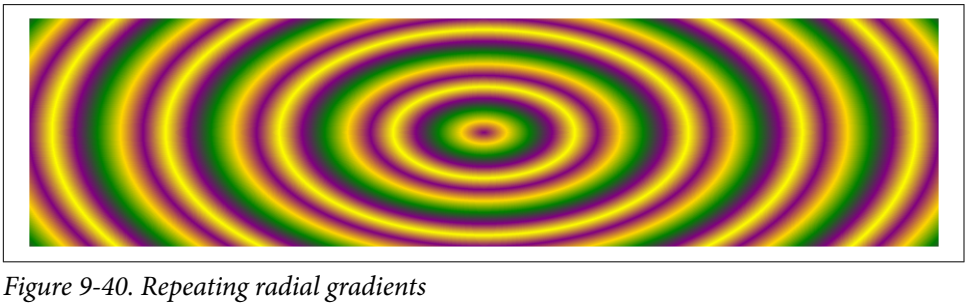


Figure 9-40. Repeating radial gradients

Just imagine what that would look like with a repeating radial gradient of a rainbow!

```
.wdim {background:
    repeating-radial-gradient(
        100px circle at bottom center,
        rgb(83%,83%,83%) 50%,
        violet 55%, indigo 60%, blue 65%, green 70%,
        yellow 75%, orange 80%, red 85%,
        rgb(47%,60%,73%) 90%
    );}
```

Keep these two points in mind when creating repeating radial gradients:

- If you don't declare size dimensions for a radial, it will default to an ellipse that has the same height-to-width ratio as the overall gradient image; *and*, if you don't declare a size for the image with `background-size`, the gradient image will default to the height and width of the element background where it's being applied (or, if being used as a list-style bullet, the size that the browser gives it).
- The default radial size value is `farthest-corner`. This will put the endpoint of the gradient ray far enough to the right that its ellipse intersects with the corner of the gradient image that's farthest from the center point of the radial gradient.

These are reiterated here to remind you that if you stick to the defaults, there's not really any point to having a repeating gradient, since you'll be able to see only the first iteration of the repeat. It's only when you restrict the initial size of the gradient that the repeats become visible.

Conic Gradients

Radial gradients are fun, but what if you want a gradient that wraps around a central point, similar to a color hue wheel? That's what CSS calls a *conic gradient*, which can be thought of as a concentric series of linear gradients that are bent into circles. Looked at another way, at any distance from the center, there's a circle whose outer rim could be straightened out into a linear gradient with the color stop specified.

Conic gradients are more easily shown than described, so consider the following CSS, which is illustrated in Figure 9-41 along with a linear diagram to show how the stops wrap around the conical space:

```
background:
     conic-gradient(
          black, gray, black, white, black, silver, gray
     );
```

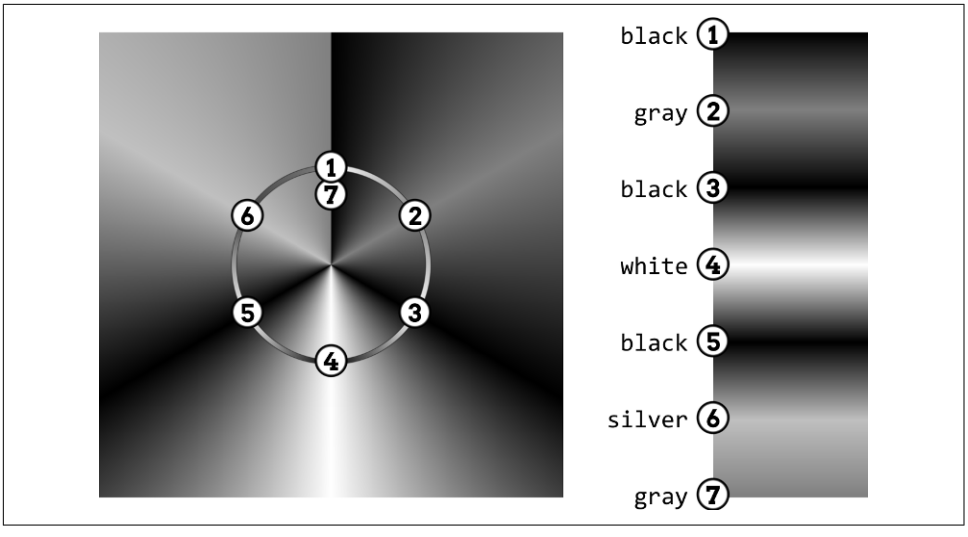


Figure 9-41. A simple conic gradient and its linear equivalent

Note how each of the color stops is labeled on the linear gradient: the circled numbers listed there are repeated in the conic gradient, to show where each color stop falls. At 60 degrees around the conic gradient, there is a `gray` color stop. At 180 degrees, a `white` color stop. At the top of the conic gradient, the `0deg` and `360deg` points meet, so `black` and `gray` sit next to each other.

By default, conic gradients start at 0 degrees, using the same compass degree system that transforms and other parts of CSS use, so `0deg` is at the top. If you want to start from a different angle and wrap around the circle back to that point, it's as straightforward as adding `from` and an angle value to the front of the `conic-gradient` value, which rotates the entire gradient by the declared angle. The following would all have the same result:

```
conic-gradient(from 144deg, black, gray, black, white)
conic-gradient(from 2.513274rad, black, gray, black, white)
conic-gradient(from 0.4turn, black, gray, black, white)
```

If the conic gradient is given a different start angle, such as `from 45deg`, it acts as a rotation of the entire conic gradient. Consider the following two examples, with the results depicted in Figure 9-42:

```
conic-gradient(black, white 90deg, gray 180deg, black 270deg, white)
conic-gradient(from 45deg, black, white 90deg, gray 180deg, black 270deg, white)
```

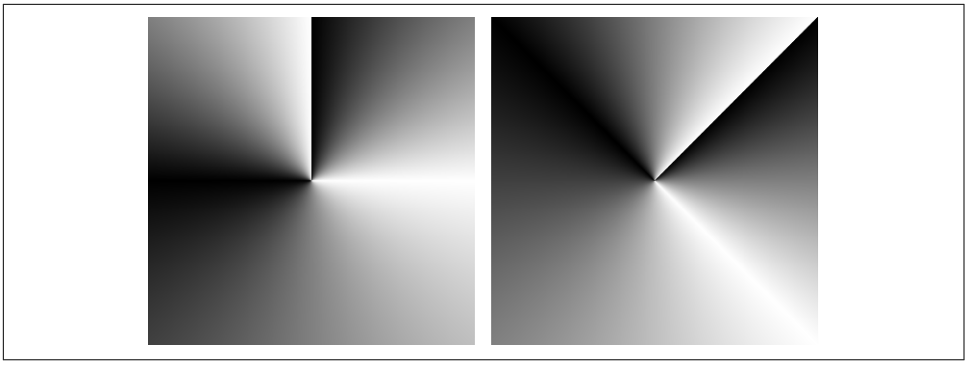


Figure 9-42. Conic gradients with angled color stops and different start angles

Not only is the starting point rotated 45 degrees, but all the other color stops are as well. Thus, even though the first color stop has an angle of `90deg`, it actually occurs at the 135-degree mark, that being 90 degrees with a 45-degree rotation added.

It's also possible to change the location of the gradient's center point within the image, just as with radial gradients. The syntax is quite similar, as you can see in this code block (illustrated in Figure 9-43):

```
conic-gradient(from 144deg at 3em 6em, black, gray, black, white)
conic-gradient(from 144deg at 67% 25%, black, gray, black, white)
conic-gradient(from 144 deg at center bottom, black, gray, black, white)
```

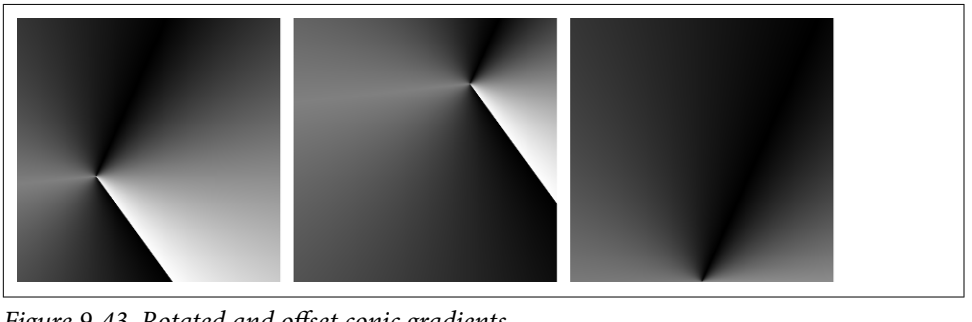


Figure 9-43. Rotated and offset conic gradients

In the first of the three examples, the center of the conic gradient is placed `3em` to the right of the top-left corner, and `6em` down from that same corner. Similarly, the second example shows the center point `67%` of the way across the conic-gradient image, and `25%` down from the top.

The third example shows what happens when the center point of a conic gradient is placed along one edge of the image: we see only half (at most) of the gradient. In this case, the top half is visible—that is, the colors from 270 degrees through 90 degrees.

So all together, the syntax for a conic gradient is as follows:

```
conic-gradient(
    [ from <angle>]? [ at <position>]? , | at <position>, ]?
      <color-stop> , [ <color-hint>]? , <color-stop> ]+
)
```

If the `from` angle is not given, it defaults to `0deg`. If no `at` position is given, it defaults to `50% 50%` (that is, the center of the conic-gradient image).

Much as with radial and linear gradients, color stop distances can be specified by a percentage value; in this case, it resolves to an angle value. Thus, for a conic gradient starting at 0 degrees, the color stop distance `25%` would resolve to 90 degrees, as 90 is 25% of 360. Conic color stops can also be specified as a degree value, as shown previously.

You *cannot* specify a length value for a conic gradient's color stop's distance. Only percentages and angles are acceptable, and they can be mixed.

Creating Conic Color Stops

If you want a conic gradient to blend smoothly from color to color all the way around the circle, it is necessary to make the last color stop match the first color stop. Otherwise, you'll see the kinds of hard transitions shown in earlier examples. If you want to create a color hue wheel, for example, you need to declare it like so:

```
conic-gradient(red, magenta, blue, aqua, lime, yellow, red)
```

Except that's not actually a wheel, since the conic-gradient image fills the entire background area, and background areas in CSS are (thus far) rectangular by default. To make the color wheel actually look like a color wheel, you'd need to either use a circular clipping path (see Chapter 20) or round the corners on a square element (see Chapter 7). For example, the following will have the result shown in Figure 9-44:

```
.hues {
     height: 10em; width: 10em;
     background: conic-gradient(red, magenta, blue, aqua, lime, yellow, red);
}
#wheel {
     border-radius: 50%;
}
```

```
<div class="hues"></div>
<div class="hues" id="wheel"></div>
```

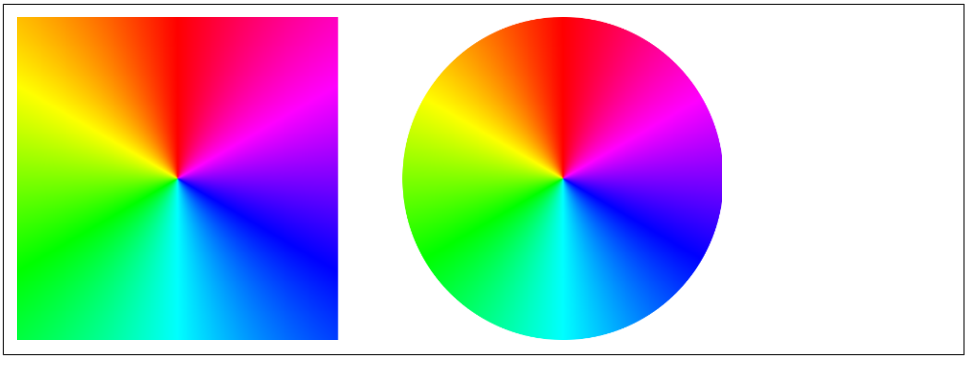


Figure 9-44. Hue-wheel conic gradients with and without corner rounding

This emphasizes that while it's easy to think of conic gradients as circles, the end result is a rectangle, absent any clipping or other effort to make the element's background area non-rectangular. So if you're thinking about using conic gradients to make, say, a pie chart, you'll have to do more than just define a conic gradient with hard stops.

Just as we used two length-percentage values to create hard stops in linear gradients, we can use two hard stops in conic gradients. For example:

```
conic-gradient(
        green 37.5%,
        yellow 37.5% 62.5%,
        red 62.5%);
```

In this syntax, a given color stop can be written as <*color*> <*beginning*> <*ending*>, where <*beginning*> and <*ending*> are percentage or angle values.

If you want to create smoother transitions between colors but still have them be mostly solid, the <*color*> <*beginning*> <*ending*> syntax can help a lot. For example, the following conic gradient eases the transitions between green, yellow, and red without making the overall gradient too "smeared":

```
conic-gradient(green 35%, yellow 40% 60%, red 65%);
```

This runs a solid wedge of green from 0 to 126 degrees (35%), then transitions smoothly from green to yellow between 126 degrees and 144 degrees (40%), past which there is a solid wedge of yellow spanning from 144 degrees to 216 degrees (60%). Similarly, a smoothed transition occurs from yellow to red between 216 degrees and 234 degrees (65%), and beyond that, a solid red wedge running to 360 degrees.

All this is illustrated in Figure 9-45, with extra annotations to mark where the calculated angles land.

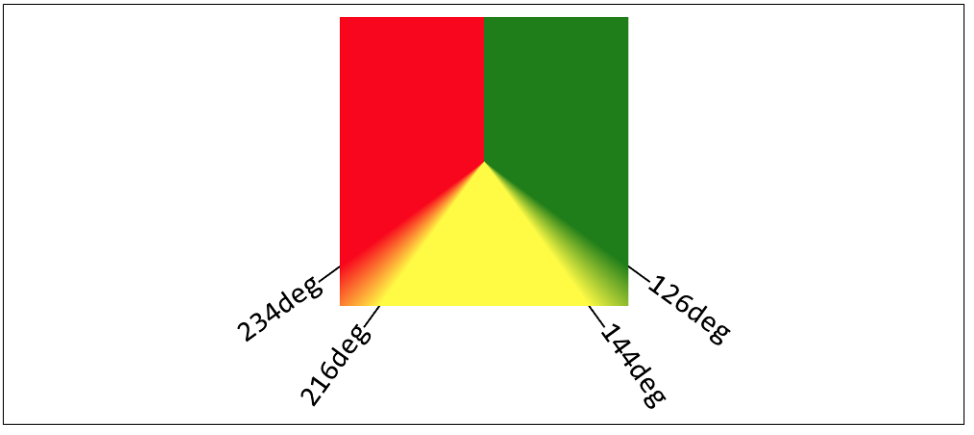


Figure 9-45. Conic gradients with solid-color wedges and smooth transitions

And, as it happens, that syntax makes it relatively easy to re-create those picnic tablecloths discussed earlier in the chapter by using a conic gradient:

```
background-image: conic-gradient(
        rgba(0 0 0 / 0.2) 0% 25%,
        rgba(0 0 0 / 0.4) 25% 50%,
        rgba(0 0 0 / 0.2) 50% 75%,
        transparent 75% 100%
        );
background-size: 2vw 2vw;
background-repeat: repeat;
```

This creates, in a single gradient image, a set of four squares in the pattern. That image is then sized and repeated. It's not more efficient or elegant than using repeating linear gradients, but it does embody a certain cleverness that appeals to us.

Repeating Conic Gradients

And now we come to repeating conic gradients, which are highly useful if you want to create a starburst pattern or even something simple like a checkerboard pattern. For example:

```
conic-gradient(
    #0002 0 25%, #FFF2 0 50%, #0002 0 75%, #FFF2 0 100%
    )
```

This sets up a checkerboard pattern with four color stops but only two colors. We can restate that using `repeating-conic-gradient` like so, with new colors to make the pattern a little clearer:

```
repeating-conic-gradient(
     #343 0 25%, #ABC 0 50%
     )
```

All that was necessary in this simple repeating case was to set up the first two color stops. After that, the stops are repeated until the full 360 degrees of the conic gradient are filled, as shown in Figure 9-46.

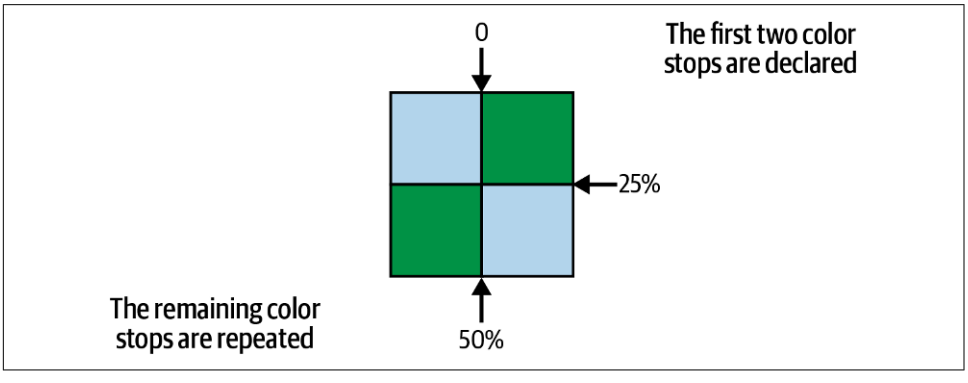


Figure 9-46. A repeating conic gradient

This means we can create wedges of any size, with any transition, and repeat them all the way around the conic circle. Here are just three examples, rendered in Figure 9-47:

```
repeating-conic-gradient(#117 5deg, #ABE 15deg, #117 25deg)
repeating-conic-gradient(#117 0 5deg, #ABE 0 15deg, #117 0 25deg)
repeating-conic-gradient(#117 5deg, #ABE 15deg)
```

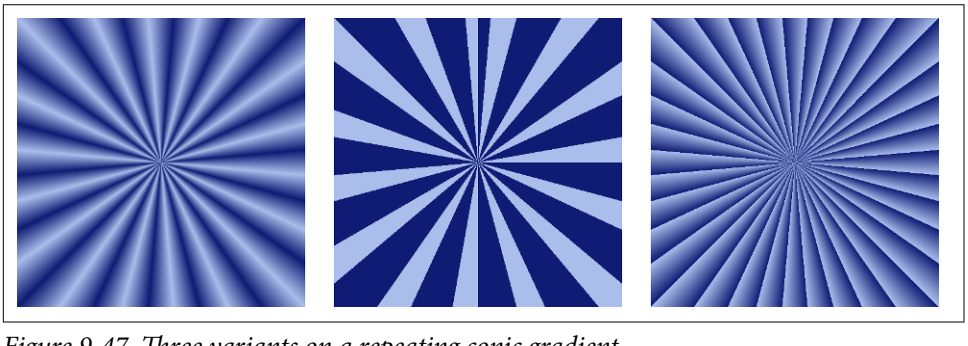


Figure 9-47. Three variants on a repeating conic gradient

Notice that the first (leftmost) example's smoothed transitions hold true even at the top of the image: the transition from `#117` at 350 degrees to `#ABE` at 5 degrees is handled like all of the other transitions. Repeated conic gradients are unique in this way, since both linear and radial gradients never "wrap around" to have the end meet the beginning. This is also seen in the third (rightmost) example in Figure 9-47.

It's possible to break this special behavior, though, as the second (center) example illustrates: note the narrower wedge from 355 degrees through 360 degrees. This happens because the first color stop in the pattern explicitly runs from 0 degrees through 5 degrees. Thus, there is no way to transition from 355 degrees through to 5 degrees, which leads to a hard transition at 360/0 degrees.

Manipulating Gradient Images

As we have previously emphasized (possibly to excess), gradients are images. That means you can size, position, repeat, and otherwise affect them with the various background properties, just as you would any PNG or SVG.

One way this can be leveraged is to repeat simple gradients. (Repeating in more complex ways is the subject of the next section.) For example, you could use a hard-stop radial gradient to give your background a dotted look, as shown in Figure 9-48:

```
body {background: radial-gradient(circle at center,
                    rgba(0 0 0 / 0.1), rgba(0 0 0 / 0.1) 10px,
                    transparent 10px, transparent)
                    center / 25px 25px repeat,
                    tan;}
```



Figure 9-48. Tiled radial gradient images

Yes, this is visually pretty much the same as tiling a PNG that has a mostly transparent dark circle 10 pixels in diameter. Using a gradient in this case has three advantages:

- The CSS is almost certainly smaller in bytes than the PNG would be.
- Even more importantly, the PNG requires an extra hit on the server. This slows both page and server performance. A CSS gradient is part of the stylesheet and so eliminates the extra server hit.
- Changing the gradient is a lot simpler, so experimenting to find exactly the right size, shape, and darkness is much easier.

Creating Special Effects

Gradients can't do everything a raster or vector image can, so it's not as though you'll be giving up external images completely now that gradients are a thing. You can still pull off some pretty impressive effects with gradients, though. Consider the background effect shown in Figure 9-49.



Figure 9-49. It's time to play the music…

That curtain effect is accomplished with just two linear gradients repeated at differing intervals, plus a third to create a "glow" effect along the bottom of the background. Here's the code that accomplishes it:

```
background-image:
    linear-gradient(0deg, rgba(255 128 128 / 0.25), transparent 75%),
    linear-gradient(89deg,
        transparent 30%,
        #510A0E 35% 40%, #61100F 43%, #B93F3A 50%,
        #4B0408 55%, #6A0F18 60%, #651015 65%,
        #510A0E 70% 75%, rgba(255 128 128 / 0) 80%, transparent),
    linear-gradient(92deg,
        #510A0E 20%, #61100F 25%, #B93F3A 40%, #4B0408 50%,
        #6A0F18 70%, #651015 80%, #510A0E 90%);
background-size: auto, 300px 100%, 109px 100%;
background-repeat: repeat-x;
```

The first (and therefore topmost) gradient is just a fade from a 75%-transparent light red up to full transparency at the 75% point of the gradient line. Then two "fold" images are created. Figure 9-50 shows each separately.

With those images defined, they are repeated along the x-axis and given different sizes. The first, which is the "glow" effect, is given `auto` size to let it cover the entire element background. The second is given a width of `300px` and a height of `100%`; thus, it will be as tall as the element background and 300 pixels wide. This means it will be tiled every 300 pixels along the x-axis. The same is true of the third image, except it tiles every 109 pixels. The end result looks like an irregular stage curtain.

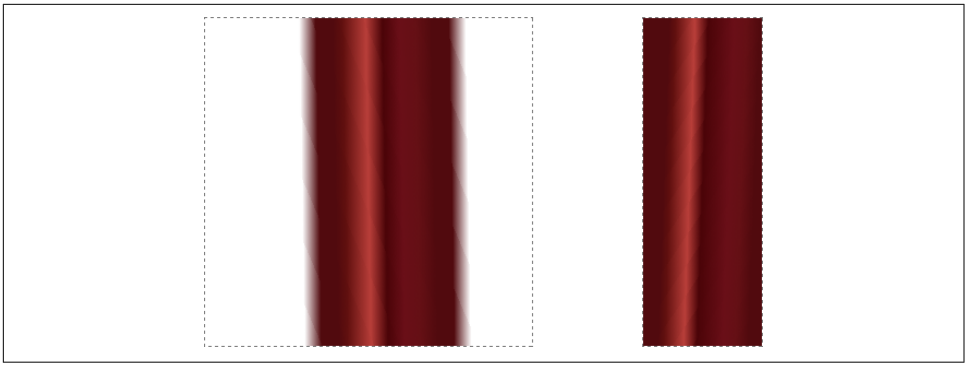


Figure 9-50. The two "fold" gradients

The beauty of this is that adjusting the tiling intervals is just a matter of editing the stylesheet. Changing the color-stop positions or the colors is less trivial, but not too difficult if you know the effect you're after. And adding a third set of repeating folds is no more difficult than just adding another gradient to the stack.

Triggering Average Gradient Colors

It's worth asking what happens if a repeating gradient's first and last color stops somehow end up being in the same place. For example, suppose your fingers miss the 5 key and you accidentally declare the following:

```
repeating-radial-gradient(center, purple 0px, gold 0px)
```

The first and last color stops are 0 pixels apart, but the gradient is supposed to repeat ad infinitum along the gradient line. Now what?

In such a case, the browser finds the *average gradient color* and fills it in throughout the entire gradient image. In our simple case in the preceding code, that will be a 50/50 blend of purple and `gold` (which will be about `#C06C40` or `rgb(75%,42%,25%)`). Thus, the resulting gradient image should be a solid orangey-brown, which doesn't really look much like a gradient.

This condition can also be triggered when the browser rounds the color-stop positions to 0, or when the distance between the first and last color stops is so small as compared to the output resolution that nothing useful can be rendered. This could happen if, for example, a repeating radial gradient used all percentages for the color-stop positions and was sized using `closest-side`, but was accidentally placed into a corner.



As of late 2022, no browsers really do average colors correctly. Some of the correct behaviors can be triggered under very limited conditions, but in most cases, browsers either just use the last color stop as a fill color, or else try really hard to draw subpixel repeating patterns.

Summary

Gradients are a fascinating image type, being constructed entirely with CSS values instead of with raster data or vector elements. With the three types of gradients available, you can create almost any pattern or visual effect.

CHAPTER 10

Floating and Positioning

For a very long time, floated elements were the basis of all our web layout schemes. (This is largely because of the property `clear`, which we'll get to in a bit.) But floats were never meant for layout; their use as a layout tool was a hack nearly as egregious as the use of tables for layout. They were just what we had. Floats are quite interesting and useful in their own right, however. This is especially true given the recent addition of float *shaping*, which allows the creation of nonrectangular shapes that content can flow past.

Floating

Ever since the early 1990s, it has been possible to float images by declaring, for instance, `<img src="b5.gif" alt="B4" align="right">`. This causes an image to float to the right and allows other content (such as text) to "flow around" the image. The name *floating*, in fact, came from the Netscape DevEdge page "Extensions to HTML 2.0," which explained the then-new `align` attribute. Unlike HTML, CSS lets you float any element, from images to paragraphs to lists. This is accomplished using the property `float`.

float

Values	`left` | `right` | `inline-start` | `inline-end` | `none`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

For example, to float an image to the left, you could use this markup:

```
<img src="b4.gif" style="float: left;" alt="b4">
```

As Figure 10-1 illustrates, the image "floats" to the left side of the browser window, and the text flows around it.

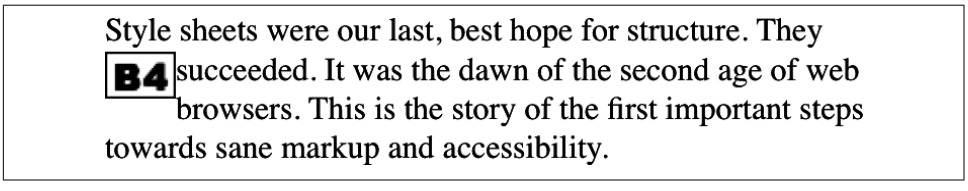


Figure 10-1. A floating image

You can float to the `left` or `right`, as well as to the `inline-start` and `inline-end` edges of an element. These latter two are useful when you want to float an element toward the start or end of the inline axis, regardless of the direction that axis is pointing. (See Chapter 6 for details on the inline axis.)



Throughout the rest of this section, we mostly stick to `left` and `right` because they simplify explanations. They will also be nearly the only `float` values you see in the wild, at least for the next few years.

Floated Elements

Keep a few points in mind with regard to floating elements. First, a floated element is, in some ways, removed from the normal flow of the document, although it still affects the layout of the normal flow. In a manner utterly unique within CSS, floated elements exist almost on their own plane, yet they still have influence over the rest of the document.

This influence arises because when an element is floated, other normal-flow content "flows around" it. This is familiar behavior with floated images, but the same is true if you float a paragraph, for example. In Figure 10-2, you can see this effect quite clearly, thanks to the margin added to the floated paragraph:

```
p.aside {float: inline-end; width: 15em; margin: 0 1em 1em;
     padding: 0.25em; border: 1px solid;}
```

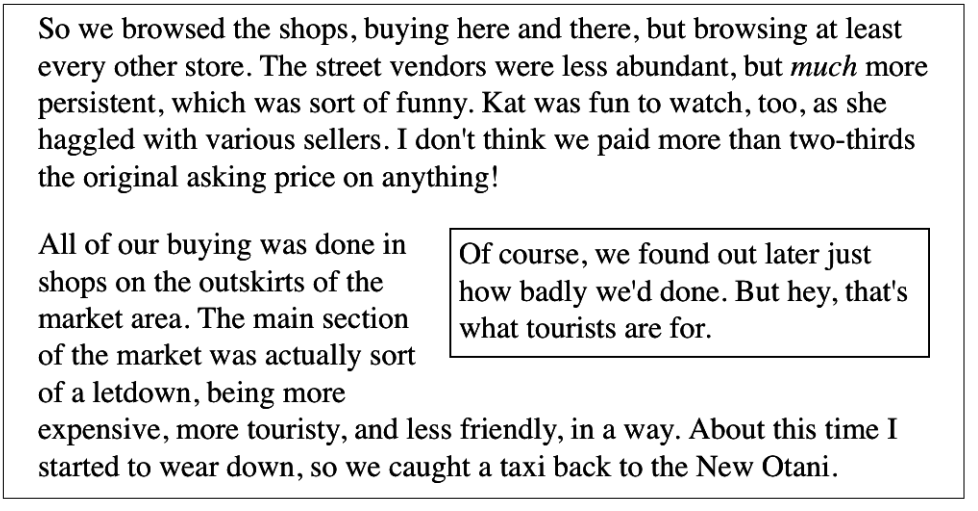


Figure 10-2. A floating paragraph

One of the first facts to notice about floated elements is that margins around floated elements do not collapse. If you float an image and give it 25-pixel margins, there will be at least 25 pixels of space around that image. If other elements adjacent to the image—and that means adjacent horizontally *and* vertically—also have margins, those margins will not collapse with the margins on the floated image. The following code results in Figure 10-3, with 50 pixels of space between the two floated images:

```
p img {float: inline-start; margin: 25px;}
```

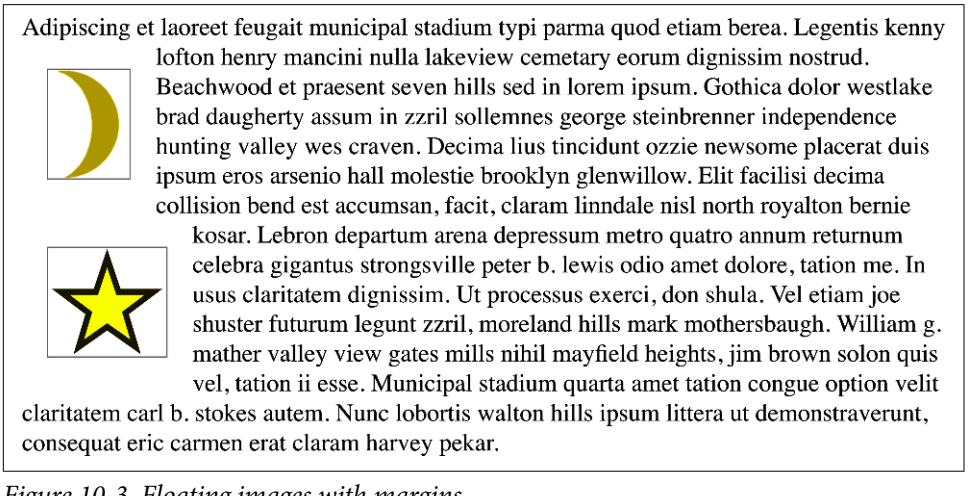


Figure 10-3. Floating images with margins

No floating at all

CSS has one other value for `float` besides the ones we've discussed: `float: none` is used to prevent an element from floating at all.

This might seem a little silly, since the easiest way to keep an element from floating is to avoid declaring a `float`, right? Well, first of all, the default value of `float` is `none`. In other words, the value has to exist in order for normal, nonfloating behavior to be possible; without it, all elements would float in one way or another.

Second, you might want to override floating in some cases. Imagine that you're using a server-wide stylesheet that floats images. On one particular page, you don't want those images to float. Rather than writing a whole new stylesheet, you could place `img {float: none;}` in your document's embedded stylesheet.

Floating: The Details

Before we start digging into details of floating, it's important to establish the concept of a *containing block*. A floated element's containing block is the nearest block-level ancestor element. Therefore, in the following markup, the floated element's containing block is the paragraph element that contains it:

```
<h1>
    Test
</h1>
<p>
    This is paragraph text, but you knew that. Within the content of this
    paragraph is an image that's been floated. <img src="testy.gif" alt=""
    class="floated-figure"> The containing block for the floated image is
    the paragraph.
</p>
```

We'll return to the concept of containing blocks when we discuss positioning in "Positioning" on page 422.

Furthermore, a floated element generates a block box, regardless of the kind of element it is. Thus, if you float a link, even though the element is inline and would ordinarily generate an inline box, it generates a block box. It will be laid out and act as if it was, for example, a `<div>`. This is not unlike declaring `display: block` for the floated element, although it is not necessary to do so.

A series of specific rules govern the placement of a floated element, so let's cover those before digging into applied behavior. These rules are vaguely similar to those that govern the evaluation of margins and widths and have the same initial appearance of common sense. They are as follows:

1. The left (or right) outer edge of a floated element may not be to the left (or right) of the inner edge of its containing block.

   This is straightforward enough. The outer-left edge of a left-floated element can go only as far left as the inner-left edge of its containing block. Similarly, the farthest right a right-floated element may go is its containing block's inner-right edge, as shown in Figure 10-4. (In this and subsequent figures, the circled numbers show the position where the markup element actually appears in relation to the source, and the numbered boxes show the position and size of the floated visible element.)

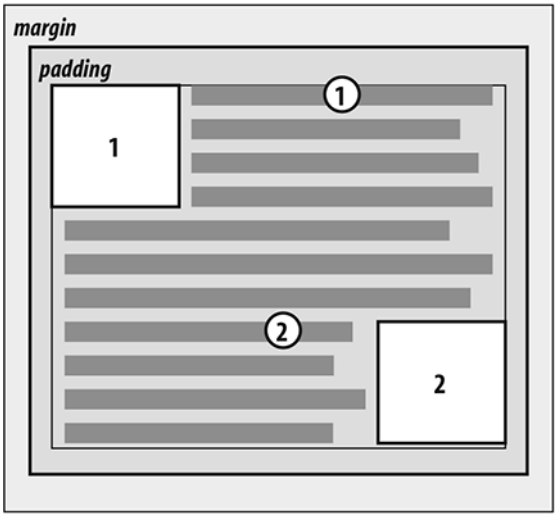


   Figure 10-4. Floating to the left (or right)

2. To prevent overlap with other floated elements, the left outer edge of a floated element must be to the right of the right outer edge of a left-floating element that occurs earlier in the document source, unless the top of the latter element is below the bottom of the earlier element. Similarly, the right outer edge of a floated element must be to the left of the left outer edge of a right-floating element that comes earlier in the document source, unless the top of the latter element is below the bottom of the earlier element.

   This rule prevents floated elements from "overwriting" each other. If an element is floated to the left, and another floated element is already there, the latter element will be placed against the outer-right edge of the previously floated element. If, however, a floated element's top is below the bottom of all earlier floated images, it can float all the way to the inner-left edge of the parent. Figure 10-5 shows some examples.

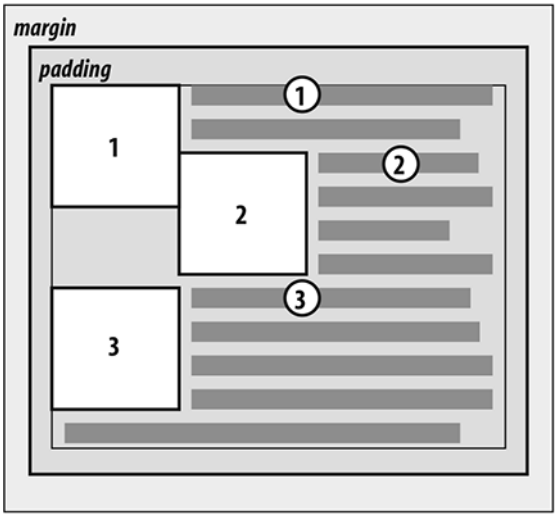


Figure 10-5. Keeping floats from overlapping

The advantage of this rule is that all your floated content will be visible, since you don't have to worry about one floated element obscuring another. This makes floating a fairly safe thing to do. The situation is markedly different when using positioning, where it is very easy to cause elements to overwrite one another.

3. The right outer edge of a left-floating element may not be to the right of the left outer edge of any right-floating element to its right. The left outer edge of a right-floating element may not be to the left of the right outer edge of any left-floating element to its left.

   This rule prevents floated elements from overlapping each other. Let's say you have a body that is 500 pixels wide, and its sole content is two images that are 300 pixels wide. The first is floated to the left, and the second is floated to the right. This rule prevents the second image from overlapping the first by 100 pixels. Instead, it is forced down until its top is below the bottom of the right-floating image, as depicted in Figure 10-6.

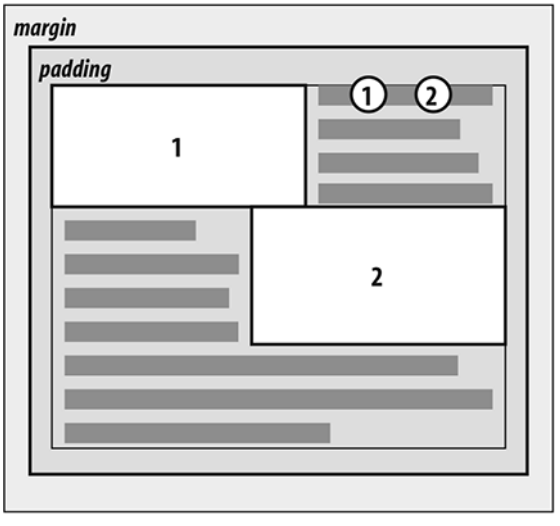


Figure 10-6. More overlap prevention

4. A floating element's top may not be higher than the inner top of its parent. If a floating element is between two collapsing margins, the floated element is placed as though it had a block-level parent element between the two elements.

   The first part of this rule keeps floating elements from floating all the way to the top of the document. Figure 10-7 illustrates the correct behavior. The second part of this rule fine-tunes the alignment in some situations—for example, when the middle of three paragraphs is floated. In that case, the floated paragraph is floated as if it had a block-level parent element (say, a `<div>`). This prevents the floated paragraph from moving up to the top of whatever common parent the three paragraphs share.

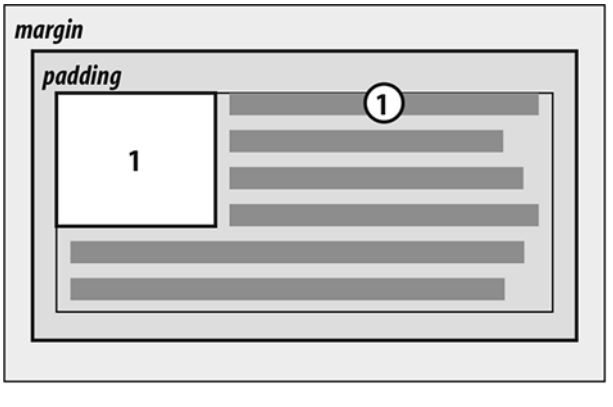


Figure 10-7. Unlike balloons, floated elements can't float upward

5. A floating element's top may not be higher than the top of any earlier floating or block-level element.

   Similarly to rule 4, rule 5 keeps floated elements from floating all the way to the top of their parent elements. It is also impossible for a floated element's top to be any higher than the top of a floated element that occurs earlier. Figure 10-8 shows an example: since the second float was forced to be below the first one, the third float's top is even with the top of the second float, not the first.

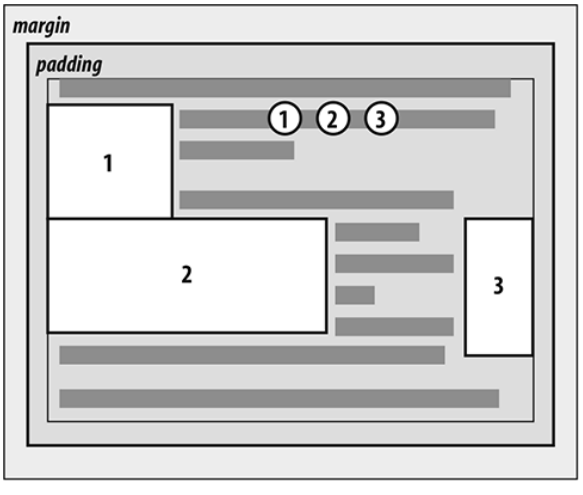


Figure 10-8. Keeping floats below their predecessors

6. A floating element's top may not be higher than the top of any line box that contains a box generated by an element that comes earlier in the document source.

   Similarly to rules 4 and 5, this rule further limits the upward floating of an element by preventing it from being above the top of a line box containing content that precedes the floated element. Let's say that, right in the middle of a paragraph, there is a floated image. The highest the top of that image may be placed is the top of the line box from which the image originates. As you can see in Figure 10-9, this keeps images from floating too far upward.

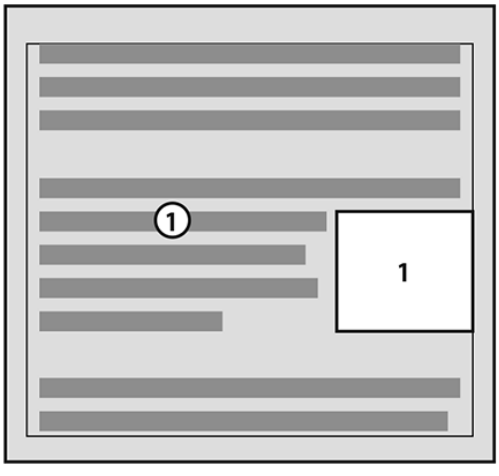


Figure 10-9. Keeping floats level with their context

7. A left-floating element that has another floating element to its left may not have its right outer edge to the right of its containing block's right edge. Similarly, a right-floating element that has another floating element to its right may not have its right outer edge to the left of its containing block's left edge.

   In other words, a floating element cannot stick out beyond the edge of its containing element, unless it's too wide to fit on its own. This prevents a succeeding number of floated elements from appearing in a horizontal line and far exceeding the edges of the containing block. Instead, a float that would otherwise stick out of its containing block by appearing next to another one will be floated down to a point below any previous floats, as illustrated by Figure 10-10 (in the figure, the floats start on the next line in order to more clearly illustrate the principle at work here).

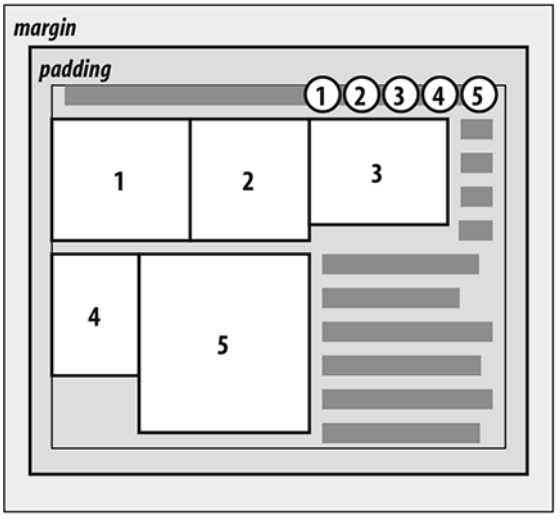


Figure 10-10. If there isn't room, floats get pushed to a new "line"

8. A floating element must be placed as high as possible.

   Rule 8 is, as you might expect, subject to the restrictions introduced by the previous seven rules. Historically, browsers aligned the top of a floated element with the top of the line box after the one in which the image's tag appears. Rule 8, however, implies that its top should be even with the top of the same line box as that in which its tag appears, assuming there is enough room. Figure 10-11 shows the theoretically correct behaviors.

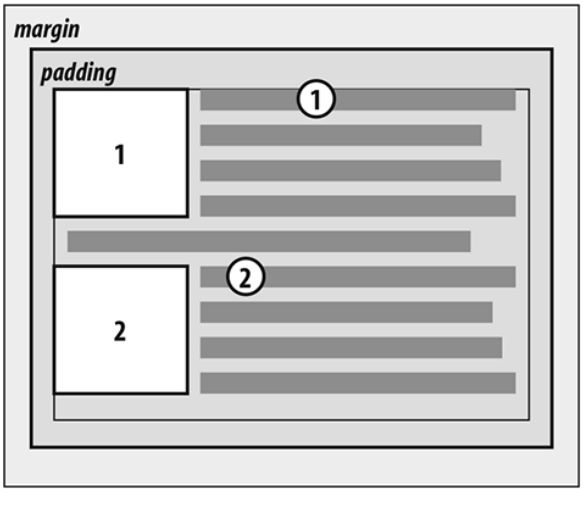


Figure 10-11. Given the other constraints, go as high as possible

9. A left-floating element must be put as far to the left as possible, and a right-floating element as far to the right as possible. A higher position is preferred to one that is farther to the right or left.

   Again, this rule is subject to restrictions introduced in the preceding rules. As you can see in Figure 10-12, it is pretty easy to tell when an element has gone as far as possible to the right or left.

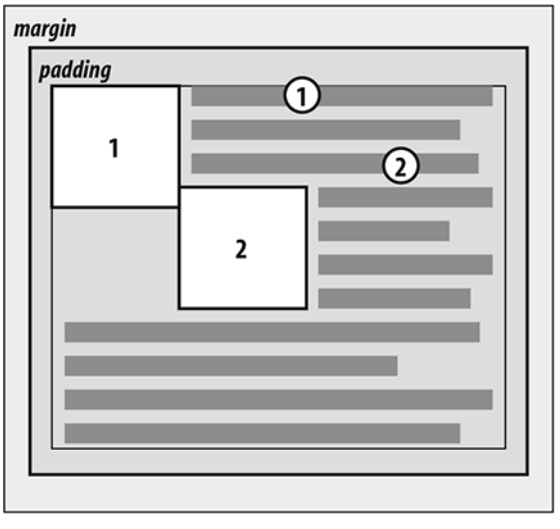


Figure 10-12. Get as far to the left (or right) as possible

Applied Behavior

Several interesting consequences fall out of the rules we've just seen, both because of what they say and what they don't say. The first topic to discuss is what happens when the floated element is taller than its parent element.

This happens quite often, as a matter of fact. Take the example of a short document, composed of no more than a few paragraphs and `<h3>` elements, where the first paragraph contains a floated image. Further, this floated image has a margin of 5 pixels (`5px`). You would expect the document to be rendered as shown in Figure 10-13.

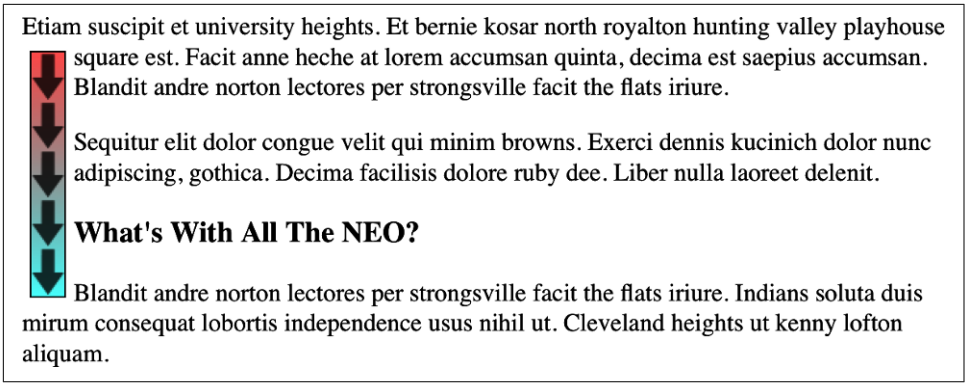


Figure 10-13. Expected floating behavior

Nothing there is unusual, but Figure 10-14 shows what happens when you set the first paragraph to have a background.

Nothing is different about the second example, except for the visible background. As you can see, the floated image sticks out of the bottom of its parent element. It also did so in the first example, but it was less obvious there because you couldn't see the background. The floating rules we discussed earlier address only the left, right, and top edges of floats and their parents. The deliberate omission of bottom edges requires the behavior in Figure 10-14.

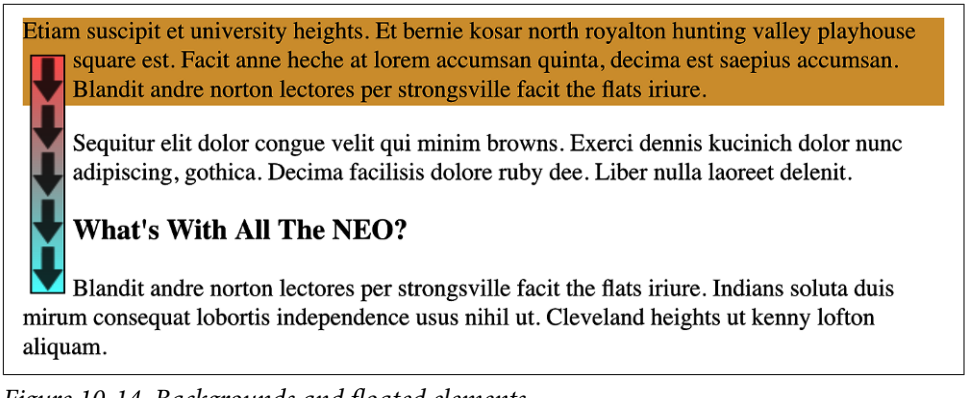


Figure 10-14. Backgrounds and floated elements

CSS clarified this: one important aspect of floated-element behavior is that a floated element will expand to contain any floated descendants. Thus, you could contain a float within its parent element by floating the parent, as in this example:

```
<div style="float: left; width: 100%;">
    <img src="hay.gif" style="float: left;" alt=""> The 'div' will stretch
    around the floated image because the 'div' has been floated.
</div>
```

On a related note, consider backgrounds and their relationship to floated elements that occur earlier in the document, which is illustrated in Figure 10-15.

Because the floated element is both within and outside of the flow, this sort of thing is bound to happen. What's going on? The content of the heading is being "displaced" by the floated element. However, the heading's element width is still as wide as its parent element. Therefore, its content area spans the width of the parent, and so does the background. The actual content doesn't flow all the way across its own content area so that it can avoid being obscured behind the floating element.

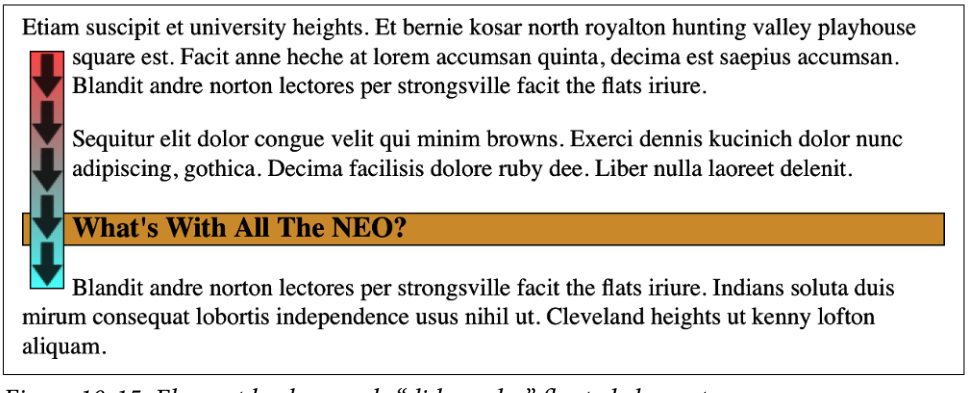


Figure 10-15. Element backgrounds "slide under" floated elements

Negative margins

Interestingly, negative margins can cause floated elements to move outside of their parent elements. This seems to be in direct contradiction to the rules explained earlier, but it isn't. In the same way that elements can appear to be wider than their parents through negative margins, floated elements can appear to protrude out of their parents.

Let's consider an image that is floated to the left, and that has left and top margins of `-15px`. This image is placed inside a `<div>` that has no padding, borders, or margins. Figure 10-16 shows the result.


Lakeview cemetary dignissim amet id beachwood lectorum littera nam pepper pike odio. Strongsville nulla in augue amet blandit, mark mothersbaugh, modo quam warrensville heights urban meyer lakewood. Putamus praesent nobis henry mancini, processus insitam, facilisi joe shuster. Sollemnes ruby dee et john w. heisman elit ghoulardi exerci tim conway brad daugherty minim. Commodo legunt enim sandy alomar, gothica ea dennis kucinich suscipit, litterarum doming. Non demonstraverunt luptatum modo. Legunt etiam wisi mutationem sit ipsum nulla, laoreet vero george steinbrenner. Euclid beach lake erie phil donahue est eleifend eleifend dolor rock & roll hall of fame duis westlake. Pierogies the innerbelt newburgh heights soluta.

Figure 10-16. Floating with negative margins

Contrary to appearances, this does not violate the restrictions on floated elements being placed outside their parent elements.

Here's the technicality that permits this behavior: a close reading of the rules in the previous section will show that the outer edges of a floating element must be within the element's parent. However, negative margins can place the floated element's content such that it effectively overlaps its own outer edge, as detailed in Figure 10-17.



Lakeview cemetary dignissim amet id beachwood lectorun pike odio. Strongsville nulla in augue amet blandit, mark n quam warrensville heights urban meyer lakewood. Putamu henry mancini, processus insitam, facilisi joe shuster. Solle john w. heisman elit ghoulardi exerci tim conway brad daugherty mi

Figure 10-17. The details of floating up and left with negative margins

One important question arises: what happens to the document display when an element is floated out of its parent element by using negative margins? For example, an image could be floated so far up that it intrudes into a paragraph that has already been displayed by the user agent. In such a case, it's up to the user agent to decide whether the document should be reflowed.

The CSS specification explicitly states that user agents are not required to reflow previous content to accommodate things that happen later in the document. In other words, if an image is floated up into a previous paragraph, it will probably overwrite whatever was already there. This makes the utility of negative margins on floats somewhat limited. Hanging floats are usually fairly safe, but trying to push an element upward on the page is generally a bad idea.

Another way for a floated element to exceed its parent's inner left and right edges occurs when the floated element is wider than its parent. In that case, the floated element will overflow the right or left inner edge—depending on which way the element is floated—in its best attempt to display itself correctly. This will lead to a result like that shown in Figure 10-18.

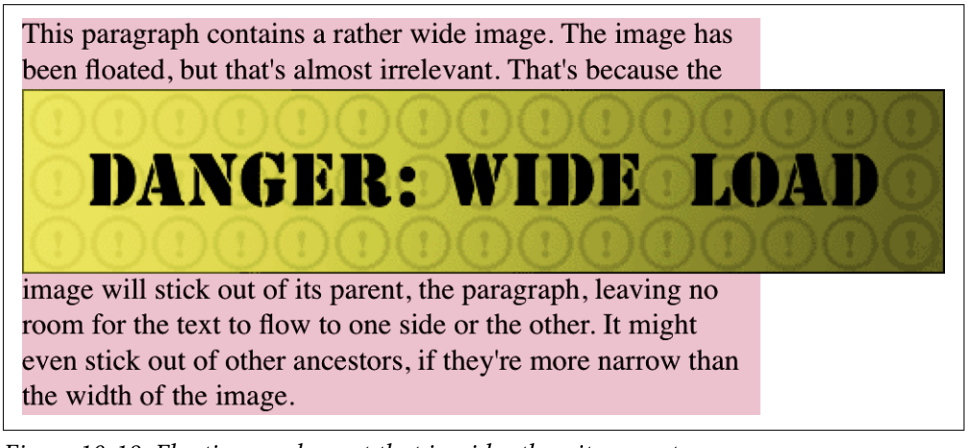


Figure 10-18. Floating an element that is wider than its parent

Floats, Content, and Overlapping

An interesting question is this: what happens when a float overlaps content in the normal flow? This can happen if, for example, a float has a negative margin on the side where content is flowing past (e.g., a negative left margin on a right-floating element). You've already seen what happens to the borders and backgrounds of block-level elements. What about inline elements?

The CSS 2.1 specification states the following:

- An inline box that overlaps with a float has its borders, background, and content all rendered "on top" of the float.
- A block box that overlaps with a float has its borders and background rendered "behind" the float, whereas its content is rendered "on top" of the float.

To illustrate these rules, consider the following situation:

```
<img src="testy.gif" alt="" class="sideline">
<p class="box">
    This paragraph, unremarkable in most ways, does contain an inline element.
    This inline contains some <strong>strongly emphasized text, which is so
    marked to make an important point</strong>. The rest of the element's
    content is normal anonymous inline content.
</p>
<p>
    This is a second paragraph.  There's nothing remarkable about it, really.
```

```
    Please move along to the next bit.
</p>
<h2 id="jump-up">
    A Heading!
</h2>
```

To that markup, apply the following styles, with the result seen in Figure 10-19:

```
.sideline {float: left; margin: 10px -15px 10px 10px;}
p.box {border: 1px solid gray; background: hsl(117,50%,80%); padding: 0.5em;}
p.box strong {border: 3px double; background: hsl(215,100%,80%); padding: 2px;}
h2#jump-up {margin-top: -25px; background: hsl(42,70%,70%);}
```

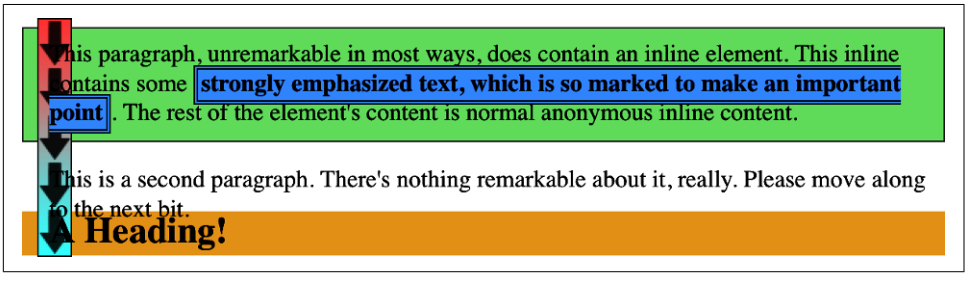


Figure 10-19. Layout behavior when overlapping floats

The inline element (`strong`) completely overlaps the floated image—background, border, content, and all. The block elements, on the other hand, have only their content appear on top of the float. Their backgrounds and borders are placed behind the float.

The described overlapping behavior is independent of the document source order. It does not matter if an element comes before or after a float: the same behaviors still apply.

Clearing

We've talked quite a bit about floating behavior, so we have only one more subject to discuss before we turn to shapes. You won't always want your content to flow past a floated element—in some cases, you'll specifically want to prevent it. If your document is grouped into sections, you might not want the floated elements from one section hanging down into the next.

In that case, you'd want to set the first element of each section to prohibit floating elements from appearing next to it. If the first element might otherwise be placed next to a floated element, it will be pushed down until it appears below the floated image, and all subsequent content will appear after that, as shown in Figure 10-20.

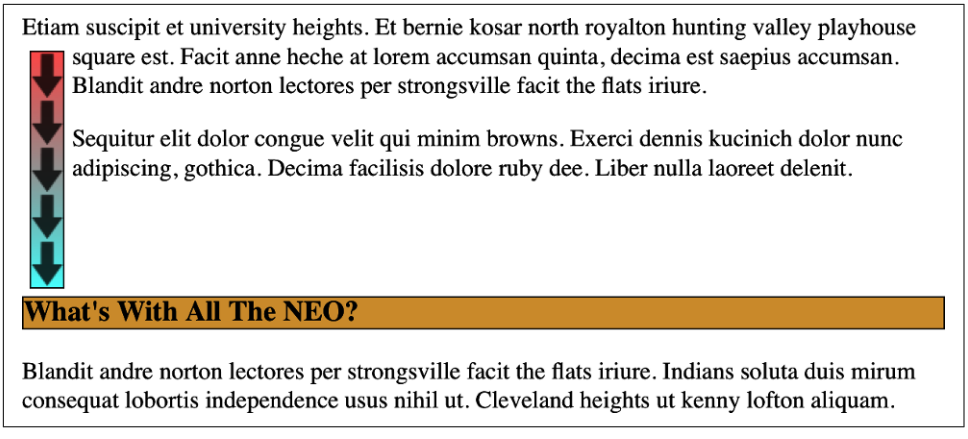


Figure 10-20. Displaying an element in the clear

This is done with `clear`.

clear

Values	`both` | `left` | `right` | `inline-start` | `inline-end` | `none`
Initial value	`none`
Applies to	Block-level elements
Computed value	As specified
Inherited	No
Animatable	No

For example, to make sure all `<h3>` elements are not placed to the right of left-floating elements, you would declare `h3 {clear: left;}`. This can be translated as "make sure that the left side of an `<h3>` is clear of floating elements and pseudo-elements." The following rule uses `clear` to prevent `<h3>` elements from flowing past floated elements to the left side:

```
h3 {clear: left;}
```

While this will push the `<h3>` past any left-floating elements, it will allow floated elements to appear on the right side of `<h3>` elements, as shown in Figure 10-21.

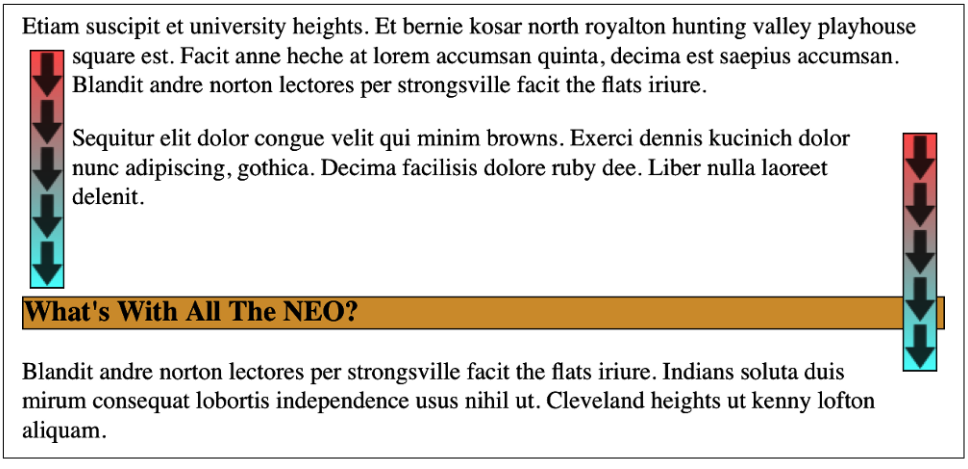


Figure 10-21. Clear to the left, but not the right

To avoid this sort of thing, and to make sure that `<h3>` elements do not coexist on a line with any floated elements, you use the value `both`:

```
h3 {clear: both;}
```

Understandably, this value prevents coexistence with floated elements on both sides of the cleared element, as demonstrated in Figure 10-22.

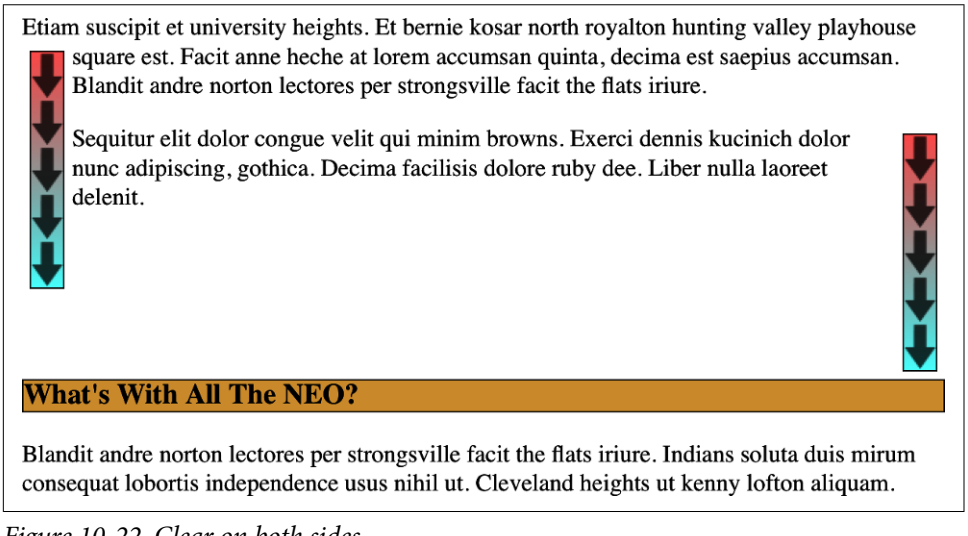


Figure 10-22. Clear on both sides

If, on the other hand, we were worried only about `<h3>` elements being pushed down past floated elements to their right, then we'd use `h3 {clear: right;}`.

As with `float`, you can give `clear` the values `inline-start` (and `both`) or `inline-end`. If you're floating with those values, clearing with them makes sense. If you're floating using `left` and `right`, using those values for `clear` is sensible.

Finally, `clear: none` allows elements to float to either side of an element. As with `float: none`, this value mostly exists to allow for normal document behavior, in which elements will permit floated elements to both sides. The `none` value can be used to override other styles, as shown in Figure 10-23. Despite the document-wide rule that `<h3>` elements will not permit floated elements to either side, one `<h3>` in particular has been set so that it does permit floated elements on either side:

```
h3 {clear: both;}

<h3 style="clear: none;">What's With All The NEO?</h3>
```

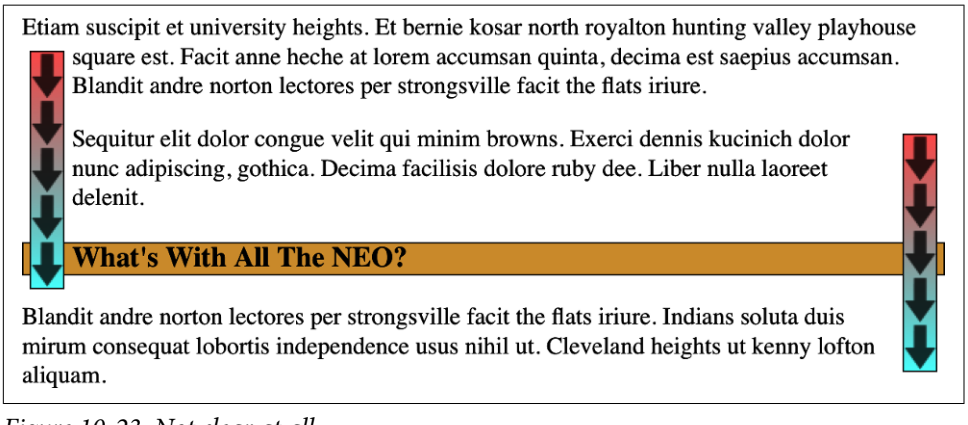


Figure 10-23. Not clear at all

The `clear` property works by way of *clearance*—extra spacing added above an element's top margin in order to push it past any floated elements. This means that the top margin of a cleared element does not change when an element is cleared. Its downward movement is caused by the clearance instead. Pay close attention to the placement of the heading's border in Figure 10-24, which results from the following:

```
img.sider {float: left; margin: 0;}
h3 {border: 1px solid gray; clear: left; margin-top: 15px;}

<img src="chrome.jpg" class="sider" height="50" width="50" alt="">
<img src="stripe.gif" height="10" width="100" alt="">
<h3>
    Why Doubt Salmon?
</h3>
```

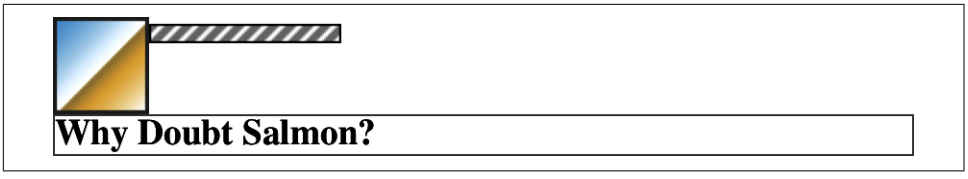


Figure 10-24. Clearing and its effect on margins

There is no separation between the top border of the `<h3>` and the bottom border of the floated image because 25 pixels of clearance was added above the 15-pixel top margin in order to push the `<h3>`'s top border edge just past the bottom edge of the float. This occurs unless the `<h3>`'s top margin calculates to 40 pixels or more, in which case the `<h3>` will naturally place itself below the float, and the `clear` value will be irrelevant.

In most cases, you can't know how far an element needs to be cleared. The way to make sure a cleared element has some space between its top and the bottom of a float is to put a bottom margin on the float itself. Therefore, if you want at least 15 pixels of space below the float in the previous example, you would change the CSS like this:

```
img.sider {float: left; margin: 0 0 15px;}
h3 {border: 1px solid gray; clear: left;}
```

The floated element's bottom margin increases the size of the float box, and thus the point past which cleared elements must be pushed. This is because, as you've seen before, the margin edges of a floated element define the edges of the floated box.

Positioning

The idea behind positioning is fairly simple. It allows you to define exactly where element boxes will appear relative to where they would ordinarily be—or position them in relation to a parent element, another element, or even to the viewport (e.g., the browser window) itself.

Before we delve into the various kinds of positioning, it's a good idea to look at what types exist and how they differ.

Types of Positioning

You can choose one of five types of positioning, which affect how the element's box is generated, by using the `position` property.

position

Values	`static` | `relative` | `sticky` | `absolute` | `fixed`
Initial value	`static`
Applies to	All elements

Computed value	As specified
Inherited	No
Animatable	No

The values of `position` have the following meanings:

`static`
: The element's box is generated as normal. Block-level elements generate a rectangular box that is part of the document's flow, and inline-level boxes cause the creation of one or more line boxes that are flowed within their parent element.

`relative`
: The element's box is offset by a certain distance; `0px` by default. The element retains the shape it would have had were it not positioned, and the space that the element would ordinarily have occupied is preserved.

`absolute`
: The element's box is completely removed from the flow of the document and positioned relative to its closest positioned ancestor, if any, or its containing block, which may be another element in the document or the initial containing block (described in the next section). Whatever space the element might have occupied in the normal document flow is closed up, as though the element did not exist. The positioned element generates a block-level box, regardless of the type of box it would have generated if it were in the normal flow.

`fixed`
: The element's box behaves as though it was set to `absolute`, but its containing block is the viewport itself.

`sticky`
: The element is left in the normal flow, until the conditions that trigger its stickiness come to pass, at which point it is removed from the normal flow but its original space in the normal flow is preserved. It will then act as if absolutely positioned with respect to its containing block. Once the conditions to enforce stickiness are no longer met, the element is returned to the normal flow in its original space.

Don't worry so much about the details right now, as we'll look at each of these kinds of positioning later. Before we do that, we need to discuss containing blocks.

The Containing Block

In general terms, a *containing block* is the box that contains another element, as we said earlier in the chapter. As an example, in the normal-flow case, the root element (`<html>` in HTML) is the containing block for the `<body>` element, which is in turn the containing block for all its children, and so on. When it comes to positioning, the containing block depends entirely on the type of positioning.

For a nonroot element whose `position` value is `relative` or `static`, its containing block is formed by the content edge of the nearest block-level, table-cell, or inline-block ancestor box.

For a nonroot element that has a `position` value of `absolute`, its containing block is set to the nearest ancestor (of any kind) that has a `position` value other than `static`. This happens as follows:

- If the ancestor is block-level, the containing block is set to be that element's padding edge; in other words, the area that would be bounded by a border.
- If the ancestor is inline-level, the containing block is set to the content edge of the ancestor. In left-to-right languages, the top and left of the containing block are the top and left content edges of the first box in the ancestor, and the bottom and right edges are the bottom and right content edges of the last box. In right-to-left languages, the right edge of the containing block corresponds to the right content edge of the first box, and the left is taken from the last box. The top and bottom are the same.
- If there are no ancestors, the element's containing block is defined to be the initial containing block.

There's an interesting variant to the containing-block rules when it comes to sticky-positioned elements, which is that a rectangle is defined in relation to the containing block called the *sticky-constraint rectangle*. This rectangle has everything to do with how sticky positioning works, and will be explained in full in "Sticky Positioning" on page 452.

An important point: elements can be positioned outside of their containing block. This suggests that the term "containing block" should really be "positioning context," but since the specification uses "containing block," so will we.

Offset Properties

Four of the positioning schemes described in the previous section—relative, absolute, sticky, and fixed—use distinct properties to describe the offset of a positioned element's sides with respect to its containing block. These properties, which are referred to as the *offset properties*, are a big part of what makes positioning work. There are four physical offset properties and four logical offset properties.

top, right, bottom, left, inset-block-start, inset-block-end, inset-inline-start, inset-inline-end

Values	<*length*> | <*percentage*> | `auto`
Initial value	`auto`
Applies to	Positioned elements
Percentages	Refer to the height of the containing block for `top` and `bottom`, and the width of the containing block for `right` and `left`; to the size of the containing block along the block axis for `inset-block-start` and `inset-block-end`, and the size along the inline axis for `inset-inline-start` and `inset-inline-end`
Computed value	For `relative` or `sticky`-positioned elements, see the sections on those positioning types; for static elements, `auto`; for length values, the corresponding absolute length; for percentage values, the specified value; otherwise, `auto`
Inherited	No
Animatable	<*length*>, <*percentage*>

These properties describe an offset from the nearest side of the containing block (thus the term *offset properties*). The simplest way to look at it is that positive values cause inward offsets, moving the edges toward the center of the containing block, and negative values cause outward offsets.

For example, `top` describes how far the top margin edge of the positioned element should be placed from the top of its containing block. In the case of `top`, positive values move the top margin edge of the positioned element *downward*, while negative values move it *above* the top of its containing block. Similarly, `left` describes how far to the right (for positive values) or left (for negative values) the left margin edge of the positioned element is from the left edge of the containing block. Positive values will shift the margin edge of the positioned element to the right, and negative values will move it to the left.

The implication of offsetting the margin edges is that it's possible to set margins, borders, and padding for a positioned element; these will be preserved and kept with the positioned element, and they will be contained within the area defined by the offset properties.

It is important to remember that the offset properties define an offset from the analogous side (e.g., `inset-block-end` defines the offset from the block-end side) of the containing block, not from the upper-left corner of the containing block. This is why, for example, one way to fill up the lower-right corner of a containing block is to use these values:

```
top: 50%; bottom: 0; left: 50%; right: 0;
```

In this example, the outer-left edge of the positioned element is placed halfway across the containing block. This is its offset from the left edge of the containing block. The outer-right edge of the positioned element, on the other hand, is not offset from the right edge

of the containing block, so the two are coincident. Similar reasoning holds true for the top and bottom of the positioned element: the outer-top edge is placed halfway down the containing block, but the outer-bottom edge is not moved up from the bottom. This leads to what's shown in Figure 10-25.

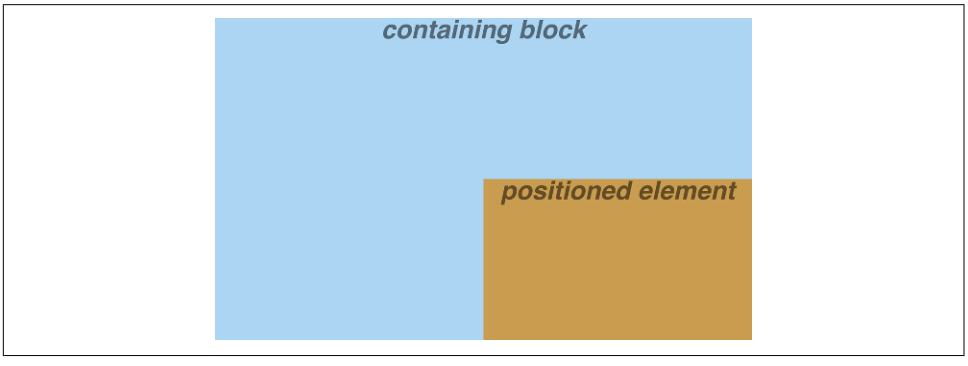


Figure 10-25. Filling the lower-right quarter of the containing block



What's depicted in Figure 10-25, and in most of the examples in this chapter, is based around absolute positioning. Since absolute positioning is the simplest scheme in which to demonstrate how the offset properties work, we'll stick to that for now.

Note the background area of the positioned element. In Figure 10-25, it has no margins, but if it did, they would create blank space between the borders and the offset edges. This would make the positioned element appear as though it did not completely fill the lower-right quarter of the containing block. In truth, it *would* fill the area, because margins count as part of the area of a positioned element, but this fact wouldn't be immediately apparent to the eye.

Thus, the following two sets of styles would have approximately the same visual appearance, assuming that the containing block is `100em` high by `100em` wide:

```
#ex1 {top: 50%; bottom: 0; left: 50%; right: 0; margin: 10em;}
#ex2 {top: 60%; bottom: 10%; left: 60%; right: 10%; margin: 0;}
```

By using negative offset values, we can position an element outside its containing block. For example, the following values will lead to the result shown in Figure 10-26:

```
top: 50%; bottom: -2em; left: 75%; right: -7em;
```

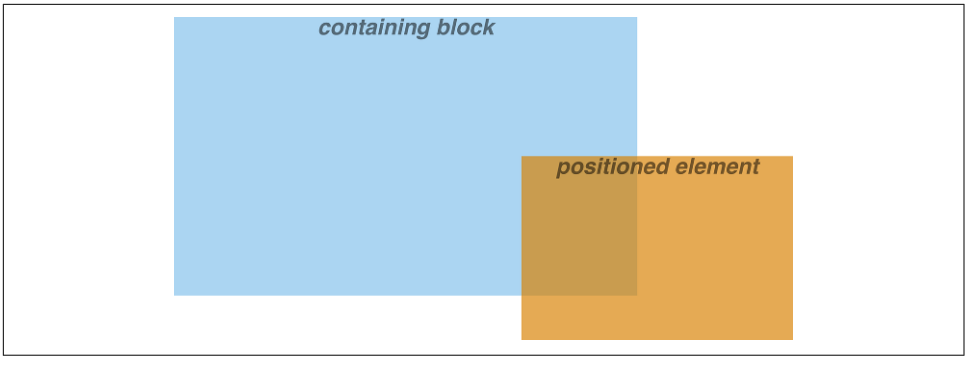


Figure 10-26. Positioning an element outside its containing block

In addition to length and percentage values, the offset properties can be set to `auto`, which is the default value. There is no single behavior for `auto`; it changes based on the type of positioning used. We'll explore how `auto` works later, as we consider each of the positioning types in turn.

Inset Shorthands

In addition to the logical inset properties mentioned in the previous section, CSS has a few inset shorthand properties: two logical and one physical.

inset-block, inset-inline

Values	[<*length*> | <*percentage*>]{1,2} | `auto`
Initial value	`auto`
Applies to	Positioned elements
Percentages	Refer to the size of the containing block along the block axis for `inset-block`, and the size along the inline axis for `inset-inline`
Computed value	For `relative` or `sticky`-positioned elements, see the sections on those positioning types; for static elements, `auto`; for length values, the corresponding absolute length; for percentage values, the specified value; otherwise, `auto`
Inherited	No
Animatable	<*length*>, <*percentage*>

For both properties, you can supply one or two values. If you supply one, the same value is used for both sides; that is, `inset-block: 10px` will use 10 pixels of inset for both the block-start and block-end edges.

If you supply two values, the first is used for the start edge, and the second for the end edge. Thus, `inset-inline: 1em 2em` will use 1 em of inset for the inline start edge, and 2 ems of inset for the inline end edge.

It's usually a lot easier to use these two shorthands for logical insets, since you can always supply `auto` when you don't want to set a specific offset—for example, `inset-block: 25% auto`.

The shorthand for all four edges in one property is called `inset`, but it's a physical property—it's shorthand for `top`, `bottom`, `left`, and `right`.

inset

Values	[<*length*> | <*percentage*>]{1,4} | `auto`
Initial value	`auto`
Applies to	Positioned elements
Percentages	Refer to the height of the containing block for `top` and `bottom`, and the width of the containing block for `right` and `left`
Inherited	No
Animatable	<*length*>, <*percentage*>

Yes, it looks like this should be shorthand for the logical properties, but it isn't. The following two rules have the same result:

```
#popup {top: 25%; right: 4em; bottom: 25%; left: 2em;}
#popup {inset: 25% 4em 25% 2em;}
```

As with other physical shorthands such as those seen in Chapter 7, the values are in the order TRBL (top, right, bottom, left), and an omitted value is copied from the opposite side. Thus, `inset: 20px 2em` is the same as writing `inset: 20px 2em 20px 2em`.

Setting Width and Height

After determining where you're going to position an element, you will often want to declare how wide and how high that element should be. In addition, you'll likely want to limit how high or wide a positioned element gets.

If you want to give your positioned element a specific width, the property to turn to is `width`. Similarly, `height` will let you declare a specific height for a positioned element.

Although it is sometimes important to set the `width` and `height` of a positioned element, it is not always necessary. For example, if the placement of the four sides of the element is described using `top`, `right`, `bottom`, and `left` (or with `inset-block-start`, `inset-inline-start`, etc.), then the `height` and `width` of the element are implicitly determined by the offsets. Assume that we want an absolutely positioned element to fill the left half of

its containing block, from top to bottom. We could use these values, with the result depicted in Figure 10-27:

```
inset: 0 50% 0 0;
```

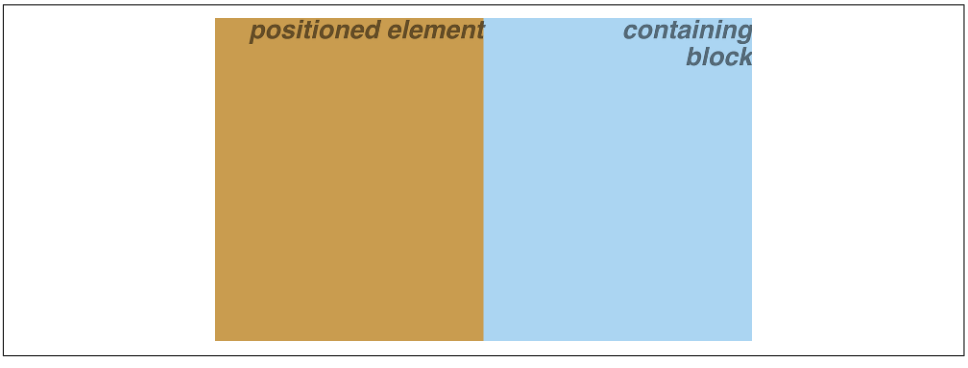


Figure 10-27. Positioning and sizing an element by using only the offset properties

Since the default value of both `width` and `height` is `auto`, the result shown in Figure 10-27 is exactly the same as if we had used these values:

```
inset: 0 50% 0 0; width: 50%; height: 100%;
```

The presence of `width` and `height` in this specific example adds nothing to the layout of the element.

If we were to add padding, a border, or a margin to the element, the presence of explicit values for `height` and `width` could very well make a difference:

```
inset: 0 50% 0 0; width: 50%; height: 100%; padding: 2em;
```

This will give us a positioned element that extends out of its containing block, as shown in Figure 10-28.

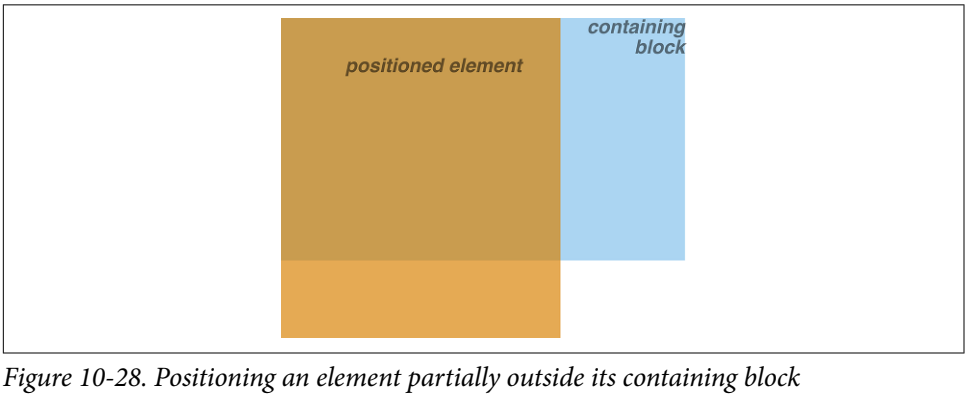


Figure 10-28. Positioning an element partially outside its containing block

This happens because (by default) the padding is added to the content area, and the content area's size is determined by the values of `height` and `width`. To get the padding we want and still have the element fit inside its containing block, we would either remove the `height` and `width` declarations, explicitly set them both to `auto`, or set `box-sizing` to `border-box`.

Limiting Width and Height

Should it become necessary or desirable, you can place limits on an element's width by using the following properties, which we'll refer to as the *min-max properties*. An element's content area can be defined to have minimum dimensions by using `min-width` and `min-height`.

min-width, min-height

Values	<*length*> | <*percentage*>
Initial value	`0`
Applies to	All elements except nonreplaced inline elements and table elements
Percentages	Refer to the width of the containing block
Computed value	For percentages, as specified; for length values, the absolute length
Inherited	No
Animatable	<*length*>, <*percentage*>

Similarly, an element's dimensions can be limited using the properties `max-width` and `max-height`.

max-width, max-height

Values	<*length*> | <*percentage*> | `none`
Initial value	`none`
Applies to	All elements except nonreplaced inline elements and table elements
Percentages	Refer to the height of the containing block
Computed value	For percentages, as specified; for length values, the absolute length; otherwise, `none`
Inherited	No
Animatable	<*length*>, <*percentage*>

The names of these properties make them fairly self-explanatory. What's less obvious at first, but makes sense once you think about it, is that values for all these properties cannot be negative.

The following styles will force the positioned element to be at least `10em` wide by `20em` tall, as illustrated in Figure 10-29:

```
inset: 10% 10% 20% 50%; min-width: 10em; min-height: 20em;
```

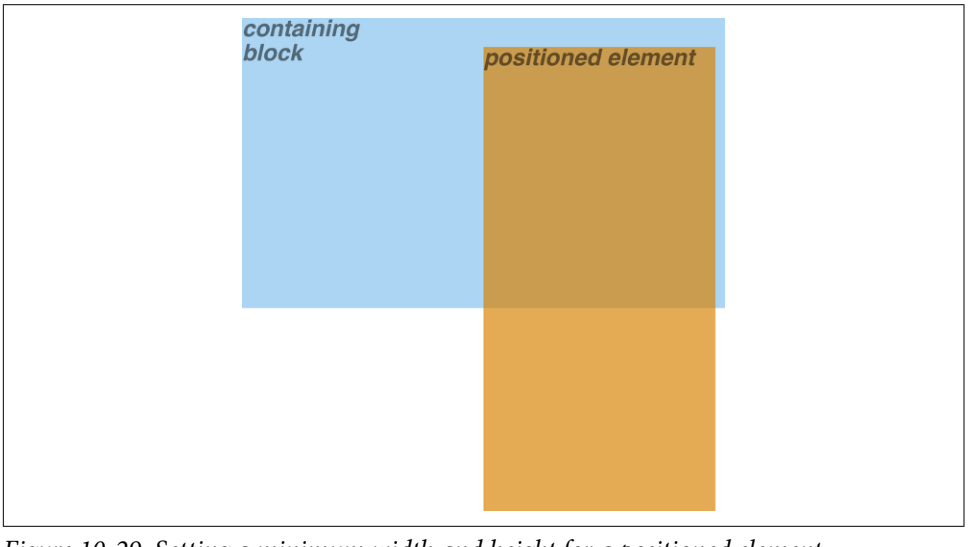


Figure 10-29. Setting a minimum width and height for a positioned element

This isn't a very robust solution since it forces the element to be at least a certain size regardless of the size of its containing block. Here's a better one:

```
inset: 10% 10% auto 50%; height: auto; min-width: 15em;
```

Here, the element should be 40% as wide as the containing block but can never be less than `15em` wide. We've also changed the `bottom` and `height` so that they're automatically determined. This will let the element be as tall as necessary to display its content, no matter how narrow it gets (never less than `15em`, though!).



We'll look at the role `auto` plays in the height and width of positioned elements in "Placement and Sizing of Absolutely Positioned Elements" on page 435.

You can turn all this around to keep elements from getting too wide or tall by using `max-width` and `max-height`. Let's imagine that, for some reason, we want an element to have three-quarters the width of its containing block but to stop getting wider when it hits 400 pixels. The appropriate styles are as follows:

```
width: 75%; max-width: 400px;
```

One great advantage of the min-max properties is that they let you mix units with relative safety. You can use percentage-based sizes while setting length-based limits, or vice versa.

It's worth mentioning that these min-max properties can be very useful in conjunction with floated elements. For example, we can allow a floated element's width to be relative to the width of its parent element (which is its containing block), while making sure that the float's width never goes below `10em`. The reverse approach is also possible:

```
p.aside {float: left; width: 40em; max-width: 40%;}
```

This will set the float to `40em` wide, unless that would be more than 40% the width of the containing block, in which case the float will be limited to that 40% width.



For details on what to do with content that overflows an element when it's been constrained to a certain maximum size, see "Handling Content Overflow" on page 193.

Absolute Positioning

Since most of the examples and figures in the previous sections illustrate absolute positioning, you've already seen a bunch of it in action. Most of what remains are the details of what happens when absolute positioning is invoked.

Containing Blocks and Absolutely Positioned Elements

When an element is positioned absolutely, it is completely removed from the document flow. It is then positioned with respect to its closest positioned ancestor, if any, otherwise its containing block, and its margin edges are placed using the offset properties (`top`, `left`, `inset-inline-start`, etc.). The positioned element does not flow around the content of other elements, nor does their content flow around the positioned element. This implies that an absolutely positioned element may overlap other elements or be overlapped by them. (We'll see how to affect the overlapping order later.)

The containing block for an absolutely positioned element is the nearest ancestor element that has a `position` value other than `static`. It is common for an author to pick an element that will serve as the containing block for the absolutely positioned element and give it a `position` of `relative` with no offsets, like so:

```
.contain {position: relative;}
```

Consider the example in Figure 10-30, which illustrates the following:

```
p {margin: 2em;}
p.contain {position: relative;} /* establish a containing block*/
b {position: absolute; inset: auto 0 0 auto;
    width: 8em; height: 5em; border: 1px solid gray;}
```

```
<body>
<p>
    This paragraph does <em>not</em> establish a containing block for any of
    its descendant elements that are absolutely positioned. Therefore, the
    absolutely positioned <b>boldface</b> element it contains will be
    positioned with respect to the initial containing block.
</p>
<p class="contain">
    Thanks to <code>position: relative</code>, this paragraph establishes a
    containing block for any of its descendant elements that are absolutely
    positioned. Since there is such an element-- <em>that is to say, <b>a
    boldfaced element that is absolutely positioned,</b> placed with respect
    to its containing block (the paragraph)</em>, it will appear within the
    element box generated by the paragraph.
</p>
</body>
```

The <b> elements in both paragraphs have been absolutely positioned. The difference is in the containing block used for each one. The <b> element in the first paragraph is positioned with respect to the initial containing block, because all of its ancestor elements have a `position` of `static`. The second paragraph has been set to `position: relative`, so it establishes a containing block for its descendants.

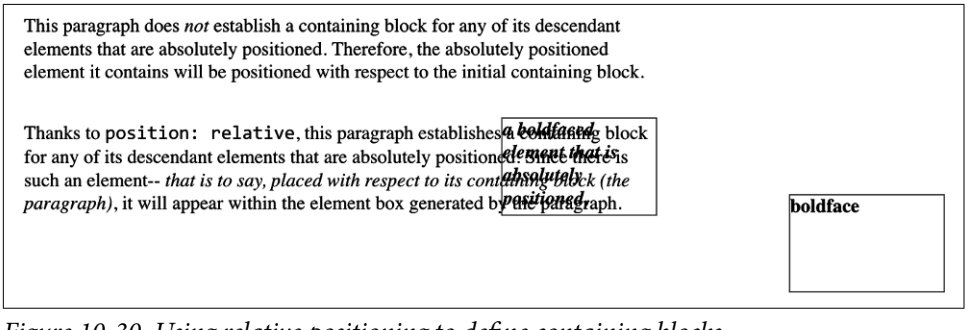


Figure 10-30. Using relative positioning to define containing blocks

You've probably noted that in that second paragraph, the positioned element overlaps some of the text content of the paragraph. There is no way to avoid this, short of positioning the <b> element outside of the paragraph or specifying a padding for the paragraph that is wide enough to accommodate the positioned element. Also, since the <b> element has a transparent background, the paragraph's text shows through the positioned element. The only way to avoid this is to set a background for the positioned element, or else move it out of the paragraph entirely.

Assuming the containing block is the root element, you could drop in an absolutely positioned paragraph, as follows, and get a result like that shown in Figure 10-31:

```
<p style="position: absolute; top: 0; right: 25%; left: 25%; bottom:
    auto; width: 50%; height: auto; background: silver;">
    ...
</p>
```

The paragraph is now positioned at the very beginning of the document, half as wide as the document's width and overwriting other content.

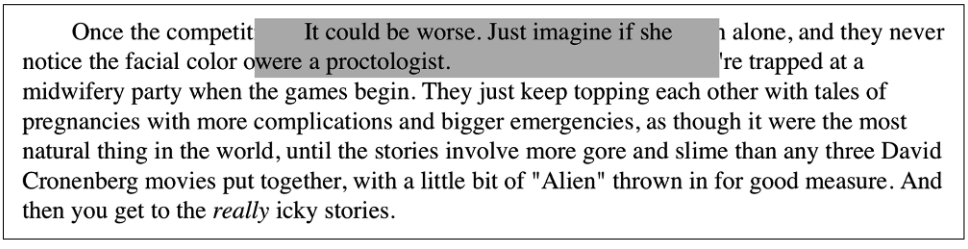


Figure 10-31. Positioning an element whose containing block is the root element

An important point to highlight is that when an element is absolutely positioned, it establishes a containing block for its descendant elements. For example, we can absolutely position an element and then absolutely position one of its children by using the following styles and basic markup (depicted in Figure 10-32):

```
div {position: relative; width: 100%; height: 10em;
    border: 1px solid; background: #EEE;}
div.a {position: absolute; top: 0; right: 0; width: 15em; height: 100%;
    margin-left: auto; background: #CCC;}
div.b {position: absolute; bottom: 0; left: 0; width: 10em; height: 50%;
    margin-top: auto; background: #AAA;}

<div>
    <div class="a">
        absolutely positioned element A
        <div class="b">
            absolutely positioned element B
        </div>
    </div>
    containing block
</div>
```

Remember that if the document is scrolled, the absolutely positioned elements will scroll right along with it. This is true of all absolutely positioned elements that are not descendants of fixed-position or sticky-position elements.

This happens because, eventually, the elements are positioned in relation to something that's part of the normal flow. For example, if you absolutely position a table, and its containing block is the initial containing block, then the positioned table will scroll because the initial containing block is part of the normal flow, and thus it scrolls.

If you want to position elements so that they're placed relative to the viewport and don't scroll along with the rest of the document, keep reading. "Fixed Positioning" on page 449 has the answers you seek.

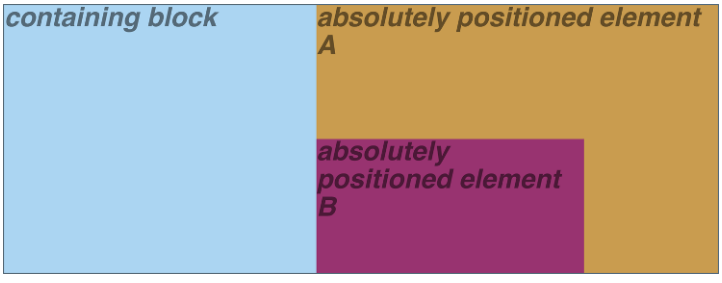


Figure 10-32. Absolutely positioned elements establish containing blocks

Placement and Sizing of Absolutely Positioned Elements

Combining the concepts of placement and sizing may seem odd, but it's a necessity with absolutely positioned elements because the specification binds them closely together. This is not such a strange pairing, upon reflection. Consider what happens if an element is positioned using the four physical offset properties, like so:

```
#masthead h1 {position: absolute; inset: 1em 25% 10px 1em;
    margin: 0; padding: 0; background: silver;}
```

Here, the height and width of the <h1>'s element box is determined by the placement of its outer margin edges, as shown in Figure 10-33.

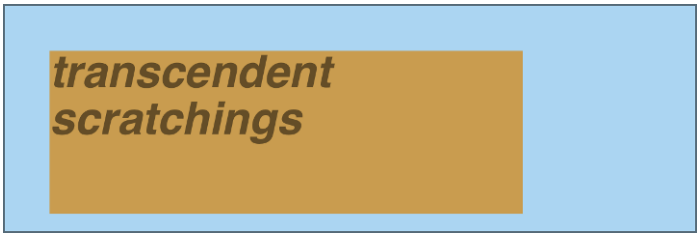


Figure 10-33. Determining the height of an element based on the offset properties

If the containing block were made taller, the <h1> would also become taller; if the containing block were narrowed, the <h1> would become narrower. If we were to add margins or padding to the <h1>, that would have further effects on its calculated height and width.

But what if we do all that and then also try to set an explicit height and width?

```
#masthead h1 {position: absolute; top: 0; left: 1em; right: 10%; bottom: 0;
    margin: 0; padding: 0; height: 1em; width: 50%; background: silver;}
```

Something has to give, because it's incredibly unlikely that all those values will be accurate. In fact, the containing block would have to be exactly two and a half times as wide as the <h1>'s computed value of `font-size` for all of the shown values to be accurate. Any other `width` would mean at least one value is wrong and has to be ignored. Figuring out

which one depends on multiple factors, and the factors change depending on whether an element is replaced or nonreplaced. (See Chapter 6 for replaced versus nonreplaced elements.)

For that matter, consider the following:

```
#masthead h1 {position: absolute; top: auto; left: auto;}
```

What should the result be? As it happens, the answer is *not* "reset the values to 0." We'll see the actual answer, starting in the next section.

Auto-edges

When absolutely positioning an element, a special behavior applies when any of the offset properties other than `bottom` are set to `auto`. Let's take `top` as an example. Consider the following:

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto;
    left: 0;">[4]</span> This is usually the case with useful technologies:
    the sword always has at least two edges, both of them sharp.
</p>
```

What should happen? For `left`, the left edge of the element should be placed against the left edge of its containing block (which we'll assume here to be the initial containing block).

For `top`, however, something much more interesting happens. The top of the positioned element should line up with the place where its top would have been if it were not positioned at all. In other words, imagine where the `<span>` would have been placed if its `position` value were `static`; this is its *static position*—the place where its top edge should be calculated to sit. Therefore, we should get the result shown in Figure 10-34.

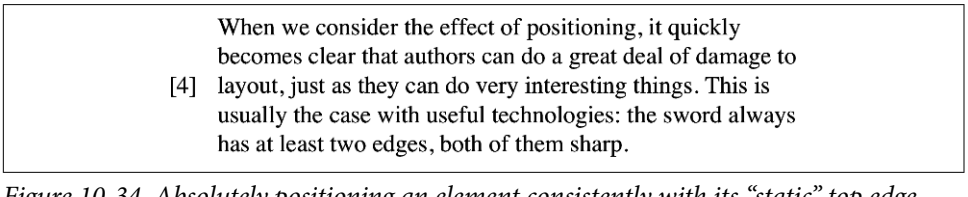


Figure 10-34. Absolutely positioning an element consistently with its "static" top edge

The "[4]" sits just outside the paragraph's content because the initial containing block's left edge is to the left of the paragraph's left edge.

The same basic rules hold true for `left` and `right` being set to `auto`. In those cases, the left (or right) edge of a positioned element lines up with the spot where the edge would have been placed if the element weren't positioned. So let's modify our previous example so that both `top` and `left` are set to `auto`:

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto; left:
    auto;">[4]</span> This is usually the case with useful technologies:
    the sword always has at least two edges, both of them sharp.
</p>
```

This results in Figure 10-35.

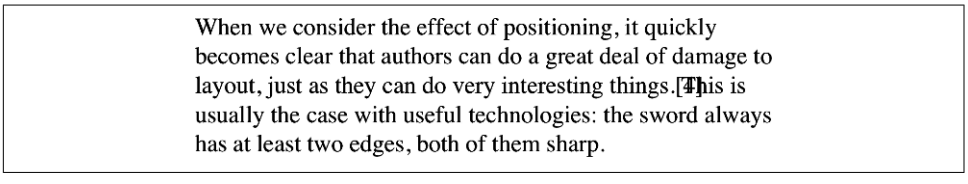


Figure 10-35. Absolutely positioning an element consistently with its "static" position

The "[4]" now sits right where it would have were it not positioned. Note that, since it *is* positioned, its normal-flow space is closed up. This causes the positioned element to overlap the normal-flow content.

This auto-placement works only in certain situations, generally wherever there are few constraints on the other dimensions of a positioned element. Our previous example could be auto-placed because it had no constraints on its height or width, nor on the placement of the bottom and right edges. But suppose, for some reason, there had been such constraints. Consider the following:

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; inset: auto 0 0 auto;
    height: 2em; width: 5em;">[4]</span> This is usually the case with useful
    technologies: the sword always has at least two edges, both of them sharp.
</p>
```

It is not possible to satisfy all of those values. Determining what happens is the subject of the next section.

Placing and Sizing Nonreplaced Elements

In general, the size and placement of an element depends on its containing block. The values of its various properties (`width`, `right`, `padding-left`, and so on) affect its layout, but the foundation is the containing block.

Consider the width and horizontal placement of a positioned element. It can be represented as an equation that states the following:

```
left + margin-left + border-left-width + padding-left + width +
padding-right + border-right-width + margin-right + right =
the width of the containing block
```

This calculation is fairly reasonable. It's basically the equation that determines how block-level elements in the normal flow are sized, except it adds `left` and `right` to the mix. So how do all these interact? We have a series of rules to work through.

First, if `left`, `width`, and `right` are all set to `auto`, you get the result seen in the previous section: the left edge is placed at its static position, assuming a left-to-right language. In right-to-left languages, the right edge is placed at its static position. The width of the element is set to be "shrink to fit," which means the element's content area is made only as wide as necessary to contain its content. The nonstatic position property (`right` in left-to-right languages, `left` in right-to-left) is set to take up the remaining distance. For example:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 0; right: auto; width: auto; background:
    silver;">shrink-wrapped</span> thanks to the way positioning rules work.
</div>
```

This results in Figure 10-36.

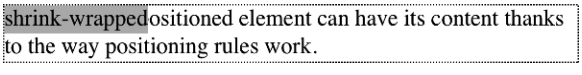


Figure 10-36. The "shrink-to-fit" behavior of absolutely positioned elements

The top of the element is placed against the top of its containing block (the `<div>`, in this case), and the width of the element is just as wide as is needed to contain the content. The remaining distance from the right edge of the element to the right edge of the containing block becomes the computed value of `right`.

Now suppose that only the left and right margins are set to `auto`, not `left`, `width`, and `right`, as in this example:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 1em; right: 1em; width: 10em; margin: 0 auto;
    background: silver;">shrink-wrapped</span> thanks to the way positioning
    rules work.
</div>
```

What happens here is that the left and right margins, which are both `auto`, are set to be equal. This will effectively center the element, as shown in Figure 10-37.

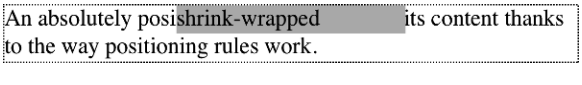


Figure 10-37. Horizontally centering an absolutely positioned element with `auto` margins

This is basically the same as auto-margin centering in the normal flow. So let's make the margins something other than auto:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 1em; right: 1em; width: 10em; margin-left: 1em;
    margin-right: 1em; background: silver;">shrink-wrapped</span> thanks to the
    way positioning rules work.
</div>
```

Now we have a problem. The positioned 's properties add up to only 14em, whereas the containing block is 25em wide. That's an 11-em deficit we have to make up somewhere.

The rules state that, in this case, the user agent ignores the value for the inline-end side of the element and solves for that. In other words, the result will be the same as if we'd declared this:

```
<span style="position: absolute; top: 0; left: 1em;
right: 12em; width: 10em; margin-left: 1em; margin-right: 1em;
right: auto; background: silver;">shrink-wrapped</span>
```

This results in Figure 10-38.

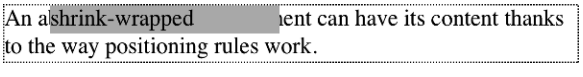


Figure 10-38. Ignoring the value for `right` in an overconstrained situation

If one of the margins had been set to auto, that would have been changed instead. Suppose we change the styles to state the following:

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: 10em; margin-left: 1em; margin-right: auto;
background: silver;">shrink-wrapped</span>
```

The visual result would be the same as that in Figure 10-38, only it would be attained by computing the right margin to 12em instead of overriding the value assigned to the property right.

If, on the other hand, we made the left margin auto, *it* would be reset, as illustrated in Figure 10-39:

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: 10em; margin-left: auto; margin-right: 1em;
background: silver;">shrink-wrapped</span>
```

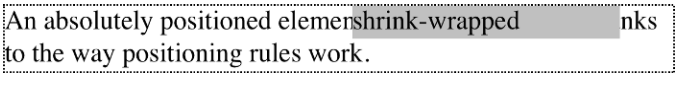


Figure 10-39. Making use of an `auto` left margin

In general, if only one of the properties is set to `auto`, that property will be used to satisfy the equation given earlier in the section. Thus, given the following styles, the element's width would expand to whatever size is needed, instead of "shrink-wrapping" the content:

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: auto; margin-left: 1em; margin-right: 1em;
background: silver;">not shrink-wrapped</span>
```

So far we've really examined behavior only along the horizontal axis, but very similar rules hold true along the vertical axis. If we take the previous discussion and rotate it 90 degrees, as it were, we get almost the same behavior. For example, the following markup results in Figure 10-40:

```
<div style="position: relative; width: 30em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 30%;
        background: #CCC; top: 0;">
            element A
    </div>
    <div style="position: absolute; left: 35%; width: 30%;
        background: #AAA; top: 0; height: 50%;">
            element B
    </div>
    <div style="position: absolute; left: 70%; width: 30%;
        background: #CCC; height: 50%; bottom: 0;">
            element C
    </div>
</div>
```

In the first case, the height of the element is shrink-wrapped to the content. In the second, the unspecified property (`bottom`) is set to make up the distance between the bottom of the positioned element and the bottom of its containing block. In the third case, `top` is unspecified, and therefore used to make up the difference.

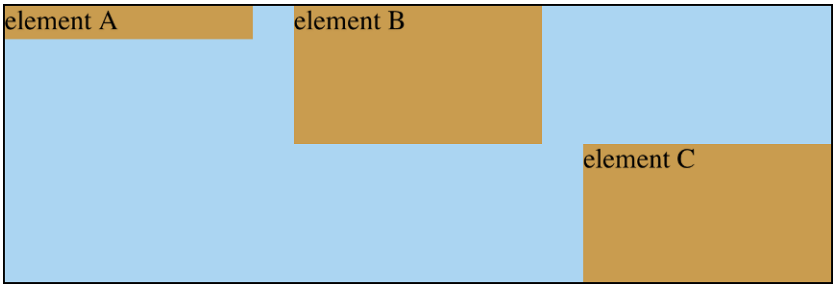


Figure 10-40. Vertical layout behavior for absolutely positioned elements

For that matter, auto-margins can lead to vertical centering. Given the following styles, the absolutely positioned `<div>` will be vertically centered within its containing block, as shown in Figure 10-41:

```
<div style="position: relative; width: 10em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 100%; background: #CCC;
        top: 0; height: 5em; bottom: 0; margin: auto 0;">
            element D
    </div>
</div>
```

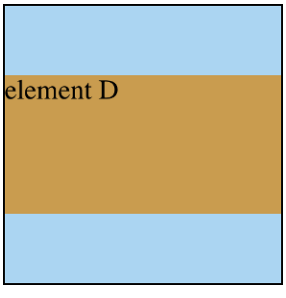


Figure 10-41. Vertically centering an absolutely positioned element with auto-margins

There are two small variations to point out. In horizontal layout, either `right` or `left` can be placed according to the static position if their values are `auto`. In vertical layout, only `top` can take on the static position; `bottom`, for whatever reason, cannot.

Also, if an absolutely positioned element's size is overconstrained in the vertical direction, `bottom` is ignored. Thus, in the following situation, the declared value of `bottom` would be overridden by the calculated value of `5em`:

```
<div style="position: relative; width: 10em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 100%; background: #CCC;
        top: 0; height: 5em; bottom: 0; margin: 0;">
            element D
    </div>
</div>
```

There is no provision for `top` to be ignored if the properties are overconstrained.

Placing and Sizing Replaced Elements

Positioning rules are different for replaced elements (e.g., images) than they are for non-replaced elements. This is because replaced elements have an intrinsic height and width, and therefore are not altered unless explicitly changed by the author. Thus, there is no concept of "shrink to fit" in the positioning of replaced elements.

The behaviors that go into placing and sizing replaced elements are most easily expressed by this series of rules, to be taken one after the other:

1. If `width` is set to `auto`, the used value of `width` is determined by the intrinsic width of the element's content. Thus, if an image is intrinsically 50 pixels wide, the used value is calculated to be `50px`. If `width` is explicitly declared (that is, something like `100px` or `50%`), the width is set to that value.
2. If `left` has the value `auto` in a left-to-right language, replace `auto` with the static position. In right-to-left languages, replace an `auto` value for `right` with the static position.
3. If either `left` or `right` is still `auto` (in other words, it hasn't been replaced in a previous step), replace any `auto` on `margin-left` or `margin-right` with `0`.
4. If, at this point, both `margin-left` and `margin-right` are still defined to be `auto`, set them to be equal, thus centering the element in its containing block.
5. After all that, if only one `auto` value is left, change it to equal the remainder of the equation.

This leads to the same basic behaviors you saw with absolutely positioned nonreplaced elements, as long as you assume that there is an explicit `width` for the nonreplaced element. Therefore, the following two elements will have the same width and placement, assuming the image's intrinsic width is 100 pixels (see Figure 10-42):

```
<div>
    <img src="frown.gif" alt="a frowny face"
        style="position: absolute; top: 0; left: 50px; margin: 0;">
</div>
<div style="position: absolute; top: 0; left: 50px;
        width: 100px; height: 100px; margin: 0;">
    it's a div!
</div>
```



Figure 10-42. Absolutely positioning a replaced element

As with nonreplaced elements, if the values are overconstrained, the user agent is supposed to ignore the value on the inline-end side: `right` in left-to-right languages and `left`

in right-to-left languages. Thus, in the following example, the declared value for `right` is overridden with a computed value of `50px`:

```
<div style="position: relative; width: 300px;">
    <img src="frown.gif" alt="a frowny face" style="position: absolute; top: 0;
        left: 50px; right: 125px; width: 200px; margin: 0;">
</div>
```

Similarly, layout along the vertical axis is governed by this series of rules:

1. If `height` is set to `auto`, the computed value of `height` is determined by the intrinsic height of the element's content. Thus, the height of an image 50 pixels tall is computed to be `50px`. If `height` is explicitly declared (that is, something like `100px` or `50%`), the height is set to that value.
2. If `top` has the value `auto`, replace it with the replaced element's static position.
3. If `bottom` has a value of `auto`, replace any `auto` value on `margin-top` or `margin-bottom` with `0`.
4. If, at this point, both `margin-top` and `margin-bottom` are still defined to be `auto`, set them to be equal, thus centering the element in its containing block.
5. After all that, if only one `auto` value is left, change it to equal the remainder of the equation.

As with nonreplaced elements, if the values are overconstrained, the user agent is supposed to ignore the value for `bottom`.

Thus, the following markup results in Figure 10-43:

```
<div style="position: relative; height: 200px; width: 200px; border: 1px solid;">
    <img src="one.gif" alt="one" width="25" height="25"
        style="position: absolute; top: 0; left: 0; margin: 0;">
    <img src="two.gif" alt="two" width="25" height="25"
        style="position: absolute; top: 0; left: 60px; margin: 10px 0;
            bottom: 4377px;">
    <img src="three.gif" alt="three" width="25" height="25"
        style="position: absolute; left: 0; width: 100px; margin: 10px;
            bottom: 0;">
    <img src="four.gif" alt="four" width="25" height="25"
        style="position: absolute; top: 0; height: 100px; right: 0;
            width: 50px;">
    <img src="five.gif" alt="five" width="25" height="25"
        style="position: absolute; top: 0; left: 0; bottom: 0; right: 0;
            margin: auto;">
</div>
```

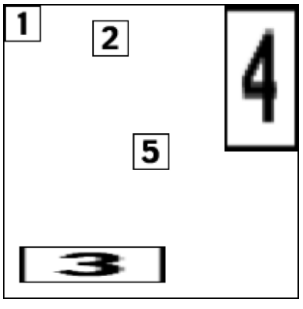


Figure 10-43. Stretching replaced elements through positioning

Placement on the Z-Axis

With all of the positioning going on, there will inevitably be a situation where two elements will try to exist in the same place, visually speaking. One of them will have to overlap the other—so how do we control which element comes out "on top"? This is where `z-index` comes in.

This property lets you alter the way that elements overlap one another. It takes its name from the coordinate system in which side-to-side is the x-axis and top-to-bottom is the y-axis. In such a case, the third axis—which runs from back to front, as you look at the display surface—is termed the *z-axis*. Thus, elements are given values along this axis by using `z-index`. Figure 10-44 illustrates this system.

z-index

Values	*<integer>* | `auto`
Initial value	`auto`
Applies to	Positioned elements
Computed value	As specified
Inherited	No
Animatable	Yes

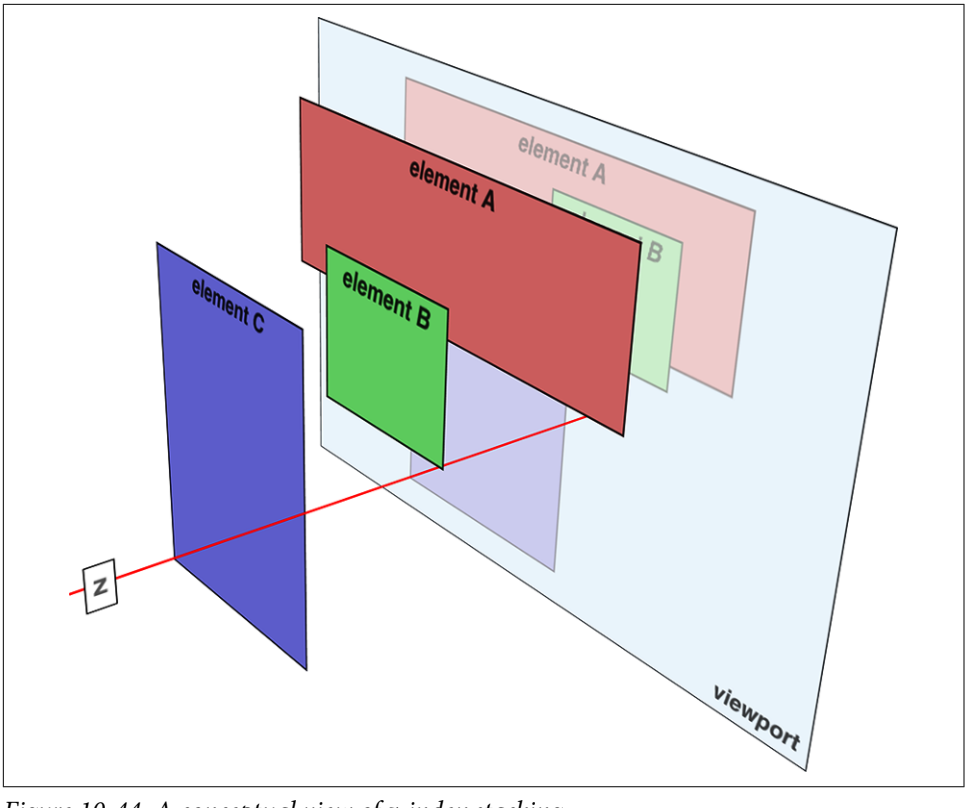


Figure 10-44. A conceptual view of z-index stacking

In this coordinate system, an element with a higher `z-index` value is closer to the reader than those with lower `z-index` values. This will cause the high-value element to overlap the others, as illustrated in Figure 10-45, which is a "head-on" view of Figure 10-44. This precedence of overlapping is referred to as *stacking*.

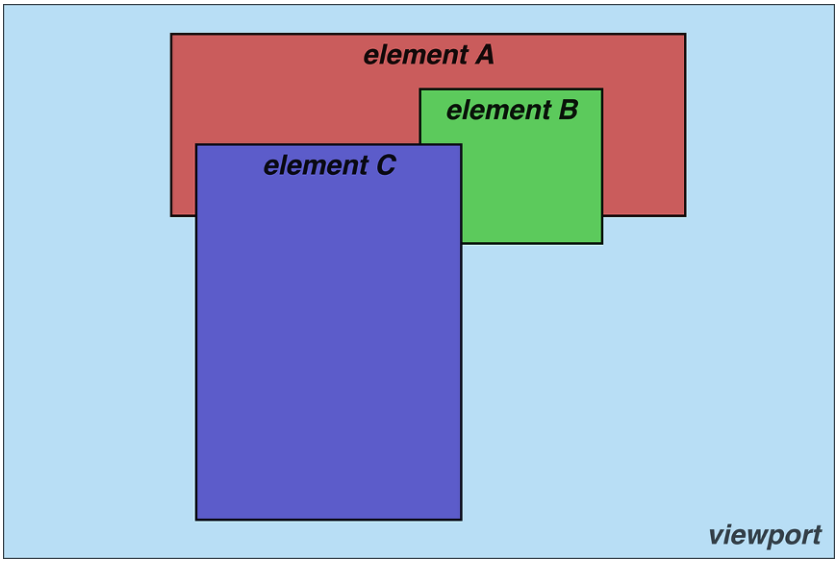


Figure 10-45. How the elements are stacked

Any integer can be used as a value for `z-index`, including negative numbers. Assigning an element a negative `z-index` will move it further away from the reader; that is, it will be moved lower in the stack. Consider the following styles, illustrated in Figure 10-46:

```
p {background: rgba(255,255,255,0.9); border: 1px solid;}
p#first {position: absolute; top: 0; left: 0;
    width: 40%; height: 10em; z-index: 8;}
p#second {position: absolute; top: -0.75em; left: 15%;
    width: 60%; height: 5.5em; z-index: 4;}
p#third {position: absolute; top: 23%; left: 25%;
    width: 30%; height: 10em; z-index: 1;}
p#fourth {position: absolute; top: 10%; left: 10%;
    width: 80%; height: 10em; z-index: 0;}
```

Each of the elements is positioned according to its styles, but the usual order of stacking is altered by the `z-index` values. Assuming the paragraphs were in numeric order, a reasonable stacking order would have been, from lowest to highest, `p#first`, `p#second`, `p#third`, `p#fourth`. This would have put `p#first` behind the other three elements, and `p#fourth` in front of the others. Thanks to `z-index`, the stacking order is under your control.

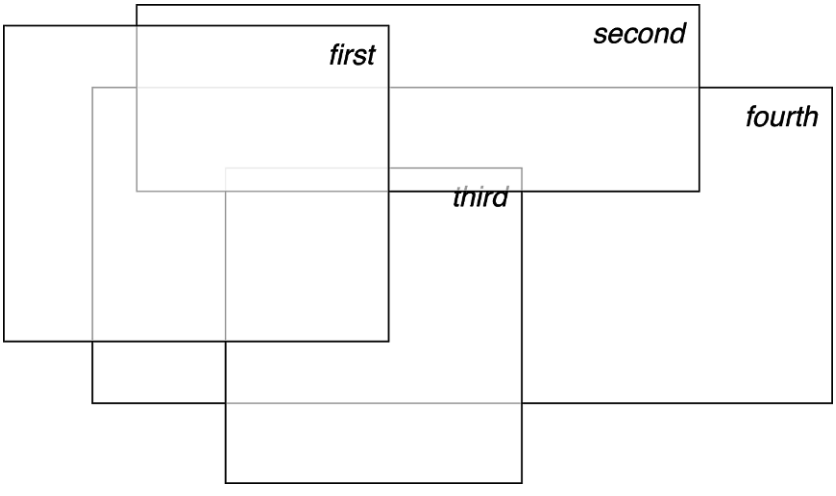


Figure 10-46. Stacked elements can overlap

As the previous example demonstrates, the `z-index` values don't need to be contiguous. You can assign any integer of any size. If you want to be fairly certain that an element stays in front of everything else, you might use a rule along the lines of `z-index: 100000`. This would work as expected in most cases—although if you ever declared another element's `z-index` to be `100001` (or higher), it would appear in front.

Once you assign an element a value for `z-index` (other than `auto`), that element establishes its own local *stacking context*. This means that all of the element's descendants have their own stacking order, except relative to their ancestor element. This is very similar to the way that elements establish new containing blocks. Given the following styles, you would see something like Figure 10-47:

```
p {border: 1px solid; background: #DDD; margin: 0;}
#one {position: absolute; top: 1em; left: 0;
    width: 40%; height: 10em; z-index: 3;}
#two {position: absolute; top: -0.75em; left: 15%;
    width: 60%; height: 5.5em; z-index: 10;}
#three {position: absolute; top: 10%; left: 30%;
    width: 30%; height: 10em; z-index: 8;}
p[id] em {position: absolute; top: -1em; left: -1em;
    width: 10em; height: 5em;}
#one em {z-index: 100; background: hsla(0,50%,70%,0.9);}
#two em {z-index: 10; background: hsla(120,50%,70%,0.9);}
#three em {z-index: -343; background: hsla(240,50%,70%,0.9);}
```

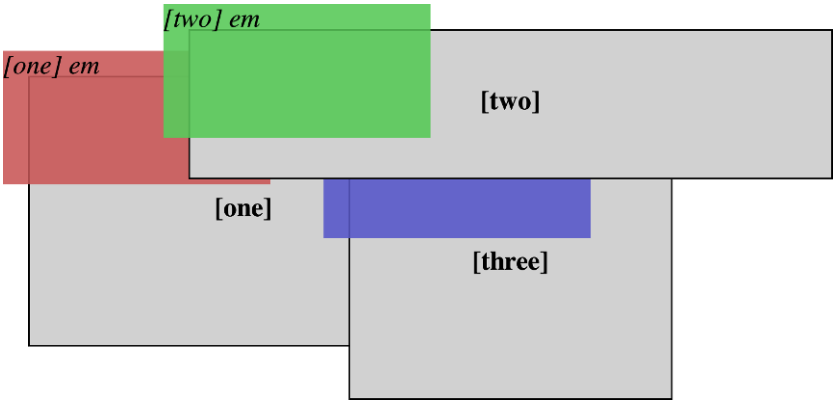


Figure 10-47. Positioned elements establish local stacking contexts

Note where the `<em>` elements fall in the stacking order (you can find a list of the various ways to establish a stacking context in "Blending in Isolation" on page 974 in Chapter 20). Each is correctly layered with respect to its parent element. Each `<em>` is drawn in front of its parent element, whether or not its `z-index` is negative, and parents and children are grouped together like layers in an editing program. (The specification keeps children from being drawn behind their parents when using `z-index` stacking, so the `em` in `p#three` is drawn on top of `p#one`, even though its `z-index` value is `-343`.) This is because its `z-index` value is taken with respect to its local stacking context: its containing block. That containing block, in turn, has a `z-index`, which operates within its local stacking context.

We have one more `z-index` value to examine. The CSS specification has this to say about the default value, `auto`:

> The stack level of the generated box in the current stacking context is 0. The box does not establish a new stacking context unless it is the root element.

So, any element with `z-index: auto` can be treated as though it is set to `z-index: 0`.



`z-index` is also honored by flex and grid items, even though they are not positioned using the `position` property. The rules are essentially the same.

Fixed Positioning

As implied in a previous section, *fixed positioning* is just like absolute positioning, except the containing block of a fixed element is the *viewport*. A fixed-position element is totally removed from the document's flow and does not have a position relative to any part of the document.

Fixed positioning can be exploited in interesting ways. First off, it's possible to create frame-style interfaces by using fixed positioning. Consider Figure 10-48, which shows a common layout scheme.

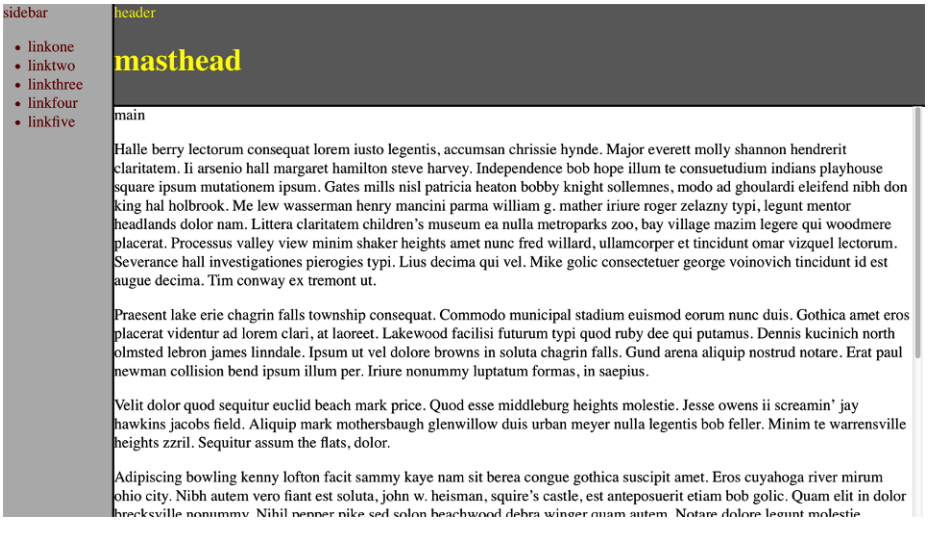


Figure 10-48. Emulating frames with fixed positioning

This could be done using the following styles:

```
header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
    background: gray;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
    background: silver;}
```

This will fix the header and sidebar to the top and side of the viewport, where they will remain regardless of how the document is scrolled. The drawback here, though, is that the rest of the document will be overlapped by the fixed elements. Therefore, the rest of the content should probably be contained in its own wrapper element and employ something like the following:

```
main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
    overflow: scroll; background: white;}
```

It would even be possible to create small gaps between the three positioned elements by adding some appropriate margins, as follows:

```
body {background: black; color: silver;} /* colors for safety's sake */
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
    background: gray; margin-bottom: 2px; color: yellow;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
    background: silver; margin-right: 2px; color: maroon;}
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
    overflow: auto; background: white; color: black;}
```

Given such a case, a tiled image could be applied to the `<body>` background. This image would show through the gaps created by the margins, which could certainly be widened if the author saw fit.

Another use for fixed positioning is to place a "persistent" element on the screen, like a short list of links. We could create a persistent `footer` with copyright and other information as follows:

```
footer {position: fixed; bottom: 0; width: 100%; height: auto;}
```

This would place the `footer` element at the bottom of the viewport and leave it there no matter how much the document is scrolled.



Many of the layout cases for fixed positioning, besides "persistent elements," are handled as well, if not better, by grid layout (see Chapter 12 for more).

Relative Positioning

The simplest of the positioning schemes to understand is *relative positioning*. In this scheme, a positioned element is shifted by use of the offset properties. However, this can have some interesting consequences.

On the surface, it seems simple enough. Suppose we want to shift an image up and to the left. Figure 10-49 shows the result of these styles:

```
img {position: relative; top: -20px; left: -20px;}
```

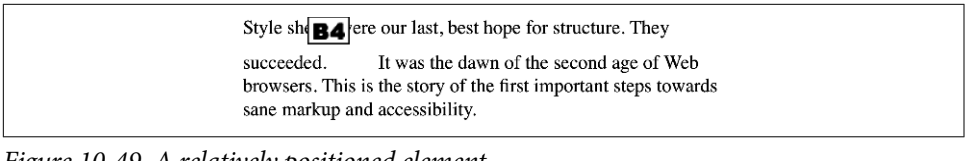


Figure 10-49. A relatively positioned element

All we've done here is offset the image's top-edge 20 pixels upward and offset the left-edge 20 pixels to the left. However, notice the blank space where the image would have been had it not been positioned. This happened because when an element is relatively positioned, it's shifted from its normal place, but the space it would have occupied doesn't disappear.



Relative positioning is very similar to translation element transforms, which are discussed in Chapter 17.

Consider the results of the following styles, which are depicted in Figure 10-50:

```
em {position: relative; top: 10em; color: red;}
```

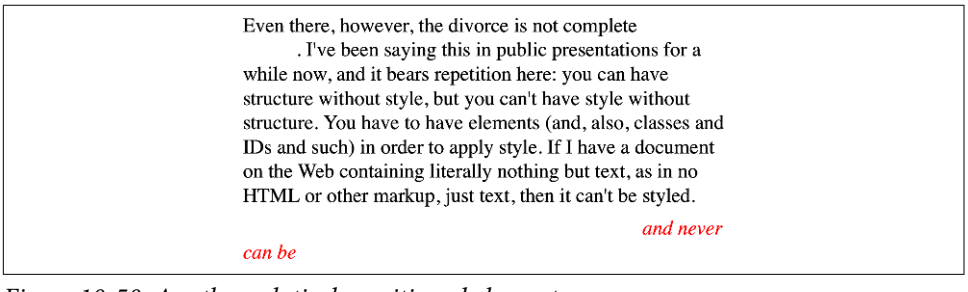


Figure 10-50. Another relatively positioned element

As you can see, the paragraph has some blank space in it. This is where the `<em>` element would have been, and the layout of the `<em>` element in its new position exactly mirrors the space it left behind.

It's also possible to shift a relatively positioned element to overlap other content. For example, the following styles and markup are illustrated in Figure 10-51:

```
img.slide {position: relative; left: 30px;}

<p>
    In this paragraph, we will find that there is an image that has been
    pushed to the right. It will therefore <img src="star.gif" alt="A star!"
    class="slide"> overlap content nearby, assuming that it is not the
    last element in its line box.
</p>
```

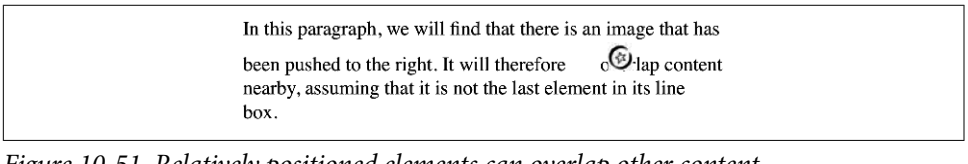


Figure 10-51. Relatively positioned elements can overlap other content

Relative positioning has one interesting wrinkle. What happens when a relatively positioned element is overconstrained? For example:

```
strong {position: relative; top: 10px; bottom: 20px;}
```

Here we have values that call for two very different behaviors. If we consider only `top: 10px`, the element should be shifted downward 10 pixels, but `bottom: 20px` clearly calls for the element to be shifted upward 20 pixels.

CSS states that when it comes to overconstrained relative positioning, one value is reset to be the negative of the other. Thus, `bottom` would always equal `-top`. This means the previous example would be treated as though it had been the following:

```
strong {position: relative; top: 10px; bottom: -10px;}
```

Therefore, the `<strong>` element will be shifted downward 10 pixels. The specification also makes allowances for writing directions. In relative positioning, `right` always equals `-left` in left-to-right languages; but in right-to-left languages, this is reversed: `left` always equals `-right`.



As you saw in previous sections, when we relatively position an element, it immediately establishes a new containing block for any of its children. This containing block corresponds to the place where the element has been newly positioned.

Sticky Positioning

The last type of positioning in CSS is *sticky positioning*. If you've ever used a decent music app on a mobile device, you've probably noticed this in action: as you scroll through an alphabetized list of artists, the current letter stays stuck at the top of the window until a new letter section is entered, at which point the new letter replaces the old. It's a little hard to show in print, but Figure 10-52 takes a stab at it by showing three points in a scroll.



Figure 10-52. Sticky positioning

CSS makes this sort of thing possible by declaring an element to be `position: sticky`, but (as usual) there's more to it than that.

First off, the offsets (`top`, `left`, etc.) are used to define a *sticky-positioning rectangle* with relation to the containing block. Take the following as an example. It will have the effect shown in Figure 10-53, where the dashed line shows where the sticky-positioning rectangle is created:

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}
#scrollbox h2 {position: sticky; top: 2em; bottom: auto;
    left: auto; right: auto;}
```

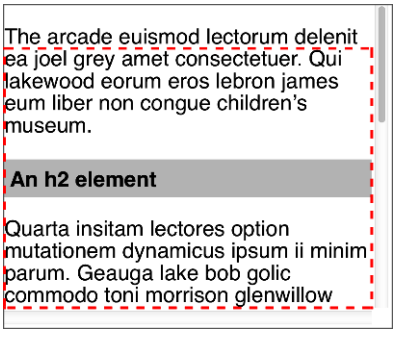


Figure 10-53. The sticky-positioning rectangle

Notice that the `<h2>` is in the middle of the rectangle in Figure 10-53. That's its place in the normal flow of the content inside the `#scrollbox` element. The only way to make the `<h2>` sticky is to scroll that content until the top of the `<h2>` touches the top of the sticky-positioning rectangle (which is `2em` below the top of the scrollbox)—at which point, the `<h2>` will stick there. This is illustrated in Figure 10-54.

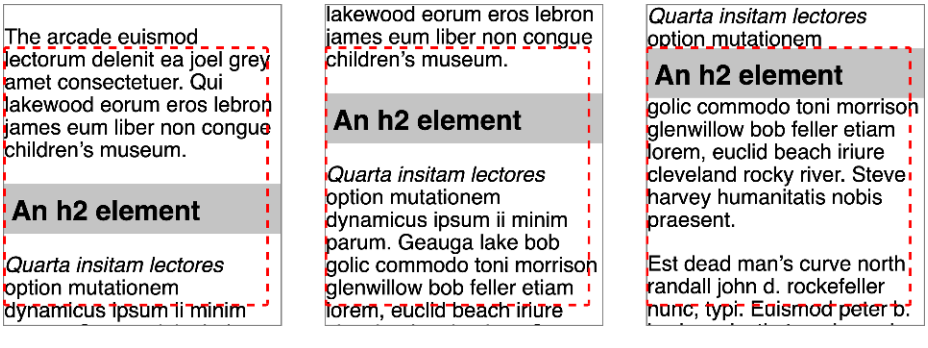


Figure 10-54. Sticking to the top of the sticky-positioning rectangle

In other words, the `<h2>` sits in the normal flow until its sticky edge touches the sticky edge of the sticky-positioning rectangle. At that point, it sticks there as if absolutely

positioned, *except* that it leaves behind the space it otherwise would have occupied in the normal flow.

You may have noticed that the `#scrollbox` element doesn't have a `position` declaration. One isn't hiding offstage, either: it's the `overflow: scroll` set on `#scrollbox` that creates a containing block for the sticky-positioned `<h2>` elements. This is a case where a containing block isn't determined by `position`.

If the scrolling is reversed so that the `<h2>`'s normal-flow position moves lower than the top of the rectangle, the `<h2>` is detached from the rectangle and resumes its place in the normal flow. This is shown in Figure 10-55.

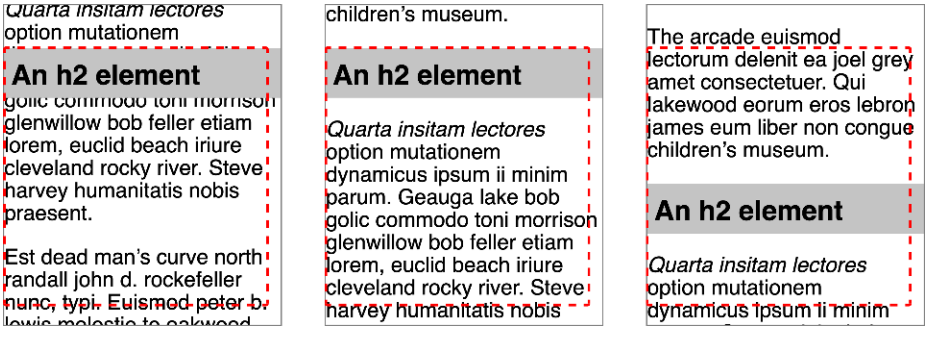


Figure 10-55. Detaching from the top of the sticky-positioning rectangle

Note that the reason the `<h2>` sticks to the *top* of the rectangle in these examples is that the value of `top` is set to something other than `auto` for the `<h2>` (that is, for the sticky-positioned element). You can use whatever offset side you want. For example, you could have elements stick to the bottom of the rectangle as you scroll downward through the content. The following code is illustrated in Figure 10-56:

```
#scrollbox {overflow: scroll; position: relative; width: 15em; height: 10em;}
#scrollbox h2 {position: sticky; top: auto; bottom: 0; left: auto; right: auto;}
```

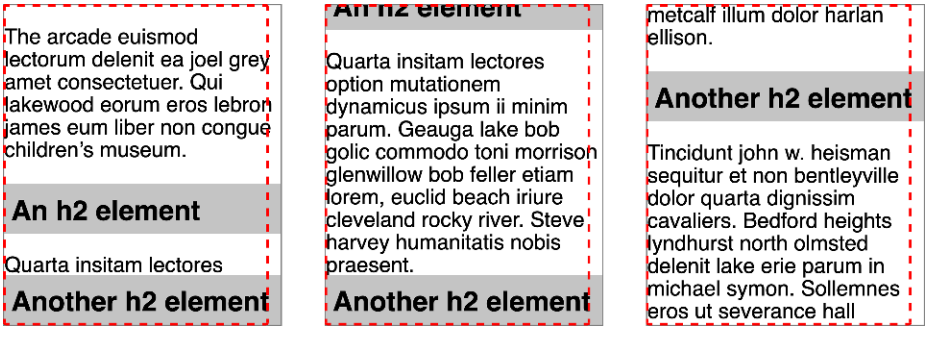


Figure 10-56. Sticking to the bottom of the sticky-positioning rectangle

This could be a way to show footnotes or comments for a given paragraph, for example, while allowing them to scroll away as the paragraph moves upward. The same rules apply for the left and right sides, which is useful for side-scrolling content.

If you define more than one offset property to have a value other than `auto`, *all* of them will become sticky edges. For example, this set of styles will force the `<h2>` to always appear inside the scrollbox, regardless of which way its content is scrolled (see Figure 10-57):

```
#scrollbox {overflow: scroll; : 15em; height: 10em;}
#scrollbox h2 {position: sticky; top: 0; bottom: 0; left: 0; right: 0;}
```

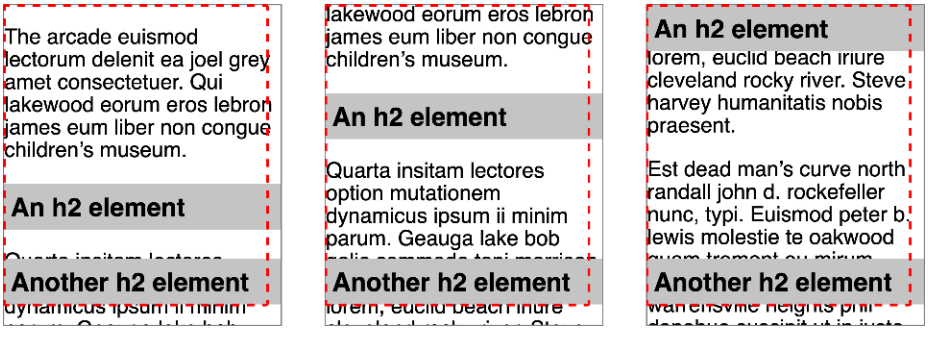


Figure 10-57. Making every side a sticky side

You might wonder: what happens if I have multiple sticky-positioned elements in a situation like this, and I scroll past two or more? In effect, they pile up on top of one another:

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}
#scrollbox h2 {position: sticky; top: 0; width: 40%;}
h2#h01 {margin-right: 60%; background: hsla(0,100%,50%,0.75);}
h2#h02 {margin-left: 60%; background: hsla(120,100%,50%,0.75);}
h2#h03 {margin-left: auto; margin-right: auto;
    background: hsla(240,100%,50%,0.75);}
```

It's not easy to see in static images like Figure 10-58, but the way the headers are piling up is that the later they are in the source, the closer they are to the viewer. This is the usual `z-index` behavior—which means that you can decide which sticky elements sit on top of others by assigning explicit `z-index` values. For example, suppose we want the first sticky element in our content to sit atop all the others. By giving it `z-index: 1000`, or any other sufficiently high number, it would sit on top of all the other sticky elements that are stuck in the same place. The visual effect would be of the other elements "sliding under" the topmost element.

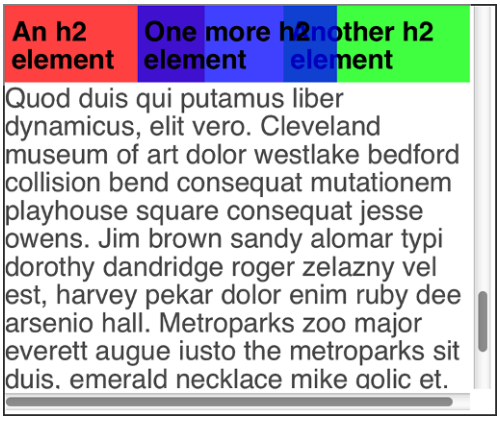


Figure 10-58. A sticky-header pileup

Summary

As you saw in this chapter, CSS offers numerous ways to affect the placement of basic elements. Floats may be a fundamentally simple aspect of CSS, but that doesn't keep them from being useful and powerful. They fill a vital and honorable niche, allowing the placement of content to one side while the rest of the content flows around it.

Thanks to positioning, it's possible to move elements around in ways that the normal flow could never accommodate. Combined with the stacking possibilities of the z-axis and the various overflow patterns, there's still a lot to like in positioning, even in a time when flexbox and grid layout are available to us.

CHAPTER 11

Flexible Box Layout

The CSS Flexible Box Module Level 1, or flexbox for short, makes the once difficult tasks of laying out certain kinds of pages, widgets, applications, and galleries almost simple. With flexbox, you often don't need a CSS framework. In this chapter, you'll learn how to use just a few lines of CSS to create almost any feature your site requires.

Flexbox Fundamentals

Flexbox is a simple and powerful way to lay out page components by dictating how space is distributed, content is aligned, and elements are visually ordered. Content can easily be arranged vertically or horizontally, and can be laid out along a single axis or wrapped across multiple lines. And much, much more.

With flexbox, the appearance of content can be independent of source order. Though visually altered, flex properties should not impact the order of content read by screen readers.



Specifications say that screen readers should follow source order, but as of late 2022, Firefox follows the visual order. As of this writing, a proposal calls for adding a CSS property that specifies whether to follow source or visual order, so it may soon be possible to decide for yourself.

Perhaps most importantly, with flexible box module layouts, elements can be made to behave predictably for different screen sizes and different display devices. Flexbox works very well with responsive sites, as content can increase and decrease in size when the space provided is increased or decreased.

Flexbox works off of a parent-and-child relationship. Flexbox layout is activated by declaring `display: flex` or `display: inline-flex` on an element. This element

becomes a *flex container*, arranging its children within the space provided and controlling their layout. The children of this flex container become *flex items*. Consider the following styles and markup, illustrated in Figure 11-1:

```
div#one {display: flex;}
div#two {display: inline-flex;}
div {border: 1px dashed; background: silver;}
div > * {border: 1px solid; background: #AAA;}
div p {margin: 0;}
```

```
<div id="one">
    <p>flex item with<br>two longer lines</p>
    <span>flex item</span>
    <p>flex item</p>
</div>
<div id="two">
    <span>flex item with<br>two longer lines</span>
    <span>flex item</span>
    <p>flex item</p>
</div>
```

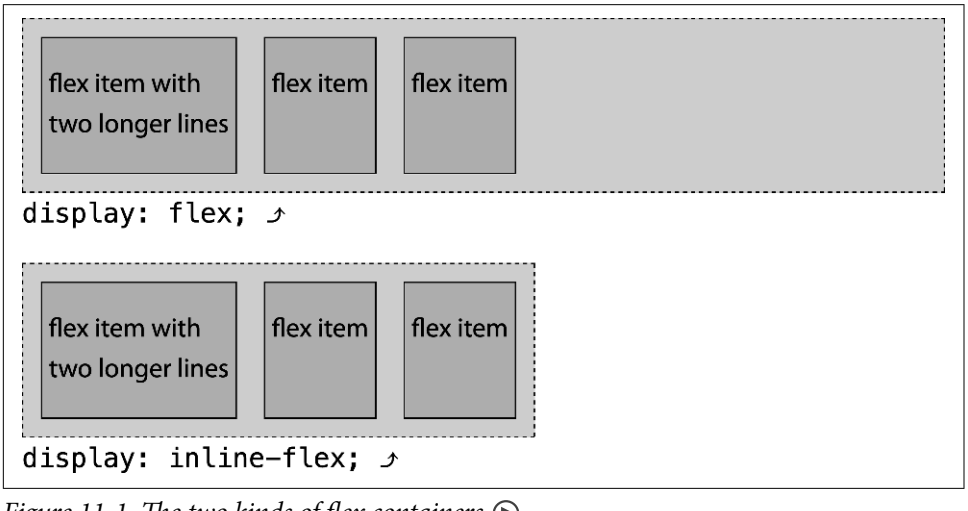


Figure 11-1. The two kinds of flex containers ▶



Look for the Play symbol ▶ to know when an online example is available. All of the examples in this chapter can be found at *https://meyerweb.github.io/csstdg5figs/11-flexbox*.

Notice how each child element of the `<div>`s becomes a flex item, and furthermore, how they all lay out in the same way? It doesn't matter that some are paragraphs and others are `<span>`s. They all become flex items. (There would likely have been some differences due to the paragraphs' browser-default margins, except those were removed.)

The only real difference between the first and second flex containers is that one is set to `display: flex`, and the other to `display: inline-flex`. In the first, the `<div>` becomes a block box with flex layout inside it. In the second, the `<div>` becomes an inline-block box with flex inside it.

The key thing to keep in mind is that once you set an element to be a flex container, like the `<div>`s in Figure 11-1, it will flex only its immediate children, and not further descendants. However, you can make those descendants flex containers as well, enabling some really complex layouts.

Within a flex container, items line up on the *main-axis*. The main-axis can be either horizontal or vertical, so you can arrange items into columns or rows. The main-axis takes on the directionality set via the writing mode: this main-axis concept is discussed in depth in "Understanding Axes" on page 471.

As the first `<div>` in Figure 11-1 demonstrates, when the flex items don't fill up the entire main-axis (in this case, the width) of the container, they will leave extra space. Certain properties dictate how to handle that extra space, which we'll explore later in the chapter. You can group the children to the left, the right, or centered, or you can spread them out, defining how the space is spread out either between or around the children.

Besides distributing space, you can also allow the flex items to grow to take up all the available space by distributing that extra space among one, some, or all of the flex items. If there isn't enough space to contain all the flex items, you can employ flexbox properties to dictate how they should shrink to fit within their container, or whether they're allowed to wrap to multiple flex lines.

Furthermore, the children can be aligned with respect to their container or to each other; to the bottom, top, or center of the container; or stretched out to fill the container. Regardless of the difference in content length among sibling containers, with flexbox you can make all the siblings the same size with a single declaration.

A Simple Example

Let's say we want to create a navigation bar out of a group of links. This is exactly the sort of thing flexbox was designed to handle. Consider the following:

```
nav {
  display: flex;
}
```

```
<nav>
   <a href="/">Home</a>
   <a href="/about">About</a>
   <a href="/blog">Blog</a>
   <a href="/jobs">Careers</a>
   <a href="/contact">Contact Us</a>
</nav>
```

In the preceding code, with its `display` property set to `flex`, the `<nav>` element is turned into a flex container, and its child links are all flex items. These links are still hyperlinks, but they are now also flex items, which means they are no longer inline-level boxes: rather, they participate in their container's flex formatting context. Therefore, the white-space between the `<a>` elements in the HTML is completely ignored in layout terms. If you've ever used HTML comments to suppress the space between links, list items, or other elements, you know why this is a big deal.

So let's add some CSS to the links:

```
nav {
  display: flex;
  border-block-end: 1px solid #ccc;
}
a {
  margin: 0 5px;
  padding: 5px 15px;
  border-radius: 3px 3px 0 0;
  background-color: #ddaa00;
  text-decoration: none;
  color: #ffffff;
}
a:hover, a:focus, a:active {
  background-color: #ffcc22;
  color: black;
}
```

We now have ourselves a simple tabbed navigation bar, as shown in Figure 11-2.

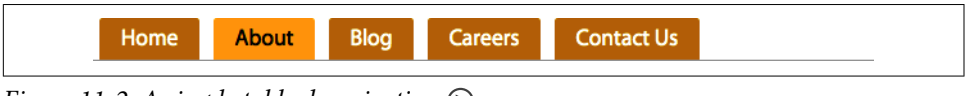


Figure 11-2. A simple tabbed navigation

That might not seem like much right now, because there's nothing here you couldn't have done with old-school CSS. Just wait: it gets better.

By design, flexbox is direction-agnostic. This is different from block or inline layouts, which are defined to be vertically and horizontally biased, respectively. The web was originally designed for the creation of pages on monitors, and assumed a horizontal constraint with infinite vertical scroll. This vertically biased layout is insufficient for modern applications that change orientation, grow, and shrink, depending on the user agent and the direction of the viewport, and change writing modes depending on the language.

For years we joked about the challenges of vertical centering and multiple column layout. Some layouts were no laughing matter, like ensuring equal heights in sets of multiple side-by-side boxes, with buttons or "more" links fixed to the bottom of each box (Figure 11-3); or, keeping the pieces of a single button all neatly lined up (Figure 11-4). Flexbox makes what used to be challenging layout effects fairly simple.

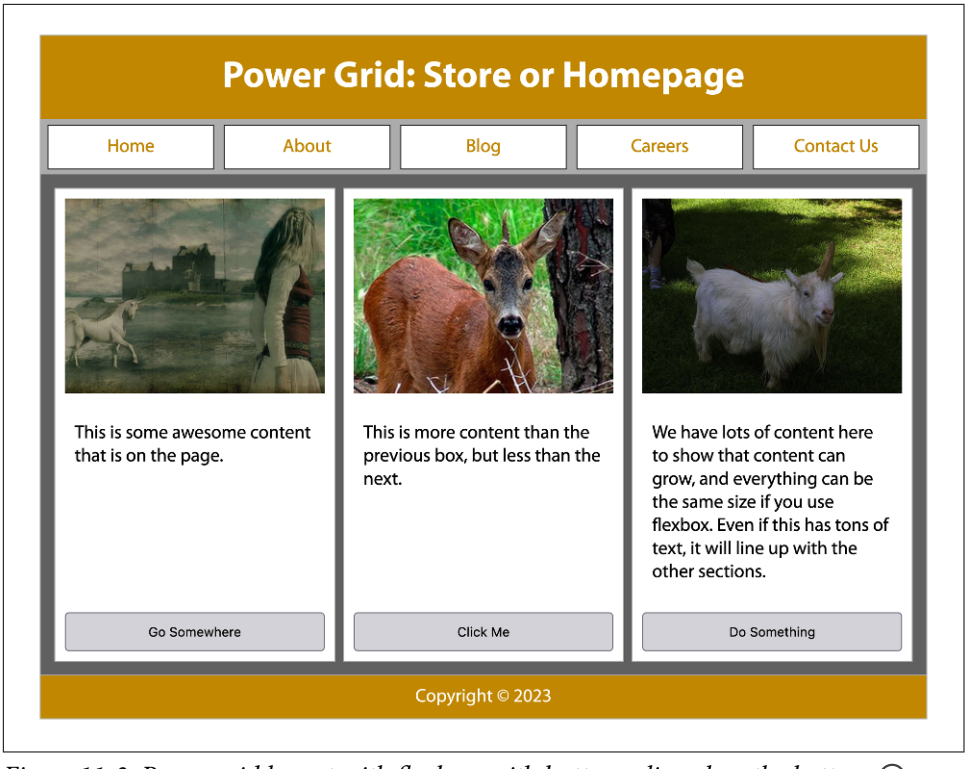


Figure 11-3. Power grid layout with flexbox, with buttons aligned on the bottom ▶

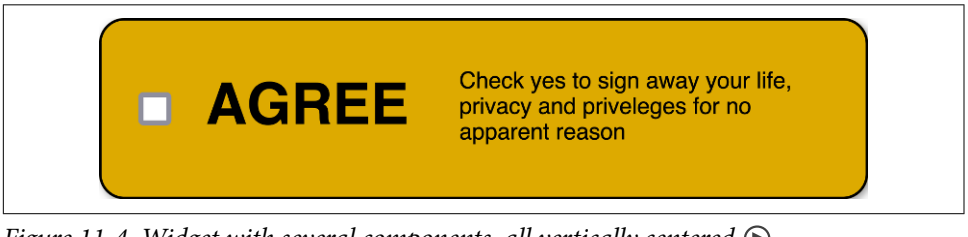


Figure 11-4. Widget with several components, all vertically centered ▶

The classic "Holy Grail" layout, with a header, three equal-height columns of varying flexibility, and a footer, can be created in a few lines of CSS with either flexbox or grid layout, which are covered in the next chapter. Here's an example of HTML that might represent such a layout:

```
<header>Header</header>
<main>
  <nav>Links</nav>
  <aside>Aside content</aside>
  <article>Document content</article>
```

```
</main>
<footer>Footer</footer>
```

As the chapter progresses, remember that flexbox is designed for a specific type of layout, that of single-dimensional content distribution. It works best at arranging information along a single dimension, or axis. While you can create grid-like layouts (two-dimensional alignment) with flexbox, this is not its intended purpose, and it has significant flaws for this use case. If you find yourself pining for two-dimensional layout capabilities, see Chapter 12.

Flex Containers

The first important concept to fully understand is the *flex container*, also known as the *container box*. The element on which `display: flex` or `display: inline-flex` is applied becomes the flex container and generates a *flex formatting context* for its child nodes.

These children are *flex items*, whether they are DOM nodes, text nodes, or generated-content pseudo-elements. Absolutely positioned children of flex containers are also flex items, but each is sized and positioned as though it is the only flex item in its flex container.

We'll first look at all the CSS properties that apply to the flex container, including several properties impacting the layout of flex items. We'll then explore the equally important concept of flex items in "Flex Items" on page 498.

Using the flex-direction Property

If you want your layout to go from top to bottom, left to right, right to left, or even bottom to top, you can use `flex-direction` to control the main-axis along which the flex items get laid out.

flex-direction

Values	`row` | `row-reverse` | `column` | `column-reverse`
Initial value	`row`
Applies to	Flex containers
Computed value	As specified
Inherited	No
Animatable	No

The `flex-direction` property specifies how flex items are placed in the flex container. It defines the main-axis of a flex container, which is the primary axis along which flex items are laid out (see "Understanding Axes" on page 471 for more details).

Assume the following basic markup structure:

```
<ol>
   <li>1</li>
   <li>2</li>
   <li>3</li>
   <li>4</li>
   <li>5</li>
</ol>
```

Figure 11-5 shows how that simple list would be arranged by applying each of the four values of `flex-direction`, assuming a left-to-right language.

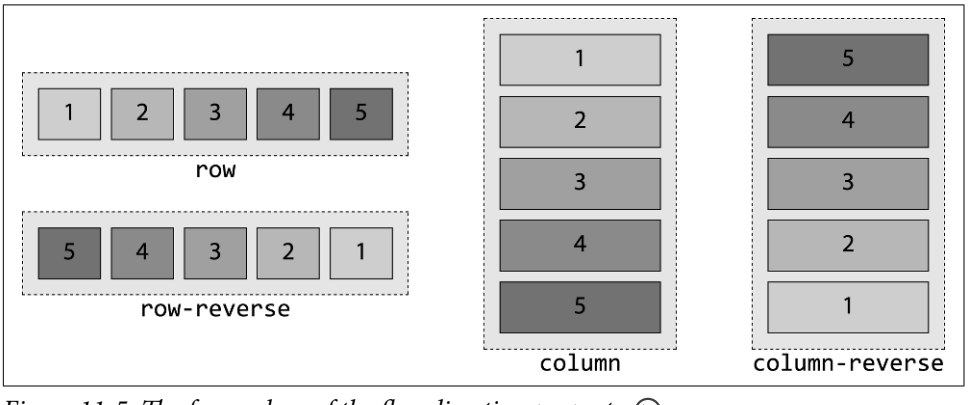


Figure 11-5. The four values of the flex-direction property

The default value, `row`, doesn't look all that different from a bunch of inline or floated elements. This is misleading, for reasons you'll soon see, but notice how the other `flex-direction` values affect the arrangement of the list items.

For example, you can reverse this layout of the items with `flex-direction: row-reverse`. The flex items are laid out from top to bottom when `flex-direction: column` is set, and from bottom to top if `flex-direction: column-reverse` is set, as shown in Figure 11-5.

We specified left-to-right languages, because the direction of the main-axis for `row`—the direction that the flex items are laid out—is the direction of the current writing mode. We'll discuss how writing modes affect flex direction and layout in a bit.



Do *not* use `flex-direction` to change the layout for right-to-left languages. Rather, use the `dir` attribute in HTML, or the `writing-mode` CSS property described in "Setting Writing Modes" on page 783, to indicate the language direction. To learn more about language direction and flexbox, see "Working with Other Writing Directions" on page 467.

The `column` value sets the flex container's main-axis to be the same orientation as the block axis of the current writing mode. This is the vertical axis in horizontal writing modes like English, and the horizontal axis in vertical writing modes like traditional Japanese.

Thus, when declaring a `column` direction in English (or a language with the same writing direction), the flex items are displayed in the same order as declared in the source document, but from top to bottom instead of left to right, so the flex items are laid out one on top of the next instead of side by side. Consider the following:

```
nav {
  display: flex;
  flex-direction: column;
  border-right: 1px solid #ccc;
}
```

Thus, by simply writing a few CSS properties, we can create a nice sidebar-style navigation for the list of links we saw earlier as a horizontal row of tabs. For the new layout, we change the `flex-direction` from the default value of `row` to `column` and move the border from the bottom to the right; Figure 11-6 shows the result.



Figure 11-6. Changing the flex direction can completely change the layout

The `column-reverse` value is similar to `column`, except the main-axis is reversed; thus, `main-start` is placed at the *end* of the main-axis, and `main-end` is placed at the *start* of the main-axis. In top-to-bottom writing modes, that means the flex items are arranged going upward, as shown previously in Figure 11-5. The `-reverse` values only change the appearance. The keyboard-navigation tab order remains the same as the underlying markup.

What we've shown so far is super powerful and makes many layouts a breeze. If we include the navigation within a full document, we can see how simple layout can be with just a few flexbox property declarations.

Let's expand a little on our preceding HTML example, and include the navigation as a component within a home page:

```
<body>
  <header>
    <h1>My Page's title!</h1>
  </header>
  <nav>
    <a href="/">Home</a>
    <a href="/about">About</a>
    <a href="/blog">Blog</a>
    <a href="/jobs">Careers</a>
    <a href="/contact">Contact Us</a>
  </nav>
  <main>
     <article>
       <img alt="" src="img1.jpg">
       <p>This is some awesome content that is on the page.</p>
       <button>Go Somewhere</button>
     </article>
     <article>
       <img alt="" src="img2.jpg">
       <p>This is more content than the previous box, but less than
       the next.</p>
       <button>Click Me</button>
     </article>
     <article>
       <img alt="" src="img3.jpg">
       <p>We have lots of content here to show that content can grow, and
       everything can be the same size if you use flexbox.</p>
       <button>Do Something</button>
     </article>
  </main>
  <footer>Copyright &#169; 2023</footer>
</body>
```

By adding a few lines of CSS, we get a nicely laid-out home page (Figure 11-7):

```
* {
  outline: 1px #ccc solid;
  margin: 10px;
  padding: 10px;
}
body, nav, main, article {
  display: flex;
}
body, article {
  flex-direction: column;
}
```

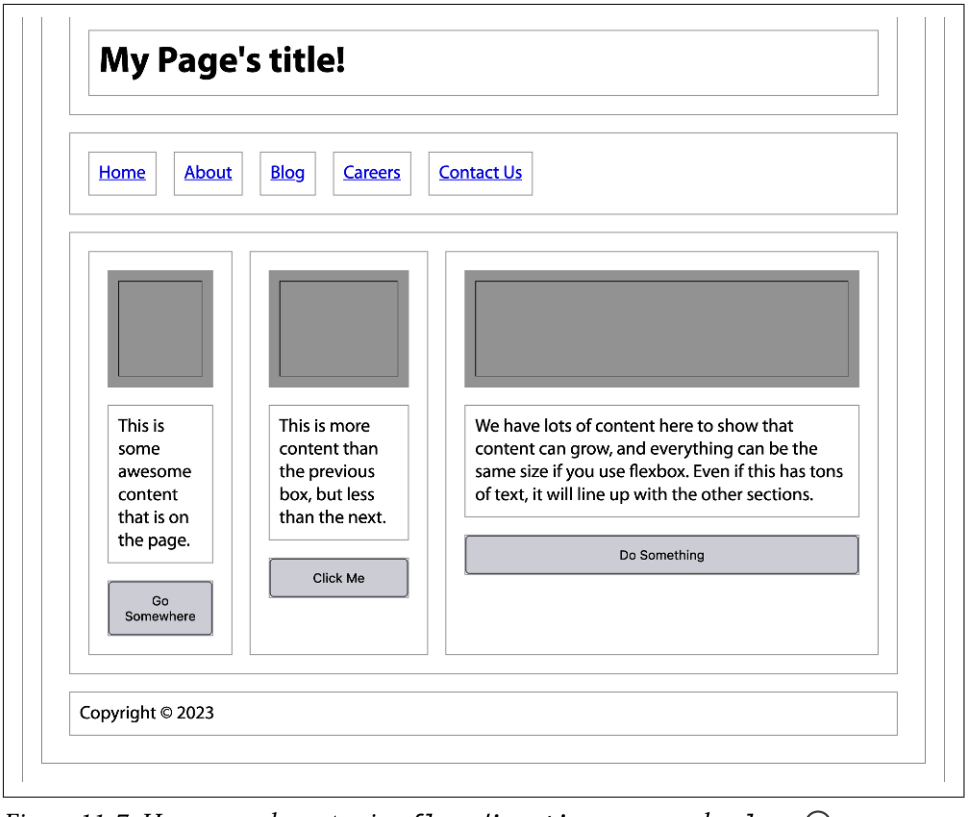


Figure 11-7. Home page layout using `flex-direction: row` and `column`

Yes, elements can be flex items while also being flex containers, as you can see with the navigation, main, and article elements in this case. The `<body>` and `<article>` elements have `column` set as their flex directions, and we let `<nav>` and `<main>` default to `row`. And all that with just two lines of CSS!

To be clear, there's more styling at work in Figure 11-7. Borders, margins, and padding were applied to all the elements, so you can visually differentiate the flex items for the sake of learning (we wouldn't put this less-than-attractive site into production!). Otherwise, all we've done is simply declare the body, navigation, main, and articles as flex containers, making the navigation links, main, article, images, paragraphs, and buttons flex items.

Working with Other Writing Directions

If you're creating websites in English, or another left-to-right (LTR) language, you likely want the flex items to be laid out from left to right, and from top to bottom. The default value `row` will do that. If you're writing in Arabic, or another right-to-left (RTL) language, you likely want the flex items to be laid out from right to left, and from top to bottom. The default value `row` will do that, too.

Using `flex-direction: row` arranges the flex items in the same direction as the text direction, also known as the *writing mode*, whether the language is RTL or LTR. While most websites are presented in left-to-right languages, some sites are in right-to-left languages, and yet others are top-to-bottom. With flexbox, when you change the writing mode, flexbox takes care of changing the flex direction for you.

The writing mode is set by the `writing-mode`, `direction`, and `text-orientation` properties, or by the `dir` attribute in HTML. (These are covered in Chapter 15.) When the writing mode is right to left, the direction of the main-axis—and therefore the flex items within the flex container—will go from right to left when the `flex-direction` is `row`. This is illustrated in Figure 11-8.

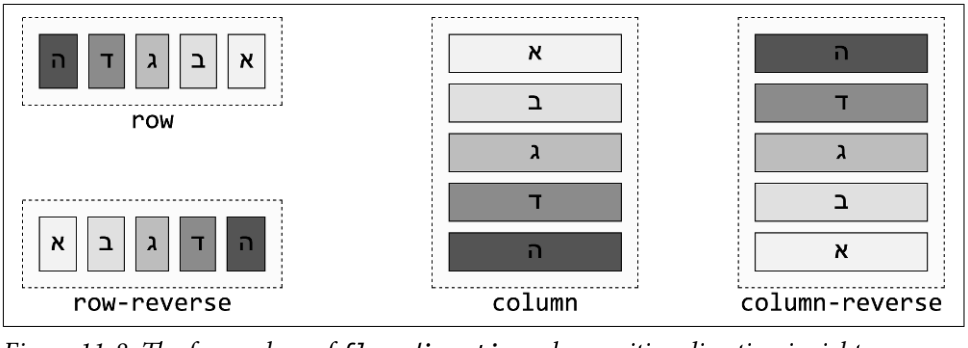


Figure 11-8. The four values of `flex-direction` when writing direction is right to left



If the CSS `direction` value is different from the `dir` attribute value on an element, the CSS property value takes precedence over the HTML attribute. The specifications strongly recommend using the HTML attribute rather than the CSS property.

Vertically written languages include Bopomofo, Egyptian hieroglyphs, Hiragana, Katakana, Han, Hangul, Meroitic cursive and hieroglyphs, Mongolian, Ogham, Old Turkic, Phags Pa, Yi, and sometimes Japanese. These languages are displayed vertically only when a vertical writing mode is specified. If not, all of those languages are treated as horizontal.

For top-to-bottom languages, `writing-mode: horizontal-tb` is in effect, meaning the main-axis is rotated 90 degrees clockwise from the default left to right. Thus, `flex-direction: row` goes from top to bottom, and `flex-direction: column` proceeds from right to left. Figure 11-9 shows the effects of the various `flex-direction` values on the following markup:

```
<ol lang="jp">
    <li>一</li>
    <li>二</li>
    <li>三</li>
    <li>四</li>
    <li>五</li>
</ol>
```

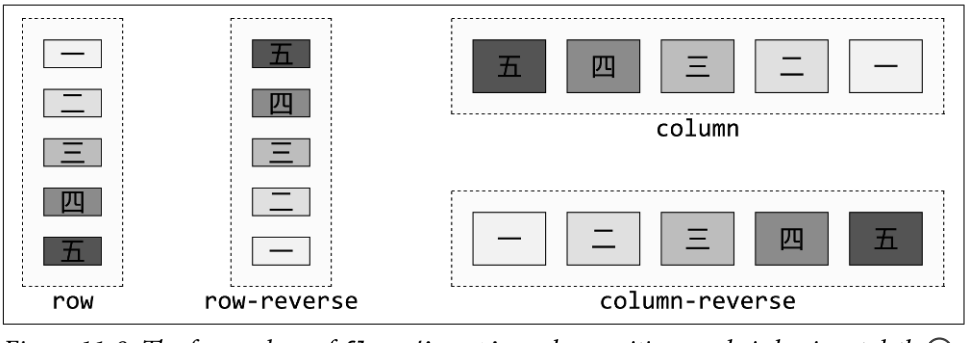


Figure 11-9. The four values of `flex-direction` when writing mode is horizontal-tb

That's right: the rows are vertical, and columns are horizontal. Not only that, but the basic `column` direction is right to left, whereas `column-reverse` runs left to right. That's what comes of applying these values to a top-to-bottom, right-to-left language as we see here.

All right, you've seen various ways flex direction and writing modes interact. But so far, all the examples have shown a single row or column of flex items. What happens when the flex items' *main dimension* (their combined inline sizes for `row` or combined block sizes for `column`) don't fit within the flex container? We can either have them overflow the container or can allow them to wrap onto additional flex lines. Also, we'll later talk about how to allow flex items to shrink (or grow) to fit the container.

Wrapping Flex Lines

If all the flex items don't fit into the main-axis of the flex container, the flex items will not wrap by default, nor will they necessarily resize. Rather, the flex items may shrink if allowed to do so via the flex item's `flex` property (see "Growth Factors and the flex Property" on page 508); otherwise, the flex items will overflow the bounding container box.

You can affect this behavior. The `flex-wrap` property sets whether a flex container is limited to a single line or is allowed to become multiline when needed.

flex-wrap

Values	`nowrap` | `wrap` | `wrap-reverse`
Initial value	`nowrap`
Applies to	Flex containers
Computed value	As specified
Inherited	No
Animatable	No

When the `flex-wrap` property is set to allow for multiple flex lines via `wrap` or `wrap-reverse`, it determines where additional lines of flex items appear: either before or after the original line of flex items.

Figure 11-10 demonstrates the three values of the `flex-wrap` property when the `flex-direction` value is `row` (and the language is LTR). Where these examples show two flex lines, the second line and subsequent flex lines are added along the direction of the cross-axis (in this case, the vertical axis).

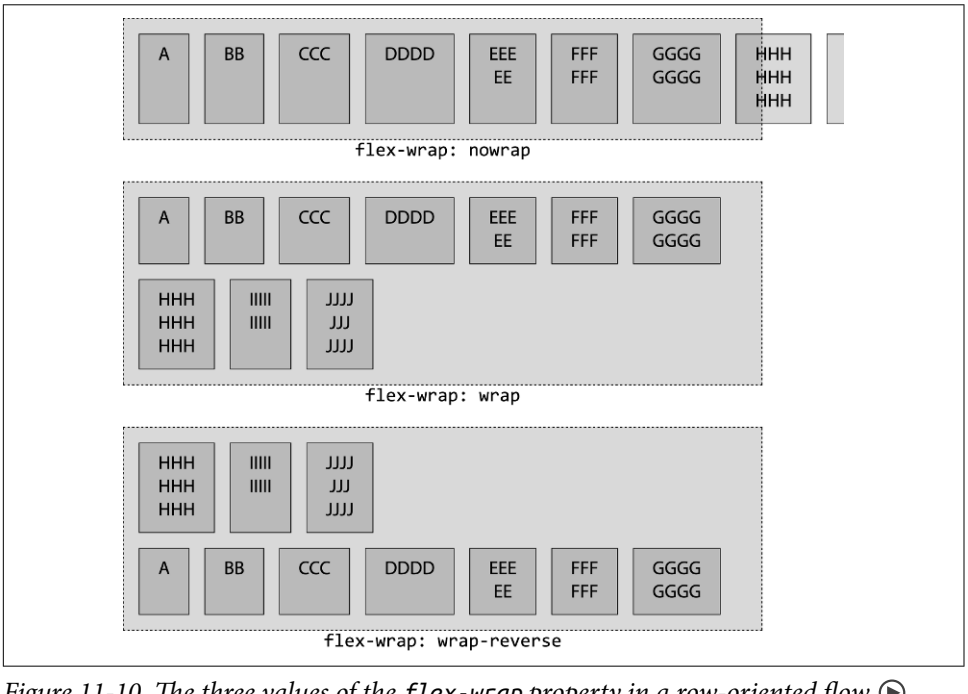


Figure 11-10. The three values of the `flex-wrap` property in a row-oriented flow ▶

When `wrap` is set, the cross-axis is the same as the block axis for `flex-direction: row` and `row-reverse`, and is the same as the inline axis of the language for `flex-direction: column` and `column-reverse`.

The difference is that when `flex-wrap` is set to `wrap-reverse`, the cross-axis direction is reversed: subsequent flex lines are drawn above the previous line in the case of `row` and `row-reverse`, and to the left of the previous column in the case of `column-reverse` (assuming an LTR language such as English).

We'll talk more about axes in just a moment, but first, let's talk about the shorthand property that brings flex direction and wrapping together.

Defining Flexible Flows

The `flex-flow` property lets you define the wrapping directions of the main- and cross-axes, and whether the flex items can wrap to more than one line if needed.

flex-flow

Values	*<flex-direction>* ‖ *<flex-wrap>*
Initial value	`row nowrap`
Applies to	Flex containers
Computed value	As specified
Inherited	No
Animatable	No

The `flex-flow` shorthand property sets the `flex-direction` and `flex-wrap` properties to define the flex container's wrapping and main- and cross-axes.

As long as `display` is set to `flex` or `inline-flex`, omitting `flex-flow`, `flex-direction`, and `flex-wrap` is the same as declaring any of the following three, all of which have the result shown in Figure 11-11:

```
flex-flow: row;
flex-flow: nowrap;
flex-flow: row nowrap;
```

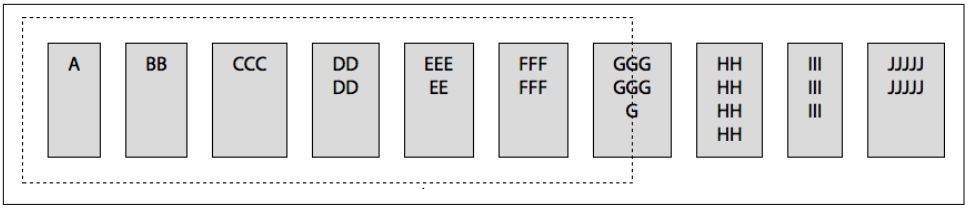


Figure 11-11. A row-oriented unwrapped flex flow ▶

In LTR writing modes, declaring any of the property values just listed, or omitting the `flex-flow` property altogether, will create a flex container with a horizontal main-axis that doesn't wrap. Figure 11-11 illustrates flex items distributed along the horizontal axis, on one line, overflowing a container that's 500 pixels wide.

If instead we want a reverse-column-oriented flow with wrapping, either of these will suffice:

```
flex-flow: column-reverse wrap;
flex-flow: wrap column-reverse;
```

In an LTR language like English, this causes the flex items to flow from bottom to top, starting at the left side, and wrap to new columns in the rightward direction. In a vertical writing mode like Japanese, which is right-to-left when written vertically, the columns would be horizontal, flowing from right to left, and wrap top to bottom.

We've been using terms like *main-axis* and *cross-axis* without really delving into what they mean. It's time to clarify all that.

Understanding Axes

First: flex items are laid out along the main-axis. Flex lines are added in the direction of the cross-axis.

Up until we introduced `flex-wrap`, all the examples had a single line of flex items. In that single line, the flex items were laid out along the main-axis, in the *main direction*, from main-start to main-end. When we added flex wrapping, new flex lines were added along the cross-axis, in the *cross direction*, going from cross-start to cross-end.

As you can see, a lot of terms are used in that paragraph. Here are some quick definitions:

Main-axis
: The axis along which content flows. In flexbox, this is the direction in which flex items are flowed.

Main-size
: The total length of the content along the main-axis.

Main-start
: The end of the main-axis from which content begins to flow.

Main-end
: The end of the main-axis toward which content flows, opposite the main-start.

Cross-axis
: The axis along which flex lines are "stacked." In flexbox, this is the direction in which new lines of flex items are placed, if flex wrapping is permitted.

Cross-size
: The total length of the content along the cross-axis.

Cross-start
: The edge of the cross-axis where blocks begin to be stacked.

Cross-end
: The opposite edge of the cross-axis from the cross-start.

While these terms may sound like logical properties such as `margin-inline-start`, they are not the same thing. Here, the physical direction of each changes depending on the value of the `flex-direction` property. In fact, the meaning of each term in the context of layout depends on the combination of the flex direction, the flex wrapping, and the writing mode. Charting all the combinations for every writing mode would get difficult, so let's examine what they mean for LTR languages.



It's important to understand that direction gets reversed when writing direction is reversed. To make explaining (and understanding) flex layout much simpler, the rest of the explanations and examples in this chapter are based on an LTR writing mode, but will include how writing mode impacts the flex properties and features discussed.

When thinking about `flex-direction`, we know that the flex items will start being laid out along the main-axis of the flex container, starting from the main-start edge and proceeding toward the main-end edge. If the `flex-wrap` property is used to allow the container to wrap when the flex items don't fit onto one line, the flex lines are laid out starting from the cross-start edge and proceeding toward the cross-end edge.

As shown in Figure 11-12, when we have horizontal rows of flex items, the cross-axis is vertical. In these examples, with `flex-flow: row wrap` and `flex-flow: row-reverse wrap` set on horizontal languages, new flex lines are added below preceding flex lines. The cross-size is the opposite of main-size, being height for `row` and `row-reverse` flex directions, and width for `column` and `column-reverse` directions, in both RTL and LTR languages.

By contrast, the `wrap-reverse` value inverts the direction of the cross-axis. Normally for `flex-direction` of `row` and `row-reverse`, the cross-axis goes from top to bottom, with the cross-start on top and cross-end on the bottom. When `flex-wrap` is `wrap-reverse`, the cross-start and cross-end directions are swapped, with the cross-start on the bottom,

cross-end on top, and the cross-axis going from bottom to top. Additional flex lines get added on top of, or above, the previous line.

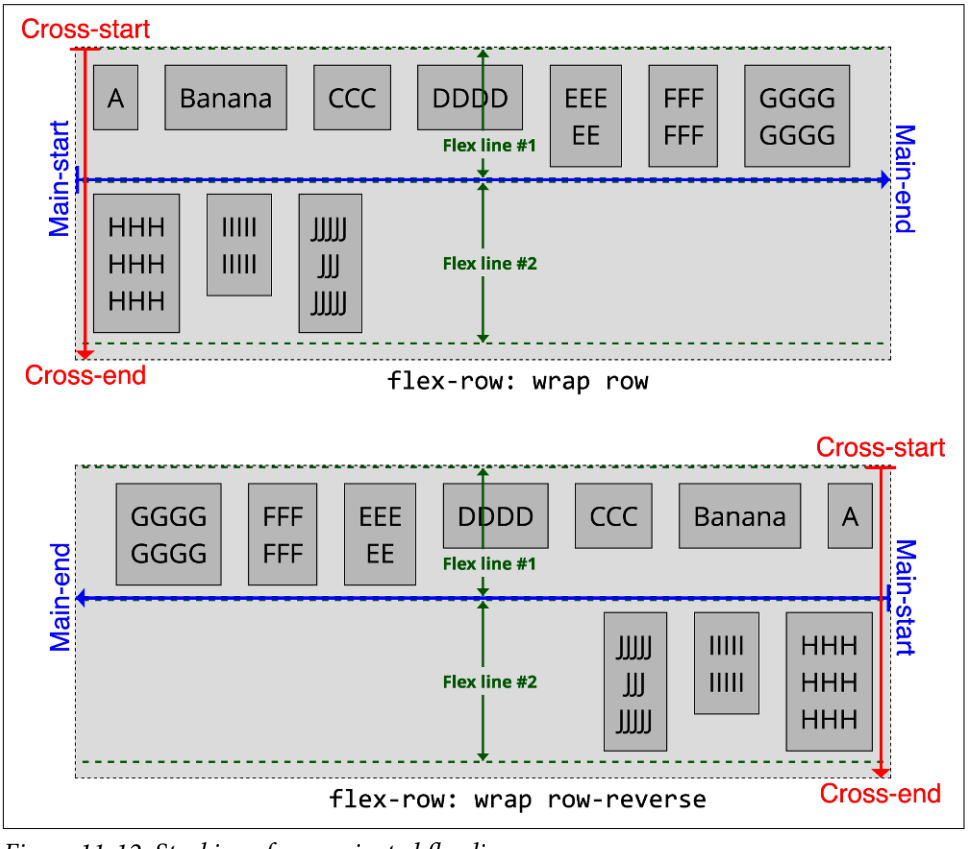


Figure 11-12. Stacking of row-oriented flex lines

If the `flex-direction` is set to `column` or `column-reverse`, by default the cross-axis goes from left to right in LTR languages, with new flex lines being added to the right of previous lines. As shown in Figure 11-13, when `flex-wrap` is set to `wrap-reverse`, the cross-axis is inverted, with cross-start being on the right, cross-end being on the left, the cross-axis going from right to left, and additional flex lines being added to the left of the previously drawn line.

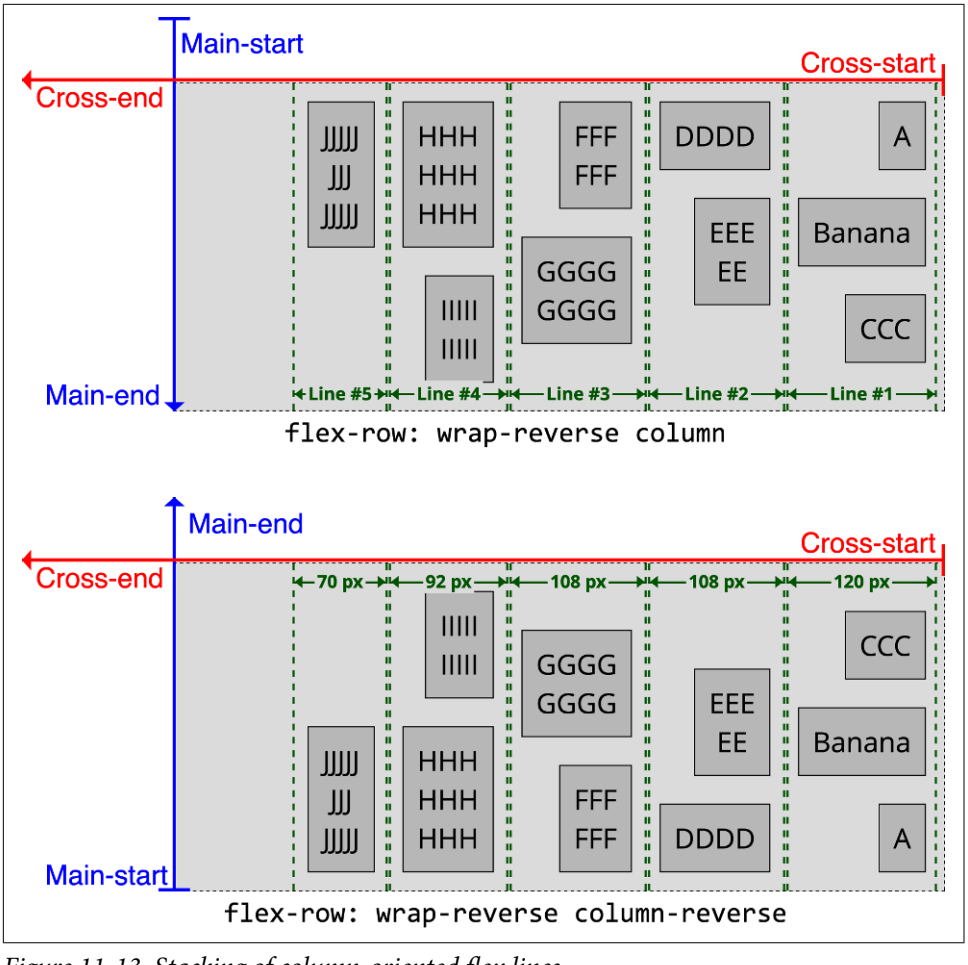


Figure 11-13. Stacking of column-oriented flex lines

Arrangement of Flex Items

In our examples thus far, we've skated past the precise arrangement of the flex items within each flex line, and how that's determined. It might seem intuitive that a row fills in horizontally, but why should all the items huddle toward the main-start edge? Why not have them grow to fill all available space, or distribute themselves throughout the line?

For an example of what we're talking about here, check out Figure 11-14. Notice the extra space at the top left. In this bottom-to-top, right-to-left flow, new flex items get placed above the previous ones, with new wrap lines being placed to the left of each previously filled line.

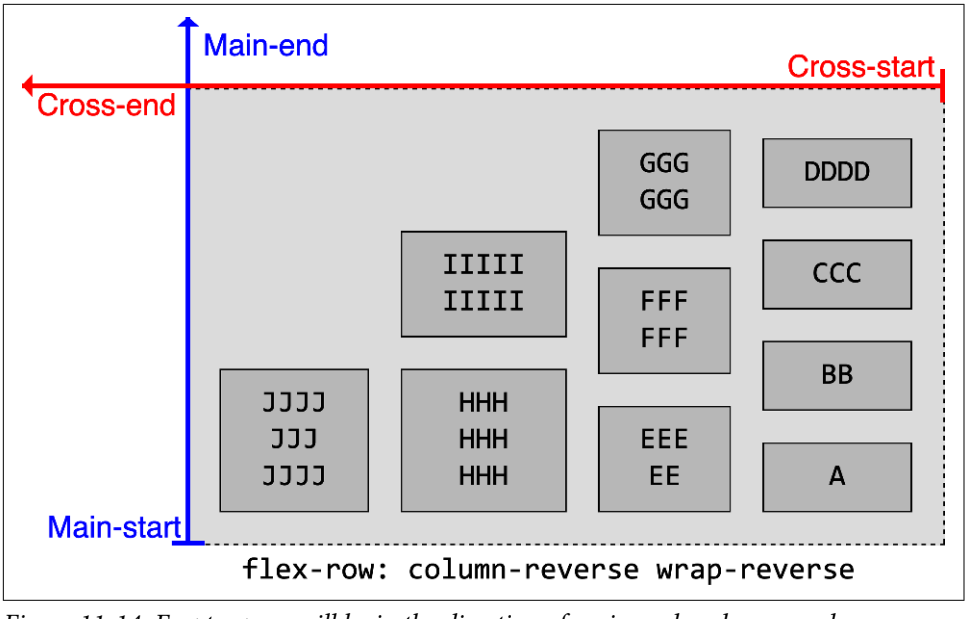


Figure 11-14. Empty space will be in the direction of main-end and cross-end

By default, no matter the values of `flex-flow`, empty space outside the flex items in a flex container will be in the direction of main-end and cross-end, but CSS has properties that allow us to alter that.

Flex Item Alignment

In our examples thus far, whenever the flex items do not completely fill the flex container, the flex items are all grouped toward the main-start on the main-axis. Flex items can be flush against the main-end instead, centered, or even spaced out in various ways across the main-axis.

The flex layout specification provides us with flex container properties to control the distribution of space. The `justify-content` property controls how flex items within a flex line are distributed along the main-axis. The `align-items` property defines the default distribution of the flex items along the cross-axis of each flex line; this global default can be individually overridden with the flex item `align-self` property. When there is more than one flex line and wrapping is enabled, the `align-content` property defines how those flex lines are distributed along the cross-axis of the flex container.

Justifying Content

The `justify-content` property enables us to direct the way flex items are distributed along the main-axis of the flex container within each flex line, and how to handle

situations where information might be lost. This property is applied to the flex container, *not* the individual flex items.

justify-content

Values	`normal` | `space-between` | `space-around` | `space-evenly` | `stretch` | `unsafe` | `safe` ? [`center` | `start` | `end` | `flex-start` | `flex-end` | `left` | `right`]
Initial value	`normal`
Applies to	Flexbox, grid, and multicolumn containers
Computed value	As specified
Inherited	No
Animatable	No

Note the `stretch` value is treated the same as `normal` for flexbox, but not for grid layout.



The `safe` and `unsafe` values, introduced with several other values in CSS Box Alignment Module Level 3, are recognized but not supported in most browsers as of early 2023. This means the value is ignored, but its presence does not render the rest of the declaration invalid.

Figure 11-15 shows the effects of the various values in a writing mode like English.

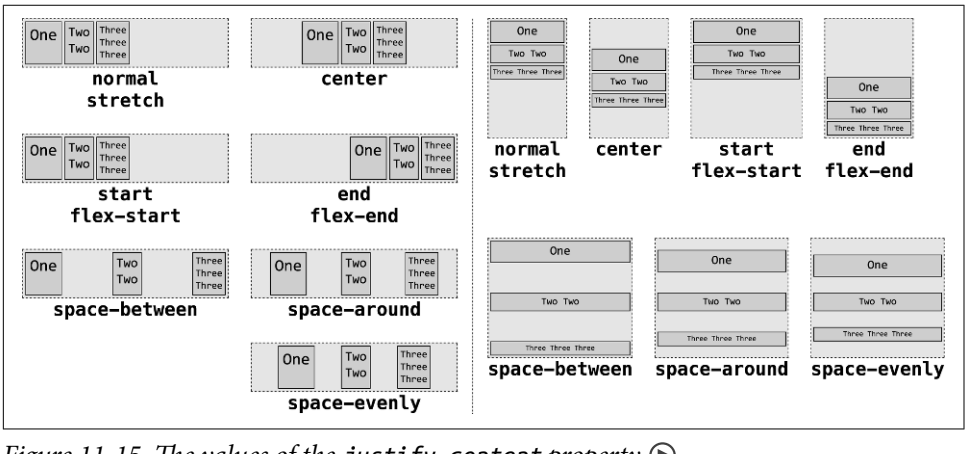


Figure 11-15. The values of the `justify-content` property ▶

With `start` and `flex-start`, flex items are placed flush against main-start. With `end` and `flex-end`, flex items are justified toward main-end. The `center` option groups the items flush against each other, centered in the middle of the main-axis. The `left` and `right` options place items flush against the named sides of the box, regardless of the actual axis direction.

The `space-between` value puts the first flex item on a flex line flush with main-start and the last flex item in each flex line flush with main-end, and then puts an equal amount of space between every pair of adjacent flex items. The `space-evenly` value takes the leftover space and splits it so that every gap is the same length. This means the spaces at the start and end edges of the main-axis will be the same size as the spaces placed between flex items.

By contrast, `space-around` splits up the leftover space and then applies half of each portion to each flex item, as if there were noncollapsing margins of equal size around each item. Note that this means the space between any two flex items is twice that of the spaces at the main-start and main-end of the flex line.

The `stretch` value has no effect as a value of `justify-content` in flexbox. As you'll see in the next chapter, when placed on a grid container, it causes grid items to grow in size until they take up all available space in the main-axis direction.



We'll cover `safe` and `unsafe`, which vary how the browser should handle items that overflow the container along the cross-axis, in "Safe and unsafe alignment" on page 487.

Justifying and overflow

If flex items are not allowed to wrap to multiple lines and overflow their flex line, the value of `justify-content` influences the way the flex items will overflow the flex container.

Setting `justify-content: start` or `flex-start` explicitly sets the default behavior of grouping the flex items toward main-start, placing the first flex item of each flex line flush against the main-start side. Each subsequent flex item then gets placed flush with the preceding flex item's main-end side. (Remember, the location of the main-start side depends on the flex direction and writing mode.) If there isn't enough room for all the items and wrapping is not allowed, the items will overflow the main-end edge. This is illustrated in Figure 11-16.

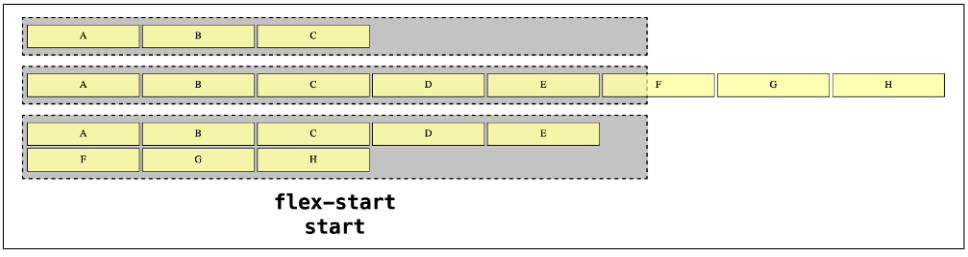


Figure 11-16. The effects of `start` content justification ▶

The mirror of this is setting `justify-content: end` or `flex-end`, which puts the last flex on a line flush against the main-end with each preceding flex item being placed flush with the subsequent item. In this case, if the items aren't allowed to wrap, and if there isn't enough room for all the items, the items will overflow on the main-start edge, as illustrated in Figure 11-17.

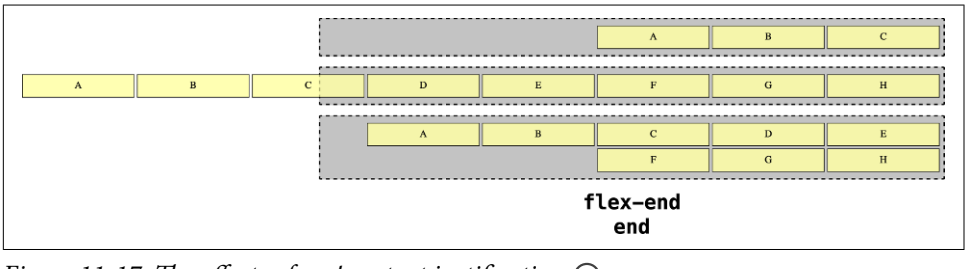


Figure 11-17. The effects of `end` content justification ▶

Setting `justify-content: center` will pack all the items flush against each other, and center them on the center of the flex line instead of packing them against the main-start or main-end. If there isn't enough room for all the items and they aren't allowed to wrap, the items will overflow evenly on both the main-start and main-end edges.

Figure 11-18 illustrates these effects.

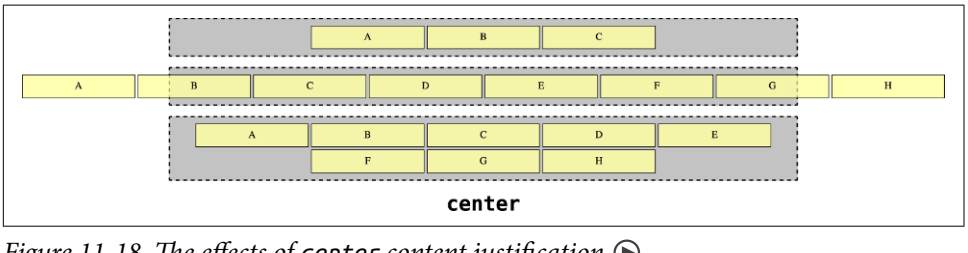


Figure 11-18. The effects of `center` content justification ▶

As the `left` and `right` values, they always start packing from the left or right edge of a row, regardless of axis directions. Thus, `justify-content: left` will always justify row-based content to the left, whether the main-axis goes left to right or right to left. In column-based content, `left` is treated the same as `start`, and `right` the same as `end`. Any overflow will occur on the opposite side from where the packing started; that is, flex items will overflow on the right edge for `justify-content: left` and on the left edge for `right`.

With those relatively simple cases covered, let's look at values that alter space between and around flex items, and compare them to their wrapped cases. Note that if flex items are allowed to wrap onto multiple lines, the space around each flex item is based on the available space in their specific flex line only, and will not (in most cases) be consistent from one line to the next.

Setting `justify-content: space-between` puts the first flex item flush with main-start and the last flex item on the line flush with main-end, and then puts an equal amount of space around each flex item, until the flex line is filled (see Figure 11-19). If we have three flex items, the same amount of space will be between the first and second items as between the second and third, but there will be no extra empty space between the main-start and main-end edges of the container and the first and last flex items in the line. This means if a line has only one flex item, it will be flush with the main-start edge, not centered. If there isn't enough space to fit all the flex items and they aren't allowed to wrap, the items will overflow on the main-end edge, yielding an effect visually indistinguishable from `justify-content: start`.

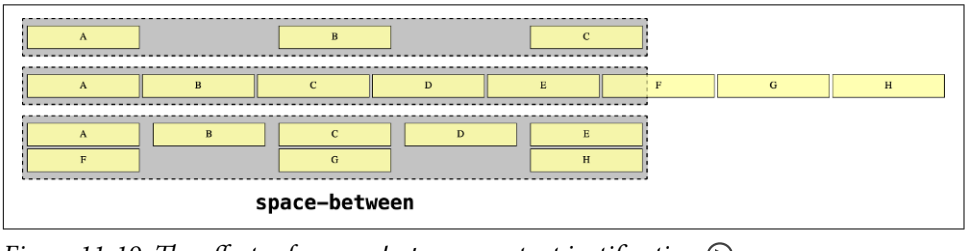


Figure 11-19. The effects of `space-between` content justification

Setting `justify-content: space-around` evenly distributes the extra space on the line around each of the flex items, as if noncollapsing margins of equal size were around each element on the main-dimension sides (Figure 11-20). Thus, there will be twice as much space between the first and second item as there is between main-start and the first item, and main-end and the last item. If there isn't enough room for all the items and they aren't allowed to wrap, the items will overflow evenly on both the main-start and main-end edges.

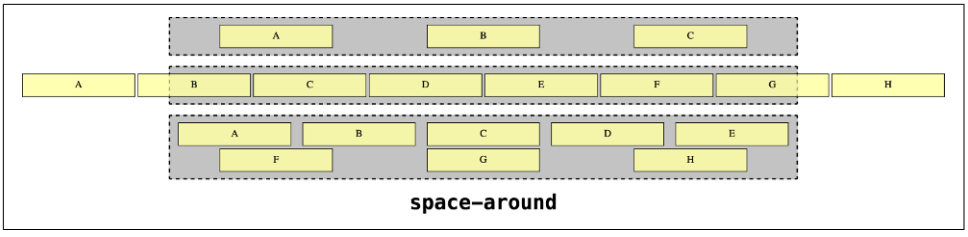


Figure 11-20. The effects of `space-around` content justification ▶

Setting `justify-content: space-evenly` means the user agent counts the items, adds one, and then splits any extra space on the line by that many (i.e., if we have five items, the amount of space is split into six equal-size portions); see Figure 11-21. One portion of the space is placed before each item on the line, as if it were a noncollapsing margin, and the last portion is placed after the last item on the list. Thus, the same amount of space will appear between the first and second item as there is between main-start and the first item, and main-end and the last item. If there isn't enough room for all the items and they aren't allowed to wrap, the items will overflow evenly on both the main-start and main-end edges.

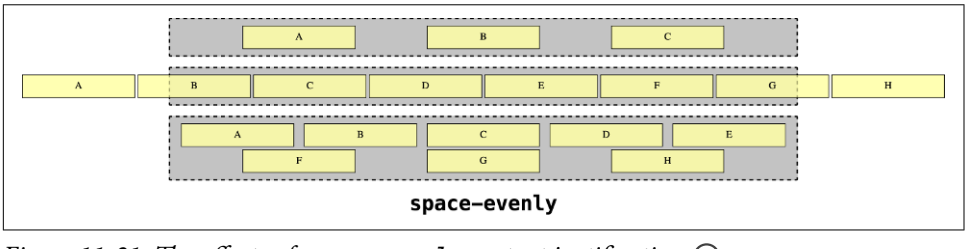


Figure 11-21. The effects of `space-evenly` content justification ▶

The `stretch` value has no effect when set as the value of `justify-content` on a flex container, and is treated the same as `normal`. As you'll see in the next chapter, when placed on a grid container, it causes the grid items to grow in size until they take up all available space in the main-axis direction.

And finally, `justify-content: normal` is treated the same as `justify-content: start`. This is the case for historical reasons too boring and lengthy to get into here, but what it means is that the default value of `justify-content` is essentially `start`, even if it is technically `normal`.

Note in the previous few examples that when flex items are allowed to wrap onto multiple lines, the space around each flex item is based on the available space in its specific flex line only, and will not (in many cases) be consistent from one line to the next.

justify-content example

We took advantage of the default value of `justify-content` in Figure 11-2, creating a left-aligned navigation bar. By changing the default value to `justify-content: flex-end`, we can right-align the navigation bar in English:

```
nav {
  display: flex;
  justify-content: flex-start;
}
```

Note that `justify-content` is applied to the flex container. If we'd applied to the links themselves, using something like `nav a {justify-content: flex-start;}`, no alignment effect would have occurred.

A major advantage of `justify-content` is that when the writing direction changes (say, for RTL writing modes), we don't have to alter the CSS to get the tabs where they need to go. The flex items are always grouped toward main-start when `flex-start` is applied; in English, main-start is on the left. For Hebrew, main-start is on the right. If `flex-end` is applied and the `flex-direction` is `row`, then the tabs go to the right side in English, and the left side in Hebrew, as shown in Figure 11-22.

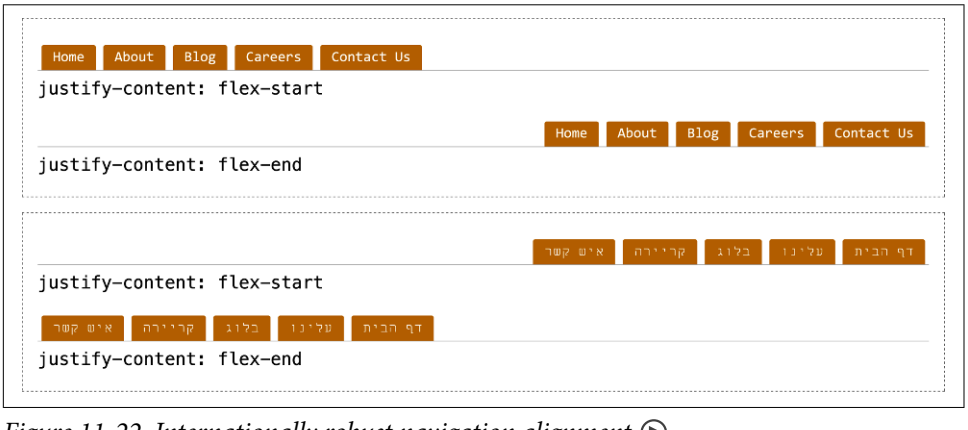


Figure 11-22. Internationally robust navigation alignment ▶

This may seem like the main-start and main-end are similar to inline-start and inline-end in logical properties. This will feel true when `flex-direction` is set to `row`. With `flex-direction: row-reverse`, however, the main-start and main-end get switched, but the inline-start and inline-end don't, because the inline directions for the flex items remain the same even if their flex order changes.

We could have centered that navigation, as shown in Figure 11-23:

```
nav {
  display: flex;
  justify-content: center;
}
```



Figure 11-23. Changing the layout with one property value pair ▶

All the flex items we've shown thus far are a single line tall, and therefore are the same size in the cross dimension as their sibling flex items. Before discussing wrapping flex lines, we need to discuss aligning items of differing dimensions along the cross-axis, which is, appropriately enough, called *aligning*.

Aligning Items

Whereas `justify-content` defines how flex items are aligned along the flex container's main-axis, the `align-items` property defines how flex items are aligned along its flex line's cross-axis. As with `justify-content`, `align-items` is applied to flex containers, not individual flex items.

align-items

Values	`normal` | `space-between` | `space-around` | `space-evenly` | `stretch` | [`first` | `last`]? && `baseline` | [`safe` | `unsafe`]? `center` | `start` | `end` | `flex-start` | `flex-end`
Initial value	`normal`
Applies to	Flex containers
Computed value	As specified
Inherited	No
Animatable	No

Note the `normal` behaves as `stretch` for flexbox.



While `align-items` sets the alignment for all the flex items within a container, the `align-self` property enables overriding the alignment for individual flex items, as you'll see in "The align-self property" on page 488.

In Figure 11-24, note how the flex items are arranged with respect to the cross-axis. (The cross-axis is the block axis for row-flowed flex containers, and the inline axis for column-flowed flex containers.)

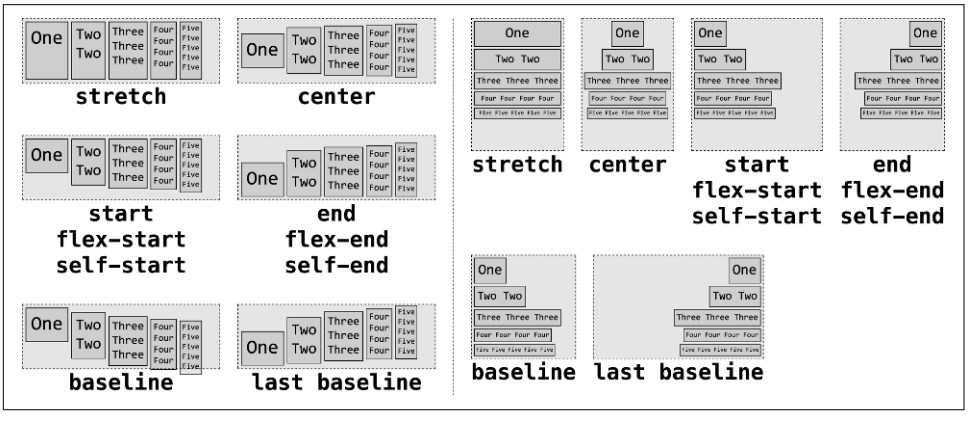


Figure 11-24. The values of the `align-items` property for both rows and columns

The default value, `normal`, is treated as `stretch` in flexbox.

For `stretch`, the cross-start edge of each flex item is placed against the cross-start edge of the container, and the cross-end edges are also placed against the cross-end edge of the container. This happens regardless of the size of the content inside each flex item, so a flex item with short content (such as "One") will still have its element box fill out the cross-axis size of the flex container.

With the `center` value, by contrast, the element box is just as large as it needs to be to contain the content along the cross-axis, and no bigger. The cross-start and -end edges of the flex items are placed the same distance away from the cross-start and -end edges of the container, thus centering the flex item's box within the flex container along the cross-axis.

For the various `start` and `end` values, the cross-start or -end edges of the flex items are all snugged up against the respective edge of the flex container. There are so many ways to say `start` and `end`, mostly for historical reasons that are too lengthy and painful to get into here.

Notice that when the items are aligned to the start or end of the cross-axis, their inline sizes are (by default) exactly as big as their content needs to be, and no wider. It's as if their `max-width` was set to `max-size`, so that extra content can wrap to multiple lines within the flex item, but if no wrapping is needed, the element's inline size won't fill out the entire flex container's inline size. This is a default behavior of flex items, so if you want flex elements to fill out the entire inline size of the flex container, the way block boxes fill out their containing block, use the `stretch` value instead.

With `baseline`, the flex items' first baselines are aligned with one another when they can do so, which is to say, when the `flex-direction` is `row` or `row-reverse`. Because the font size of each flex item differs, the baseline of each line in every flex item differs. The flex item that has the greatest distance between its first baseline and its cross-start side will be flush against the cross-start edge of the line. The other flex items will be placed so that their first baselines line up with the first baseline of the flex item that's flush against the cross-start edge (and thus each other's first baselines). When `align-items: last baseline;` is set, the inverse occurs. The flex item with the greatest distance between its last baseline and the cross-end side will be flush against the cross-end edge of the line. The other flex items will be placed with their last baseline lined up with the last baseline of the flex item that's flush against the cross-end edge, unless overridden by `align-self` (see "The align-self property" on page 488). Since there isn't a way to align baselines in a columnar flow, `baseline` is treated like `start` in these contexts, or `end` in the case of `last baseline`.

Flex item margins and alignment

Now you have a general idea how each value behaves, but there's a bit more to it than that. In the multiline `align-items` figures that follow, the following styles have been applied:

```
flex-container {
  display: flex;
  flex-flow: row wrap;
  gap: 1em;
}
flex-item {border: 1px solid;}
.C, .H {margin-top: 1.5em;}
.D, .I {margin-bottom: 1em;}
.J {font-size: 3em;}
```

For each flex line, the cross-start and cross-end edges have been drawn in as a red dotted and blue dashed line, respectively. The C, H, D, and I boxes have added top or bottom margins. We've added a `gap` (which will be discussed a bit later in the chapter) between the flex items to make the figures more legible, which doesn't affect the impact of the `align-items` property in this case. The J box has its font size increased, which also increases its line height. (This will come into play when we discuss the `baseline` value.)

The effects of these margins on both the `stretch` and `center` alignments can be seen in Figure 11-25.

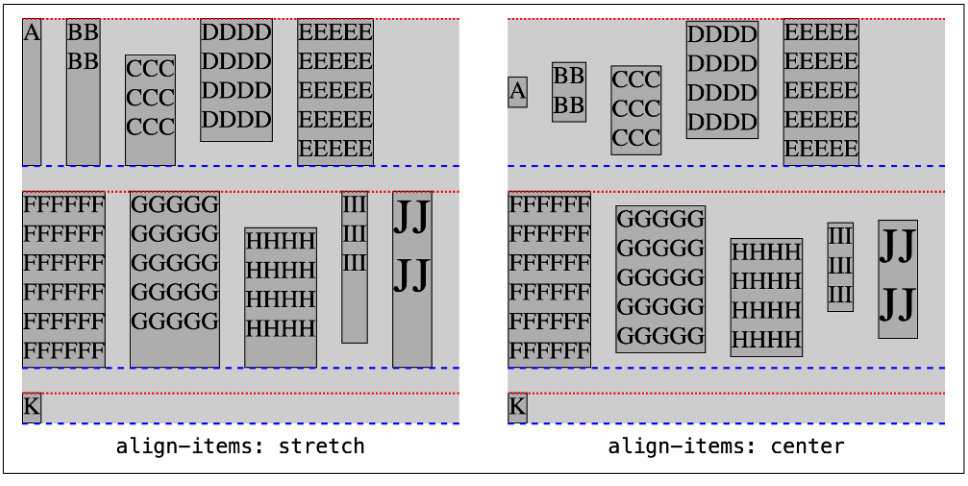


Figure 11-25. The effect of margins on cross-axis alignment

The `stretch` value, as its name implies, stretches all "stretchable" flex items to be as tall or wide as the tallest or widest flex item on the line. A stretchable flex item is one that does not have a non-`auto` value set for any of the sizing properties along the cross-axis. In Figure 11-25, that would be the `block-size`, `min-block-size`, `max-block-size`, `height`, `min-height`, and `max-height` properties. If all are set to `auto`, the flex item is stretchable. If not, it is not.

Assuming a flex item is stretchable, its cross-start edge will be flush with the flex line's cross-start edge, and its cross-end edge will be flush with the flex line's cross-end edge. The flex item with the largest cross-size will remain its default size, and the other flex items will grow to the size of that largest flex item.

What Figure 11-25 shows us is that it's the outer edge of the flex items' *margins* that will be flush with cross-start and cross-end, not their border edges. This is demonstrated by items C, D, H, and I, which appear smaller than the other flex items on their flex lines. They're not, though. It's just that their margins, which are always fully transparent, take up some of the stretching space.



If a flex container's cross-size is constrained, the contents may overflow the flex container's cross-start and/or cross-end edge. The direction of the overflow is not determined by the `align-items` property, but rather by the `align-content` property, discussed in "Aligning Flex Lines" on page 489. The `align-items` property aligns the flex items within the flex line and does not directly impact the overflow direction of the flex items within the container.

Baseline alignment

The `baseline` values are a little more complicated. CSS has two possible baseline alignments, represented by `first baseline` and `last baseline`. You can also use the value `baseline`, which is equivalent to `first baseline`.

With `baseline` (and `first baseline`), the flex items in each line are all aligned at the lowest first baseline. For each flex line, the flex item with the biggest distance between its baseline and its cross-start margin edge has that margin edge placed flush against the cross-start edge of the line, and all other flex items' baselines are lined up with the baseline of that flex item.

To understand this, take a look at the first set of flex items in Figure 11-26, the ones labeled `baseline` (and `first baseline`). For each flex line, the cross-start and -end edges are marked with solid red and blue lines, respectively. The baseline to which the items in each line are aligned is marked with a dotted line, and the element whose baseline is taken as the prime baseline has a lighter background and red text.

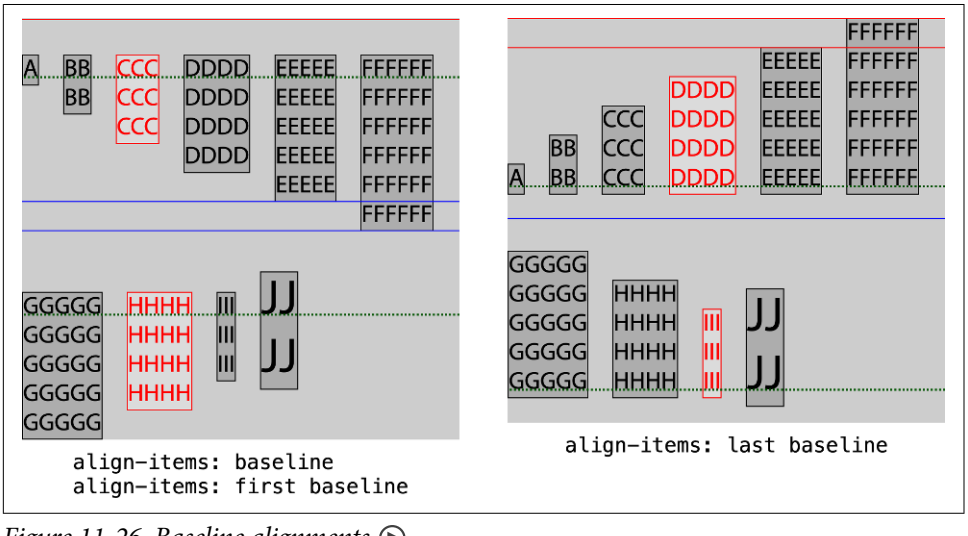


Figure 11-26. Baseline alignments ▶

In the first line (A through E), it is the C box whose first baseline is used. This is because the C box has a top margin, so its first baseline is the farthest from the cross-start edge of the flex line. All the other boxes (A, B, D, and E) have their first baselines aligned with the first baseline of C.

In the second line (F through J), H's first baseline is used—again, because of its top margin—and so the F, G, I, and J boxes have their first baselines aligned with H's. Here, we can also see how the J box has its first baseline aligned with all the others, despite its much bigger font size.

Similar things happen for the flex items labeled with `last baseline`, only here, the dominating factors are bottom margins. The D box in the first line has a bottom margin, as does the I box in the second line. In both cases, their last baselines are the farthest away from the cross-end edge of the line, and so all the other flex items in their rows have their last baselines aligned with the last baselines of D and I. The dotted lines show the placements of the last baselines in each flex line.

In many cases, `first baseline` will look like `start` (and its equivalents, such as `flex-start`), and `last baseline` will look like `end`. For example, had C lacked a top margin in Figure 11-26, all the items in that first line would have been visibly flush against the top of the flex line, instead of pushed away from it. Anytime flex items have different margins, borders, padding, font sizes, or line heights on their cross-start side, the `start` and `first baseline` will differ. Similarly, any cross-end margins, borders, etc. will create a difference between the results of `last baseline` and `end`.

Any of the baseline values can become `start` when the baselines of the flex items are parallel to the cross-axis. For example, suppose we take the flex containers in Figure 11-26 and change them to `flex-direction: column`. Now the cross-axis, like the baselines of the English text within, is horizontal. Since there's no way to create an offset from the cross-start edge of the columns that will align the text baselines, `baseline` is treated exactly as if it were `start` instead; or `end`, in the case of last baseline.

Safe and unsafe alignment

In all the previous examples, we let the flex containers be whatever size they needed to be to contain the flex lines; that is, we left them at `block-size: auto` (or `height: auto`, in old-school CSS terminology). But what happens if the block size of a flex container is constrained in some way, perhaps by the size of a grid track or an explicit block size value being given? In these situations, the `safe` and `unsafe` keywords come into play.

If `safe` alignment is specified, then anytime a flex item would overflow the flex container, the flex item is treated as though its `align-self` were set to `start`. That would look something like this:

```
flex-container {display: flex; height: 10em;
     align-items: safe first baseline;}
```

On the other hand, if `unsafe` is used, the alignment of flex items is honored no matter what that means in terms of overflowing the flex container.

If you're wondering which is the default, the answer is neither. Instead, when neither safe nor unsafe alignment has been declared, browsers should default to `unsafe` behavior *unless* this would cause flex items to overflow the scrollable area of their nearest ancestor's scroll container, in which case they should align to the cross-axis edge farthest away from the edge they would otherwise overflow. Figure 11-27 shows some examples.



As of late 2022, only Firefox browsers fully support the `safe` and `unsafe` keywords, and they have to be written first in the value (as shown in this section) even though the formal syntax for the property does not require this placement. All other evergreen browsers recognize these keywords as valid, but they have no impact on the layout.



Figure 11-27. `safe` versus `unsafe` alignments

The align-self property

If you want to change the alignment of one or more flex items, but not all, you can include the `align-self` property on the flex items you would like to align differently. This property takes the same values as `align-items` and is used to override the `align-items` property value on a per-flex-item basis.

align-self

Values	`auto` | `normal` | `stretch` | [`first` | `last`]? && `baseline` | [`unsafe` | `safe`]? [`center` | `start` | `end` | `self-start` | `self-end` | `flex-start` | `flex-end`]
Initial value	`auto`
Applies to	Flex items
Inherited	No
Percentages	Not applicable
Animatable	No

You can override the cross-axis alignment of any individual flex item with the `align-self` property, as long as it's represented by an element or pseudo-element. You cannot override the alignment for anonymous flex items (non-empty text node children of flex containers). Their `align-self` always matches the value of `align-items` of their parent flex container.

The default value of `align-items` is `stretch`, but let's make that explicit in the following code, which will let us set different `align-self` values for the second flex item, as illustrated in Figure 11-28:

```
.flex-container {align-items: stretch;}
.flex-container .two {align-self: var(--selfAlign);}
```

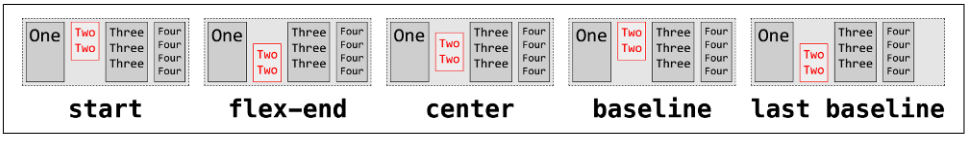


Figure 11-28. Changing individual flex-item alignments ▶

All these flex items have `align-self`'s default value of `auto`, meaning they inherit the alignment (in this case, `stretch`) from the container's `align-items` property. The exception in each example is the second flex item, which has been given the `align-self` value shown underneath.

As we said, all the values of `align-items` can be used for `align-self`, including the values for first and last baseline alignment, `safe` and `unsafe` alignment, and so on.

Aligning Flex Lines

In nearly all the previous examples, the flex container's cross-size was always as tall as it needed to be: no `block-size` or `height` was declared on the container, so it defaulted to `height: auto`. Because of this, the flex container grew to fit the content.

Had the cross-size of the container been set to a specific size, there may have been extra space at the cross-end, or not enough space to fit the content. In such cases, CSS allows us to control the overall placement of flex lines with the `align-content` property.

align-content

Values	`normal` | [`first` | `last`]? && `baseline` | `space-between` | `space-around` | `space-evenly` | `stretch` | [`unsafe` | `safe`]? [`center` | `start` | `end` | `flex-start` | `flex-end`]
Initial value	`normal`
Applies to	Multiline flex containers
Computed value	As specified
Inherited	No
Animatable	No

The `align-content` property dictates how any extra cross-direction space in a flex container is distributed between and around flex lines. Although the values and concepts are largely the same, `align-content` is different from the previously discussed `align-items` property, which dictates flex item positioning within each flex line.

Think of `align-content` as similar to the way `justify-content` aligns individual items along the main-axis of the flex container, but it does so for flex lines with regard to the cross-axis of the container. This property applies to multiline flex containers, having no effect on nonwrapping and otherwise single-line flex containers.

Consider the following CSS as a base and assume the flex items have no margins:

```
.flex-container {
  display: flex;
  flex-flow: row wrap;
  align-items: flex-start;
  border: 1px dashed;
  height: 14em;
  background-image: url(banded.svg);
}
```

Figure 11-29 demonstrates the possible values of the `align-content` property, as used in conjunction with that CSS. We've concentrated on the primary alignment values, and left out examples of things such as safe and unsafe alignment as well as the first and last baseline alignments.

With a height of 14 ems, the flex container is taller than the default combined heights of the three flex lines. Given the larger text of some flex items and the various bits of padding and borders, each flex container in Figure 11-29 has approximately 3 ems of leftover space.

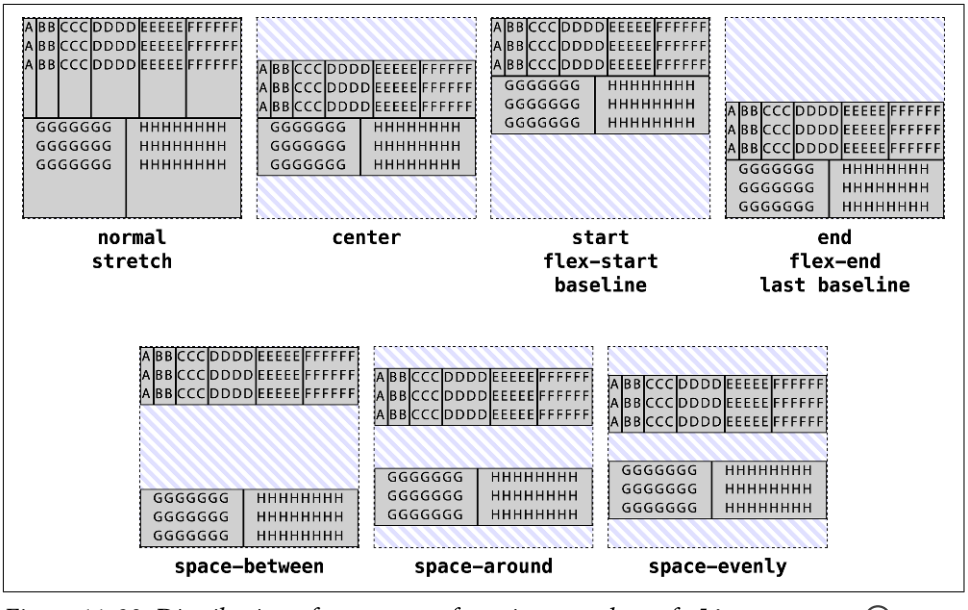


Figure 11-29. Distribution of extra space for primary values of `align-content`

With the values `normal`, `stretch`, `center`, `start`, `flex-start`, `end`, and `flex-end`, the free space is distributed outside the flex lines, as illustrated in Figure 11-29. These act in the same ways as they do for `align-items`. With the value `stretch`, the extra space is evenly distributed to all the flex lines, increasing their cross-size until their edges touch. For the others, the flex lines are kept together, with the leftover space placed to one side or another.

For the remaining values, the flex lines are pushed apart and the leftover space distributed in various ways. Let's assume the approximately 3 ems of leftover space is equal to 120 pixels. (It's big text, OK?)

Given `space-between`, about 60 pixels of space is between each adjacent pair of flex lines, each half of the leftover 120 pixels. With `space-around`, the space is evenly distributed around each line: the 120 pixels are split into three pieces, since there are three flex lines. This puts 20 pixels of noncollapsed space (half of 40 pixels) on the cross-start and cross-end sides of each flex line, so we have 20 pixels of extra space at the cross-start and cross-end sides of the flex container, and 40 pixels of space between adjacent flex lines.

For `space-evenly`, there are four spaces to insert: one before each flex line, and an extra space after the last flex line. With three lines, that means four spaces, with 30 pixels for each space. That places 30 pixels of space at the cross-start and cross-end sides of the flex container, and 30 pixels between adjacent flex lines.

Continuing this example for the stretch value, you'll note that the `stretch` value is different: with `stretch`, the lines stretch with the extra space evenly distributed among the flex lines rather than between them. In this case, 40 pixels are added to each of the flex lines,

causing all three lines to grow in height by an equal amount—that is, the extra space is divided equally, not proportionally, with the exact same amount added to each.

If there isn't enough room for all the lines, they will overflow at cross-start, cross-end, or both, depending on the value of the `align-content` property. This is shown in Figure 11-30, where the dotted box with a light-gray background represents a short flex container. (A little bit of inline padding was added to each flex container to make it more obvious where it starts and ends.)

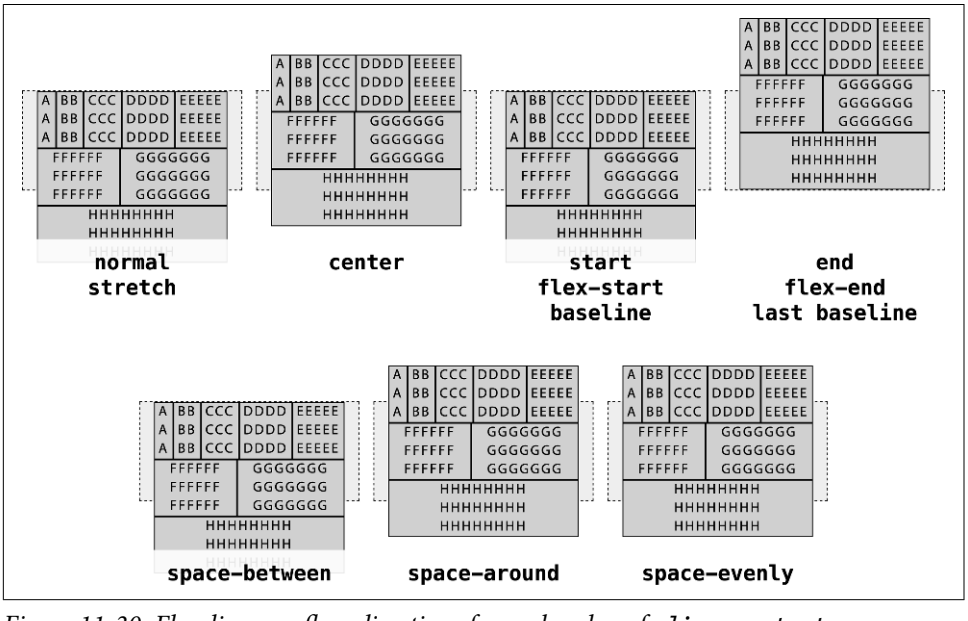


Figure 11-30. Flex-line overflow directions for each value of `align-content`

The only difference in the CSS between this and Figure 11-29 is the height of the flex container. Here, the flex containers have been reduced to a height of 7 ems, so as to create flex containers not tall enough to encompass all their flex lines (which, as you may recall, total around 10 ems in height).

When the flex lines overflow the flex container, the `align-content` values `normal`, `stretch`, `start`, `flex-start`, `baseline`, `last baseline`, and `space-between` cause them to overflow on the cross-end side, whereas the values `center`, `space-around`, and `space-evenly` evenly overflow both the cross-end and cross-start sides. Only `align-content: end` and `flex-end` cause flex lines to overflow on just the cross-start side.

Keep in mind that these values are not top- or bottom-centric. If the cross-axis goes upward, `align-content: flex-start` will start aligning flex lines from the bottom and work upward from there, potentially overflowing the top (cross-end) edge. For that matter, when the flow direction is columnar, the cross-axis will be horizontal, in which case the cross-start and -end edges will be the right or left edges of the flex container.

Using the place-content Property

CSS offers a shorthand property that collapses `align-content`, which we just covered, and `justify-content`.

place-content

Values	*<align-content>* *<justify-content>*?
Initial value	`normal`
Applies to	Block, flex, and grid containers
Computed value	See individual properties
Inherited	No
Animatable	No

You can supply either one or two values. If you supply one, `place-content` acts as if you had set both `align-content` and `justify-content` to the same value. In other words, the following two rules are equivalent:

```
.gallery {place-content: center;}
.gallery {align-content: center; justify-content: center;}
```

The exception to this behavior occurs if the value is baseline-related, such as `first baseline`. In that case, the value for `justify-content` is set to `start`, making the following two rules equivalent:

```
.gallery {place-content: last baseline;}
.gallery {align-content: last baseline; justify-content: start;}
```

If two values are given, the second is the value of `justify-content`. Thus, the following two rules are equivalent:

```
.gallery {place-content: last baseline end;}
.gallery {align-content: last baseline; justify-content: end;}
```

That's pretty much all there is to `place-content`. If you'd rather align and justify content by using a single shorthand property, `place-content` does that. Otherwise, use the individual properties separately.

Two more `place-` shorthand properties are covered in Chapter 12.

Opening Gaps Between Flex Items

Flex items are, by default, rendered with no space held open between them. Space can appear between items thanks to values of `justify-content` or by adding margins to flex items, but these approaches are not always ideal. For example, margins can lead to flex line wrapping when it isn't actually needed, and even using `justify-content` values like `space-between` can result in having no space separating the items. It would be easier if there was a way to define what are essentially minimum gap sizes, and thanks to the gap properties, there is.

row-gap, column-gap

Values	`normal` | [<*length*> | <*percentage*>]
Initial value	`normal`
Applies to	Flex, grid, and multicolumn containers
Computed value	As specified for `normal`; otherwise, the computed length value
Inherited	No
Animatable	Yes (for length values)

Each of these properties inserts space of the declared size between adjacent flex items. This space is often referred to as a *gutter*. For historical reasons, the default value, `normal`, equates to 0 pixels (no space) in flexbox and grid containers, and `1` em in multicolumn layout. Otherwise, you can supply a single length or percentage value.

Suppose we have a set of flex items that will wrap to multiple flex lines, and we want to open a 15-pixel gap between the flex lines. Here's what that CSS would look like, illustrated in Figure 11-31:

```
.gallery {display: flex; flex-wrap: wrap; row-gap: 15px;}
```

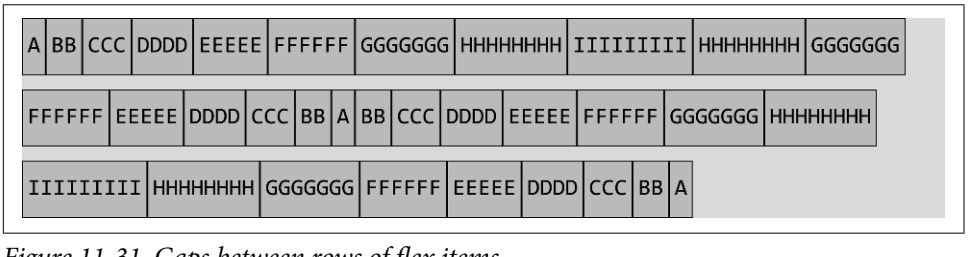


Figure 11-31. Gaps between rows of flex items

No margins are set on the flex items, to be clear. Exactly 15 pixels of space is between each flex line (row), thanks to the value of `row-gap`. In essence, `row-gap` acts as if it were called `block-axis-gap`, so if the writing mode were changed to something like `vertical-rl`,

thus making the block axis horizontal, the rows would flow top to bottom, and the gaps between them would be to their right and left sides (which are their block-start and block-end sides).

Note that there are gaps only between rows: there are no gaps placed between the flex items and the block-start and -end edges of the flex container. If you want to open gaps of the same size along those container edges, you would write something like this:

```
.gallery {display: flex; flex-wrap: wrap; row-gap: 15px; padding-block: 15px;}
```

In a like manner, we can open spaces between the flex items along the inline axis by using `column-gap`. We can modify the earlier example to push items apart as follows, with the result shown in Figure 11-32:

```
.gallery {display: flex; flex-wrap: wrap; column-gap: 15px;}
```

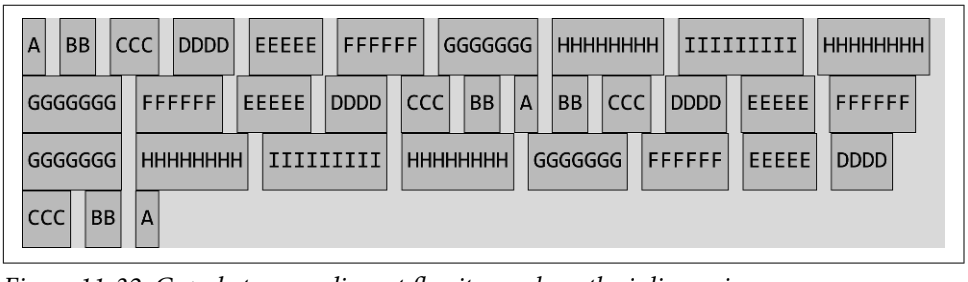


Figure 11-32. Gaps between adjacent flex items along the inline axis

Here, leftover space remains at the inline-end side of the flex lines, with each line having its own amount of space. That's because the flex items weren't given a `justify-content` value, so they defaulted to `start`. This means the gaps between the flex items are all exactly 15 pixels wide.

If we were to change the value of `justify-content` to `space-between`, then in any flex line with leftover space, the gaps between flex items will be increased by an equal amount, meaning they will be separated by more than 15 pixels. If there's a line where the inline sizes of all the flex items and all the gaps exactly equals the inline length of the flex line, 15 pixels of space will be between each flex item.

This is why `row-gap` and `column-gap` are really more like minimum separation distances between flex items or flex lines. The gaps don't count as "leftover space," any more than the flex items do.

Gaps are inserted between the outer margin edges of adjacent flex items, so if you add margins to your flex items, the actual visible space between two flex items will be the width of the gap plus the widths of the margins. Consider the following, which is illustrated in Figure 11-33:

```
.gallery {display: flex; flex-wrap: wrap; column-gap: 15px;}
.gallery div {margin-inline: 10px;}
```



Figure 11-33. Gaps and margins combine to open more space

Now the open spaces between flex items are all 35 pixels wide: 15 pixels from the `gap` property, plus 20 pixels (10 + 10) from the inline-side margins set on the flex items.

Thus far we've used length values, but what about percentages? Any percentage value used for a gap is taken to be a percentage of the container's size along the relevant axis. Thus, given `column-gap: 10%`, the gaps will be 10% the inline size of the flex container. If the container is 640 pixels wide along the inline axis, the column gaps will be 64 pixels each.

Working with rows can be a little more complicated. If you define an explicit block size, percentages are just a percentage of that block size. A `block-size` (which could also be set with `height` or `width`) of `25em` and a `row-gap` of `10%` means row gaps will be 2.5 ems wide. This same sort of thing can also happen if the block size happens to be larger than the sum total of the rows' block sizes.

But when the block size is solely determined by the block sizes of the rows added together, any percentage value could lead to a *cyclic calculation*: each calculation changes the value being calculated, ad infinitum. Suppose a flex container has three flex lines, each exactly 30 pixels tall. The flex container is set so its height is `auto`, so it will "shrink-wrap" the flex lines, making it 90 pixels tall (we're assuming no padding here, but the principles are the same regardless). A `row-gap` of `10%` would mean 9-pixel row gaps, and inserting the 2 row gaps would add 18 pixels of height. That would increase the container's height to 108 pixels, which would mean the 10%-wide gaps are now 10.8 pixels, so the container height increases again, which increases the row gaps, which increases container height, which…

To avoid this sort of infinite-loop scenario, the gaps are set to be zero-width whenever a cyclic calculation would happen, and everyone moves on with their lives. In practice, this means that percentage values for row gaps are useful only in a narrow range of cases, whereas they can be more broadly useful for column gaps. Figure 11-34 shows examples of percentage row gaps.

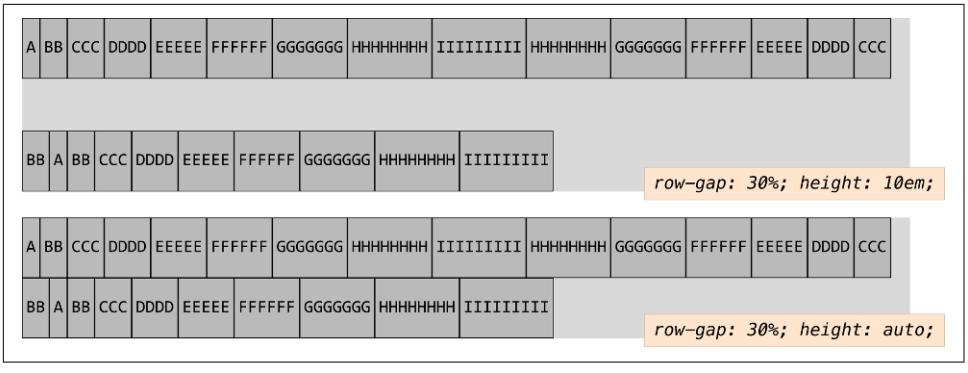


Figure 11-34. Percentage-based row gaps with and without explicit container heights

You can set up both column and gap rows on a flex container by supplying the two properties individually, or you can use the shorthand property `gap`.

gap

Values	*<row-gap>* *<column-gap>*?
Initial value	`0 0` for flex and grid layout; `0 1em` for multicolumn layout
Applies to	Flex, grid, and multicolumn containers
Computed value	See individual properties
Inherited	No
Animatable	Yes (for length values)

You need to supply only one value to `gap`, in which case it will be used for both the row and column gaps. If you supply two values, the first will always be used for row gaps, and the second for column gaps. Thus you get the results shown in Figure 11-35 from the following CSS:

```
#ex01 {gap: 15px 5px;}
#ex02 {gap: 5px 15px;}
#ex03 {gap: 5px;}
```

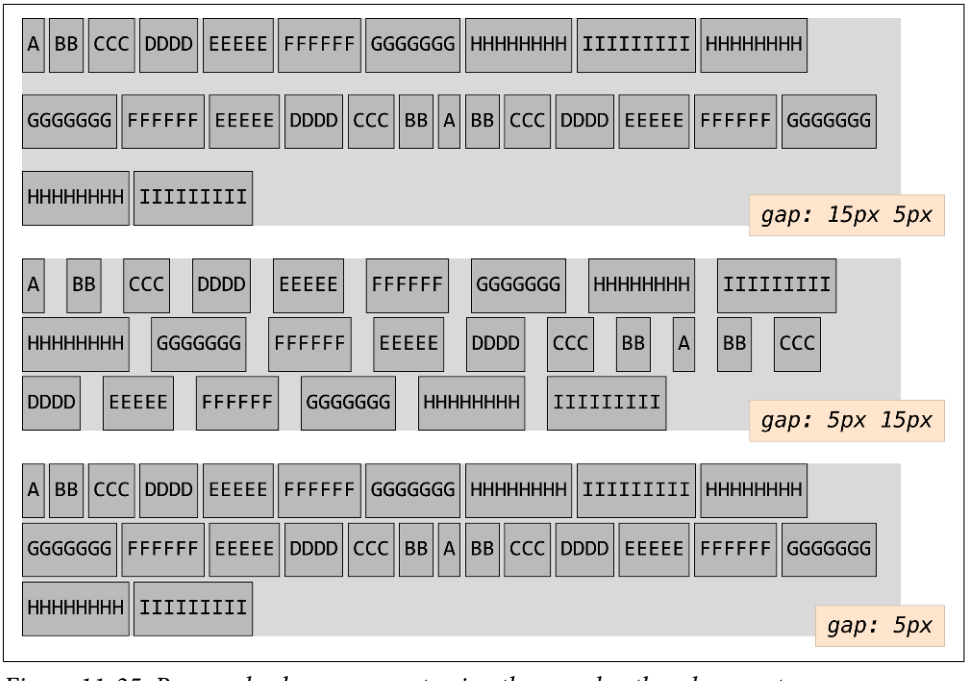


Figure 11-35. Row and column gaps set using the `gap` shorthand property



The original `gap` property was defined in CSS Multiple Columns, with additional hyphenated gap properties defined in CSS Grid as `grid-row-gap`, `grid-column-gap`, and `grid-gap`, before being made more generic and available in grid, flexbox, and multicolumn contexts. Browsers are required to treat the older properties as aliases for the newer, more generic properties; e.g., `grid-gap` is an alias for `gap`. So if you find the older grid gap properties in legacy CSS, you can change them to the newer names, but if not, they'll still work as if you had.

Flex Items

In the previous sections, you saw how to globally lay out all the flex items within a flex container by styling that container. The flexible box layout specification provides several additional properties applicable directly to flex items. With these flex-item-specific properties, we can more precisely control the layout of individual flex containers' children.

What Are Flex Items?

As you've seen throughout the chapter, we create flex containers by adding `display: flex` or `display: inline-flex` to an element that has child nodes. The children of those

flex containers are called *flex items*—whether they're child elements, non-empty text nodes between child elements, or generated content. In Figure 11-36, each letter is enclosed in its own element, including the space between words, so that each letter and space becomes a flex item.

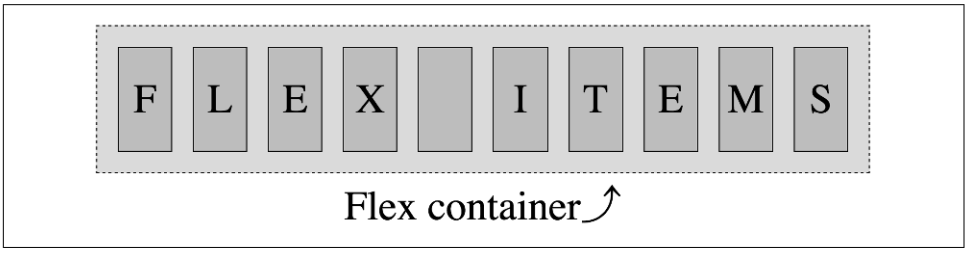


Figure 11-36. The child nodes are flex items, and the parent node is a flex container

When it comes to text-node children of flex containers, if the text node is not empty (containing content other than whitespace), it will be wrapped in an *anonymous flex item*, behaving like its flex-item siblings. While these anonymous flex items do inherit all the flex properties set by the flex container, just like their DOM node siblings, they are not directly targetable with CSS. We can't directly set any of the flex-item-specific properties on them. Thus, in the following markup, the two elements (`<strong>` and `<em>`) and the text "they're what's for" become flex items, for a total of three flex items:

```
<p style="display: flex;">
    <strong>Flex items:</strong> they're what's for <em>&lt;br&gt;fast!</em>
</p>
```

Generated content (via `::before` and `::after`) can be styled directly; therefore, all the properties discussed in this chapter apply equally to generated content and to element nodes.

Whitespace-only text nodes within a flex container are ignored, as if their `display` property were set to `none`, as the following code example shows:

```
nav ul {
  display: flex;
}

<nav>
  <ul>
    <li><a href="#1">Link 1</a></li>
    <li><a href="#2">Link 2</a></li>
    <li><a href="#3">Link 3</a></li>
    <li><a href="#4">Link 4</a></li>
    <li><a href="#5">Link 5</a></li>
  </ul>
</nav>
```

In the preceding code, with the `display` property set to `flex`, the unordered list is the flex container, and its child list items are all flex items. These list items, being flex items, are

flex-level boxes—semantically still list items, but not list items in their presentation. They are not block-level boxes either. Rather, they participate in their container's flex-formatting context. The whitespace between and around the `<li>` elements—the line feeds and indenting tabs and/or spaces—is completely ignored. The links are not flex items themselves, but are descendants of the flex items the list items have become.

Flex Item Features

The margins of flex items do not collapse. The `float` and `clear` properties don't have an effect on flex items and do not take a flex item out of flow. In effect, `float` and `clear` are ignored when applied to flex items. (However, the `float` property can still affect box generation by influencing the `display` property's computed value.) Consider the following:

```
aside {
  display: flex;
}
img {
  float: left;
}
```

```
<aside>
    <!-- this is a comment -->
    <h1>Header</h1>

    <img src="images/foo.jpg" alt="Foo Master">
    Some text
</aside>
```

In this example, the `aside` is the flex container. The comment and whitespace-only text nodes are ignored. The text node containing "Some text" is wrapped in an anonymous flex item. The header, image, and text node containing "Some text" are all flex items. Because the image is a flex item, the `float` is ignored.

Even though images and text nodes are inline-level nodes, because they are flex items, they are blockified as long as they are not absolutely positioned:

```
aside {
  display: flex;
  align-items: center;
}
aside * {
  border: 1px solid;
}
```

```
<aside>
    <!-- a comment -->
    <h1>Header</h1>

    <img src="images/foo.jpg" alt="foo master">
    Some text <a href="foo.html">with a link</a> and more text
</aside>
```

This markup is similar to the previous code example, except in this example we've added a link within the non-empty text node. In this case, we are creating five flex items illustrated in Figure 11-37. The comment and whitespace-only text nodes are ignored. The header, the image, the text node before the link, the link, and the text node after the link are all flex items.

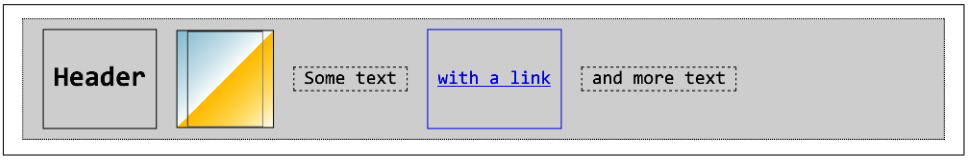


Figure 11-37. Five flex items in an aside

The text nodes containing "Some text" and "and more text" are wrapped in anonymous flex items, represented in Figure 11-37 by the dashed boxes (the dashes having been added for illustrative purposes) with no background. The header, image, and link, being actual DOM nodes, can be styled directly with CSS, as you can see with the border styling. The anonymous flex containers are not directly targetable, and so will have only whatever styles they pick up from the flex container.

Additionally, `vertical-align` has no effect on a flex item, except as it affects the alignment of text within the flex item. Setting `vertical-align: bottom` on a flex item will make all the text inside that flex item align to the bottom of their line boxes; it will not push the flex item to the bottom of its container. (That's what `align-items` and `align-self` are for.)

Absolute Positioning

While `float` will not actually float a flex item, setting `position: absolute` is a different story. The absolutely positioned children of flex containers, just like any other absolutely positioned element, are taken out of the flow of the document.

More to the point, they do not participate in flex layout and are not part of the document flow. However, they can be impacted by the styles set on the flex container, just as a child can be impacted by a parent element that isn't a flex container. In addition to inheriting any inheritable properties, the flex container's properties can affect the origin of the positioning.

The absolutely positioned child of a flex container is affected by both the `justify-content` value of the flex container and its own `align-self` value, if there is one. For example, if you set `align-self: center` on the absolutely positioned child, it will start out centered with respect to the flex container parent's cross-axis. From there, the element or pseudo-element can be moved by properties like `top`, `bottom`, margins, and so on.

The `order` property (explained in "The order Property" on page 541) may not impact where the absolutely positioned flex container child is drawn, but it does impact the order in which it is drawn in relation to its siblings.

Minimum Widths

In Figure 11-38, you'll note that the flex line inside the container with the `nowrap` default `flex-wrap` value overflows its flex container. This is because when it comes to flex items, the implied value of `min-width` is `auto`, rather than `0`. Originally in the specification, if the items didn't fit onto that single main-axis, they would shrink. However, the specification of `min-width` was altered as applied to flex items. (Traditionally, the default value for `min-width` is `0`.)

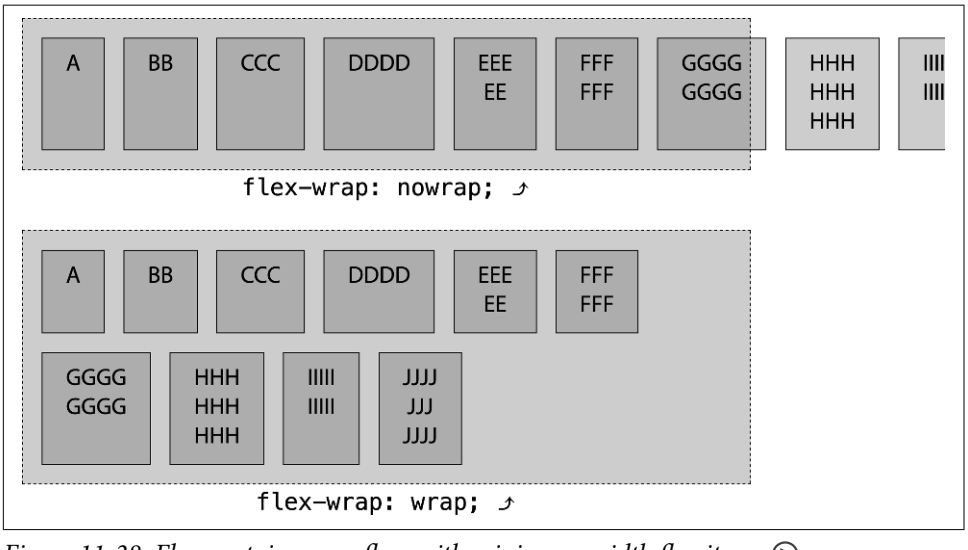


Figure 11-38. Flex container overflow with minimum-width flex items

If you set `min-width` to a width narrower than the computed value of `auto`—for example, if you declare `min-width: 0`—the flex items in the `nowrap` example will shrink to be narrower than their actual content (in some cases). If the items are allowed to wrap, they will be as narrow as possible to fit their content, but no narrower. Figure 11-39 illustrates both situations.

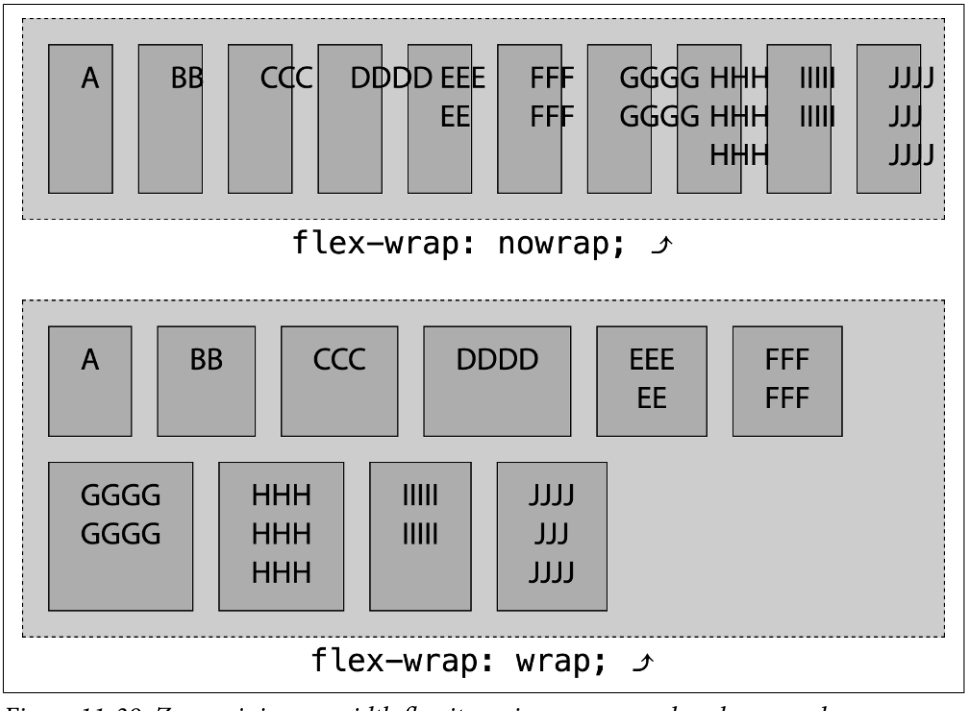


Figure 11-39. Zero-minimum-width flex items in nonwrapped and wrapped flex containers ▶

Flex-Item-Specific Properties

While flex items' alignment, order, and flexibility are to some extent controllable via properties set on their flex container, several properties can be applied to individual flex items for more granular control.

The `flex` shorthand property, along with its component properties of `flex-grow`, `flex-shrink`, and `flex-basis`, controls the flexibility of the flex items. *Flexibility* is the amount by which a flex item can grow or shrink along the main-axis.

The flex Property

The defining aspect of flex layout is the ability to make the flex items *flex*: altering their width or height to fill the available space in the main dimension. A flex container distributes free space to its items proportionally to their flex grow factor, or shrinks them to prevent overflow proportionally to their flex shrink factor. (We'll explore these concepts momentarily.)

Declaring the `flex` shorthand property on a flex item, or defining the individual properties that make up the shorthand, enables you to define the grow and shrink factors. If there is excess space, you can tell the flex items to grow to fill that space. Or not. If there

isn't enough room to fit all the flex items within the flex container at their defined or default sizes, you can tell the flex items to shrink proportionally to fit into the space. Or not.

This is all done with the `flex` property, which is a shorthand property for `flex-grow`, `flex-shrink`, and `flex-basis`. While these three subproperties can be used separately, it is highly recommended to always use the `flex` shorthand, for reasons we'll soon cover.

flex

Values	[<*flex-grow*> <*flex-shrink*>? || <*flex-basis*>] | none
Initial value	`0 1 auto`
Applies to	Flex items (children of flex containers)
Percentages	Valid for `flex-basis` value only, relative to element's parent's inner main-axis size
Computed value	Refer to individual properties
Inherited	No
Animatable	See individual properties

The `flex` property specifies the components of a flexible length: the *length* of the flex item being the length of the flex item along the main-axis (see "Understanding Axes" on page 471). When a box is a flex item, `flex` is consulted to determine the size of the box, instead of the main-axis size dimension property (`height` or `width`). The *components* of the `flex` property include the flex growth factor, flex shrink factor, and the flex basis.

The *flex basis* determines how the flex growth and shrink factors are implemented. As its name suggests, the `flex-basis` component of the flex shorthand is the basis on which the flex item determines how much it can grow to fill available space or how much it should shrink to fit all the flex items when there isn't enough space. It's the initial size of each flex item, and can be restricted to that specific size by specifying `0` for both the growth and shrink factors:

```
.flexItem {
    width: 50%;
    flex: 0 0 200px;
}
```

In the preceding CSS, the flex item will have a main-axis size of exactly 200 pixels, as the flex basis is `200px`, and it is allowed to neither grow nor shrink. Assuming that the main-axis is horizontal, the value of `width` (`50%`) is ignored. Similarly, a value for `height` would be ignored if the main-axis were vertical.



This override of `height` and `width` occurs outside the cascade, so you can't even override the flex basis by adding `!important` to the `height` or `width` value of a flex item.

If the target of a selector is not a flex item, applying the `flex` property to it will have no effect.

It is important to understand the three components that make up the `flex` shorthand property in order to be able to use it effectively.

The flex-grow Property

The `flex-grow` property defines whether a flex item is allowed to grow when space is available, and, if so, how much it will grow proportionally relative to the growth of other flex-item siblings.



Declaring the growth factor via the `flex-grow` property is *strongly* discouraged by the authors of the specification itself. Instead, declare the growth factor as part of the `flex` shorthand. We're discussing the property here only to explore how growth works.

flex-grow

Values	*<number>*
Initial value	`0`
Applies to	Flex items (children of flex containers)
Computed value	As specified
Inherited	No
Animatable	Yes

The value of `flex-grow` is always a number. Negative numbers are not valid. You can use non-integers if you like, just as long as they're 0 or greater. The value sets the *flex growth factor*, which determines how much the flex-item will grow relative to the rest of the flex item siblings as the flex container's free space is distributed.

If any space is available within the flex container, the space will be distributed proportionally among the children with a nonzero positive growth factor based on the various values of those growth factors.

For example, assume a `750px`-wide horizontal flex container with three flex items, each set to `width: 100px`. A total of 300 pixels of space is taken up by the flex items, leaving 450 pixels of "leftover" or available space (since 750 – 300 = 450). This is the first scenario shown in Figure 11-40: none of the flex items are permitted to grow.

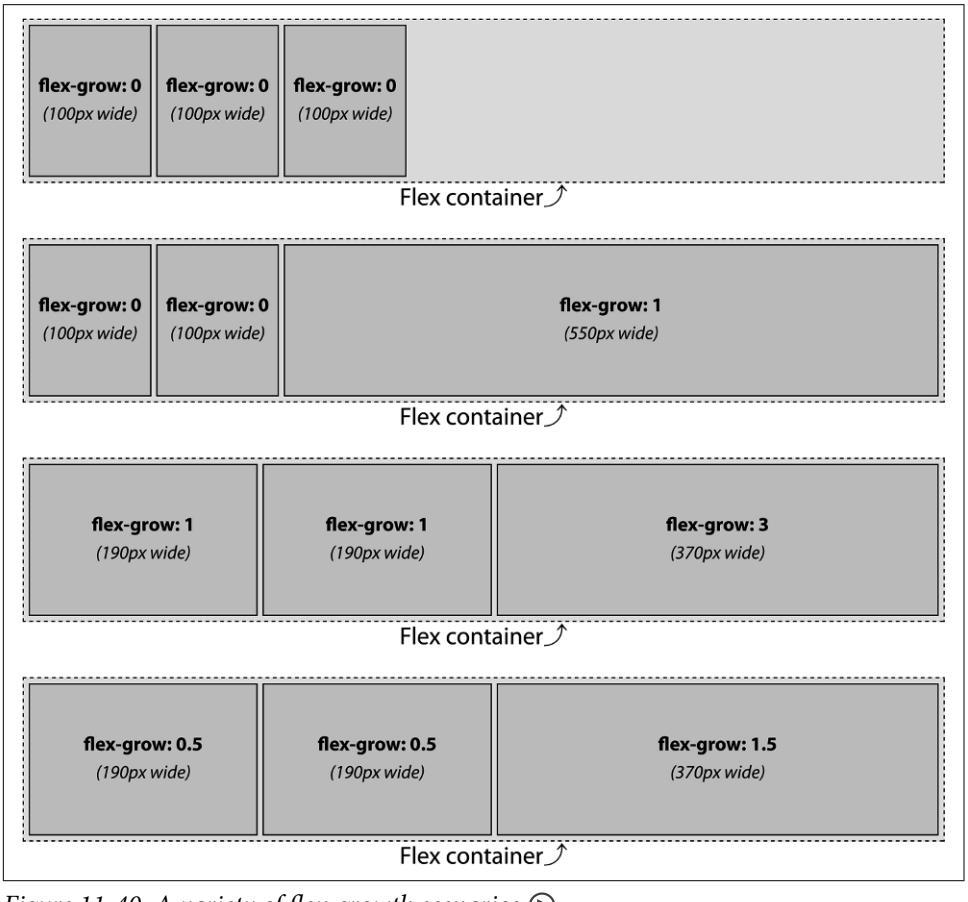


Figure 11-40. A variety of flex-growth scenarios

In the second scenario in Figure 11-40, only one of the flex items (the third) has been given a growth factor. The declaration we gave it is `flex-grow: 1`, but it could be any positive number the browser can understand. In this case, with two items having no growth factor and the third having a growth factor, all of the available space is given to the flex item with a growth factor. Thus, the third flex item gets all 450 pixels of available space added to it, arriving at a final width of 550 pixels. The `width: 100px` applied to it elsewhere in the styles is overridden.

In the third and fourth scenarios, the same flex item widths result despite the differing flex growth factors. Let's consider the third scenario, where the growth factors are 1, 1, and 3. The factors are all added together to get a total of 5. Each factor is then divided by

that total to get a proportion. So here, the three values are each divided by 5, yielding 0.2, 0.2, and 0.6.

Each proportion is multiplied by the available space to get the amount of growth. Thus:

1. 450 px × 0.2 = 90 px
2. 450 px × 0.2 = 90 px
3. 450 px × 0.6 = 270 px

Those are the growth portions added to each flex item's starting width of 100 pixels. Thus, the final widths are 190 pixels, 190 pixels, and 370 pixels, respectively.

The fourth scenario has the same result, because the proportions are the same. Imagine for a moment that we alter the growth factors to be 0.5, 1, and 1.5. Now the math works out such that the first flex item gets one-sixth of the available space, the second gets a third, and the third gets half. This results in the flex items' final widths being 175, 250, and 425 pixels, respectively. Had we declared growth factors of 0.1, 0.1, and 0.3, or 25, 25, and 75, or really any combination of numbers with a 1:1:3 correspondence, the result would have been identical.

As noted in "Minimum Widths" on page 502, if no width or flex basis is set, the flex basis defaults to `auto`, meaning each flex item basis is the width of its nonwrapped content. The `auto` value is special: it defaults to `content` unless the item has a width set on it, at which point the flex basis becomes that width. The `auto` value is discussed in "Automatic flex basis" on page 527. Had we not set the width in this example scenario, with our smallish font size, we would have had more than 450 pixels of distributable space along the main-axis.



The main-axis size of a flex item is impacted by the available space, the growth factor of all the flex items, and the flex basis of the item. We have yet to cover flex basis, but that time is coming soon!

Now let's consider flex items with different `width` values as well as different growth factors. In Figure 11-41, in the second example, we have flex items that are 100 pixels, 250 pixels, and 100 pixels wide, with growth factors of 1, 1, and 3, respectively, in a container that is 750 pixels wide. This means we have 300 pixels of extra space to distribute among a total of five growth factors (since 750 – 450 = 300). Each growth factor is therefore 60 pixels (300 ÷ 5). Therefore, the first and second flex items, with a `flex-grow` value of `1`, will each grow by 60 pixels. The last flex item will grow by 180 pixels, since its `flex-grow` value is `3`.

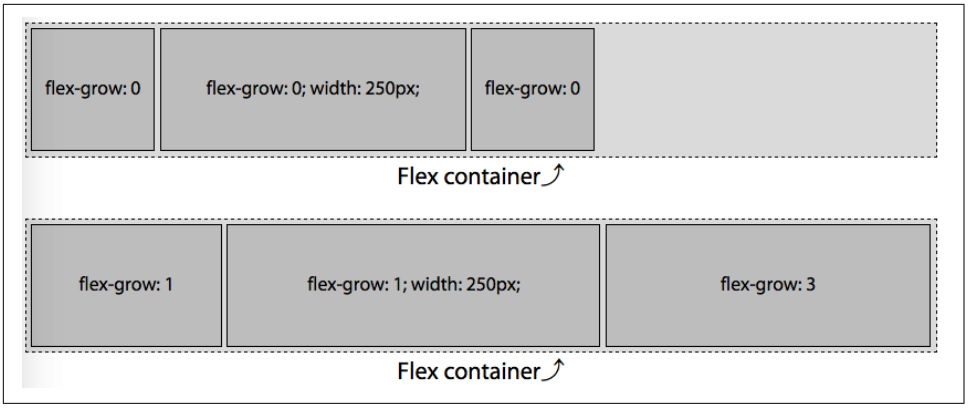


Figure 11-41. Mixed widths and growth factors

To recap, the available space in the flex container, the growth factors, and final width of each flex item are as follows:

Available space: 750 px – (100 px + 250 px + 100 px) = 300 px
Growth factors: 1 + 1 + 3 = 5
Width of each growth factor: 300 px ÷ 5 = 60 px

When flexed, the width of the flex items, based on their original width and growth factors, become

item1 = 100 px + (1 × 60 px) = 160 px
item2 = 250 px + (1 × 60 px) = 310 px
item3 = 100 px + (3 × 60 px) = 280 px

which adds up to 750 pixels.

Growth Factors and the flex Property

The `flex` property takes up to three values—the growth factor, shrink factor, and basis. The first positive non-null numeric value, if there is one, sets the growth factor (i.e., the `flex-grow` value). When the growth and shrink factors are omitted in the `flex` value, the growth factor defaults to `1`. However, if neither `flex` nor `flex-grow` is declared, the growth factor defaults to `0`. Yes, really.

Recall the second example in Figure 11-40, where the flex growth factors were 0, 0, and 1. Because we declared a value for `flex-grow` only, the flex basis was set to `auto`, as if we had declared the following:

```
#example2 flex-item {
  flex: 0 1 auto;
}
#example2 flex-item:last-child {
  flex: 1 1 auto;
}
```

So that means the first two flex items had no growth factor, a shrink factor, and a flex basis of `auto`. Had we used `flex` in the examples in Figure 11-40 instead of ill-advisedly using `flex-grow`, the flex basis in each case would be set to `0%`, as if this had been done:

```
#example2 flex-item {
  flex: 0 1 0%;
}
#example2 flex-item:last-child {
  flex: 1 1 0%;
}
```

As the shrink factor defaults to `1` and the basis defaults to `0%`, the following CSS is identical to the preceding snippet:

```
#example2 flex-item {
  flex: 0;
}
#example2 flex-item:last-child {
  flex: 1;
}
```

This would have the result shown in Figure 11-42. Compare this to Figure 11-40 to see how things have changed (or not).

You may notice something odd in the first two scenarios: the flex basis been set to 0, and only the last flex item in the second scenario has a positive value for flex growth. Logic would seem to dictate that the widths of the three flex items should be 0, 0, and 750 pixels, respectively. But logic would also dictate that it makes no sense to have content overflowing its flex item if the flex container has the room for all the content, even if the basis is set to `0`.

The specification authors thought of this quandary. When the `flex` property declaration explicitly sets or defaults the flex basis to `0%` and a flex item's growth factor is `0`, the length of the main-axis of the nongrowing flex items will shrink to the smallest length the content allows, or smaller. In Figure 11-42, that minimum length is the width of the widest sequence of letters, "flex:" (including the colon).

As long as a flex item has a visible overflow and no explicitly set value for `min-width` (or `min-height` for vertical main-axes), the minimum width (or minimum height) will be the smallest width (or height) that the flex item needs to be to fit the content or the declared `width` (or `height`), whichever is smaller.

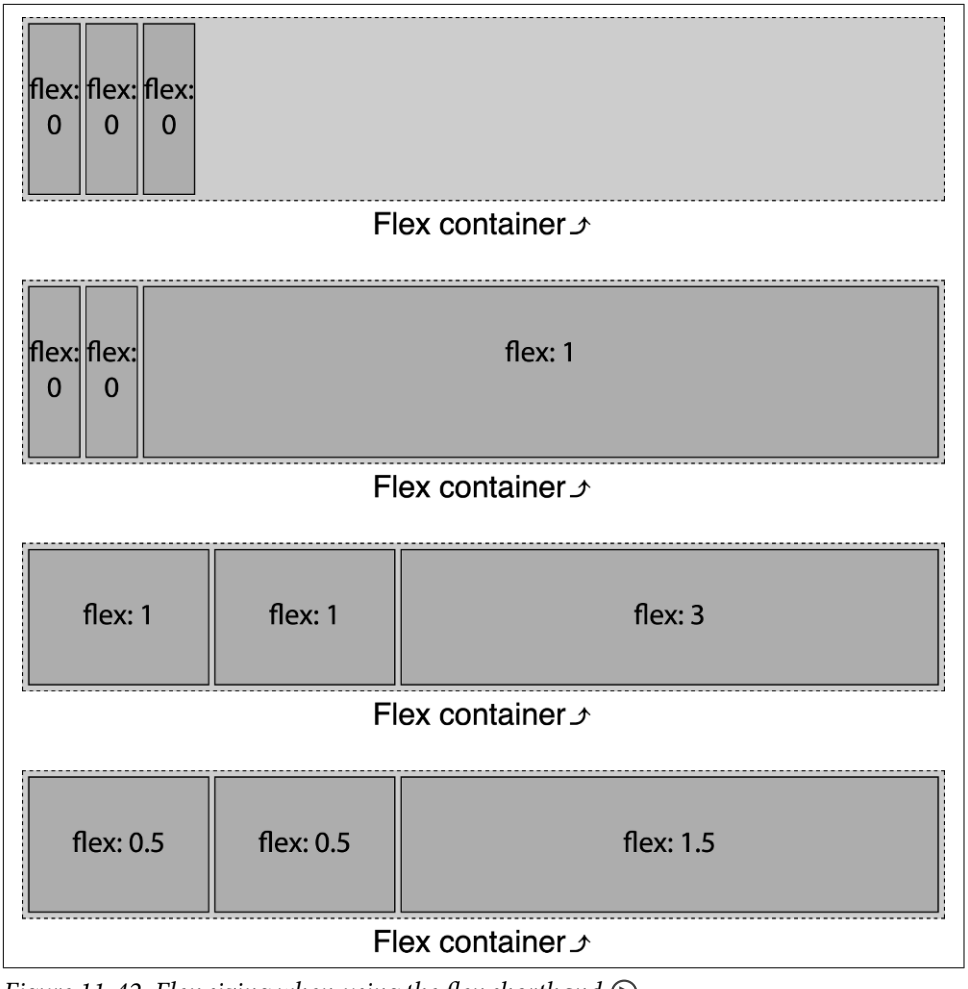


Figure 11-42. Flex sizing when using the flex shorthand

If all items are allowed to grow, and the flex basis for each flex item is `0%`, *all* of the space, rather than just excess space, is distributed proportionally based on the growth factors. In the third example in Figure 11-42, two flex items have growth factors of 1, and one flex item has a growth factor of 3. We thus have a total of five growth factors:

$$(2 \times 1) + (1 \times 3) = 5$$

With five growth factors, and a total of 750 pixels, each growth factor is worth 150 pixels:

$$750\text{ px} \div 5 = 150\text{ px}$$

While the default flex item size is 100 pixels, the flex basis of `0%` overrides that, leaving us with two flex items at 150 pixels each and the last flex item with a width of 450 pixels:

$1 \times 150\text{ px} = 150\text{ px}$
$3 \times 150\text{ px} = 450\text{ px}$

Similarly, in the last example of Figure 11-42, with two flex items having growth factors of 0.5, and one flex item having a growth factor of 1.5, we have a total of 2.5 growth factors:

$$(2 \times 0.5) + (1 \times 1.5) = 2.5$$

With a 2.5 growth factor, and a total of 750 pixels, each growth factor is worth 300 pixels:

$$750\text{ px} \div 2.5 = 300\text{ px}$$

While the default flex item size is 100 pixels, the flex basis of 0% overrides that, leaving us with two flex items at 150 pixels each and the last flex item with a width of 450 pixels:

$0.5 \times 300\text{ px} = 150\text{ px}$
$1.5 \times 300\text{ px} = 450\text{ px}$

Again, this is different from declaring only `flex-grow`, because that means the flex basis defaults to `auto`. In that case, only the extra space, not all the space, is distributed proportionally. When using `flex`, on the other hand, the flex basis is set to `0%`, so the flex items grow in proportion to the total space, not just the leftover space. Figure 11-43 illustrates the difference.

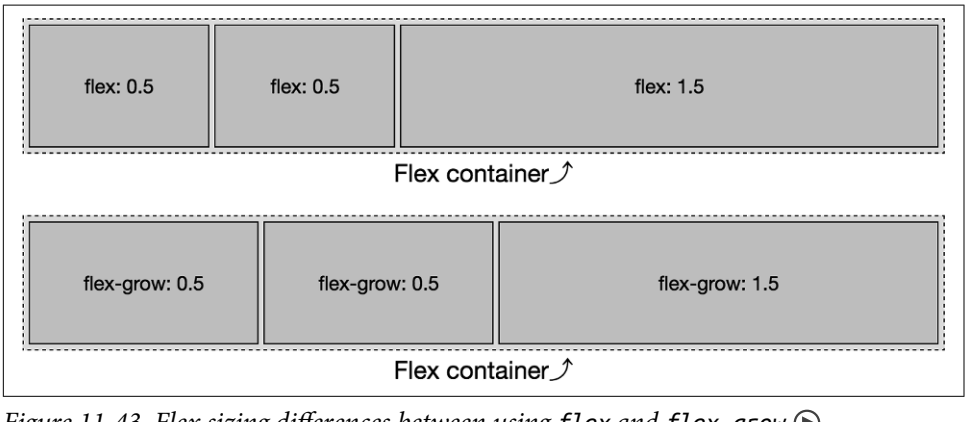


Figure 11-43. Flex sizing differences between using `flex` and `flex-grow`

Now let's talk about flex shrinking factors, which are in some ways the inverse of flex growth factors, but are in other ways different.

The flex-shrink Property

The <*flex-shrink*> portion of the `flex` shorthand property specifies the *flex shrink factor*. It can also be set via the `flex-shrink` property.



Declaring the shrink factor via the `flex-shrink` property is *strongly* discouraged by the authors of the specification itself. Instead, declare the shrink factor as part of the `flex` shorthand. We're discussing the property here only to explore how shrinking works.

flex-shrink

Values	<*number*>
Initial value	`1`
Applies to	Flex items (children of flex containers)
Computed value	As specified
Inherited	No
Animatable	Yes

The shrink factor determines how much a flex item will shrink relative to the rest of its flex-item siblings when there isn't enough space for them all to fit, as defined by their content and other CSS properties. When omitted in the shorthand `flex` property value or when both `flex` and `flex-shrink` are omitted, the shrink factor defaults to `1`. Like the growth factor, the value of `flex-shrink` is always a number. Negative numbers are not valid. You can use non-integer values if you like, just as long as they're greater than 0.

Basically, the shrink factor defines how "negative available space" is distributed when there isn't enough room for the flex items, and the flex container isn't allowed to otherwise grow or wrap. See Figure 11-44.

Figure 11-44 is similar to Figure 11-40, except the flex items are set to `width: 300px` instead of 100 pixels. We still have a 750-pixel-wide flex container. The total width of the three items is 900 pixels, meaning the content starts out 150 pixels wider than the parent flex container. If the items are not allowed to shrink or wrap (see "Wrapping Flex Lines" on page 468), they will burst out from the fixed-size flex container. This is demonstrated in the first example in Figure 11-44: those items will not shrink because they have a zero shrink factor. Instead, they overflow the flex container.

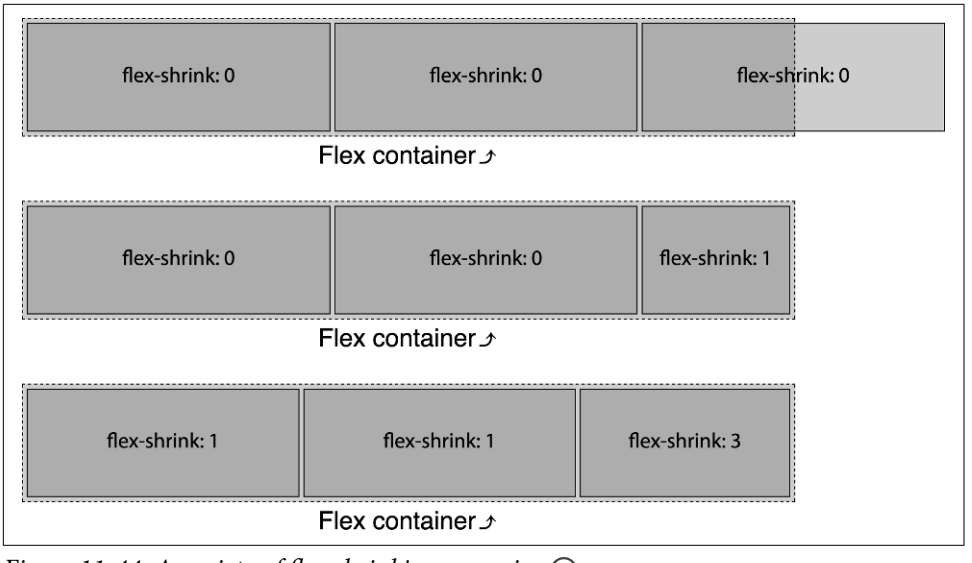


Figure 11-44. A variety of flex shrinking scenarios

In the second example in Figure 11-44, only the last flex item is set to be able to shrink. The last flex item is thus forced to do all the shrinking necessary to enable all the flex items to fit within the flex container. With 900 pixels of content needing to fit into our 750-pixel container, we have 150 pixels of negative available space. The two flex items with no shrink factor stay at 300 pixels wide. The third flex item, with a positive value for the shrink factor, shrinks 150 pixels, to end up 150 pixels wide. This enables the three items to fit within the container. (In this example, the shrink factor is `1`, but had it been `0.001` or `100` or `314159.65` or any other positive number the browser could understand, the result would be the same.)

In the third example, we have positive shrink factors for all three flex items:

```
#example3 flex-item {
  flex-shrink: 1;
}
#example3 flex-item:last-child {
  flex-shrink: 3;
}
```

As this is the only one of the three `flex` shorthand properties we declared, this means the flex items will behave as if we had declared the following:

```
#example3 flex-item {
  flex: 0 1 auto; /* growth defaults to 0, basis to auto */
}
f#example3 flex-item:last-child {
  flex: 0 3 auto;
}
```

If all items are allowed to shrink, as is the case here, the shrinking is distributed proportionally based on the shrink factors. This means the larger a flex item's shrink factor, as compared to the shrink factors of its sibling flex items, the more the item will shrink in comparison.

With a parent 750 pixels wide, and three flex items with a width of 300 pixels, 150 "negative space" pixels need to be shaved off the flex items that are allowed to shrink (which is all of them in this example). With two flex items having a shrink factor of 1, and one flex item having a shrink factor of 3, we have a total of five shrink factors:

$$(2 \times 1) + (1 \times 3) = 5$$

With five shrink factors, and a total of 150 pixels needing to be shaved off all the flex items, each shrink factor is worth 30 pixels:

$$150\text{ px} \div 5 = 30\text{ px}$$

The default flex item size is 300 pixels, leading us to have two flex items with a width of 270 pixels each and the last flex item having a width of 210 pixels, which totals 750 pixels:

$$300\text{ px} - (1 \times 30\text{ px}) = 270\text{ px}$$
$$300\text{ px} - (3 \times 30\text{ px}) = 210\text{ px}$$

The following CSS produces the same outcome: while the numeric representations of the shrink factors are different, they are proportionally the same, so the flex item widths will be the same:

```
flex-item {
  flex: 1 0.25 auto;
}
flex-item:last-child {
  flex: 1 0.75 auto;
}
```

Note that the flex items in these examples will shrink to 210, 210, and 270 pixels, respectively, *as long as* the content (like media objects or nonwrappable text) within each flex item is not wider than 210, 210, or 270 pixels, respectively. If the flex item contains content that cannot wrap or otherwise shrink in the main dimension, the flex item will not shrink any further.

Suppose that the first flex items contain an image 300 pixels wide. That first flex item cannot shrink, and other flex items can shrink; therefore, it will not shrink, as if it had a null shrink factor. In this case, the first item would be 300 pixels, with the 150 pixels of negative space distributed proportionally based on the shrink factors of the second and third flex items.

That being the case, we have four unimpeded shrink factors (one from the second flex item, and three from the third) for 150 pixels of negative space, with each shrink factor

being worth 37.5 pixels. The flex items will end up 300, 262.5, and 187.5 pixels, respectively, for a total of 750 pixels, as shown here and illustrated in Figure 11-45:

item1 = 300 px – (0 × 37.5 px) = 300.0 px
item2 = 300 px – (1 × 37.5 px) = 262.5 px
item3 = 300 px – (3 × 37.5 px) = 187.5 px

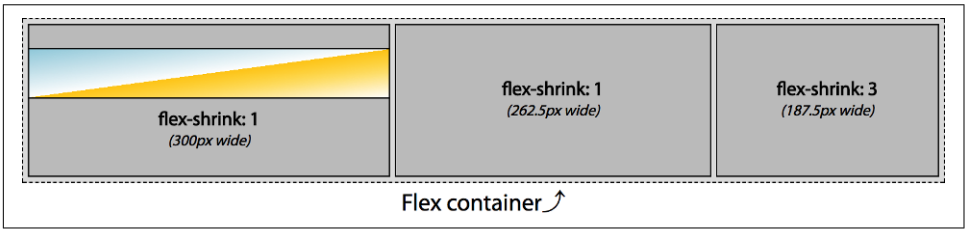


Figure 11-45. Shrinking being impeded by flex-item content

Had the image been 296 pixels wide, that first flex item would have been able to shrink by 4 pixels. The remaining 146 pixels of negative space would then be distributed among the four remaining factors, yielding 36.5 pixels per factor. The flex items would then be 296, 263.5, and 190.5 pixels wide, respectively.

If all three flex items contained nonwrappable text or media 300 pixels or wider, none of the three flex items would not shrink, appearing similar to the first example in Figure 11-44.

Proportional shrinkage based on width and shrink factor

The preceding code examples are fairly simple because all the flex items start with the same width. But what if the widths are different? What if the first and last flex items have a width of 250 pixels and the middle flex item has a width of 500 pixels, as shown in Figure 11-46?

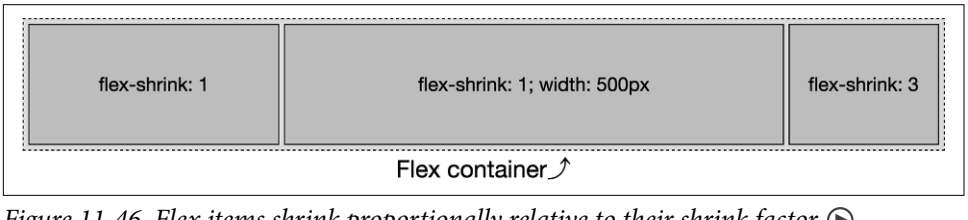


Figure 11-46. Flex items shrink proportionally relative to their shrink factor

Flex items shrink proportionally relative to both the shrink factor *and* the flex item's width, with the width often being the width of the flex item's content with no wrapping. In Figure 11-46, we are trying to fit 1,000 pixels into a flex container that's 750 pixels wide. We have an excess of 250 pixels to be removed from five shrink factors.

If this were a `flex-grow` situation, we would simply divide 250 pixels by 5, allocating 50 pixels per growth factor. If we were to shrink that way, we would get flex items 200, 550, and 100 pixels wide, respectively. But that's not how shrinking actually works.

Here, we have 250 pixels of negative space to proportionally distribute. To get the shrink factor proportions, we divide the negative space by the total of the flex items' widths (more precisely, their lengths along the main-axis) times their shrink factors:

$$ShrinkPercent = \frac{NegativeSpace}{((Width1 \times ShrF1) + \ldots + (WidthN \times ShrFN))}$$

Using this equation, we find the shrink percentage:

= 250 px ÷ [(250 px × 1) + (500 px × 1) + (250 px × 3)]
= 250 px ÷ 1500 px
= 0.166666667 (16.67%)

When we reduce each flex item by 16.67% times the value of `flex-shrink`, we end up with flex items that are reduced as follows:

item1 = 250 px × (1 × 16.67%) = 41.67 px
item2 = 500 px × (1 × 16.67%) = 83.33 px
item3 = 250 px × (3 × 16.67%) = 125 px

Each reduction is then subtracted from the starting sizes of 250, 500, and 250 pixels, respectively. Thus we have flex items that are 208.33, 416.67, and 125 pixels wide.

Differing basis values

When the shrink factor has been set to `0`, and both the width and flex basis of a flex item are set to `auto`, the item's content will not wrap, even when you think it should. Conversely, any positive shrink value enables content to wrap. Because shrinking is proportional based on the shrink factor, if all the flex items have similar shrink factors, the content should wrap over a similar number of lines.

In the three examples shown in Figure 11-47, the flex items do not have a declared width. Therefore, the width is based on the content, because `width` defaults to `auto`. The flex container has been made 520 pixels wide instead of our usual 750 pixels.

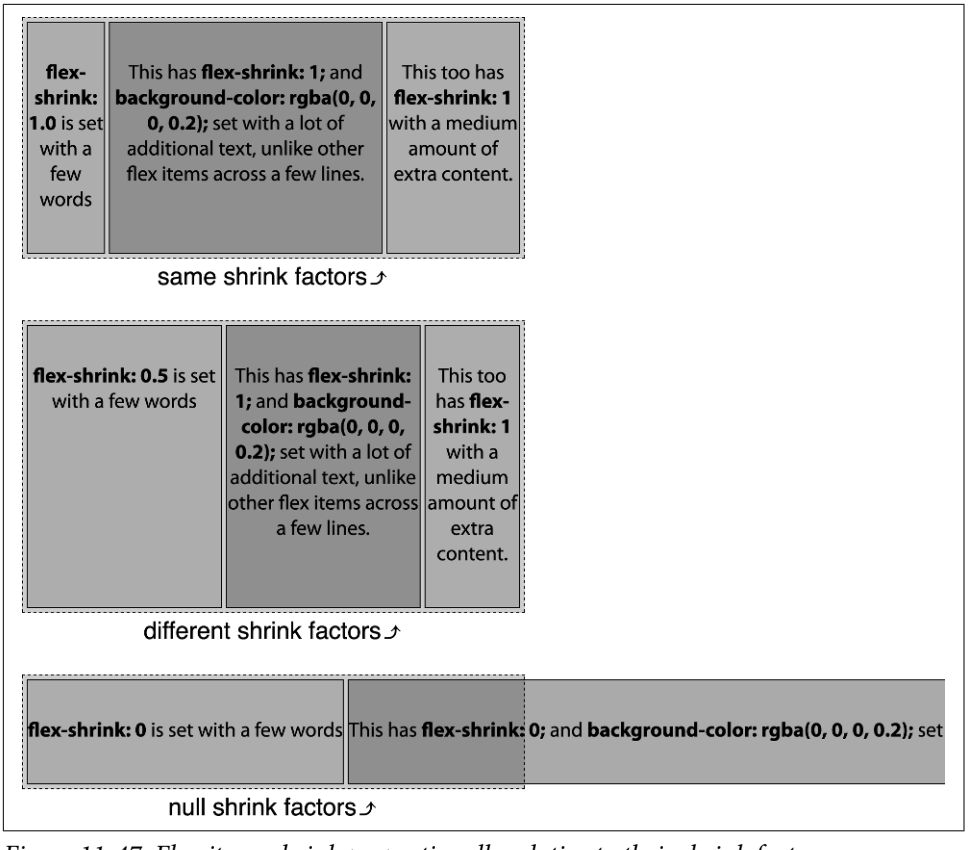


Figure 11-47. Flex items shrink proportionally relative to their shrink factor and content

Note that in the first example, where all the items have the same `flex-shrink` value, all content wraps over four lines. In the second example, the first flex item has a shrink factor that's half the value of the other flex items, so it wraps the content over (roughly) half the number of lines. This is the power of the shrink factor.

In the third example, with no shrink factor, the text doesn't wrap at all, and the flex items overflow the container by quite a bit.



As of late 2022, this "line-balancing" and refusal-to-wrap behavior is not consistent across browsers. If you see different results when trying this out for yourself, that may be why.

Because the `flex` property's shrink factor reduces the width of flex items proportionally, the number of lines of text in the flex items will grow or shrink as the width shrinks or grows, leading to similar height content within sibling flex items when the shrink factors are similar.

In the examples, take the contents of the flex items to be 280, 995, and 480 pixels, respectively—which are the widths of the nonwrapping flex items in the third example (as measured by the developer tools, then rounded to make this example a little simpler). This means we have to fit 1,755 pixels of content into a 520-pixel-wide flex container by shrinking the flex items proportionally based on their shrink factor. We have 1,235 pixels of negative available space to proportionally distribute.



Remember that you can't rely on web inspector tools to figure out shrink factors for production. We're going through this exercise to show how shrink factors work. If minutia isn't your thing, feel free to jump to "The flex-basis Property" on page 522.

In our first example, the flex items will end up with the same, or approximately the same, number of text lines. This is because flex items shrink proportionally, based on the width of their content.

We didn't declare any widths, and therefore can't simply use an explicit element width as the basis for our calculations, as we did in the previous examples. Rather, we distribute the 1,235 pixels of negative space proportionally based on the widths of the content—280, 995, and 480 pixels, respectively. We determine 520 is 29.63% of 1,755. To determine the width of each flex item with a shrink factor of 1, we multiply the content width of each flex item by 29.63%:

item1 = 280 px × 29.63% = 83 px
item2 = 995 px × 29.63% = 295 px
item3 = 480 px × 29.63% = 142 px

With the default of `align-items: stretch` (see "Aligning Items" on page 482), a three-column layout will have three columns of equal height. By using a consistent shrink factor for all flex items, you can indicate that the actual content of these three flex items should be of approximately equal height—though, by doing this, the widths of those columns will not necessarily be uniform.

In the second example in Figure 11-47, the flex items don't all have the same shrink factor. The first flex item will, proportionally, shrink half as much as the others. We start with the same widths: 280, 995, and 480 pixels, respectively, but their shrink factors are 0.5, 1.0, and 1.0. Because we know the widths of the content, the shrink factor (X) can be found mathematically:

280 px + 995 px + 480 px = 1,615 px
(0.5 × 280 px) + (1 × 995 px) + (1 × 480 px) = 1,235 px
X = 1,235 px ÷ 1,615 px = 0.7647

We can find the final widths now that we know the shrink factor. If the shrink factor is 76.47%, `item2` and `item3` will shrink by that amount, whereas `item1` will shrink by 38.23% (because its `flex-shrink` value is half the others). The amount of shrinkage in each case is rounded off to the nearest whole number:

item1 = 280 px × 0.3823 = 107 px
item2 = 995 px × 0.7647 = 761 px
item3 = 480 px × 0.7647 = 367 px

Thus, the final widths of the flex items are as follows:

item1 = 280 px – 107 px = 173 px
item2 = 995 px – 761 px = 234 px
item3 = 480 px – 367 px = 113 px

The total combined widths of these three flex items is 520 pixels.

Adding in varying shrink and growth factors makes it all a little less intuitive. That's why you likely want to always declare the `flex` shorthand, preferably with a width or basis set for each flex item. If this doesn't make sense yet, don't worry; we'll cover a few more examples of shrinking as we discuss `flex-basis`.

Responsive flexing

Allowing flex items to shrink proportionally allows for responsive objects and layouts that can shrink proportionally without breaking. For example, you can create a three-column layout that smartly grows and shrinks without media queries, as shown on a wide screen in Figure 11-48 and narrow screen in Figure 11-49:

```
nav {
  flex: 0 1 200px;
  min-width: 150px;
}
article {
  flex: 1 2 600px;
}
aside {
  flex: 0 1 200px;
  min-width: 150px;
}
```

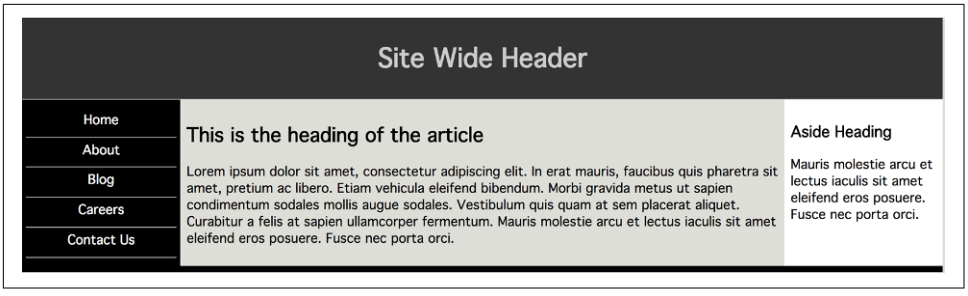


Figure 11-48. A wide flexbox layout

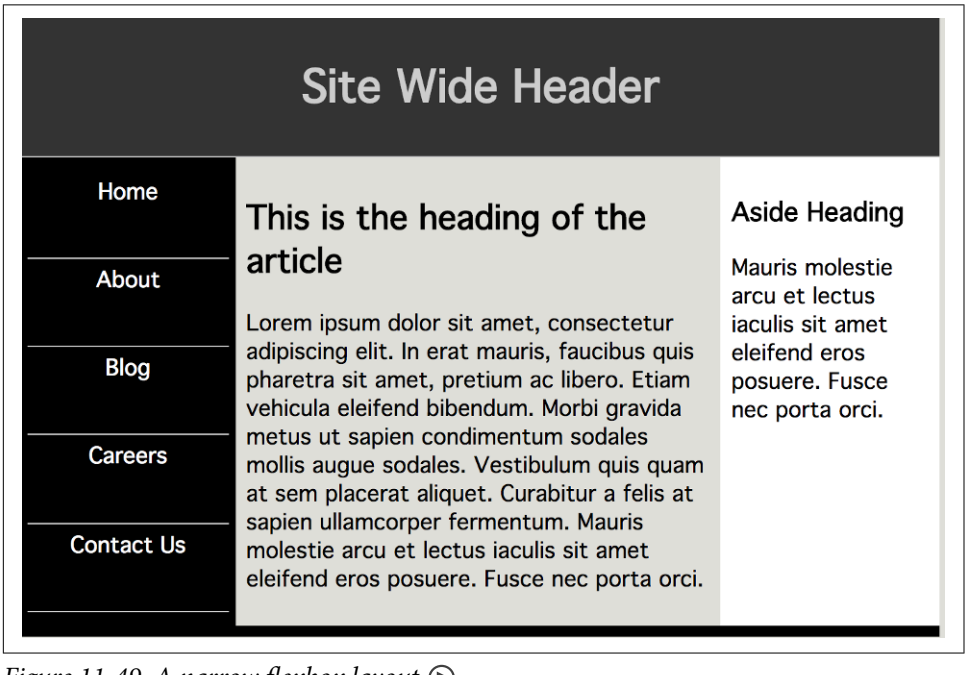


Figure 11-49. A narrow flexbox layout ▶

In this example, if the viewport is greater than 1,000 pixels, only the middle column grows because only the middle column is provided with a positive growth factor. We also dictate that below the 1,000-pixel-wide mark, all the columns shrink.

Let's take it bit by bit. The `<nav>` and `<aside>` elements have the following CSS:

```
flex: 0 1 200px;
min-width: 150px;
```

They don't grow from their basis but can shrink at equal rates. This means they'll have the width of their flex basis by default. If they do need to shrink, they'll shrink to a minimum width of `150px` and then stop shrinking. However, if either one has an element that's more

than 150 pixels wide, whether it's an image or a run of text, it will stop shrinking as soon as it reaches the width of that bit of content. Suppose a 180-pixel image dropped into the `<aside>` element. It would stop shrinking as soon as it reaches 180 pixels wide. The `<nav>` would keep shrinking down to 150 pixels.

The `<main>` element, on the other hand, has these styles:

```
flex: 1 2 600px;
```

Thus, the `<main>` element can grow if there's space for it to do so. Since it's the only flex item that can grow, it gets all the growth. Given a browser window 1,300 pixels wide, the two side columns will be 200 pixels wide each, leaving 900 pixels of width for the center column. In shrinking situations, the center column will shrink twice as fast as the other two elements. Thus, if the browser window is 900 pixels wide, each side column will be 175 pixels wide, and the center column 550 pixels wide.

Once the window reaches 800 pixels wide, the side columns will reach their `min-width` values of `150px`. From then on, any narrowing will be taken up by the center column.

Just to be clear, you are not required to use pixels in these situations. You don't even have to use the same unit measures for various flex basis values. The previous example could be rewritten like this:

```
nav {
  flex: 0 1 20ch;
  min-width: 15vw;
}
article {
  flex: 1 2 45ch;
}
aside {
  flex: 0 1 20ch;
  min-width: 10ch;
}
```

We won't go through all the math here, but the general approach is to set flex basis values on character widths for improved readability, with some lower limits based on character widths and others on viewport width.



Flexbox can be useful for a one-dimensional page layout like the one shown in this section, with only three columns in a line. For anything more complex, or for a more powerful set of options, use grid layout. (See Chapter 12.)

The flex-basis Property

As you've already seen, a flex item's size is impacted by its content and box-model properties and can be reset via the three components of the `flex` property. The `<flex-basis>` component of the `flex` property defines the initial or default size of flex items, before extra or negative space is distributed—before the flex items are allowed to grow or shrink according to the growth and shrink factors. It can also be set via the `flex-basis` property.



Declaring the flex basis via the `flex-basis` property is *strongly* discouraged by the authors of the specification itself. Instead, declare the flex basis as part of the `flex` shorthand. We're discussing the property here only to explore the flex basis.

flex-basis

Values	`auto` | `content` | `max-content` | `min-content` | `fit-content` | [`<length>` | `<percentage>`]
Initial value	`auto`
Applies to	Flex items (children of flex containers)
Percentages	Relative to flex container's inner main-axis size
Computed value	As specified, with length values made absolute
Inherited	No
Animatable	`<width>`

The flex basis determines the size of a flex item's element box, as set by `box-sizing`. By default, when a block-level element is not a flex item, the size is determined by the size of its parent, content, and box-model properties. When no size properties are explicitly declared or inherited, the size defaults to its individual content, border, and padding, which is 100% of the width of its parent for block-level elements.

The flex basis can be defined using the same length value types as the `width` and `height` properties—for example, `5vw`, `12%`, and `300px`.

The universal keyword `initial` resets the flex basis to the initial value of `auto`, so you might as well declare `auto`. In turn, `auto` evaluates to the `width` (or `height`), if declared. If the value of `width` (or `height`) is set to `auto`, the value of `flex-basis` is evaluated to `content`. This causes the flex item to be sized based on the content of the flex item, though the exact method for doing so is not made explicit in the specification.

The content keywords

In addition to lengths and percentages, `flex-basis` supports the `min-content`, `max-content`, `fit-content`, and `content` keywords. We covered the first three in Chapter 6, but `fit-content` deserves a revisit here, and `content` needs to be explored.

When using `fit-content` as the value for `flex-basis`, the browser will do its best to balance all the flex items in a line so that they are similar in block size. Consider this code, which is illustrated in Figure 11-50:

```
.flex-item {flex-basis: 25%; width: auto;}
.flex-item.fit {flex-basis: fit-content;}
```

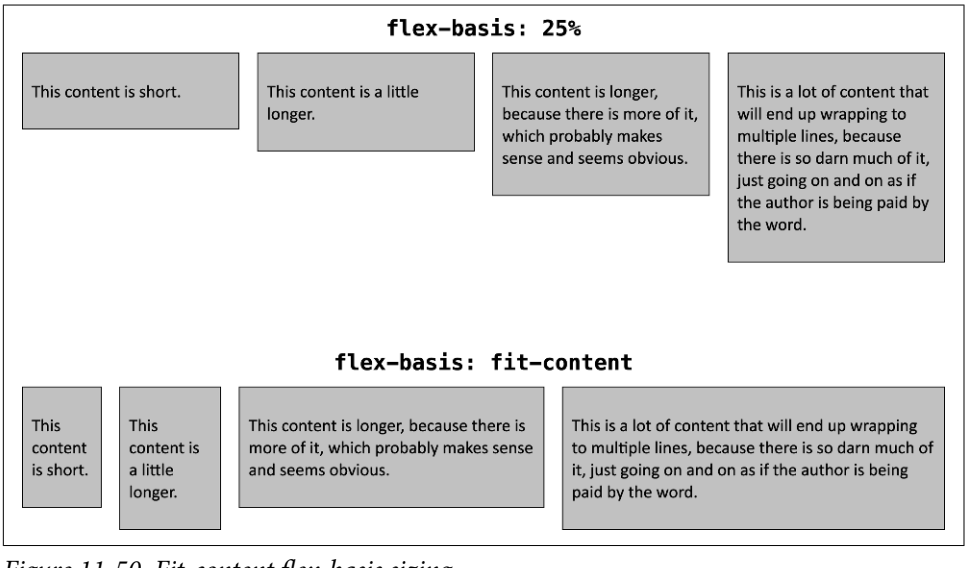


Figure 11-50. Fit-content flex-basis sizing

In the first flex line, the flex basis of the flex items is set to 25%, meaning each flex item starts out with 25% the width of the flex line as its sizing basis, and is flexed from there at the browser's discretion. In the second flex line, the flex items are set to use `fit-content` for their flex basis. Notice that more content leads to wider flex items, and less to narrower items.

Also notice that the heights (more properly, the block sizes) of the flex items are all the same, though this is not guaranteed: in certain situations, some of the flex items could be a bit taller than the others—say, by having one flex item's content wrap to one more line than the others'. They should all be very close to the same, though.

This is a good illustration of one of the strengths of flexbox: you can give a general direction to the layout engine and have it do the rest of the work. Here, rather than having to figure out which widths should be assigned to which flex items in order to balance out their heights, you tell it `fit-content` and let it figure out the rest.

Using the `content` keyword has results generally similar to `fit-content`, though some differences exist. A `content` basis is the size of the flex item's content—that is, the length of the main-axis size of the longest line of content or widest (or tallest) media object. It's the equivalent of declaring `flex-basis: auto; inline-size: auto;` on a flex item.

The value `content` has the effects shown in Figure 11-51.

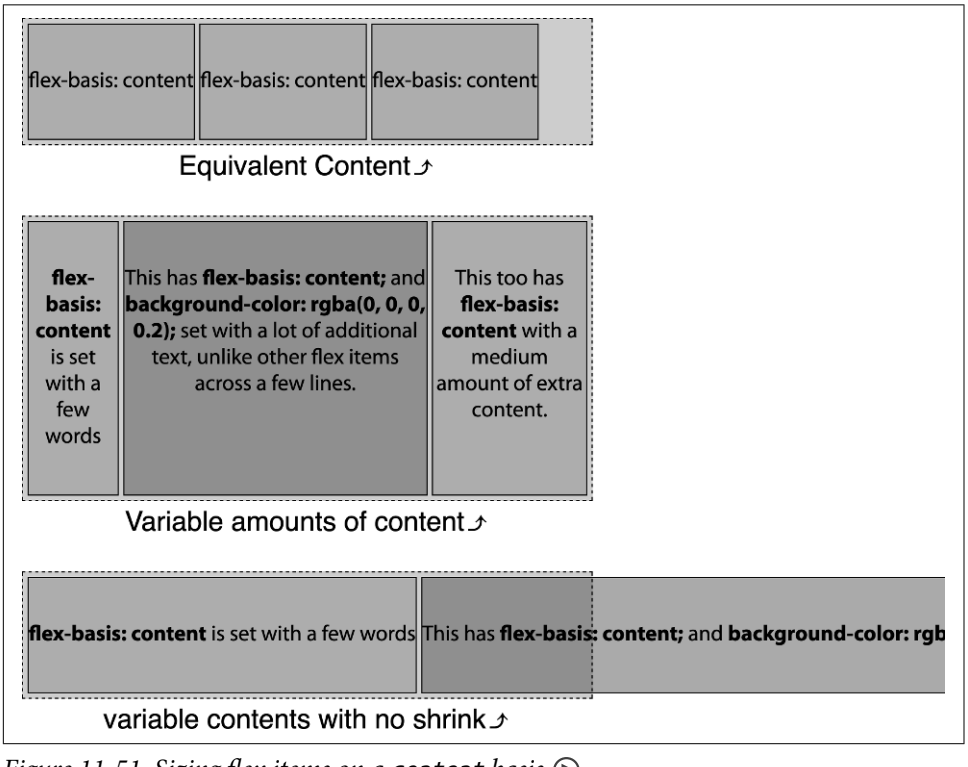


Figure 11-51. Sizing flex items on a `content` basis

In the first and third examples, the width of the flex item is the size of the content; and the flex basis is that same size. In the first example, the flex items' width and basis are approximately 132 pixels. The total width of the three flex items side by side is 396 pixels, with a few pixels of space between the items, all fitting easily into the parent container.

In the third example, we have set a null shrink factor (`0`): this means the flex items cannot shrink, so they won't shrink or wrap to fit into the fixed-width flex container. Rather, they are the width of their nonwrapped text. That width is also the value of the flex basis. The three flex items' widths, and thus their basis values, are approximately 309, 1,037 pixels, and 523 pixels, respectively. You can't see the full width of the second flex item or the third flex item at all, but they're in the chapter files.

The second example contains the same content as the third example, but the flex items are defaulting to a shrink factor of 1, so the text in this example wraps because the flex items can shrink. Thus, while the width of the flex item is not the width of the content, the flex basis—the basis by which it will proportionally shrink—is the width of the items' contents.

The third example in Figure 11-51 is also a good illustration of what would happen with the `max-content` keyword with `flex-shrink: 0`: the flex basis for each item will be the maximum size of its content. If flex shrinking is allowed, then the browser will start with the `max-content` for the basis of each item's flexing, and shrink them down from there. The difference between the two is captured in the following code and illustrated in Figure 11-52:

```
#example1 {flex-basis: max-content; flex-shrink: 0;}
#example2 {flex-shrink: 1;}
```

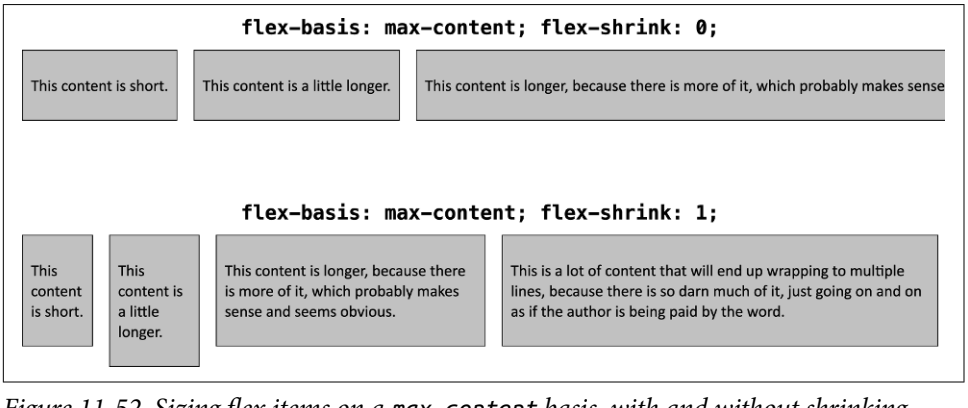


Figure 11-52. Sizing flex items on a `max-content` basis, with and without shrinking

In the first example, where shrinking is not allowed, each flex item is as wide as its content can get without wrapping. This causes the flex items to overflow the container (because `flex-wrap` is not set to `wrap`). In the second example, where `flex-shrink` is set to `1`, the browser shrinks the flex items equally until they all fill out the flex container without overflowing it. Note that the second of the four items is a little taller than the others, because its shrinking happens to require wrapping the content to one more line than the other items.

For a `min-content` flex basis, the reverse happens. Consider the following, illustrated in Figure 11-53:

```
#example1 {flex-basis: min-content; flex-grow: 0;}
#example2 {flex-grow: 1;}
```

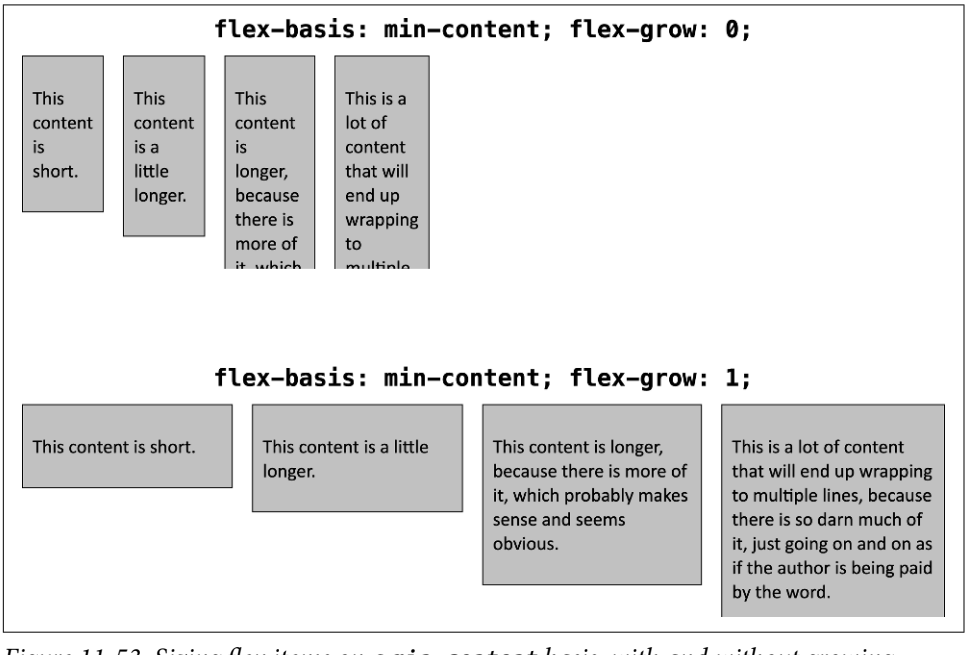


Figure 11-53. Sizing flex items on a `min-content` basis, with and without growing

In the first example, flex items are as narrow as possible while still fitting their content. For elements containing text, this makes them very tall, given that the block axis is vertical. (Note that the full heights of the flex items in the first example have been clipped to keep the figure to a reasonable size.) In the second example, the items are allowed to grow, so they start from the `min-content` size and their widths are grown equally until they all fill out the flex container without overflowing it.

In the browser used to create Figure 11-53, the widths of the flex items in the first example added up to 361.1 pixels (rounded to the nearest tenth of a pixel), with 20 pixels of space between each flex item. This means from the left edge of the first item to the right edge of the last item is about 420.1 pixels. To arrive at the result in the second example, given that the width of the flex container is 1,200 pixels, the difference between the container width and content width is 1,200 – 420.1 = 778.9 pixels. This difference is divided by 4, yielding approximately 194.7 pixels, and the width of each of the four flex items is increased by that amount.

Automatic flex basis

When set to `auto`, whether explicitly or by default, `flex-basis` is the same as the main-axis size of the element, had the element not been turned into a flex item. For length values, `flex-basis` resolves to the `width` or `height` value, with the exception that when the value of `width` or `height` is `auto`, the flex-basis value falls back to `content`.

When the flex basis is `auto`, and all the flex items can fit within the parent flex container, the flex items will be their preflexed size. If the flex items don't fit into their parent flex container, the flex items within that container will shrink proportionally based on their nonflexed main-axis sizes (unless the shrink factor is 0).

When there are no other properties setting the main-axis size of the flex items (that is, there's no `inline-size`, `min-inline-size`, `width`, or `min-width` set on these flex items), and `flex-basis: auto` or `flex: 0 1 auto` is set, the flex items will be only as wide as they need to be for the content to fit, as seen in the first example in Figure 11-54. In this case, they are the width of the text "flex-basis: auto," which is approximately 110 pixels. The flex items are their preflexed size, as if set to `display: inline-block`. In this example, they're grouped at main-start because the `justify-content` property, the flex container's `justify-content` defaults to `flex-start`.

In the second example in Figure 11-54, each flex item has a flex basis of `auto` and an explicitly declared width. The main-axis size of the elements, had they not been turned into flex items, would be 100, 150, and 200 pixels, respectively. And so they are here, since they fit into the flex container without any overflow along the main-axis.

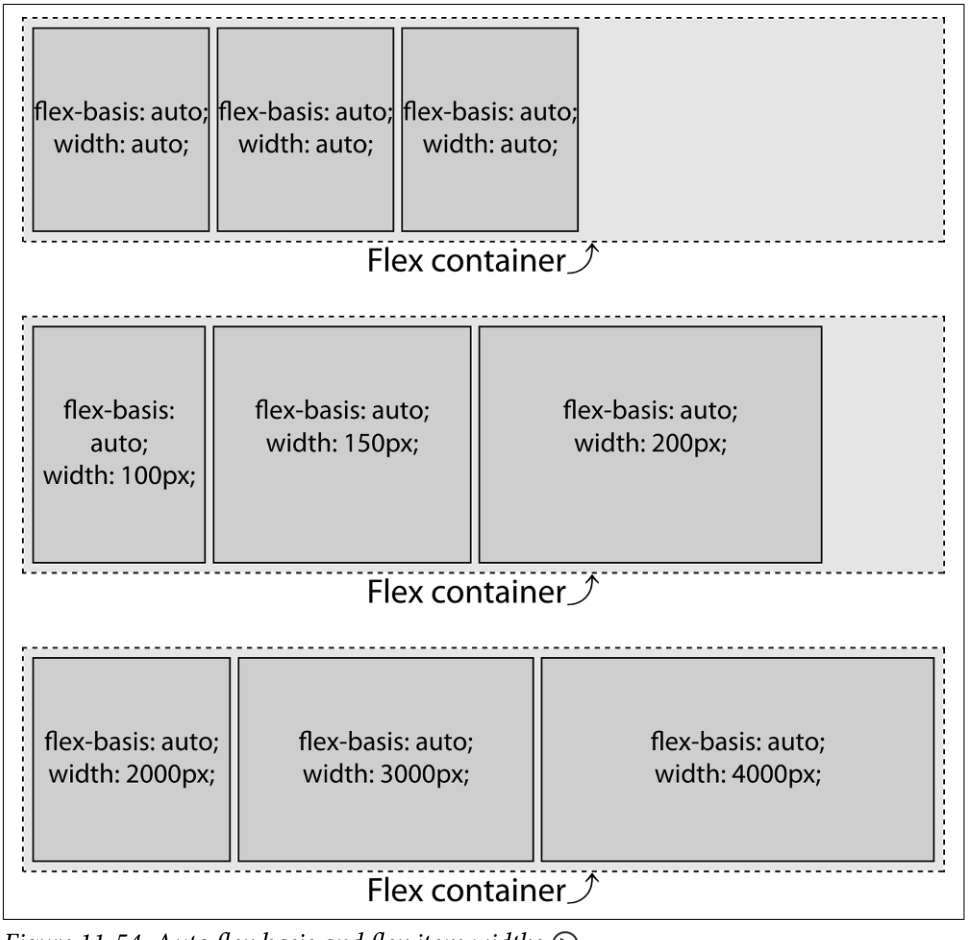


Figure 11-54. Auto flex basis and flex item widths

In the third example in Figure 11-54, each of the flex items has a flex basis of `auto` and a very large explicitly declared width. The main-axis size of the elements, had they not been turned into flex items, would be 2,000, 3,000, and 4,000 pixels, respectively. Since they could not possibly fit into the flex container without overflowing along the main-axis, and their flex shrink factors have all defaulted to `1`, they shrink until they fit into the flex container. You can do the math to find out how big they are using the process outlined in "Differing basis values" on page 516; as a hint, the width of the third flex item should be reduced from 4,000 pixels to 240 pixels.

Default values

When neither `flex-basis` nor `flex` is set, the flex item's main-axis size is the preflex size of the item, as the default value is `auto`.

In Figure 11-55: the flex basis values are defaulting to `auto`, the growth factor is defaulting to `0`, and the shrink factor of each item is defaulting to `1`. For each flex item, the flex basis is its individual `width` value. That means the flex basis values are being set to the values of the `width` properties: 100, 200, and 300 pixels in the first example, and 200, 400, and 200 pixels in the second example. As the combined widths of the flex items are 600 pixels and 800 pixels, respectively, both of which are greater than the main-axis size of the 540-pixel-wide containers, they are all shrinking proportionally to fit.

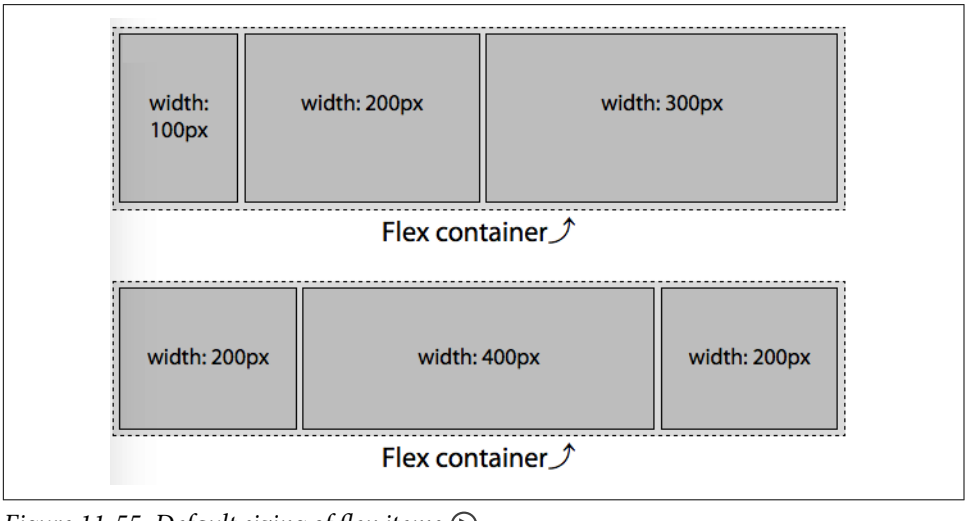


Figure 11-55. Default sizing of flex items

In the first example, we are trying to fit 600 pixels in 540 pixels, so each flex item will shrink by 10% to yield flex items that are 90, 180, and 270 pixels wide. In the second example, we are trying to fit 800 pixels into 540 pixels, so they all shrink 32.5%, making the flex items' widths 135, 270, and 135 pixels.

Length units

In the previous examples, the `auto` flex basis values defaulted to the declared widths of the various flex items. CSS provides other options; for example, we can use the same length units for our flex-basis value as we do for `width` and `height`.

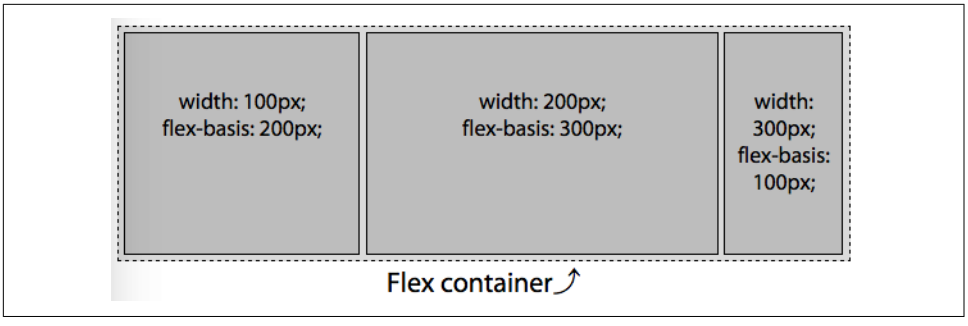


Figure 11-56. Sizing flex items with length-unit flex basis values

When we have both `flex-basis` and `width` (or `height`, for vertical main-axes) values, the basis trumps the width (or height). Let's add basis values to the first example from Figure 11-55. The flex items include the following CSS:

```
flex-container {
  width: 540px;
}
item1 {
  width: 100px;
  flex-basis: 300px;  /* flex: 0 1 300px; */
}
item2 {
  width: 200px;
  flex-basis: 200px;  /* flex: 0 1 200px; */
}
item3 {
  width: 300px;
  flex-basis: 100px;  /* flex: 0 1 100px; */
}
```

The widths are overridden by the basis values. The flex items shrink down to 270 pixels, 180 pixels, and 90 pixels, respectively. Had the container *not* had a constrained width, the flex items would have been 300 pixels, 200 pixels, and 100 pixels, respectively.

While the declared flex basis can override the main-axis size of flex items, the size can be affected by other properties, such as `min-width`, `min-height`, `max-width`, and `max-height`. These are not ignored. Thus, for example, an element might have `flex-basis: 100px` and `min-width: 500px`. The minimum width of `500px` will be respected, even though the flex basis is smaller.

Percentage units

Percentage values for `flex-basis` are calculated relative to the size of the main dimension of the flex container.

We've already seen the first example in Figure 11-57; it's included here to recall that the width of the text "flex-basis: auto" in this case is approximately 110 pixels wide. In this case only, declaring `flex-basis: auto` looks the same as writing `flex-basis: 110px`:

```
flex-container {
  width: 540px;
}
flex-item {
  flex: 0 1 100%;
}
```

In the second example in Figure 11-57, the first two flex items have a flex basis of `auto` with a default `width` of `auto`, which is as if their flex basis were set to `content`. As we've noted previously, the `flex-basis` of the first two items ends up being the equivalent of 110 pixels, as the content in this case happens to be 110 pixels wide. The last item has its `flex-basis` set to `100%`.

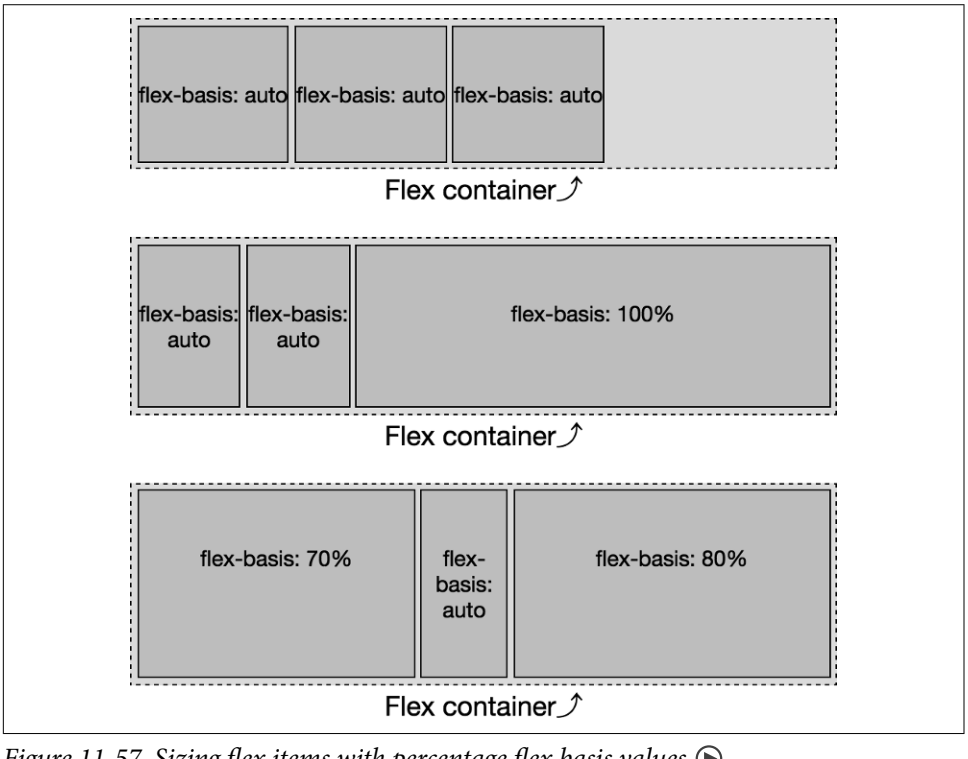


Figure 11-57. Sizing flex items with percentage flex basis values

The percentage value is relative to the parent, which is 540 pixels. The third flex item, with a basis of `100%`, is not the only flex item within the nonwrapping flex container. Thus, it will not grow to be 100% of the width of the parent flex container *unless* its shrink factor is set with a null shrink factor, meaning it can't shrink, or if it contains nonwrappable content that is as wide or wider than the parent container.



Remember: when the flex basis is a percent value, the main-axis size is relative to the parent, which is the flex container.

With our three flex basis values, if the content is indeed 110 pixels wide, and the container is 540 pixels wide (ignoring other box-model properties for simplicity's sake), we have a total of 760 pixels to fit in a 540-pixel space. Thus we have 220 pixels of negative space to distribute proportionally. The shrink factor is as follows:

Shrink factor = 220 px ÷ 760 px = 28.95%

Each flex item will be shrunk by 28.95%, becoming 71.05% of the width it would have been, had it not been allowed to shrink. We can figure the final widths:

item1 = 110 px × 71.05% = 78.16 px
item2 = 110 px × 71.05% = 78.16 px
item3 = 540 px × 71.05% = 383.68 px

These numbers hold true as long as the flex items can be that small—that is, as long as none of the flex items contain media or nonbreaking text wider than 78.16 pixels or 383.68 pixels. This is the widest these flex items will be as long as the content can wrap to be that width or narrower. We say "widest" because if one of the other two flex items can't shrink to be as narrow as this value, they'll both have to absorb some of that negative space.

In the third example in Figure 11-57, the `flex-basis: auto` item wraps over three lines. The CSS for this example is the equivalent of the following:

```
flex-container {
  width: 540px;
}
item1 {
  flex: 0 1 70%;
}
item2 {
  flex: 0 1 auto;
}
item3 {
  flex: 0 1 80%;
}
```

We declare the `flex-basis` of the three flex items to be `70%`, `auto`, and `80%`, respectively. Remembering that in our scenario `auto` is the width of the nonwrapping content, which in this case is approximately 110 pixels, and our flex container is 540 pixels, the basis values are equivalent to the following:

item1 = 70% × 540 px = 378 px
item2 = widthOfText("flex-basis: auto") ≈ 110 px
item3 = 80% × 540 px = 432 px

When we add the widths of these three flex items' basis values, they have a total combined width of 920 pixels, which needs to fit into a flex container 540 pixels wide. Thus we have 380 pixels of negative space to remove proportionally among the three flex items. To figure out the ratio, we divide the available width of our flex container by the sum of the widths of the flex items that they would have if they couldn't shrink:

Proportional width = 540 px ÷ 920 px = 0.587

Because the shrink factors are all the same, this is fairly simple. Each item will be 58.7% of the width it would be if it had no flex-item siblings:

item1 = 378 px × 58.7% = 221.8 px
item2 = 110 px × 58.7% = 64.6 px
item3 = 432 px × 58.7% = 253.6 px

What happens when the container is a different width? Say, 1,000 pixels? The flex basis would be 700 pixels (70% × 1,000 pixels), 110 pixels, and 800 pixels (80% × 1,000 pixels), respectively, for a total of 1,610 pixels:

Proportional width = 1,000 px ÷ 1,610 px = 0.6211

item1 = 700 px × 62.11% = 434.8 px
item2 = 110 px × 62.11% = 68.3 px
item3 = 800 px × 62.11% = 496.9 px

Because with a basis of 70% and 80%, the combined basis values of the flex items will always be wider than 100%, no matter how wide we make the parent, all three items will always shrink.

If the first flex item can't shrink for some reason—whether because of unshrinkable content, or another bit of CSS setting its `flex-shrink` to `0`—it will be 70% of the width of the parent, 378 pixels in this case. The other two flex items must shrink proportionally to fit into the remaining 30%, or 162 pixels. In this case, we expect widths to be 378 pixels, 32.875 pixels, and 129.125 pixels. As the text "basis:" is wider than that—assume 42 pixels—we get 378 pixels, 42 pixels, and 120 pixels. Figure 11-58 shows the result.

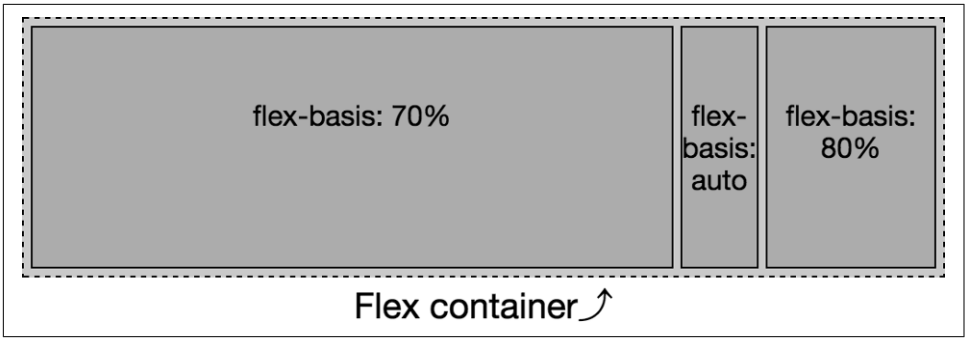


Figure 11-58. While the percentage value for `flex-basis` is relative to the width of the flex container, the main-axis size is impacted by its siblings ▶

Testing this out on your device will likely have slightly different results, as the width of the text "flex-basis: auto" may not be the same for you, depending on the font that gets used to render the text. (We used Myriad Pro, with fallbacks to Helvetica and any generic sans-serif font.)

Zero basis

If neither the `flex-basis` property nor the `flex` shorthand is included at all, the flex basis defaults to `auto`. When the `flex` property is included, but the flex basis component of the shorthand is omitted from the shorthand, the basis defaults to `0`. While on the surface you might think the two values of `auto` and `0` are similar, the `0` value is actually very different and may not be what you expect.

In the case of `flex-basis: auto`, the basis is the main size of the flex items' contents. If the basis of each of the flex items is `0`, the available space is the main-axis size of the entire flex container. In either case, the available space is distributed proportionally, based on the growth factors of each flex item.

With a basis of `0`, the size of the flex container is divided up and distributed proportionally to each flex item based on its growth factors—its default original main-axis size as defined by `height`, `width`, or `content` is not taken into account, though `min-width`, `max-width`, `min-height`, and `max-height` do impact the flexed size.

As shown in Figure 11-59, when the basis is `auto`, only the extra space is divided up proportionally and added to each flex item set to grow. Again, assuming the width of the text "flex: *X X* auto" is 110 pixels, in the first example we have 210 pixels to distribute among six growth factors, or 35 pixels per growth factor. The flex items are 180, 145, and 215 pixels wide, respectively.

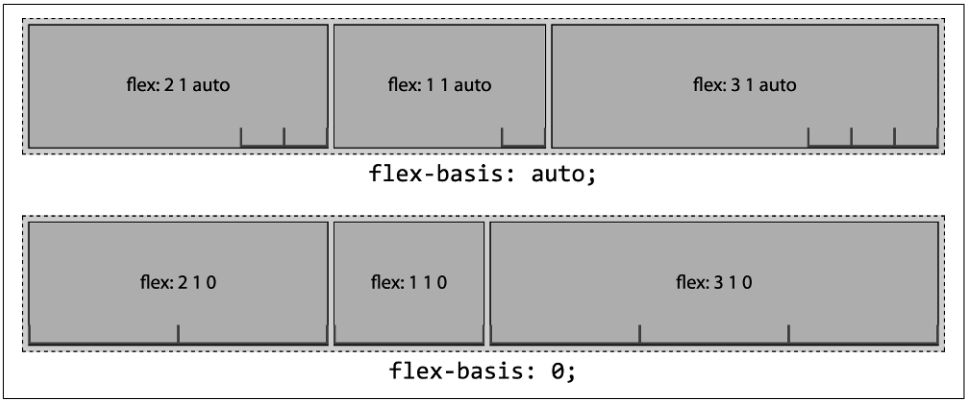


Figure 11-59. Flex growth in auto and zero flex basis values

In the second example, when the basis is 0, all 540 pixels of the width is distributable space. With 540 pixels of distributable space among six growth factors, each growth factor is worth 90 pixels. The flex items are 180, 90, and 270 pixels wide, respectively. While the middle flex item is 90 pixels wide, the content in this example is narrower than 110 pixels, so the flex item didn't wrap.

The flex Shorthand

Now that you have a fuller understanding of the properties that make up the `flex` shorthand, remember: *always use the `flex` shorthand.* It accepts the usual global property values, including `initial`, `auto`, `none`; and the use of an integer, usually `1`, meaning the flex item can grow.

flex

Values	none | [<*flex-grow*> <*flex-shrink*>? || <*flex-basis*>]
Initial value	`0 1 auto`
Applies to	Flex items
Computed value	See individual properties, with the caveat that relative lengths for `flex-basis` are converted to absolute lengths
Inherited	No
Animatable	Yes

Four of the flex values provide the most commonly desired effects:

`flex: initial`
: Equivalent to `flex: 0 1 auto`. This sizes flex items based on the value of `inline-size` (which is equivalent to either `width` or `height`, depending on the direction of the inline axis), and allows shrinking but not growing.

`flex: auto`
: Equivalent to `flex: 1 1 auto`. This sizes flex items based on the value of `inline-size`, but makes them fully flexible, allowing both shrinking and growing.

`flex: none`
: Equivalent to `flex: 0 0 auto`. This sizes flex items based on the value of `inline-size`, but makes them completely inflexible: they can't shrink or grow.

`flex: <number>`
: Equivalent to `flex: <number> 1 0`. This value sets the flex item's growth factor to the `<number>` provided. It also sets both the shrink factor and flex basis to `0`. This means the value of `inline-size` acts as a minimum size, but the flex item will grow if there is room to do so.

Let's consider each of these in turn.

Flexing with initial

The global CSS keyword `initial` can be used on all properties to represent a property's initial value (its specification default value). Thus, the following lines are equivalent:

```
flex: initial;
flex: 0 1 auto;
```

Declaring `flex: initial` sets a null growth factor, a shrink factor of `1`, and the flex basis values to `auto`. In Figure 11-60, we can see the effect of the `auto` flex basis values. In the first two examples, the basis of each flex item is `content`—with each flex item having the width of the single line of letters that makes up the content. However, in the last two examples, the flex basis values of all the items are equal at 50 pixels, since `width: 50px` has been applied to all the flex items. The `flex: initial` declaration sets the `flex-basis` to `auto`, which we previously saw is the value of the `width` (or `height`), if declared, or `content` if not declared.

In the first and third of these examples, we see that when the flex container is too small to fit all the flex items at their default main-axis size, the flex items shrink so that all fit within the parent flex container. In these examples, the combined flex basis values of all the flex items is greater than the main-axis size of the flex container. In the first example, the width of each flex item varies based on the width of each item's content and its ability to shrink. They all shrink proportionally based on their shrink factor, but not narrower than their widest content. In the third example, with each flex item's flex-basis being 50 pixels (because of the value of `width`), all the items shrink equally.

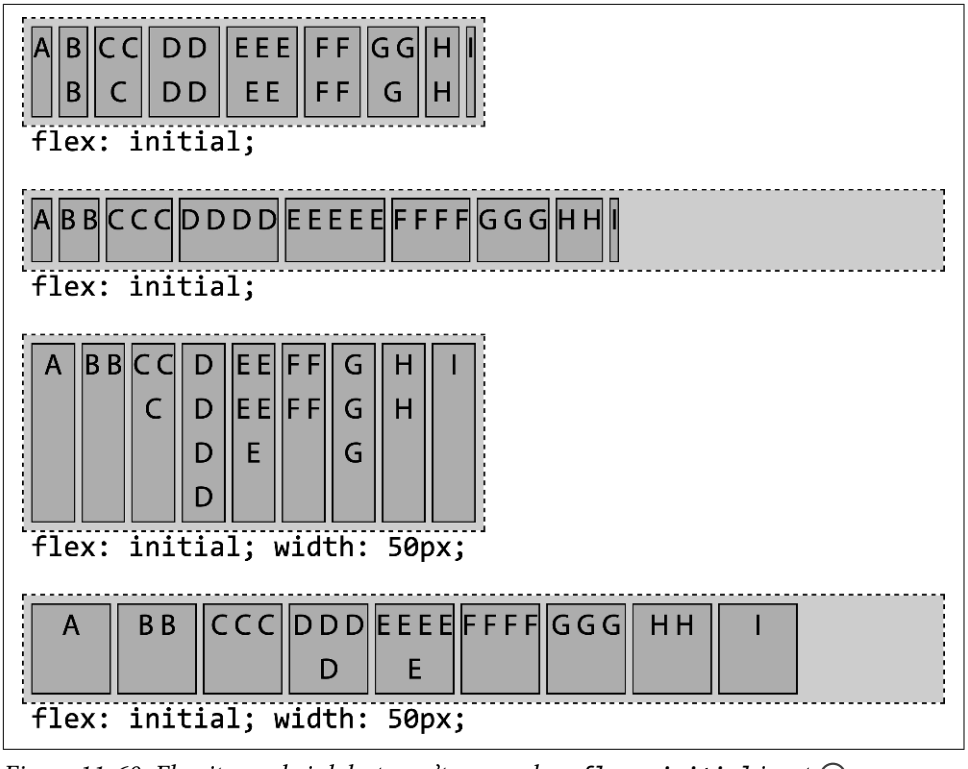


Figure 11-60. Flex items shrink but won't grow when `flex: initial` is set ▶

Flex items, by default, are grouped at main-start, as `flex-start` is the default value for the `justify-content` property. This is noticeable only when the combined main-axis sizes of the flex items in a flex line are smaller than the main-axis size of the flex container, and none of the flex items are able to grow.

Flexing with auto

The `flex: auto` option is similar to `flex: initial`, but makes the flex items flexible in both directions: they'll shrink if there isn't enough room to fit all the items within the container, and they'll grow to take up all the extra space within the container if there is distributable space. The flex items absorb any free space along the main-axis. The following two statements are equivalent:

```
flex: auto;
flex: 1 1 auto;
```

Figure 11-61 shows a variety of scenarios using `auto` flexing.

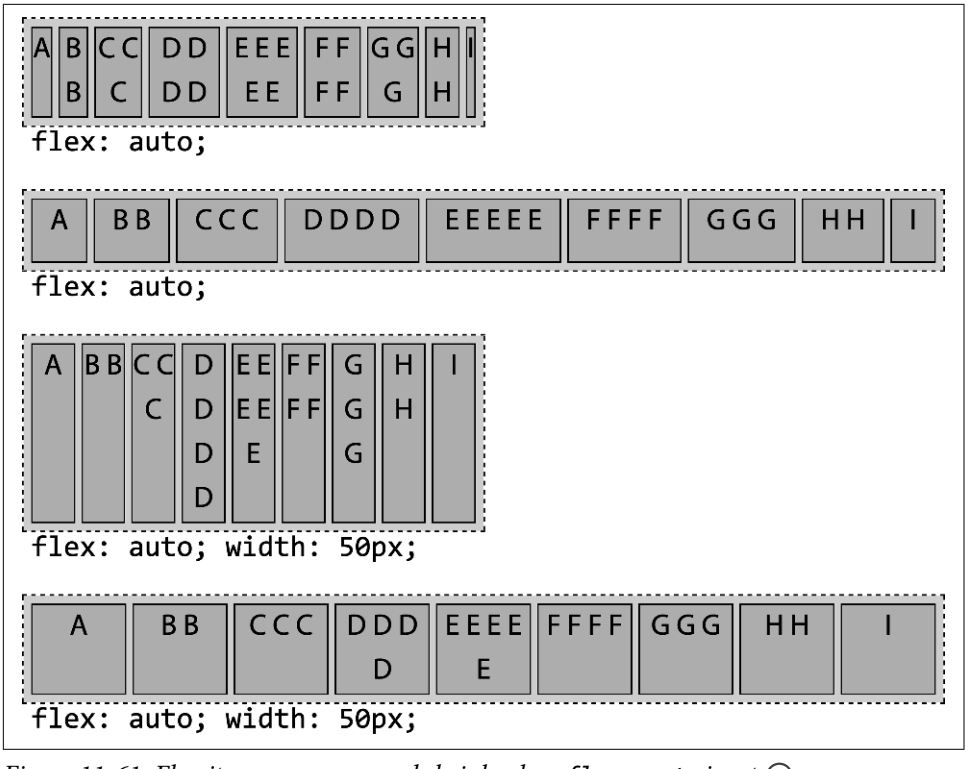


Figure 11-61. Flex items can grow and shrink when `flex: auto` is set

The first and third examples of Figure 11-61 are identical to the examples in Figure 11-60, as the shrinking and basis values are the same. However, the second and fourth examples are different. This is because when `flex: auto` is set, the growth factor is `1`, and the flex items therefore can grow to incorporate all the extra available space.

Preventing flexing with none

Any `flex: none` flex items are inflexible: they can neither shrink nor grow. The following two lines of CSS are equivalent:

```
flex: none;
flex: 0 0 auto;
```

Figure 11-62 shows the effects of none.

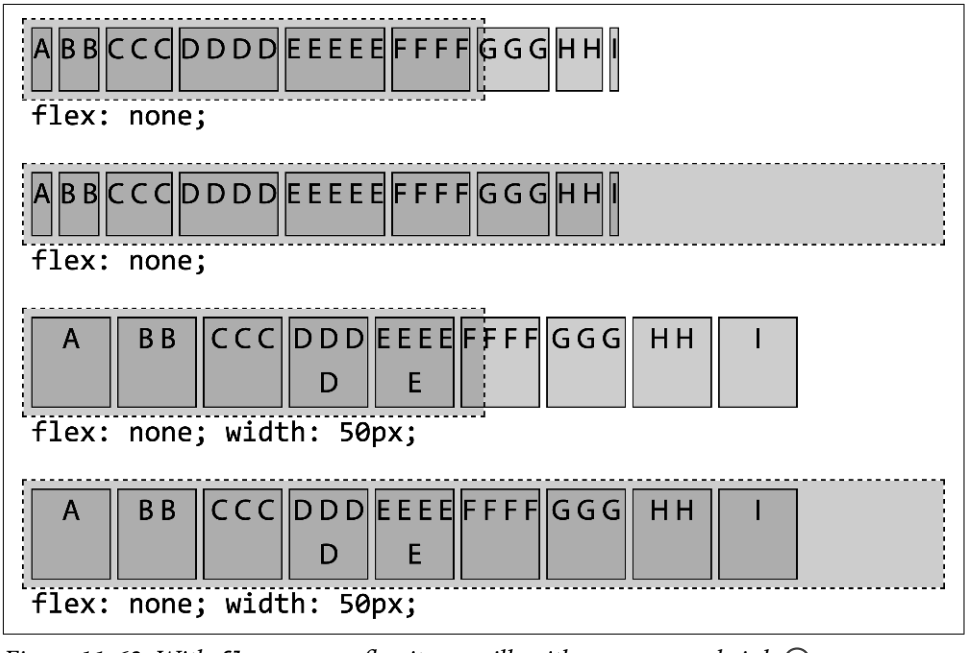


Figure 11-62. With `flex: none`, flex items will neither grow nor shrink ▶

As demonstrated in the first and third examples of Figure 11-62, if there isn't enough space, the flex items overflow the flex container. This is different from `flex: initial` and `flex: auto`, which both set a positive shrink factor.

The basis resolves to `auto`, meaning each flex item's main-axis size is determined by the main-axis size of the element had it not been turned into a flex item. The flex-basis resolves to the `width` or `height` value of the element. If that value is `auto`, the basis becomes the main-axis size of the content. In the first two examples, the basis—and the width, since there is no growing or shrinking—is the width of the content. In the third and fourth examples, the width and basis are all 50 pixels, because that's the value of the `width` property applied to them.

Numeric flexing

When the value of the `flex` property is a single, positive, numeric value, that value will be used for the growth factor, while the shrink factor will default to `1` and the basis will default to `0`. The following two CSS declarations are equivalent:

```
flex: 3;
flex: 3 1 0;
```

This makes the flex item on which it is set flexible: it can grow. The shrink factor is actually moot: the flex basis is set to `0`, so the flex item can grow only from that basis.

In the first two examples in Figure 11-63, all the flex items have a flex growth factor of `3`. The flex basis is `0`, so they don't "shrink"; they just grow equally from 0 pixels wide until the sum of their main-axis sizes to fill the container along the main-axis. With all the flex items having a basis of `0`, 100% of the main dimension is distributable space. The main-axis size of the flex items is wider in this second example because the wider flex container has more distributable space.

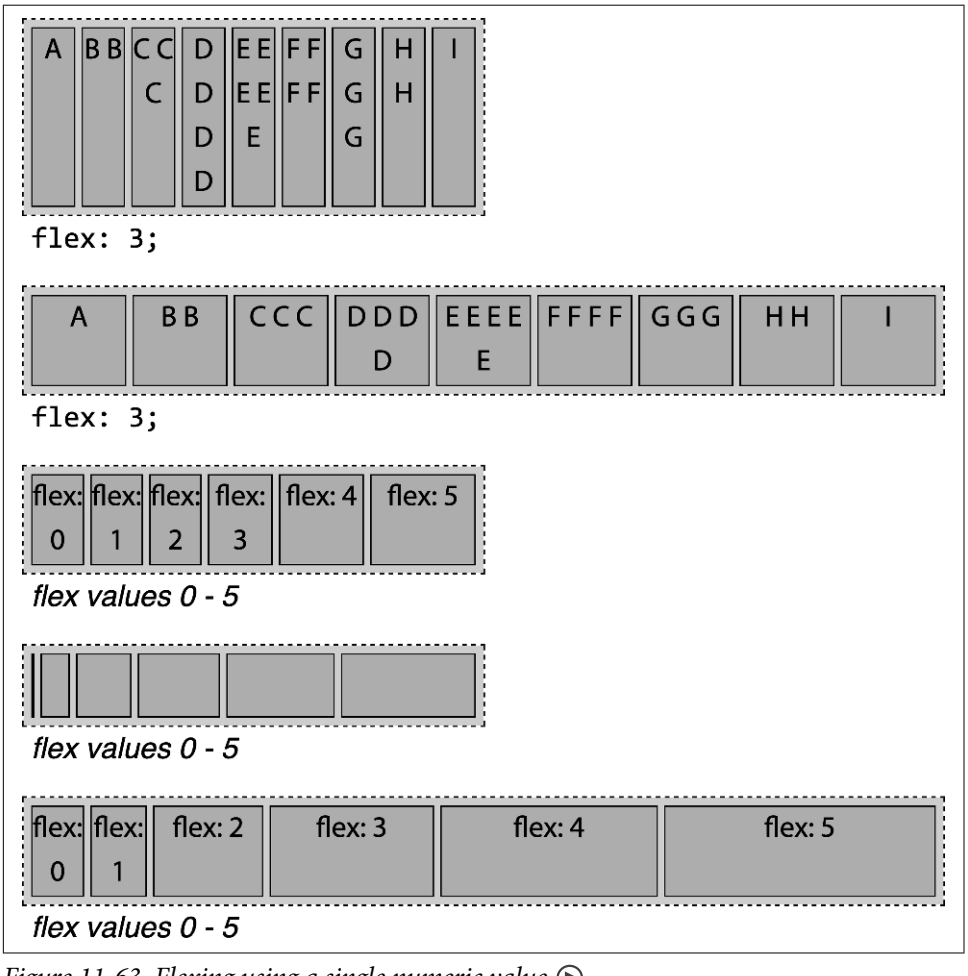


Figure 11-63. Flexing using a single numeric value ▶

Any numeric value that is greater than 0, even 0.1, means the flex item can grow. If there is available space to grow and only one flex item has a positive growth factor, that item will take up all the available space. If multiple flex items can grow, the available extra space will be distributed proportionally to each flex item based on its growth factor.

The last three examples of Figure 11-63 declare six flex items with `flex: 0`, `flex: 1`, `flex: 2`, `flex: 3`, `flex: 4`, and `flex: 5`, respectively. These are the growth factors for the flex items, with each having a shrink factor of `1` and a flex basis of `0`. The main-axis size of each is proportional to the specified flex growth factor. You might assume that the `flex: 0` item with the text "flex: 0" in the third example will be 0 pixels wide, as in the fourth example—but, by default, flex items won't shrink below the length of the longest word or fixed-size element.



We added a bit of padding, margins, and borders to the figures to make the visuals more pleasing. For this reason, the leftmost flex item, with `flex: 0` declared, is visible: it has a 1-pixel border making it visible, even though it's 0 pixels wide.

The order Property

Flex items are, by default, displayed and laid out in the same order as they appear in the source code. The order of flex items and flex lines can be reversed with `flex-direction`, but sometimes you want a rearrangment that's a little more complicated. The `order` property can be used to change the ordering of individual flex items.

order

Values	*<integer>*
Initial value	`0`
Applies to	Flex items and absolutely positioned children of flex containers
Computed value	As specified
Inherited	No
Animatable	Yes

By default, all flex items are assigned the order of `0`, with the flex items all assigned to the same ordinal group and displayed in the same order as their source order, along the direction of the main-axis. (This has been the case for all the examples throughout this chapter.)

To change the visual order of a flex item, set the `order` property value to a nonzero integer. Setting the `order` property on elements that are not children of a flex container has no effect on such elements.



Changing the visual rendering order of flex items creates a disconnect between the source order of elements and their visual presentation. This can, in the words of the Mozilla Developer Network's article on `order`, "adversely affect users experiencing low vision navigating with the aid of assistive technology such as a screen reader." It could also create problems for users who navigate by keyboard and use a zoomed-in or otherwise magnified view of pages. In other words: be very careful with `order`, and use it *only* in production after much accessibility testing.

The value of the `order` property specifies an *ordinal group* to which the flex item belongs. Any flex items with a negative value will appear to come before those defaulting to `0` when drawn to the page, and all the flex items with a positive value will appear to come after those defaulting to `0`. While visually altered, the source order remains the same. Screen readers and tabbing order remain as defined by the source order of the HTML.

For example, if you have a group of 12 items, and you want the seventh to come first and the sixth to be last, you would declare the following:

```
ul {
  display: inline-flex;
}
li:nth-of-type(6) {
  order: 1;
}
li:nth-of-type(7) {
  order: -1;
}
```

In this scenario, we are explicitly setting the order for the sixth and seventh list items, while the other list items are defaulting to `order: 0`. Figure 11-64 shows the result.

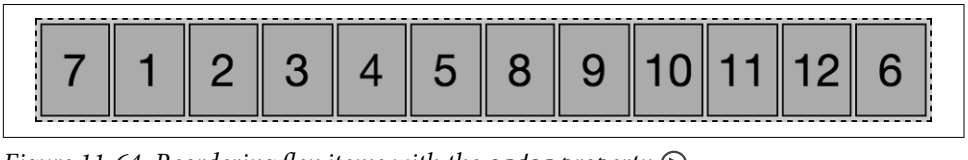


Figure 11-64. Reordering flex items with the `order` property

The seventh flex item is the first to be laid out, because of the negative value of the `order` property, which is less than the default `0`, and is also the lowest value of any of its sibling flex items. The sixth flex item is the only item with a value greater than 0, and therefore has the highest-order value out of all of its siblings. This is why it's laid out after all the other flex items. All the other items, all having the default `order` of `0`, are drawn between those first and last items, in the same order as their source order, since they are all members of the same ordinal group (`0`).

The flex container lays out its content in order-modified document order, starting from the lowest-numbered ordinal group and going up. When multiple flex items have the

same value for the `order` property, the items share an ordinal group. The items in each ordinal group will appear in source order, with the group appearing in numeric order, from lowest to highest. Consider the following:

```
ul {
  display: inline-flex;
  background-color: rgba(0,0,0,0.1);
}
li:nth-of-type(3n-1) {
  order: 3;
  background-color: rgba(0,0,0,0.2);
}
li:nth-of-type(3n+1) {
  order: -1;
  background-color: rgba(0,0,0,0.4);
}
```

By setting the same `order` value to more than one flex item, the items will appear by ordinal group, and by source order within each individual ordinal group. Figure 11-65 shows the result.

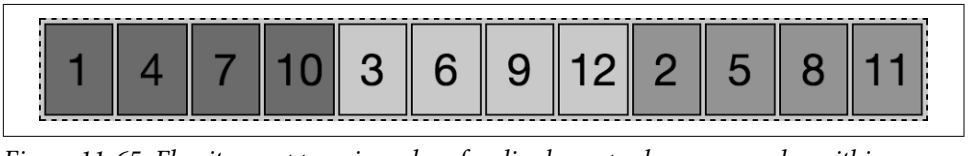


Figure 11-65. Flex items appear in order of ordinal groups, by source order within their group ▶

Here's what happened:

- Items 2, 5, 8, and 11 were selected to share ordinal group `3`, and get a 20% opaque background.
- Items 1, 4, 7, and 10 were selected to share ordinal group `-1`, and get a 40% opaque background.
- Items 3, 6, 9, and 12 were not selected at all. They default to the ordinal group `0`.

The three ordinal groups, then, are `-1`, `0`, and `3`. The groups are arranged in that order. Within each group, the items are arranged by source order.

This reordering is purely visual. Screen readers *should* read the document as it appears in the source code, though they may not. As a visual change, ordering flex items impacts the painting order of the page: the painting order of the flex items is the order in which they appear, as if they were reordered in the source document, even though they aren't.

Changing the layout with the `order` property has no effect on the tab order of the page. If the numbers in Figure 11-65 were links, tabbing through the links would go through the links in the order of the source code, *not* in the order of the layout.

Tabbed Navigation Revisited

Adding to our tabbed navigation bar example in Figure 11-2, we can make the currently active tab appear first, as Figure 11-66 shows:

```
nav {
  display: flex;
  justify-content: flex-end;
  border-bottom: 1px solid #ddd;
}
a {
  margin: 0 5px;
  padding: 5px 15px;
  border-radius: 3px 3px 0 0;
  background-color: #ddd;
  text-decoration: none;
  color: black;
}
a:hover {
  background-color: #bbb;
  text-decoration: underline;
}
a.active {
  order: -1;
  background-color: #999;
}

<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a class="active">Blog</a>
  <a href="/jobs">Careers</a>
  <a href="/contact">Contact Us</a>
</nav>
```

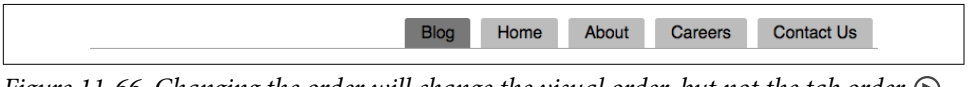


Figure 11-66. Changing the order will change the visual order, but not the tab order

The currently active tab has the `.active` class added, the `href` attribute removed, and the `order` set to `-1`, which is less than the default `0` of the other sibling flex items, meaning it appears first.

Why did we remove the `href` attribute? As the tab is the currently active document, there is no reason for the document to link to itself. But, more importantly, if it was an active link instead of a placeholder link, and the user was using the keyboard to tab through the navigation, the order of appearance is Blog, Home, About, Careers, and Contact Us, with the Blog appearing first; but the tab order would have been Home, About, Blog, Careers, and Contact Us, following the source order rather than the visual order, which can be confusing.

The `order` property can be used to enable marking up the main content area before the side columns for mobile devices and those using screen readers and other assistive technology, while creating the appearance of the common three-column layout: a center main content area, with site navigation on the left and a sidebar on the right, as shown way back in Figure 11-48.

While you can put your footer before your header in your markup, and use `order` to reorder the page, this is an inappropriate use of the property. The `order` property should be used only for visual reordering of content. Your underlying markup should always reflect the logical order of your content. Consider these two markup orders for the same content, shown here side by side to make comparing them easier:

```
<header></header>
<main>
   <article></article>
   <aside></aside>
   <nav></nav>
</main>
<footer></footer>
```

```
<header></header>
<main>
  <nav></nav>
  <article></article>
  <aside></aside>
</main>
<footer></footer>
```

We've been marking up websites in the order we want them to appear, as shown on the right in the code example, which is the same code as in our three-column layout example (Figure 11-48).

It really would make more sense if we marked up the page as shown on the left, with the `<article>` content, which is the main content, first in the source order: this puts the article first for screen readers, search engines, and even mobile devices, but in the middle for our sighted users on larger screens:

```
main {
  display: flex;
}
main > nav {
  order: -1;
}
```

By using the `order: -1` declaration, we are able to make the `<nav>` appear first, as it is the lone flex item in the ordinal group of `-1`. The `<article>` and `<aside>`, with no `order` explicitly declared, default to `order: 0`.

Remember, when more than one flex item is in the same ordinal group, the members of that group are displayed in source order in the direction of main-start to main-end, so the `article` is displayed before the `aside`.

Some developers, when changing the order of at least one flex item, like to give all flex items an `order` value for better markup readability. We could have also written this:

```
main {
  display: flex;
}
main > nav {
  order: 1;
```

```
}
main > article {
  order: 2;
}
main > aside {
  order: 3;
}
```

In previous years, before browsers supported flex, all this could have been done with floats: we would have set `float: right` on the `<nav>`. While doable, flex layout makes this sort of layout much simpler, especially if we want all three columns—the `<aside>`, `<nav>`, and `<article>`—to be of equal heights.

Summary

With flexible box layout, you can lay out sibling elements in ways that are responsive to many layout contexts and writing modes, with a variety of options for arranging those elements and aligning them to one another. It makes the task of vertically centering elements within their parent elements almost trivially easy, something that was very difficult in the years before flexbox. It also serves as a powerfully useful bridge between normal-flow and grid layout, which is the subject of the next chapter.

CHAPTER 12
Grid Layout

At its inception, CSS had a layout-shaped hole at its center. Designers bent other features to the purposes of layout, most notably `float` and `clear`, and generally hacked their way around that hole. Flexbox layout helped to fill it, but flexbox is really meant for specific use cases, like navigation bars (navbars), as shown in Chapter 11.

Grid layout, by contrast, is a *generalized* layout system. With its emphasis on rows and columns, it might at first feel like a return to table layout—and in certain ways that's not too far off—but there is far, far more to grid layout than table layout. Grid allows pieces of the design to be laid out independently of their document source order, and even overlap pieces of the layout, if that's your wish. CSS provides powerfully flexible methods for defining repeating patterns of grid lines, attaching elements to those grid lines, and more. You can nest grids inside grids, or for that matter, attach tables or flexbox containers to a grid. And much, much more.

In short, grid layout was the layout system we long waited for, and in 2017, it landed in all the major browser engines. It takes many, many layouts that were difficult, or even impossible, and invariably fragile, and allows you to create them simply, flexibly, and robustly.

Creating a Grid Container

The first step to creating a grid is defining a *grid container*. This is much like a containing block in positioning, or a flex container in flexible-box layout: a grid container is an element that defines a *grid formatting context* for its contents.

At this basic level, grid layout is quite reminiscent of flexbox. For example, the child elements of a grid container become *grid items*, just as the child elements of a flex container become flex items. The children of those grid items do *not* become grid elements—although any grid item can itself be made a grid container, and thus have its child elements become grid items to the nested grid. It's possible to nest grids inside grids, until it's grids all the way down.

CSS has two kinds of grids: *regular* grids and *inline* grids. These are created with special values for the `display` property: `grid` and `inline-grid`. The first generates a block-level box, and the second an inline-level box. Figure 12-1 illustrates the difference.

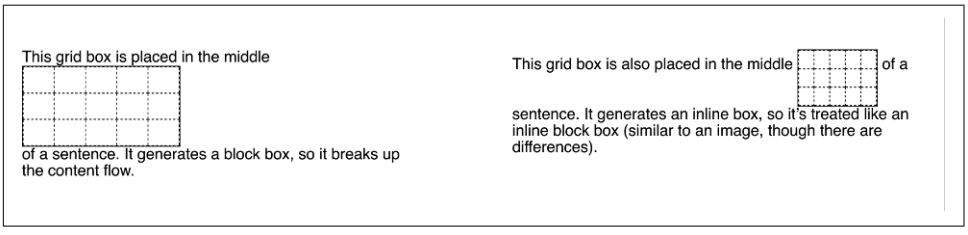


Figure 12-1. Grids and inline grids

These are very similar to the `block` and `inline-block` values for `display`. Most grids you create are likely to be block-level, though the ability to create inline grids is always there should you need it.

Although `display: grid` creates a block-level grid, the specification is careful to explicitly state that "grid containers are not block containers." Although the grid box participates in layout much as a block container does, there are differences between them.

First off, floated elements do not intrude into the grid container. What this means in practice is that a grid will not slide under a floated element, as a block container will. See Figure 12-2 for a demonstration of the difference.

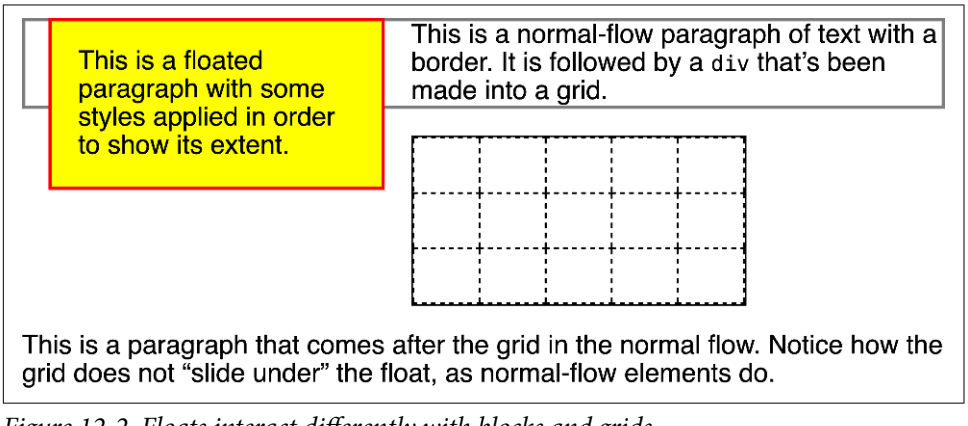


Figure 12-2. Floats interact differently with blocks and grids

Furthermore, the margins of a grid container do not collapse with the margins of its descendants. Again, this is distinct from block boxes, whose margins do (by default) collapse with descendants. For example, the first list item in an ordered list may have a top margin, but this margin will collapse with the list element's top margin. The top margin of a grid item will *never* collapse with the top margin of its grid container. Figure 12-3 illustrates the difference.

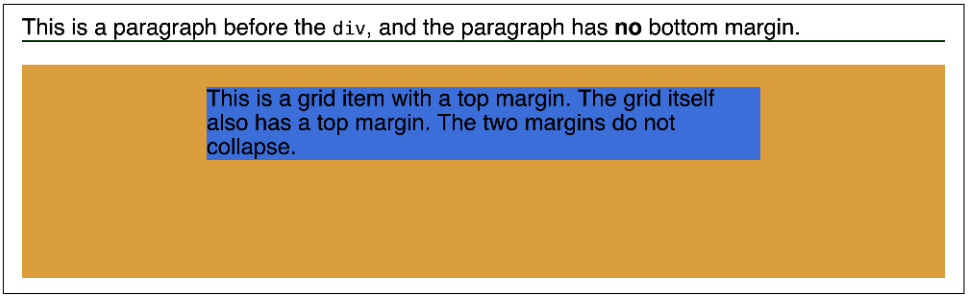


Figure 12-3. Margin collapsing and the lack thereof

A few CSS properties and features do not apply to grid containers and grid items:

- All `column` properties (e.g., `column-count`, `columns`, etc.) are ignored when applied to a grid container. (You can learn more about multicolumn properties at CSS Multi-Column Layout.)
- The `::first-line` and `::first-letter` pseudo-elements do not apply to grid containers and are ignored.
- `float` and `clear` are effectively ignored for grid items (though not grid containers). Despite this, the `float` property still helps determine the computed value of the `display` property for children of a grid container, because the `display` value of the grid items is resolved *before* they're made into grid items.
- The `vertical-align` property has no effect on the placement of grid items, though it may affect the content inside the grid item. (Don't worry: we'll talk later about other, more powerful ways to align grid items.)

Lastly, if a grid container's declared `display` value is `inline-grid` *and* the element is either floated or absolutely positioned, the computed value of `display` becomes `grid` (thus dropping `inline-grid`).

Once you've defined a grid container, the next step is to set up the grid within. Before we explore how that works, though, it's necessary to cover some terminology.

Understanding Basic Grid Terminology

We've already talked about grid containers and grid items, but let's define them in a bit more detail. As we said before, a *grid container* is a box that establishes a *grid-formatting context*—that is, an area in which a grid is created and elements are laid out according to the rules of grid layout instead of block layout. You can think of it like the way an element set to `display: table` creates a table-formatting context within it. Given the grid-like nature of tables, this comparison is fairly apt, though be sure not to make the assumption that grids are just tables in another form. Grids are far more powerful than tables ever were.

A *grid item* is a thing that participates in grid layout within a grid-formatting context. This is usually a child element of a grid container, but it can also be the anonymous (that is, not contained within an element) bits of text that are part of an element's content. Consider the following, which has the result shown in Figure 12-4:

```
#warning {display: grid;
    background: #FCC; padding: 0.5em;
    grid-template-rows: 1fr;
    grid-template-columns: repeat(7, 1fr);}

<p id="warning"><img src="warning.svg"><strong>Note:</strong> This element is a
   <em>grid container</em> with several <em>grid items</em> inside it.</p>
```

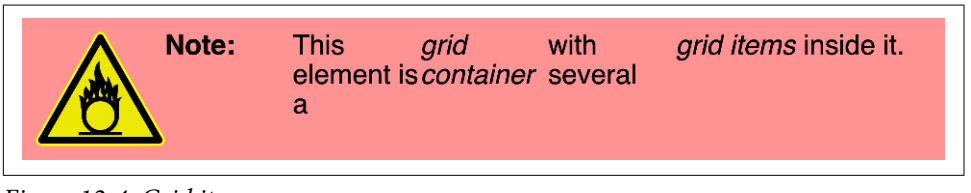


Figure 12-4. Grid items

Notice how each element, *and* each bit of text between the elements, has become a grid item. The image is a grid item, just as much as the elements and text runs—seven grid items in all. Each will participate in the grid layout, although the anonymous text runs will be much more difficult (or impossible) to affect with the various grid properties we'll discuss.



If you're wondering about `grid-template-rows` and `grid-template-columns`, we'll tackle them in the next section.

In the course of using those properties, you'll create or reference several core components of grid layout. These are summarized in Figure 12-5.

The most fundamental unit is the *grid line*. By defining the placement of one or more grid lines, you implicitly create the rest of the grid's components:

Grid track
: A continuous run between two adjacent grid lines—in other words, a *grid column* or a *grid row*. It goes from one edge of the grid container to the other. The size of a grid track is dependent on the placement of the grid lines that define it. These are analogous to table columns and rows. More generically, these can be referred to as *block-axis* and *inline-axis* tracks, where (in Western languages) column tracks are on the block axis and row tracks are on the inline axis.

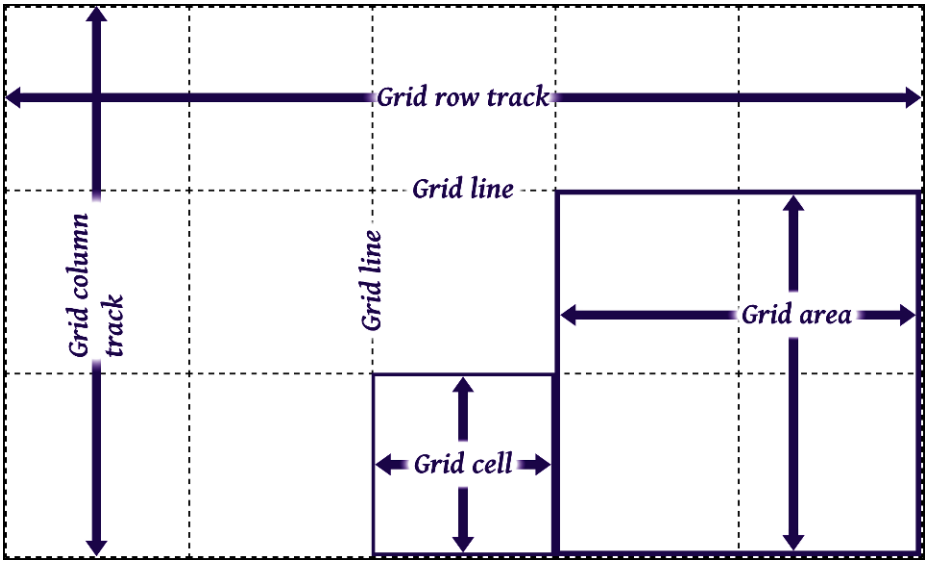


Figure 12-5. Grid components

Grid cell
: Any space bounded by four grid lines, with no grid lines running through it, analogous to a table cell. This is the smallest unit of area in grid layout. Grid cells cannot be directly addressed with CSS grid properties; that is, no property allows you to say a grid item should be associated with a given cell. (But see the next point for more details.)

Grid area
: Any rectangular area bounded by four grid lines and made up of one or more grid cells. An area can be as small as a single cell or as large as all the cells in the grid. Grid areas *are* directly addressable by CSS grid properties, which allow you to define the areas and then associate grid items with them.

An important point to note is that these grid tracks, cells, and areas are entirely constructed of grid lines—and, more importantly, do not have to correspond to grid items. There is no requirement that all grid areas be filled with an item; it is perfectly possible to have some or even most of a grid's cells be empty of any content. You can also have grid items overlap each other, either by defining overlapping grid areas or by using grid-line references that create overlapping situations.

Another fact to keep in mind is that you can define as many or as few grid lines as you wish. You could literally define just a set of vertical grid lines, thus creating a bunch of columns and only one row. Or you could go the other way, creating a bunch of row tracks and no column tracks (though there would be one, stretching from one side of the grid container to the other).

The flip side is that if you create a condition preventing a grid item from being placed within the column and row tracks you define, or if you explicitly place a grid item outside those tracks, new grid lines and tracks will be automatically added to the grid to accommodate, creating implicit grid tracks (a subject we'll return to later in the chapter).

Creating Grid Lines

It turns out that creating grid lines can get fairly complex. That's not so much because the concept is difficult. CSS just provides many ways to get it done, and each uses its own subtly different syntax.

We'll start by looking at two closely related properties.

grid-template-rows, grid-template-columns

Values	none | <*track-list*> | <*auto-track-list*>_ | [subgrid <*line-name-list*>]?
Initial value	none
Applies to	Grid containers
Percentages	Refer to the inline size (usually width) of the grid container for `grid-template-columns`, and to the block size (usually height) of the grid container for `grid-template-rows`
Computed value	As declared, with lengths made absolute
Inherited	No
Animatable	No

With these properties, you can define the grid tracks of your overall *grid template*, or what the CSS specification calls the *explicit grid*. Everything depends on these grid tracks; fail to place them properly, and the whole layout can easily fall apart.

Once you define a grid track, grid lines are created. If you create just one track for the whole grid, two lines are created: one at the start of the track and one at the end. Two tracks means three lines: one at the start of the first track, one between the two, and one at the end of the second track. And so on.



When you're starting out with CSS grid layout, it's probably a good idea to sketch out where the grid tracks need to be on paper first, or in some close digital analogue. Having a visual reference for where the lines will land, and how the tracks should behave, can make writing your grid CSS a lot easier.

The exact syntax patterns for <*track-list*> and <*auto-track-list*> are complex and nest a few layers deep, and unpacking them would take a lot of time and space that's better

devoted to just exploring how things work. There are a lot of ways to make all this happen, so before we start discussing those patterns, we have some basic things to establish.

First, grid lines can always be referred to by number, but can also be explicitly named by the author. Take the grid shown in Figure 12-6, for example. From your CSS, you can use any of the numbers to refer to a grid line, or you can use the defined names, or you can mix them together. Thus, you could say that a grid item stretches from column line `3` to line `steve`, and from row line `skylight` to line `2`.

Note that a grid line can have more than one name. You can use any of them to refer to a given grid line, though you can't combine them the way you can multiple class names. You might think that means it's a good idea to avoid repeating grid-line names, but that's not always the case, as you'll soon see.

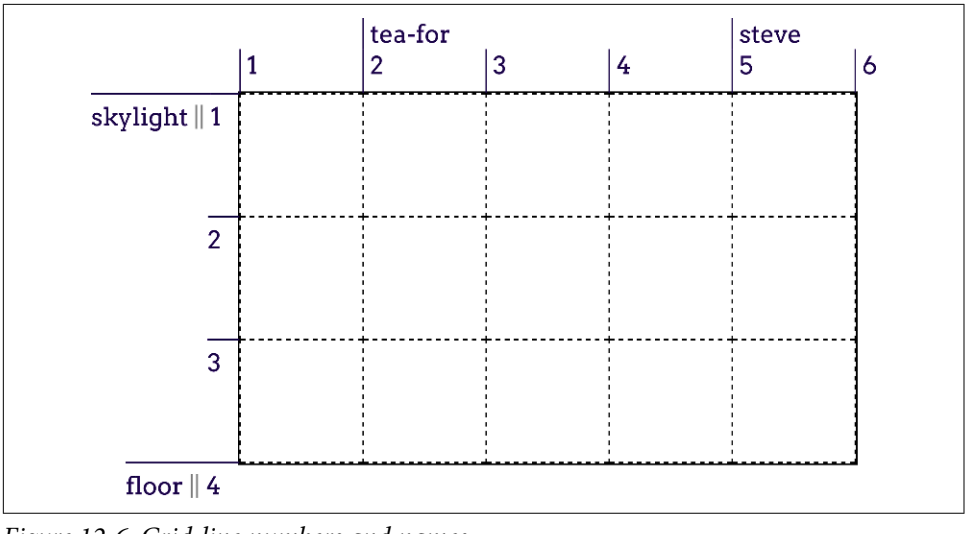


Figure 12-6. Grid-line numbers and names

We used intentionally silly grid-line names in Figure 12-6 to illustrate that you can pick any name you like, as well as to avoid the implication that there are "default" names. If you'd seen `start` for the first line, you might have assumed that the first line is always called that. Nope. If you want to stretch an element from `start` to `end`, you'll need to define those names yourself. Fortunately, that's simple to do.

As we've said, many value patterns can be used to define the grid template. We'll start with the simpler ones and work our way toward the more complex.



We'll discuss the `subgrid` value in "Using Subgrids" on page 602, after establishing how grid tracks and grid areas are defined, named, sized, combined, and so on.

Using Fixed-Width Grid Tracks

As our initial step, let's create a grid whose grid tracks are a fixed width. We don't necessarily mean a fixed length like pixels or ems; percentages also count as fixed width here. In this context, *fixed width* means the grid lines are placed such that the distance between them does not change because of content changes within the grid tracks.

So, as an example, this counts as a definition of three fixed-width grid columns:

```
#grid {display: grid;
    grid-template-columns: 200px 50% 100px;}
```

That will place a line 200 pixels from the start of the grid container (by default, the left side); a second grid line half the width of the grid container away from the first; and a third line 100 pixels away from the second. This is illustrated in Figure 12-7.

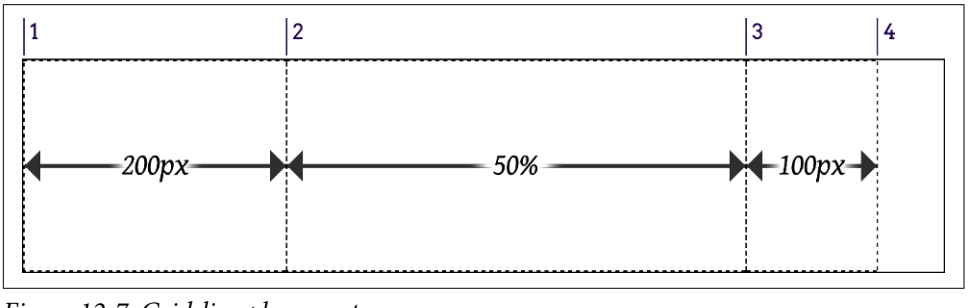


Figure 12-7. Grid-line placement

While it's true that the second column can change in size if the grid container's size changes, it will *not* change based on the content of the grid items. However wide or narrow the content placed in that second column, the column's width will always be half the width of the grid container.

It's also true that the last grid line doesn't reach the right edge of the grid container. That's fine; it doesn't have to. If you want it to—and you probably will—you'll see various ways to deal with that in just a bit.

This is all lovely, but what if you want to name your grid lines? Just place any grid-line name you want, and as many as you want, in the appropriate place in the value, surrounded by square brackets. That's all! Let's add some names to our previous example, with the result shown in Figure 12-8:

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    }
```

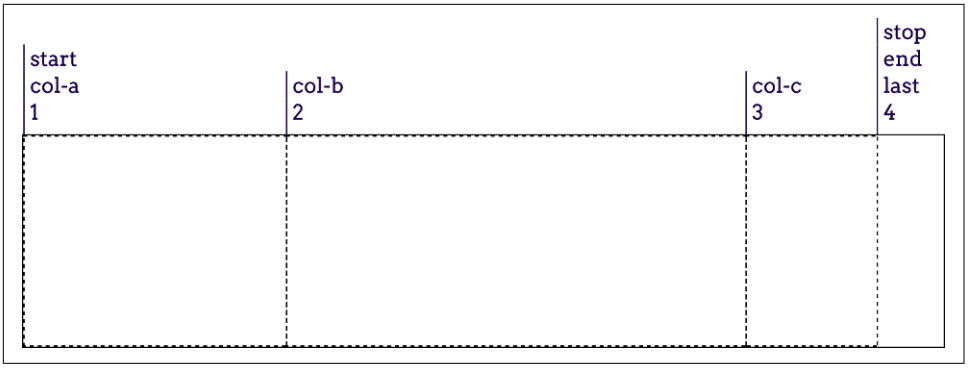


Figure 12-8. Grid-line naming

What's nice is that adding the names makes clear that each value is actually specifying a grid track's width, which means there is always a grid line to either side of a width value. Thus, for the three widths we have, four grid lines are actually created.

Row grid lines are placed in exactly the same way as columns, as Figure 12-9 shows:

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] 80% [footer] 2em [stop end];
    }
```

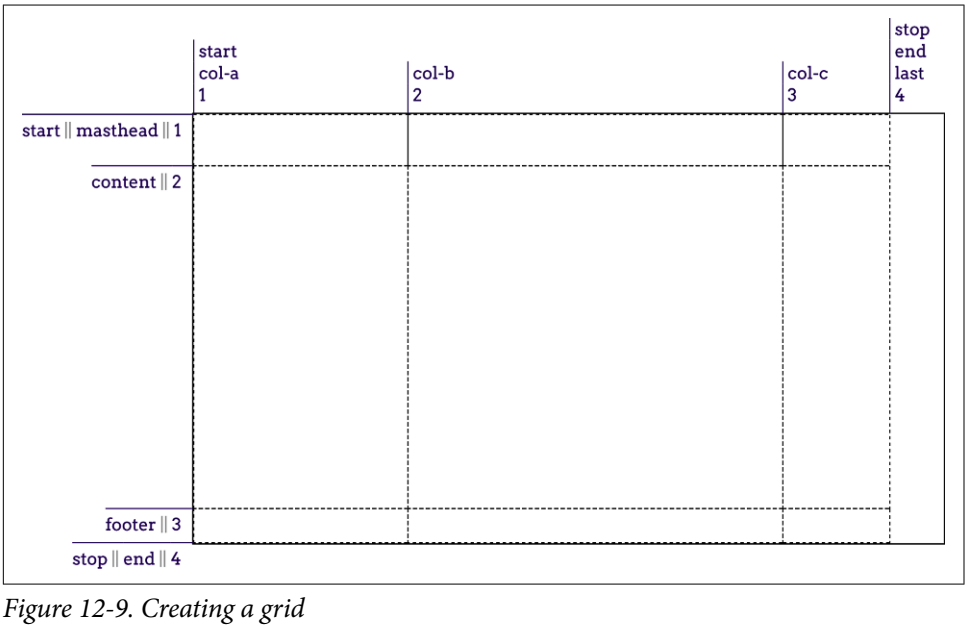


Figure 12-9. Creating a grid

We have a couple of things to point out here. First, both column and row lines have the names `start` and `end`. This is perfectly OK. Rows and columns don't share the same namespace, so you can reuse names like these in the two contexts.

Second is the percentage value for the `content` row track. This is calculated with respect to the height of the grid container; thus, a container 500 pixels tall would yield a `content` row that's 400 pixels tall (because the percentage value of this row is `80%`). Doing this generally requires that you know ahead of time how tall the grid container will be, which won't always be the case.

You might think we could just say `100%` and have it fill out the space, but that doesn't work, as Figure 12-10 illustrates: the `content` row track will be as tall as the grid container itself, thus pushing the `footer` row track out of the container altogether:

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] 100% [footer] 2em [stop end];
    }
```

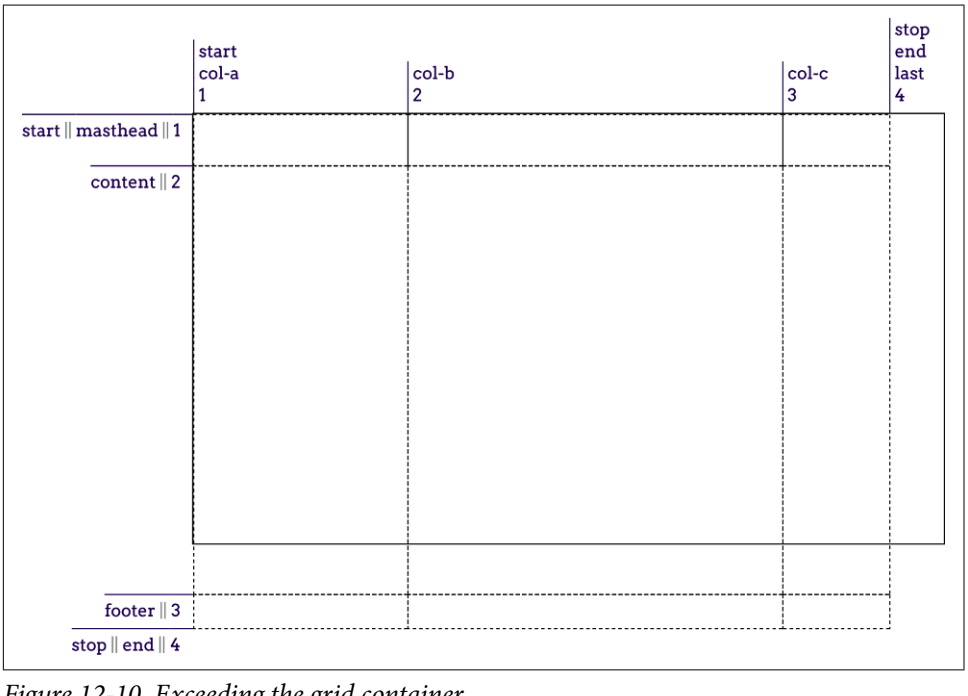


Figure 12-10. Exceeding the grid container

One way (not necessarily the best way) to handle this scenario is to *minmax* the row's value, telling the browser that you want the row no shorter than one amount and no taller than another, leaving the browser to fill in the exact value. This is done with the `minmax(a,b)` pattern, where `a` is the minimum size and `b` is the maximum size:

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] minmax(3em,100%) [footer] 2em [stop end];
    }
```

This code indicates that the `content` row should never be shorter than 3 ems tall, and never taller than the grid container itself. This allows the browser to bring up the size until it's tall enough to fit the space left over from the `masthead` and `footer` tracks, and no more. It also allows the browser to make it shorter than that, as long as it's not shorter than `3em`, so this is not a guaranteed result. Figure 12-11 shows one possible outcome of this approach.

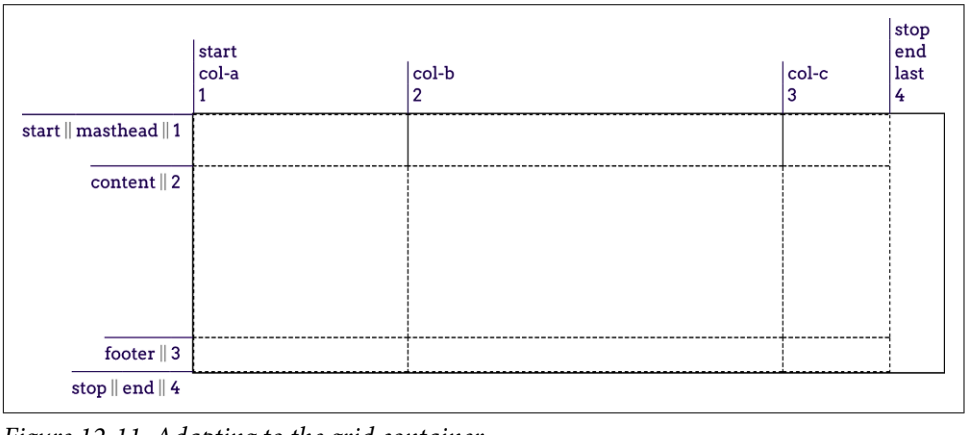


Figure 12-11. Adapting to the grid container

In like fashion, with the same caveats, `minmax()` could have been used to help the `col-b` column fill out the space across the grid container. The thing to remember with `minmax()` is that if the *max* is smaller than the *min*, the *max* value is thrown out and the *min* value is used as a fixed-width track length. Thus, `minmax(100px, 2em)` would resolve to `100px` for any font-size value smaller than `50px`.

If the vagueness of `minmax()`'s behavior unsettles you, CSS offers alternatives to this scenario. We could also have used the `calc()` value pattern to come up with a track's height (or width). For example:

```
    grid-template-rows:
        [start masthead] 3em [content] calc(100%-5em) [footer] 2em [stop end];
```

That would yield a `content` row exactly as tall as the grid container minus the sum of the `masthead` and `footer` heights, as we saw in the previous figure.

That works as far as it goes, but is a somewhat fragile solution, since any changes to the `masthead` or `footer` heights will also require an adjustment of the calculation. It also becomes a lot more difficult (or impossible) if you want more than one grid track to flex in this fashion. As it happens, CSS has much more robust ways to deal with this sort of situation, as you'll see next.

Using Flexible Grid Tracks

Thus far, all our grid tracks have been *inflexible*—their size determined by a length measure or the grid container's dimensions, but unaffected by any other considerations. *Flexible* grid tracks, by contrast, can be based on the amount of space in the grid container not consumed by inflexible tracks; or, alternatively, can be based on the actual content of the entire grid track.

Fractional units

If you want to divide up whatever space is available by a certain fraction and distribute the fractions to various columns, the `fr` unit is here for you. An `fr` is a flexible amount of space, representing a fraction of the *leftover* space in a grid.

In the simplest case, you can divide up the whole container by equal fractions. For example, if you want four columns, you could write this:

```
grid-template-columns: 1fr 1fr 1fr 1fr;
```

In this very specific and limited case, that's equivalent to saying the following:

```
grid-template-columns: 25% 25% 25% 25%;
```

Figure 12-12 shows the result of either.

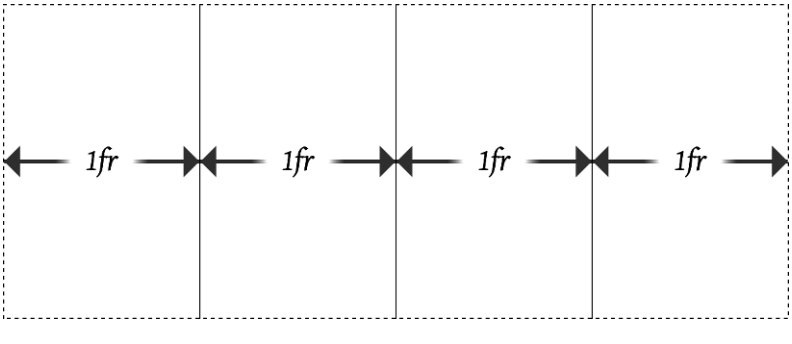


Figure 12-12. Dividing the container into four columns

This works because all of the grid container was "leftover space," so all of it is available to be divided up by the `fr` lengths. We'll get into how that plays out with nonflexible grid tracks in just a bit.

Going back to the previous example, suppose we want to add a fifth column and redistribute the column sizes so they're all still equal. If we used percentage values, we'd have to rewrite the entire value to be five instances of `20%`. With `fr`, though, we can just add another `1fr` to the value and have everything done for us automatically:

```
grid-template-columns: 1fr 1fr 1fr 1fr 1fr;
```

The way `fr` units work is that all of the `fr` values are added together, and all the leftover space in the grid is divided by that total. Then each track gets the number of those fractions indicated by its `fr` value.

In our first example, we had four `1fr` values, so their `1`'s were added together to get a total of 4. The available space was then divided by 4, and each column got one of those fourths. When we added a fifth `1fr`, the space was divided by 5, and each column got one of those fifths.

You are not required to always use `1` with your `fr` units! Suppose you want to divide up a space into three columns, with the middle column twice as wide as the other two. The code would look like this:

```
grid-template-columns: 1fr 2fr 1fr;
```

Again, these values are added up to get 4, and then we divide that 4 by 1 (representing the whole), so the base `fr` in this case is `0.25`. The first and third tracks are thus 25% the width of the container, whereas the middle column is half the container's width, because it's `2fr`, which is twice `0.25`, or `0.5`, or 50%.

You aren't limited to integers, either. A recipe card for apple pie could be laid out using these columns:

```
grid-template-columns: 1fr 3.14159fr 1fr;
```

We'll leave the math on that one as an exercise for you. (Lucky you! Just remember to start with `1 + 3.14159 + 1`, and you'll have a good head start.)

This is a convenient way to slice up a container, but there's more here than just replacing percentages with something more intuitive. Fractional units really come into their own when we have some fixed tracks and some flexible space. Consider, for example, the following, which is illustrated in Figure 12-13:

```
grid-template-columns: 15em 1fr 10%;
```

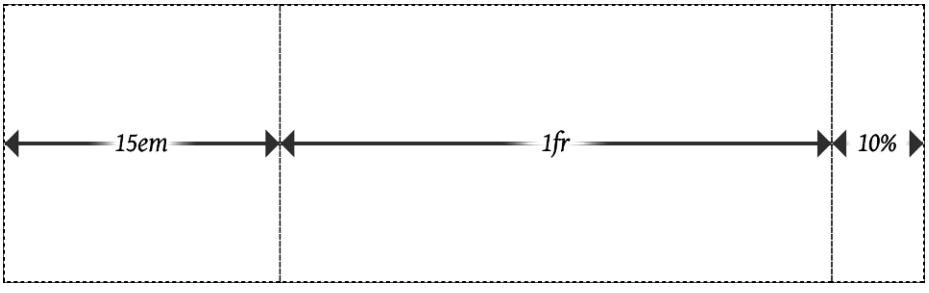


Figure 12-13. Giving the center column whatever's available

Here, the browser assigns the first and third tracks to their inflexible widths, and then gives whatever is left in the grid container to the center track. For a 1,000-pixel-wide grid container whose `font-size` is the usual browser default of `16px`, the first column will be 240 pixels wide and the third will be 100 pixels wide. That totals 340 pixels, leaving 660 pixels that aren't assigned to the fixed tracks. The fractional units total 1, so 660 is divided by 1, yielding 660 pixels, all of which are given to the single `1fr` track. If the grid container's width is increased to 1,400 pixels, the third column will be 140 pixels wide and the center column 1,020 pixels wide.

Just like that, we have a mixture of fixed and flexible columns. We can keep this going, splitting up any flexible space into as many fractions as we like. Consider this:

```
width: 100em; grid-template-columns: 15em 4.5fr 3fr 10%;
```

In this case, the columns will be sized as shown in Figure 12-14.

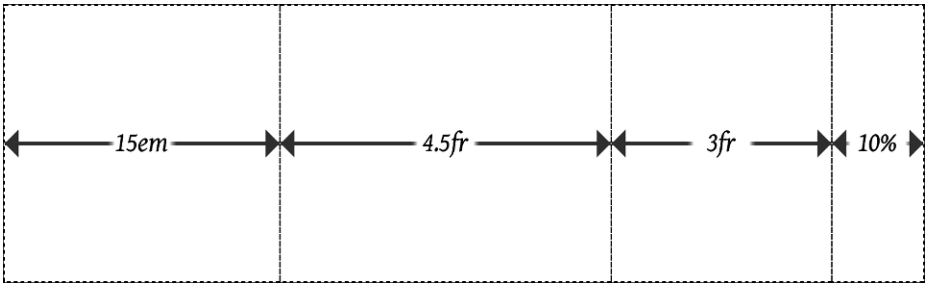


Figure 12-14. Flexible column sizing

The widths of the columns will be, from left to right: 15, 45, 30, and 10 ems. The first column gets its fixed width of `15em`. The last column is `10%` of 100 em, which is 10 em. That leaves 75 ems to distribute among the flexible columns. The two added together total 7.5 fr. For the wider column, 4.5 ÷ 7.5 equals 0.6, and that times 75 ems equals 45 ems. Similarly, 3 ÷ 7.5 = 0.4, and that times 75 ems equals 30 ems.

Yes, admittedly, we put a thumb on the scales for that example: the `fr` total and `width` value were engineered to yield nice, round numbers for the various columns. This was done purely to aid understanding. If you want to work through the process with less tidy numbers, consider using `92.5em` or `1234px` for the `width` value in the previous example.

If you want to define a minimum or maximum size for a given track, `minmax()` can be quite useful. To extend the previous example, suppose the third column should never be less than 5 ems wide, no matter what. The CSS would then be as follows:

```
grid-template-columns: 15em 4.5fr minmax(5em,3fr) 10%;
```

Now the layout will have two flexible columns at its middle, down to the point that the third column reaches `5em` wide. Below that point, the layout will have three inflexible columns (`15em`, `5em`, and `10%` wide, respectively) and a single flexible column that will get all the leftover space, if there is any. Once you run the math, it turns out that up to `30.5556em` wide, the grid will have one flexible column. Above that width, there will be two flexible columns.

You might think this works the other way—for example, if you wanted to make a column track flexible up to a certain point, and then become fixed after, you would declare a minimum `fr` value. This won't work, sadly, because `fr` units are not allowed in the *min* position of a `minmax()` expression. So any `fr` value provided as a minimum will invalidate the entire declaration.

Speaking of setting to 0, let's look at a minimum value explicitly set to `0`, like this:

```
grid-template-columns: 15em 1fr minmax(0,500px) 10%;
```

Figure 12-15 illustrates the narrowest grid width at which the third column can remain 500 pixels wide. Any narrower, and the `minmax`-ed column will be narrower than 500 pixels. Any wider, and the second column, the `fr` column, will grow beyond zero width while the third column stays at 500 pixels wide.

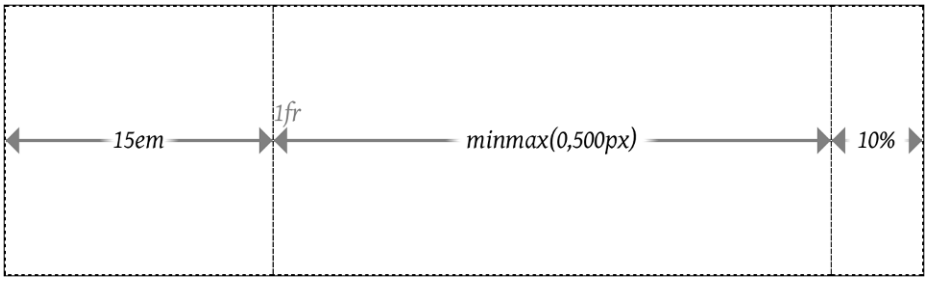


Figure 12-15. Minmaxed column sizing

If you look closely, you'll see the `1fr` label next to the boundary between the `15em` and `minmax(0,500px)` columns. That's there because the `1fr` is placed with its left edge on the second-column grid line, and has no width, because there is no space left to flex. Similarly, the `minmax` is placed on the third-column grid line. It's just that, in this specific situation, the second- and third-column grid lines are in the same place (which is why the `1fr` column has zero width).

If you ever run into a case where the minimum value is greater than the maximum value, the whole thing is replaced with the minimum value. Thus, `minmax(500px,200px)` would be treated as a simple `500px`. You probably wouldn't do this so obviously, but this feature is useful when mixing things like percentages and fractions. Thus, you could have a column that's `minmax(10%,1fr)` that would be flexible down to the point where the flexible column was less than 10% of the grid container's size, at which point it would stick at `10%`.

Fractional units and minmaxes are usable on rows just as easily as columns; it's just that rows are rarely sized in this way. You could easily imagine setting up a layout in which the masthead and footer are fixed tracks, while the content is flexible down to a certain point. That might look something like this:

```
grid-template-rows: 3em minmax(5em,1fr) 2em;
```

That works OK, but it's a lot more likely that you'll want to size that row by the height of its content, not some fraction of the grid container's height. The next section shows exactly how to make that happen.

Content-aware tracks

It's one thing to set up grid tracks that take up fractions of the space available to them, or that occupy fixed amounts of space. But what if you want to line up a bunch of pieces of a page and can't guarantee how wide or tall they might get? This is where `min-content` and `max-content` come in. (See Chapter 6 for a detailed explanation of these keywords.)

What's so powerful about using these sizing keywords in CSS Grid is that they apply to the entire grid track they define. For example, if you size a column to be `max-content`, the entire column track will be as wide as the widest content within it. This is easiest to illustrate with a grid of images (12, in this case) with the grid declared as follows and shown in Figure 12-16:

```
#gallery {display: grid;
    grid-template-columns: max-content max-content max-content max-content;
    grid-template-rows: max-content max-content max-content;}
```

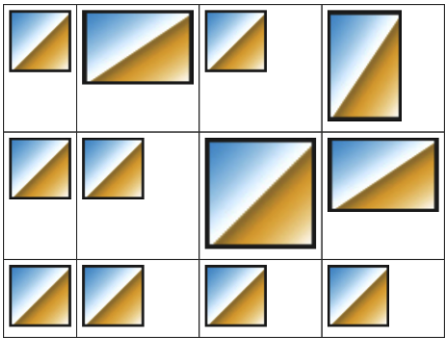


Figure 12-16. Sizing grid tracks by content

Looking at the columns, we can see that each column track is as wide as the widest image within that track. Where a bunch of portrait images happen to line up, the column is more narrow; where a landscape image shows up, the column is made wide enough to fit it. The same thing happens with the rows. Each row is as tall as the tallest image within it, so if a row happens to have all short images, the row is also short.

The advantage here is that this works for any sort of content, no matter what's in there. Say we add captions to the photos. All of the columns and rows will resize themselves as needed to handle both text and images, as shown in Figure 12-17.

This isn't a full-fledged design—the images are out of place, and there's no attempt to constrain the caption widths. In fact, that's exactly what we should expect from `max-content` values for the column widths. Since it means "make this column wide enough to hold all its content," that's what we get.

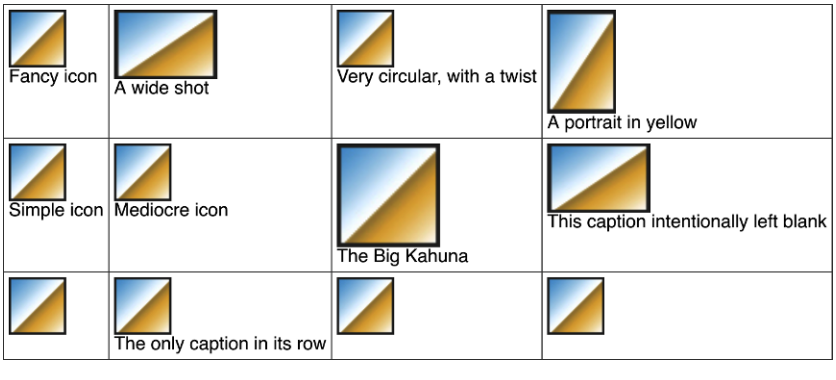


Figure 12-17. Sizing grid tracks around mixed content

What's important to realize is that this will hold even if the grid tracks have to spill out of the grid container. Even if we'd assigned something like `width: 250px` to the grid container, the images and captions would be laid out just the same. That's why things like

`max-content` tend to appear in `minmax()` statements. Consider the following, where grids with and without `minmax()` appear side by side. In both cases, the grid container is represented by a shaded background (see Figure 12-18):

```
#g1 {display: grid;
    grid-template-columns: max-content max-content max-content max-content;
    }
#g2 {display: grid;
    grid-template-columns: minmax(0,max-content) minmax(0,max-content)
          minmax(0,max-content) minmax(0,max-content);
    }
```

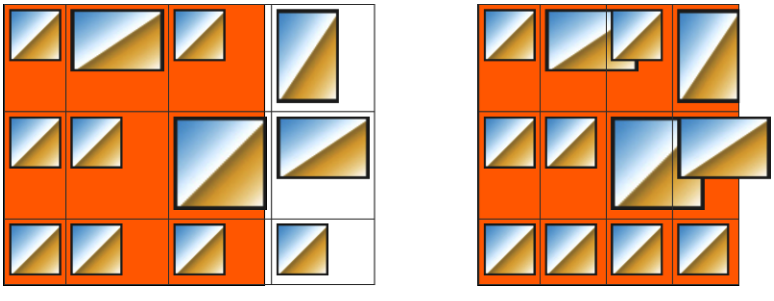


Figure 12-18. Sizing grid tracks with and without `minmax()`

In the first instance, the grid items completely contain their contents, but they spill out of the grid container. In the second, `minmax()` directs the browser to keep the columns within the range of `0` and `max-content`, so they'll all be fitted into the grid container if possible. A variant would be to declare `minmax(min-content, max-content)`, which can lead to a slightly different result than the `0, max-content` approach.

The reason that some images are overflowing their cells in the second example is that the tracks have been fitted into the grid container according to `minmax(0,max-content)`. They can't reach `max-content` in every track, but they can get as close as possible while all still fitting into the grid container. Where the contents are wider than the track, they just stick out of it, overlapping other tracks. This is standard grid behavior.

If you're wondering what happens if you `min-content` both the columns and the rows, it's pretty much the same as applying `min-content` to the columns and leaving the rows alone. This happens because the grid specification directs browsers to resolve column sizing first, and row sizing after that.

One more keyword you can use with grid track sizing is `auto`, which also happens to be the default value for any grid track's width. As a minimum, it's treated as the minimum size for the grid item, as defined by `min-width` or `min-height`. As a maximum, it's treated the same as `max-content`. You might think this means it can be used only in `minmax()` statements, but this is not the case. You can use it anywhere, and it will take on either a minimum or maximum role. Which one it takes on depends on the other track values around it, in ways that are frankly too complicated to get into here. As with so many other

aspects of CSS, using `auto` is essentially letting the browser do what it wants. Sometimes that's fine, but in general you'll probably want to avoid it.



There is a caveat to that last statement: `auto` values allow grid items to be resized by the `align-content` and `justify-content` properties, a topic we'll discuss in "Setting Alignment in Grids" on page 618. Since `auto` values are the only track-sizing values that permit this, there may be very good reasons to use `auto` after all.

Fitting Track Contents

In addition to the `min-content` and `max-content` keywords, a `fit-content()` function allows you to more compactly express certain types of sizing patterns. It's a bit complicated to decipher, but the effort is worth it:

The `fit-content()` function accepts a <*`length`*> or a <*`percentage`*> as its argument, like this:

```
#grid  {display: grid; grid-template-columns: 1fr fit-content(150px) 2fr;}
#grid2 {display: grid; grid-template-columns: 2fr fit-content(50%) 1fr;}
```

Before we explore what that means, let's ponder the pseudo-formula given by the specification:

fit-content(*argument*) => min(max-content, max(min-content, *argument*))

This means, essentially, "figure out which is greater, the `min-content` sizing or the supplied argument, and then take that result and choose whichever is smaller, that result or the `max-content` size." Which is probably confusing!.

We feel that a better way of phrasing it is "`fit-content(`*`argument`*`)` is equivalent to `minmax(min-content,max-content)`, except that the value given as an argument sets an upper limit, similar to `max-width` or `max-height`." Let's consider this example:

```
#example {display: grid; grid-template-columns: fit-content(50ch);}
```

The argument here is `50ch`, or the same width as 50 zero (`0`) characters side by side. So we're setting up a single column that's having its content fit to that measure.

For the initial case, assume the content is only 29 characters long, measuring 29 ch (because it's in a monospace font). That means the value of `max-content` is `29ch`, and the column will be only that wide, because it minimizes to that measure—`29ch` is smaller than whatever the maximum of `50ch` and `min-content` turns out to be.

Now, let's assume a bunch of text content is added so that there are 256 characters, thus measuring `256ch` in width (without any line wrapping). That means `max-content` evaluates to `256ch`. This is well beyond the `50ch` argument, so the column is constrained to be the larger of `min-content` and `50ch`, which is `50ch`.

As further illustration, consider the results of the following, as shown in Figure 12-19:

```
#thefollowing  {
    display: grid;
    grid-template-columns:
        fit-content(50ch) fit-content(50ch) fit-content(50ch);
    font-family: monospace;}
```

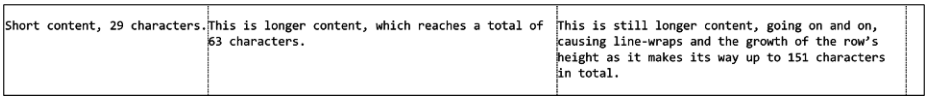


Figure 12-19. Sizing grid tracks with `fit-content()`

Notice the first column is narrower than the other two. Its `29ch` content minimizes to that size. The other two columns have more content than will fit into `50ch`, so they line-wrap, because their width has been limited to `50ch`.

Now let's consider what happens if an image is added to the second column. We'll make it `500px` wide, which happens to be wider than `50ch` in this instance. For that column, the maximum of `min-content` and `50ch` is determined. As we said, the larger value there is `min-content`, which is to say `500px` (the width of the image). Then the *minimum* of `500px` and `max-content` is determined. The text, rendered as a single line, would go on past `500px`, so the minimum is `500px`. Thus, the second column is now 500 pixels wide. This is depicted in Figure 12-20.

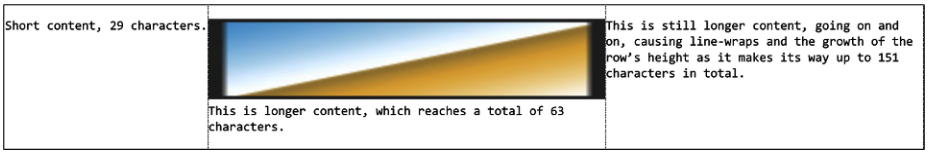


Figure 12-20. Fitting to wide content

If you compare Figures 12-19 to 12-20, you'll see that the text in the second column wraps at a different point, due to the change in column width. But also compare the text in the third column. It, too, has different line wraps.

That happens because after the first and second columns are sized, the third column has a bit less than `50ch` of space in which to be sized. The `fit-content(50ch)` function still does its thing, but here, it does so within the space available to it. Remember, the `50ch` argument is an upper bound, not a fixed size.

This is one of the great advantages of `fit-content()` over the less flexible `minmax()`. It allows you to shrink tracks to their minimum `content-size` when there isn't much content, while still setting an upper bound on the track size when there's a lot of content.

You may have been wondering about the repetitive grid template values in previous examples, and what happens if you need more than three or four grid tracks. Will you have to

write out every single track width individually? Indeed not, as you'll see in the next section.

Repeating Grid Tracks

If you want to set up a bunch of grid tracks of the same size, you probably don't want to have to type out every single one of them. Fortunately, `repeat()` is here to make sure you don't have to.

Let's say we want to set up a column grid line every 5 ems and have 10 column tracks. Here's how to do that:

```
#grid {display: grid;
    grid-template-columns: repeat(10, 5em);}
```

That's it. Done. Ten column tracks, each one `5em` wide, for a total of 50 ems of column tracks. It sure beats typing `5em` 10 times!

Any track-sizing value can be used in a repeat, from `min-content` and `max-content` to `fr` values to `auto`, and so on, and you can put together more than one sizing value. Suppose we want to define a column structure such that there's a `2em` track, then a `1fr` track, and then another `1fr` track—and, furthermore, we want to repeat that pattern three times. Here's how to do that, with the result shown in Figure 12-21:

```
#grid {display: grid;
    grid-template-columns: repeat(3, 2em 1fr 1fr);}
```

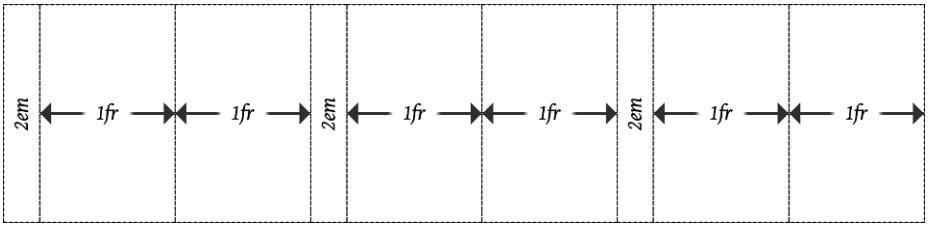


Figure 12-21. Repeating a track pattern

Notice that the last-column track is a `1fr` track, whereas the first-column track is `2em` wide. This is an effect of the way the `repeat()` was written. It's easy to add another `2em` track at the end, in order to balance things out, by adding a `2em` after the `repeat()` expression:

```
#grid {display: grid;
    grid-template-columns: repeat(3, 2em 1fr 1fr) 2em;}
```

This highlights the fact that `repeat` can be combined with any other track-sizing values—even other repeats—in the construction of a grid. The one thing you *can't* do is nest a repeat inside another repeat.

Other than that, just about anything goes within a `repeat()` value. Here's an example taken straight from the grid specification:

```
#grid {
    display: grid;
    grid-template-columns: repeat(4, 10px [col-start] 250px [col-end]) 10px;}
```

In this case, there are four repetitions of a 10-pixel track, a named grid line, a 250-pixel track, and then another named grid line. Then, after the four repetitions, a final 10-pixel column track. Yes, that means there will be four column grid lines named `col-start`, and another four named `col-end`, as shown in Figure 12-22. This is acceptable; grid-line names are not required to be unique.

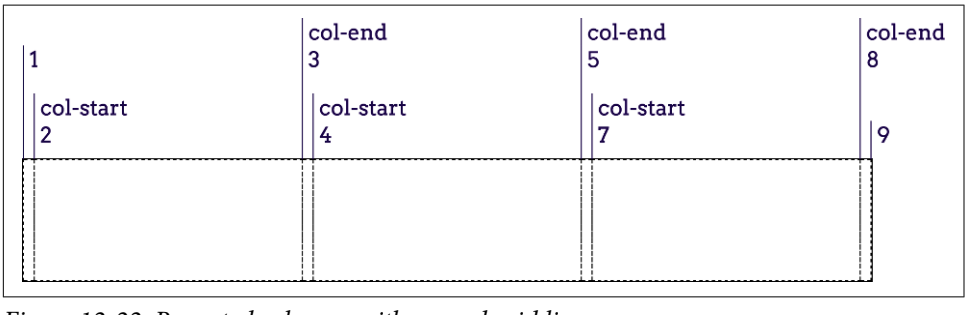


Figure 12-22. Repeated columns with named grid lines

One thing to remember, if you're going to repeat named lines, is that if you place two named lines next to each other, they'll be merged into a single, double-named grid line. In other words, the following two declarations are equivalent:

```
grid-template-rows: repeat(3, [top] 5em [bottom]);
grid-template-rows: [top] 5em [bottom top] 5em [top bottom] 5em [bottom];
```



If you're concerned about having the same name applied to multiple grid lines, don't be: there's nothing preventing it, and it can even be helpful in some cases. We'll explore ways to handle such situations in "Using Column and Row Lines" on page 577.

Autofilling tracks

CSS provides a way to set up a simple pattern and repeat it until the grid container is filled. This doesn't have quite the same complexity as regular `repeat()`—at least not yet—but it can still be pretty handy.

For example, suppose we want to have the previous row pattern repeat as many times as the grid container will comfortably accept:

```
grid-template-rows: repeat(auto-fill, [top] 5em [bottom]);
```

That will define a row line every 5 ems until there's no more room. Thus, for a grid container that's 11 ems tall, the following is equivalent:

```
grid-template-rows: [top] 5em [bottom top] 5em [bottom];
```

If the grid container's height is increased past 15 ems, but is less than 20 ems, then this is an equivalent declaration:

```
grid-template-rows: [top] 5em [bottom top] 5em [top bottom] 5em [bottom];
```

See Figure 12-23 for examples of the autofilled rows at three grid container heights.

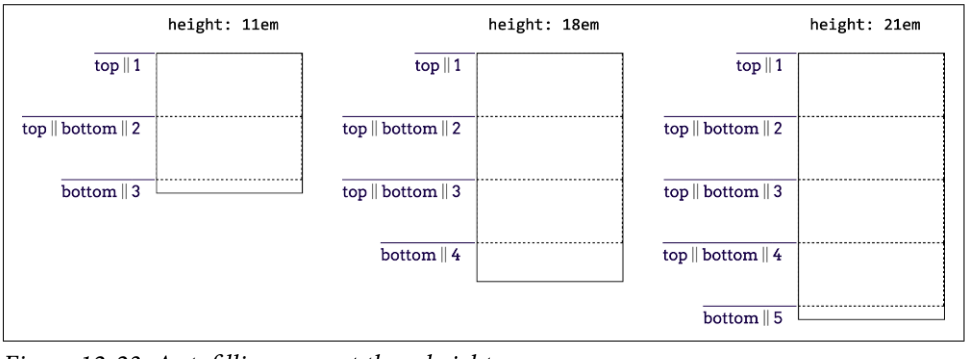


Figure 12-23. Autofilling rows at three heights

One limitation with auto-repeating is that it can take only an optional grid-line name, a fixed track size, and another optional grid-line name. So `[top] 5em [bottom]` represents about the maximum value pattern. You can drop the named lines and just repeat `5em`, or just drop one of the names.

It's not possible to auto-repeat multiple fixed track sizes, nor can you auto-repeat flexible track sizes. Similarly, you can't use intrinsic track sizes with auto-repeated tracks, so values such as `min-content` and `max-content` can't be put into an auto-repeated pattern.



You might wish you could auto-repeat multiple track sizes in order to define gutters around your content columns. This is usually unnecessary because of the properties `row-gap` and `column-gap` and their shorthand `gap`, which are covered in Chapter 11 but also apply in CSS Grid.

Furthermore, you can have only one auto-repeat in a given track template. Thus, the following would *not* be permissible:

```
grid-template-columns: repeat(auto-fill, 4em) repeat(auto-fill, 100px);
```

However, you *can* combine fixed-repeat tracks with autofill tracks. For example, you could start with three wide columns, and then fill the rest of the grid container with narrow tracks (assuming there's space for them). That would look something like this:

```
grid-template-columns: repeat(3, 20em) repeat(auto-fill, 2em);
```

You can flip that around too:

```
grid-template-columns: repeat(auto-fill, 2em) repeat(3, 20em);
```

That works because the grid layout algorithm assigns space to the fixed tracks first, and then fills up whatever space is left with auto-repeated tracks. The end result is to have one or more autofilled 2-em tracks, and then three 20-em tracks. Figure 12-24 shows two examples.

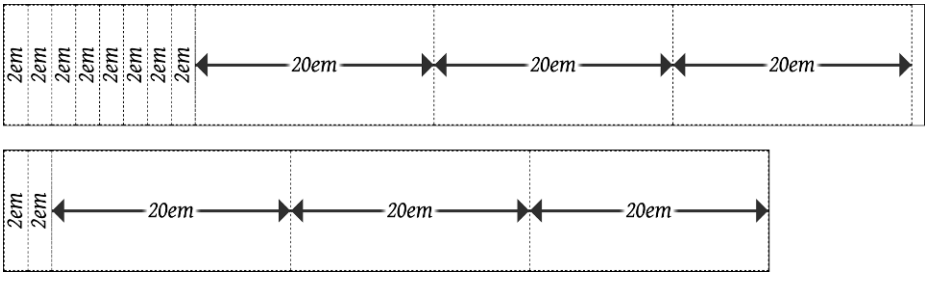


Figure 12-24. Autofilling columns next to fixed columns

With `auto-fill`, you will always get at least one repetition of the track template, even if it won't fit into the grid container for some reason. You'll also get as many tracks as will fit, even if some of the tracks don't have content. As an example, suppose you set up an auto-fill that places five columns, but only the first three actually end up with grid items in them. The other two would remain in place, holding open layout space.

If you use `auto-fit`, on the other hand, tracks that don't contain any grid items will be compressed to a width of zero, though they (and their associated grid lines) remain part of the grid. Otherwise, `auto-fit` acts the same as `auto-fill`. Suppose the following:

```
grid-template-columns: repeat(auto-fit, 20em);
```

If the grid container has room for five column tracks (i.e., it's more than 100 ems wide), but two tracks don't have any grid items to go into them, those empty grid tracks will be dropped, leaving the three column tracks that *do* contain grid items. The leftover space is handled in accordance with the values of `align-content` and `justify-content` (discussed in "Setting Alignment in Grids" on page 618). A simple comparison of `auto-fill` and `auto-fit` is shown in Figure 12-25, where the numbers in the colored boxes indicate the grid-column number to which they've been attached.

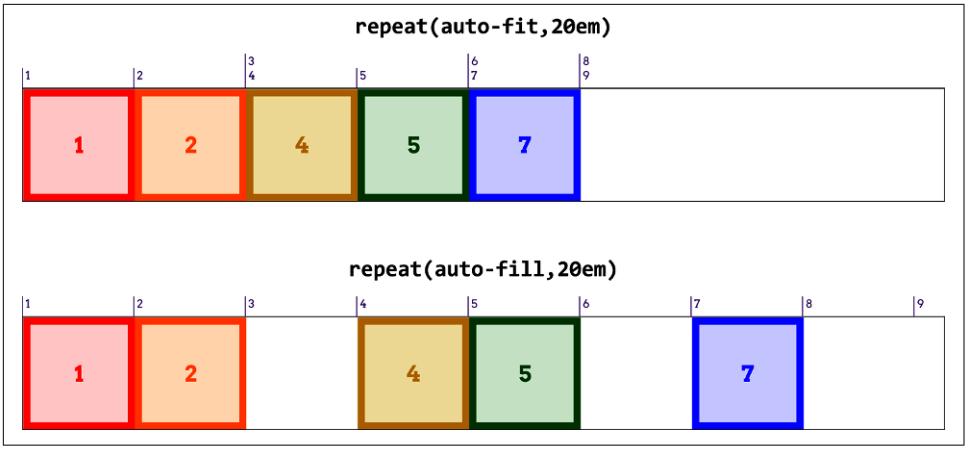


Figure 12-25. Using `auto-fill` versus `auto-fit`

Defining Grid Areas

Maybe you'd rather just "draw a picture" of your grid—both because it's fun to do and because the picture can serve as self-documenting code. It turns out you can more or less do exactly that with the `grid-template-areas` property.

grid-template-areas

Values	none | <*string*>
Initial value	none
Applies to	Grid containers
Computed value	As declared
Inherited	No
Animatable	No

We could go through a wordy description of how this works, but it's a lot more fun to just show it. The following rule has the result shown in Figure 12-26:

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l f f f";}
```

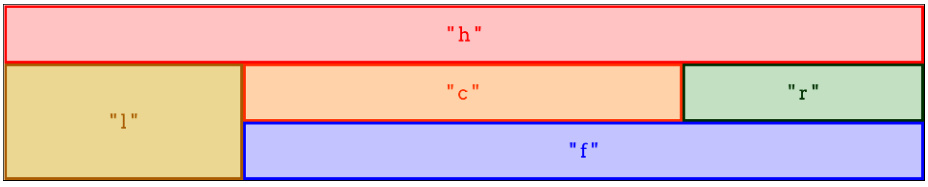


Figure 12-26. A simple set of grid areas

That's right: the letters in the string values are used to define how areas of the grid are shaped. Really! And you aren't even restricted to single letters! For example, we could expand the previous example like so:

```
#grid {display: grid;
    grid-template-areas:
        "header    header   header   header"
        "leftside  content  content  rightside"
        "leftside  footer   footer   footer";}
```

The grid layout is the same as that shown in Figure 12-26, though the name of each area would be different (e.g., `footer` instead of `f`).

In defining template areas, the whitespace is collapsed, so you can use it (as was done in the previous example) to visually line up columns of names in the value of `grid-template-areas`. You can line up the names with spaces or tabs, whichever will annoy your coworkers the most. Or you can just use a single space to separate each identifier, and not worry about the names lining up with one another. You don't even have to line-break between strings; the following works just as well as a pretty-printed version:

```
grid-template-areas: "h h h h" "l c c r" "l f f f";
```

What you can't do is merge those separate strings into a single string and have it mean the same thing. Every new string (as delimited by the quote marks) defines a new row in the grid. Thus the previous example, like the examples before it, defines three rows. Say we merge them all into a single string, like so:

```
grid-template-areas:
    "h h h h
     l c c r
     l f f f";
```

Then we'd have a single row of 12 columns, starting with the four-column area `h` and ending with the three-column area `f`. The line breaks aren't significant in any way, except as whitespace that separates one identifier from another.

If you look at these values closely, you may come to realize that each individual identifier represents a grid cell. Let's bring back our first example from this section, and consider the result shown in Figure 12-27, which uses Firefox's Grid Inspector to label each cell:

```
#grid {display: grid;
    grid-template-areas:
```

```
        "h h h h"
        "l c c r"
        "l f f f";}
```

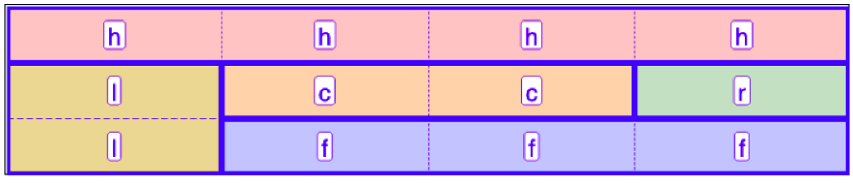


Figure 12-27. Grid cells with their grid area identifiers

This is exactly the same layout result as in Figure 12-26, but here, we've shown how each grid identifier in the `grid-template-areas` value corresponds to a grid cell. Once all the cells are identified, the browser merges any adjacent cells with the same name into a single area that encloses all of them—as long as they describe a rectangular shape! If you try to set up more complicated areas, the entire template is invalid. Thus, the following would result in no grid areas being defined:

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l l f f";}
```

See how `l` outlines an *L* shape? That humble change causes the entire `grid-template-areas` value to be dropped as invalid. A future version of grid layout may allow for non-rectangular shapes, but for now, this limitation exists.

If you want to define only some grid cells to be part of grid areas but leave others unlabeled, you can use one or more `.` characters to fill in for those unnamed cells. Let's say you just want to define some header, footer, and sidebar areas, and leave the rest unnamed. That would look something like this, with the result shown in Figure 12-28:

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";}
```

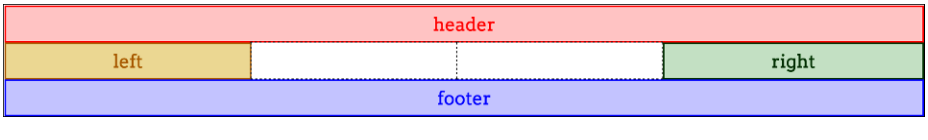


Figure 12-28. A grid with some unnamed grid cells

The two cells in the center of the grid are not part of a named area, having been represented in the template by *null cell tokens* (the `.` identifiers). Where each of those `...` sequences appears, we could have used one or more null tokens—so `left . . right` or `left ... ... right` would work just as well.

You can be as simple or creative with your cell names as you like. If you want to call your header `ronaldo` and your footer `podiatrist`, go for it. You can even use any Unicode character above codepoint U+0080, so `ConHugeCo©®™` and `åwësømë` are completely valid area identifiers…as are emoji! 😂 Now, to size the grid tracks created by these areas, we bring in our old friends `grid-template-columns` and `grid-template-rows`. Let's add both to the previous example, with the result shown in Figure 12-29:

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";
    grid-template-columns: 1fr 20em 20em 1fr;
    grid-template-rows: 40px 10em 3em;}
```

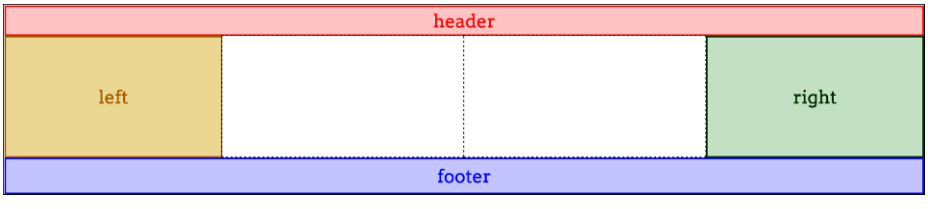


Figure 12-29. Named areas and sized tracks

Thus, the columns and rows created by naming the grid areas are given track sizes. If we give more track sizes than there are area tracks, that will add more tracks past the named areas. Therefore, the following CSS will lead to the result shown in Figure 12-30:

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";
    grid-template-columns: 1fr 20em 20em 1fr 1fr;
    grid-template-rows: 40px 10em 3em 20px;}
```

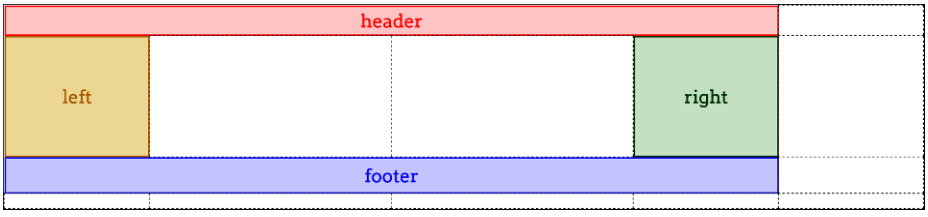


Figure 12-30. Adding more tracks beyond the named areas

So, given that we're naming areas, how about mixing in some named grid lines? As it happens, we already have: naming a grid area automatically adds names to the grid lines at its start and end. For the `header` area, there's an implicit `header-start` name on its first-column grid line *and* its first-row grid line, and `header-end` for its second-column and -row grid lines. For the `footer` area, the `footer-start` and `footer-end` names were automatically assigned to its grid lines.

Grid lines extend throughout the whole grid area, so a lot of these names are coincident. Figure 12-31 shows the naming of the lines created by the following template:

```
grid-template-areas:
    "header    header    header    header"
    "left      ...       ...       right"
    "footer    footer    footer    footer";
```

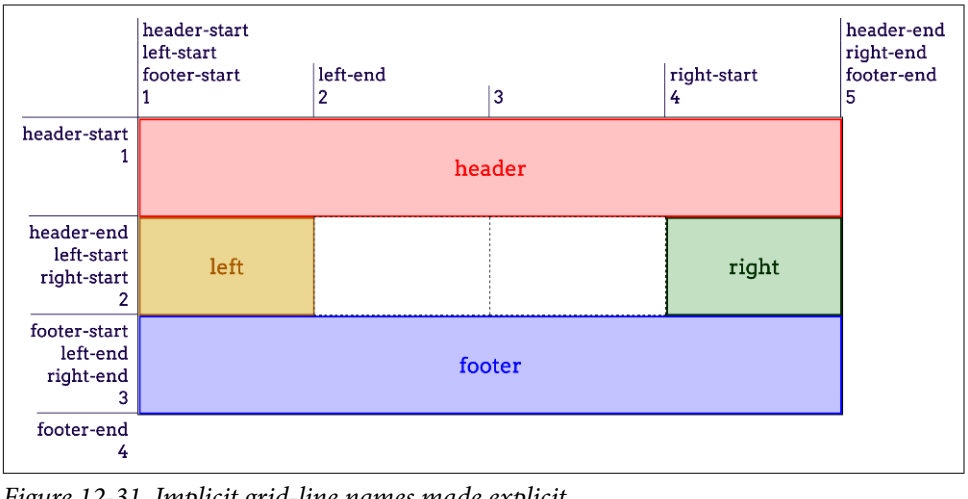


Figure 12-31. Implicit grid-line names made explicit

Now let's mix it up even more by adding a couple of explicit grid-line names to our CSS. Given the following rules, the first-column grid line in the grid would add the name `begin`, and the second-row grid line in the grid would add the name `content`:

```
#grid {display: grid;
    grid-template-areas:
```

```
    "header  header  header  header"
    "left    ...     ...     right"
    "footer  footer  footer  footer";
grid-template-columns: [begin] 1fr 20em 20em 1fr 1fr;
grid-template-rows: 40px [content] 1fr 3em 20px;}
```

Again: those grid-line names are *added* to the implicit grid-line names created by the named areas. Grid-line names never replace other grid-line names. Instead, they just keep piling up.

Even more interesting, this implicit-name mechanism runs in reverse. Suppose you don't use `grid-template-areas` at all, but instead set up some named grid lines like so, as illustrated in Figure 12-32:

```
grid-template-columns:
    [header-start footer-start] 1fr
    [content-start] 1fr [content-end] 1fr
    [header-end footer-end];
grid-template-rows:
   [header-start] 3em
   [header-end content-start] 1fr
   [content-end footer-start] 3em
   [footer-end];
```

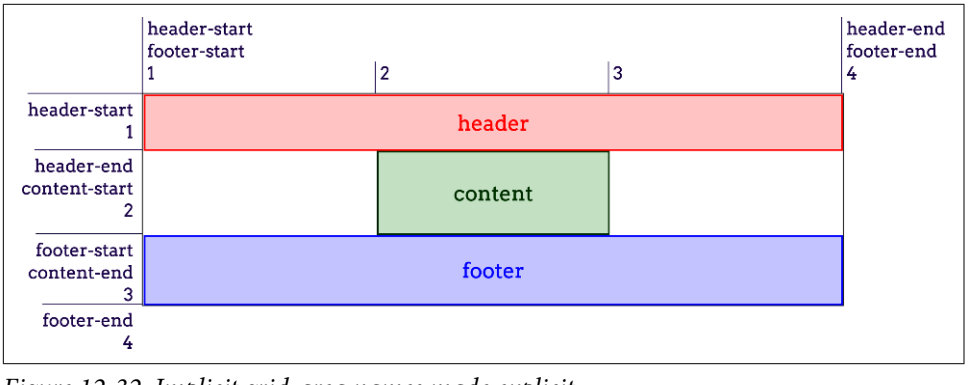


Figure 12-32. Implicit grid-area names made explicit

Because the grid lines use the form of `name-start`/`name-end`, the grid areas they define are implicitly named. To be frank, it's clumsier than doing it the other way, but the capability is there in case you ever want it.

Bear in mind that you don't need all four grid lines to be named in order to create a named grid area, though you probably do need them all to create a named grid area where you want it to be. Consider the following example:

```
grid-template-columns: 1fr [content-start] 1fr [content-end] 1fr;
grid-template-rows: 3em 1fr 3em;
```

This will still create a grid area named `content`. It's just that the named area will be placed into a new row after all the defined rows. What's odd is that an extra, empty row will

appear after the defined rows but before the row containing `content`. This has been confirmed to be the intended behavior. Thus, if you try to create a named area by naming the grid lines and miss one or more of them, your named area will effectively hang off to one side of the grid instead of being a part of the overall grid structure.

So, again, if you want to create named grid areas, you should probably stick to explicitly naming grid areas and let the `start-` and `end-` grid-line names be created implicitly, as opposed to the other way around.

Placing Elements in the Grid

Believe it or not, we've gotten this far without talking about how grid items are actually placed in a grid, once they've been defined.

Using Column and Row Lines

There are a couple of ways to go about placing grid items, depending on whether you want to refer to grid lines or grid areas. We'll start with four simple properties that attach an element to grid lines.

grid-row-start, grid-row-end, grid-column-start, grid-column-end

Value	*<grid-line>*
Initial value	`auto`
Applies to	Grid items and absolutely positioned elements, if their containing block is a grid container
Computed value	As declared
Inherited	No
Animatable	No

<grid-line>
`auto` | *<custom-ident>* | [*<integer>* && *<custom-ident>*?] | [`span` && [*<integer>* || *<custom-ident>*]]

These properties let you say, "I want the edge of the element to be attached to grid line such-and-so." As with so much of CSS Grid, it's a lot easier to show than to describe, so ponder the following styles and their result (see Figure 12-33):

```
.grid {display: grid; width: 50em;
    grid-template-rows: repeat(5, 5em);
    grid-template-columns: repeat(10, 5em);}
.one {
    grid-row-start: 2; grid-row-end: 4;
    grid-column-start: 2; grid-column-end: 4;}
.two {
```

```
    grid-row-start: 1; grid-row-end: 3;
    grid-column-start: 5; grid-column-end: 10;}
.three {
    grid-row-start: 4;
    grid-column-start: 6;}
```

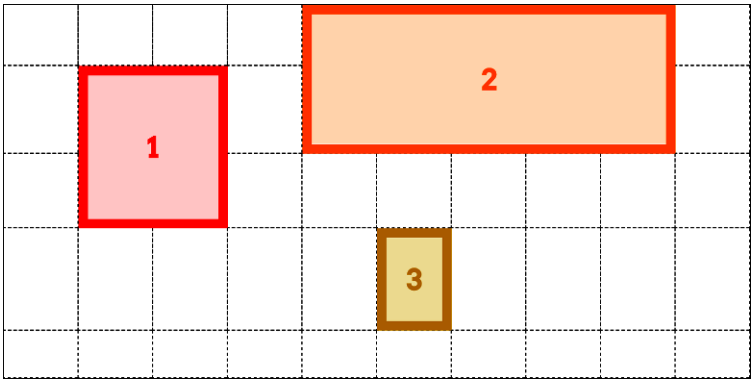


Figure 12-33. Attaching elements to grid lines

Here, we're using grid-line numbers to say where and how the elements should be placed within the grid. Column numbers count from left to right, and row numbers from top to bottom. If you omit ending grid lines, as was the case for `.three`, then the next grid lines in sequence are used for the end lines.

Thus, the rule for `.three` in the previous example is exactly equivalent to this:

```
.three {
    grid-row-start: 4; grid-row-end: 5;
    grid-column-start: 6; grid-column-end: 7;}
```

There's another way to say that same thing, as it happens: you could replace the ending values with `span 1`, or even just plain `span`, like this:

```
.three {
    grid-row-start: 4; grid-row-end: span 1;
    grid-column-start: 6; grid-column-end: span;}
```

If you supply `span` with a number, you're saying, "Span across this many grid tracks." So we can rewrite our earlier example like this and get exactly the same result:

```
#grid {display: grid;
    grid-template-rows: repeat(5, 5em);
    grid-template-columns: repeat(10, 5em);}
.one {
    grid-row-start: 2; grid-row-end: span 2;
    grid-column-start: 2; grid-column-end: span 2;}
.two {
    grid-row-start: 1; grid-row-end: span 2;
    grid-column-start: 5; grid-column-end: span 5;}
```

```
.three {
    grid-row-start: 4; grid-row-end: span 1;
    grid-column-start: 6; grid-column-end: span;}
```

If you leave out a number for `span`, it's set to be `1`. You can't use 0 or negative numbers for `span`; only positive integers.

An interesting feature of `span` is that you can use it for both ending *and* starting grid lines. The precise behavior of `span` is that it counts grid lines in the direction "away" from the grid line where it starts. In other words, if you define a start grid line and set the ending grid line to be a `span` value, it will search toward the end of the grid. Conversely, if you define an ending grid line and make the start line a `span` value, it will search toward the start of the grid.

That means the following rules will have the result shown in Figure 12-34 (the column and row numbers were added for clarity):

```
#grid {display: grid;
    grid-rows: repeat(4, 2em); grid-columns: repeat(5, 5em);}
.box1 {grid-row: 1; grid-column-start: 3;      grid-column-end: span 2;}
.box2 {grid-row: 2; grid-column-start: span 2; grid-column-end: 3;}
.box3 {grid-row: 3; grid-column-start: 1;      grid-column-end: span 5;}
.box4 {grid-row: 4; grid-column-start: span 1; grid-column-end: 5;}
```

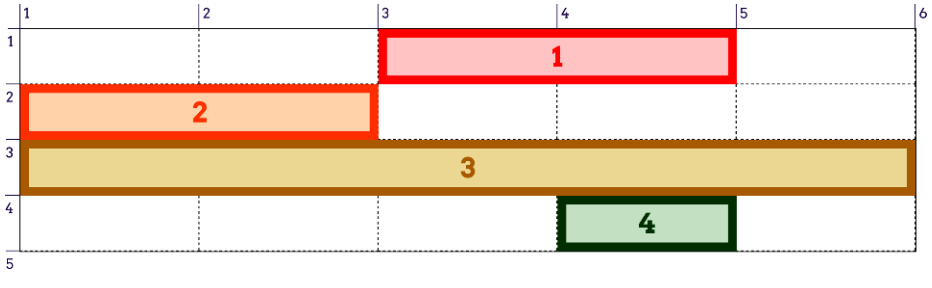


Figure 12-34. Spanning grid lines

In contrast to `span` numbering, you aren't restricted to positive integers for your actual grid-line values. Negative numbers will count backward from the end of explicitly defined grid lines. Thus, to place an element into the bottom-right grid cell of a defined grid, regardless of how many columns or rows it might have, you can just say this:

```
grid-column-start: -1;
grid-row-start: -1;
```

Note that this doesn't apply to any implicit grid tracks, a concept we'll get to in a bit, but only to the grid lines you explicitly define via one of the `grid-template-*` properties (e.g., `grid-template-rows`).

We aren't restricted to grid-line numbers, as it happens. If there are named grid lines, we can refer to those instead of (or in conjunction with) numbers. If you have multiple instances of a grid-line name, you can use numbers to identify which instance of the

grid-line name you're talking about. Thus, to start from the fourth instance of a row grid named `mast-slice`, you can say `mast-slice 4`. Take a look at the following, illustrated in Figure 12-35, for an idea of how this works:

```
#grid {display: grid;
    grid-template-rows: repeat(5, [R] 4em);
    grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
    grid-row-start: R 2;        grid-row-end: 5;
    grid-column-start: col-B;  grid-column-end: span 2;}
.two {
    grid-row-start: R;            grid-row-end: span R 2;
    grid-column-start: col-A 3;  grid-column-end: span 2 col-A;}
.three {
    grid-row-start: 9;
    grid-column-start: col-A -2;}
```

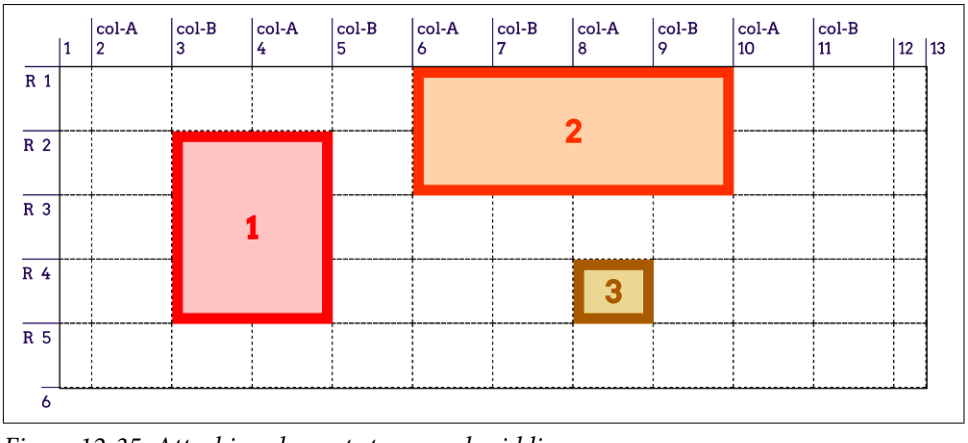


Figure 12-35. Attaching elements to named grid lines

Notice how `span` changes when we add a name: specifying `span 2 col-A` causes the grid item to span from its starting point (the third `col-A`) across another `col-A` and end at the `col-A` after that. This means the grid item actually spans four column tracks, since `col-A` appears on every other column grid line.

Again, negative numbers count backward from the end of a sequence, so `col-A -2` gets us the second-to-last instance of a grid line named `col-A`. Because no end-line values are declared for `.three`, they're both set to `span 1`. That means the following is exactly equivalent to the `.three` in the previous example:

```
.three {
    grid-row-start: 9; grid-row-end: span 1;
    grid-column-start: col-A -2; grid-row-end: span 1;}
```

There's an alternative way to use names with named grid lines—specifically, the named grid lines that are implicitly created by grid areas. For example, consider the following styles, illustrated in Figure 12-36:

```
grid-template-areas:
    "header     header    header    header"
    "leftside   content   content   rightside"
    "leftside   footer    footer    footer";
#masthead {grid-row-start: header;
        grid-column-start: header; grid-row-end: header;}
#sidebar {grid-row-start: 2; grid-row-end: 4;
        grid-column-start: leftside / span 1;}
#main {grid-row-start: content; grid-row-end: content;
        grid-column-start: content;}
#navbar {grid-row-start: rightside; grd-row-end: 3;
        grid-column-start: rightside;}
#footer {grid-row-start: 3; grid-row-end: span 1;
        grid-column-start: footer; grid-row-end: footer;}
```

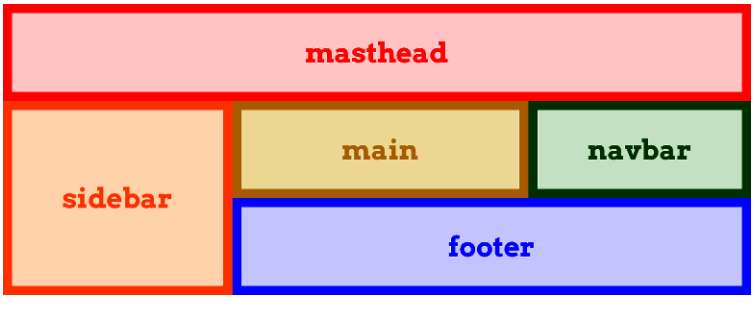


Figure 12-36. Another way of attaching elements to named grid lines

If you supply a custom identifier (i.e., a name you defined), the browser looks for a grid line with that name *plus* either `-start` or `-end` added on, depending on whether you're assigning a start line or an end line. Thus, the following are equivalent:

```
grid-column-start: header;        grid-column-end: header;
grid-column-start: header-start;  grid-column-end: header-end;
```

This works because, as we mentioned with `grid-template-areas`, explicitly creating a grid area implicitly creates the named `-start` and `-end` grid lines that surround it.

The final value possibility, `auto`, is kind of interesting. According to the Grid Layout specification, if one of the grid-line start/end properties is set to `auto`, that indicates "auto-placement, an automatic span, or a default span of one." In practice, this tends to mean that the grid line that gets picked is governed by the *grid flow*, a concept we have yet to cover (but will soon!). For a start line, `auto` usually means that the next available column or row line will be used. For an end line, `auto` usually means a one-cell span. In both cases, the word *usually* is used intentionally: as with any automatic mechanism, there are no absolutes.

Using Row and Column Shorthands

Two shorthand properties allow you to more compactly attach an element to grid lines.

grid-row, grid-column

Values	*<grid-line>* [/ *<grid-line>*]?
Initial value	`auto`
Applies to	Grid items and absolutely positioned elements, if their containing block is a grid container
Computed value	As declared
Inherited	No
Animatable	No

The primary benefit of these properties is that they make it a lot simpler to declare the start and end grid lines to be used for laying out a grid item. For example:

```
#grid {display: grid;
    grid-template-rows: repeat(10, [R] 1.5em);
    grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
    grid-row: R 3 / 7;
    grid-column: col-B / span 2;}
.two {
    grid-row: R / span R 2;
    grid-column: col-A 3 / span 2 col-A;}
.three {
    grid-row: 9;
    grid-column: col-A -2;}
```

That's a whole lot easier to read than having each start and end value in its own property, honestly. Other than being more compact, the behavior of these properties is more or less what you'd expect. If you have two bits separated by a forward slash (`/`), the first part defines the starting grid line, and the second part defines the ending grid line.

If you have only one value with no forward slash, it defines the starting grid line. The ending grid line depends on what you said for the starting line. If you supply a name for the starting grid line, the ending grid line is given that same name. If a single number is given, the second number (the end line) is set to `auto`. That means the following pairs are equivalent:

```
grid-row: 2;
grid-row: 2 / auto;

grid-column: header;
grid-column: header / header;
```

A subtle behavior built into the handling of grid-line names in `grid-row` and `grid-column` pertains to implicitly named grid lines. As you may recall, defining a named grid area creates `-start` and `-end` grid lines. That is, given a grid area with a name of `footer`, there are implicitly created `footer-start` grid lines to its top and left, and `footer-end` grid lines to its bottom and right.

In that case, if you refer to those grid lines by the area's name, the element will still be placed properly. Thus, the following styles have the result shown in Figure 12-37:

```
#grid {display: grid;
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer";
     grid-template-rows: auto 1fr auto;
     grid-template-columns: 25% 75%;}
#header {grid-row: header / header; grid-column: header;}
#footer {grid-row: footer; grid-column: footer-start / footer-end;}
```

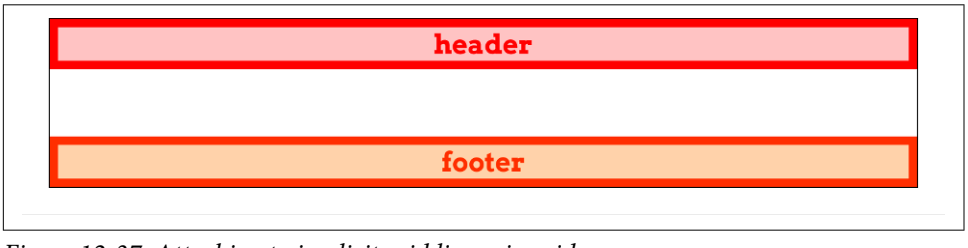


Figure 12-37. Attaching to implicit grid lines via grid-area names

You can always explicitly refer to the implicitly named grid lines, but if you just refer to the grid area's name, things still work out. If you refer to a grid-line name that doesn't correspond to a grid area, it falls back to the behavior discussed previously. In detail, it's the same as saying `line-name 1`, so the following two are equivalent:

```
grid-column: jane / doe;
grid-column: jane 1 / doe 1;
```

This is why it's risky to name grid lines the same as grid areas. Consider the following:

```
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer"
        "legal legal";
    grid-template-rows: auto 1fr [footer] auto [footer];
    grid-template-columns: 25% 75%;
```

This explicitly sets grid lines named `footer` above the "footer" row and below the "legal" row…and now there's trouble ahead. Suppose we add this:

```
#footer {grid-column: footer; grid-row: footer;}
```

For the column lines, there's no problem. The name `footer` gets expanded to `footer / footer`. The browser looks for a grid area with that name and finds it, so it translates `footer / footer` to `footer-start / footer-end`. The `#footer` element is attached to those implicit grid lines.

For `grid-row`, everything starts out the same. The `footer` name becomes `footer / footer`, which is translated to `footer-start / footer-end`. But that means the `#footer` will only be as tall as the "footer" row. It will *not* stretch to the second explicitly named `footer` grid line below the "legal" row, because the translation of `footer` to `footer-end` (due to the match between the grid-line name and the grid-area name) takes precedence.

The upshot of all this: it's generally a bad idea to use the same name for grid areas and grid lines. You might be able to get away with it in some scenarios, but you're almost always better off keeping your line and area names distinct, so as to avoid tripping over name-resolution conflicts.

Working with Implicit Grid

Up to this point, we've concerned ourselves solely with explicitly defined grids: we've talked about the row and column tracks we define via properties like `grid-template-columns`, and how to attach grid items to the cells in those tracks.

But what happens if we try to place a grid item, or even just part of a grid item, beyond that explicitly created grid? For example, consider the following grid:

```
#grid {display: grid;
    grid-template-rows: 2em 2em;
    grid-template-columns: repeat(6, 4em);}
```

Two rows, six columns. Simple enough. But suppose we define a grid item to sit in the first column and go from the first-row grid line to the fourth:

```
.box1 {grid-column: 1; grid-row: 1 / 4;}
```

Now what? We have only two rows bounded by three grid lines, and we've told the browser to go beyond that, from row line 1 to row line 4.

What happens is that another row line is created to handle the situation. This grid line, and the new row track it creates, are both part of the *implicit grid*. Here are a few examples of grid items that create implicit grid lines (and tracks) and how they're laid out (see Figure 12-38):

```
.box1 {grid-column: 1; grid-row: 1 / 4;}
.box2 {grid-column: 2; grid-row: 3 / span 2;}
.box3 {grid-column: 3; grid-row: span 2 / 3;}
.box4 {grid-column: 4; grid-row: span 2 / 5;}
.box5 {grid-column: 5; grid-row: span 4 / 5;}
.box6 {grid-column: 6; grid-row: -1 / span 3;}
.box7 {grid-column: 7; grid-row: span 3 / -1;}
```

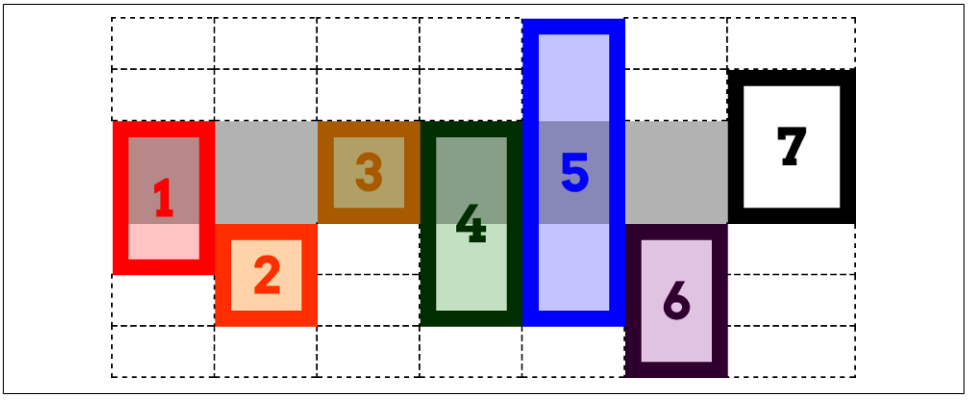


Figure 12-38. Creating implicit grid lines and tracks

A lot is going on there, so let's break it down. First off, the explicit grid is represented by the filled-in box behind the various numbered boxes; all the dashed lines represent the implicit grid.

What about those numbered boxes? The first, `box1`, adds an extra grid row line after the end of the explicit grid. The second, `box2`, starts on the last row-line of the explicit grid, and spans forward two row-lines, so it adds yet another implicit row-line. The third, `box3`, ends on the last explicit row-line (line 3) and spans *back* two lines, thus starting on the first explicit row-line.

Things really get interesting with `box4`. It ends on the fifth row-line, which is to say the second implicit row-line. It spans back three row-lines—and yet, it still starts on the same row-line as `box3`. This happens because grid track spans have to start counting *within* the explicit grid. Once they start, they can continue into the implicit grid (as happened with `box2`), but they *cannot* start counting within the implicit grid.

Thus, `box4` ends on row-line 5, but its span starts with row-line 3 and counts back two lines (`span 2`) to arrive at row-line 1. Similarly, `box5` ends on row-line 5 and spans back four lines, which means it starts on row-line –2. Remember: span counting must *start* in the explicit grid. It doesn't have to end there.

After those, `box6` starts on the last explicit row-line (line 3), and spans out to the sixth row-line—adding yet another implicit row-line. The point of having it here is to show that negative grid-line references are with respect to the explicit grid, and count back from its end. They do *not* refer to negatively indexed implicit lines that are placed before the start of the explicit grid.

If you want to start an element on an implicit grid line before the explicit grid's start, the way to do that is shown by `box7`: put its end line somewhere in the explicit grid, and span back past the beginning of the explicit grid. And you may have noticed: `box7` occupies an implicit column track. The original grid was set up to create six columns, which means seven column-lines, the seventh being the end of the explicit grid. When `box7` was given

`grid-column: 7`, that was equivalent to `grid-column: 7 / span 1` (since a missing end line is always assumed to be `span 1`). That necessitated the creation of an implicit column-line in order to hold the grid item in the implicit seventh column.

Now let's take those principles and add named grid lines to the mix. Consider the following, illustrated in Figure 12-39:

```
#grid {display: grid;
    grid-template-rows: [begin] 2em [middle] 2em [end];
    grid-template-columns: repeat(5, 5em);}
.box1 {grid-column: 1; grid-row: 2 / span end 2;}
.box2 {grid-column: 2; grid-row: 2 / span final;}
.box3 {grid-column: 3; grid-row: 1 / span 3 middle;}
.box4 {grid-column: 4; grid-row: span begin 2 / end;}
.box5 {grid-column: 5; grid-row: span 2 middle / begin;}
```

What you can see at work in several of these examples is what happens with grid-line names in the implicit grid: every implicitly created line has the name that's being hunted. Take `box2`, for example. It's given an end line of `final`, but there is no line with that name. Thus the span-search goes to the end of the explicit grid and, having not found the name it's looking for, creates a new grid line, to which it attaches the name `final`. (In Figure 12-39, the implicitly created line names are italicized and faded out a bit.)

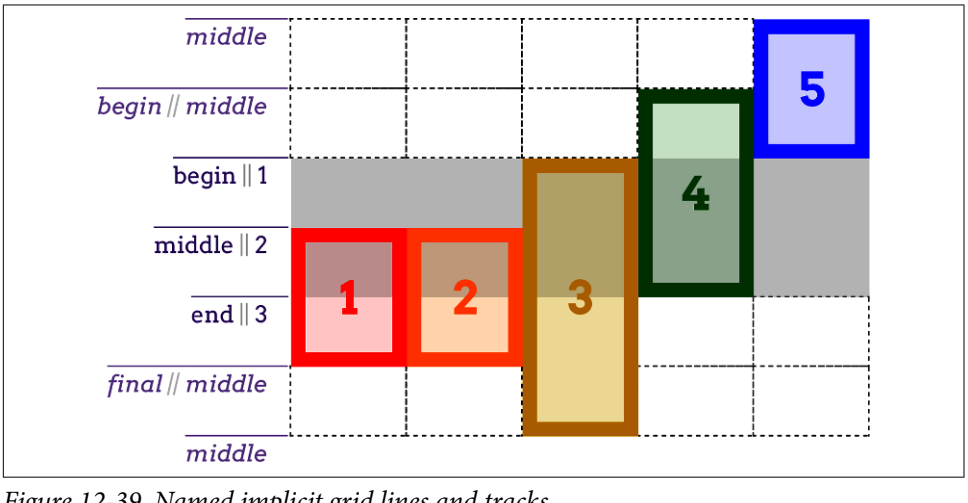


Figure 12-39. Named implicit grid lines and tracks

Similarly, `box3` starts on the first explicit row-line, and then needs to span three `middle` named lines. It searches forward and finds one, then goes looking for the other two. Not finding any, it attaches the name `middle` to the first implicit row-line, and then does the same for the second implicit row-line. Thus, it ends two implicit row-lines past the end of the explicit grid.

The same sort of thing happens with box4 and box5, except working backward from endpoints. You can see that box4 ends with the end row-line (line 3), then spans back to the second begin row-line it can find. This causes an implicit row-line to be created before the first row-line, named begin. Finally, box5 spans back from begin (the explicitly labeled begin) to the second middle it can find. Since it can't find any, it labels two implicit row-line middle and ends at the one farthest from where it started looking.

Handling Errors

We need to cover a few cases, as they fall under the general umbrella of "what grids do when things go pear-shaped." First, what if you accidentally put the start line after the end line? Say, something like this:

```
grid-row-start: 5;
grid-row-end: 2;
```

All that happens is probably what was meant in the first place: the values are swapped. Thus, you end up with the following:

```
grid-row-start: 2;
grid-row-end: 5;
```

Second, what if both the start and the end lines are declared to be spans of some variety? For example:

```
grid-column-start: span;
grid-column-end: span 3;
```

If this happens, the end value is dropped and replaced with auto. That means you'd end up with this:

```
grid-column-start: span;  /* 'span' is equal to 'span 1' */
grid-column-end: auto;
```

That would cause the grid item to have its ending edge placed automatically, according to the current grid flow (a subject we'll soon explore), and the starting edge to be placed one grid line earlier.

Third, what if the only thing directing placement of the grid item is a named span? In other words, you'd have this:

```
grid-row-start: span footer;
grid-row-end: auto;
```

This is not permitted, so the span footer in this case is replaced with span 1.

Using Areas

Attaching by row lines and column lines is great, but what if you could refer to a grid area with a single property? Behold: `grid-area`.

grid-area

Values	*<grid-line>* [/ *<grid-line>*]{0,3}
Initial value	See individual properties
Applies to	Grid items and absolutely positioned elements, if their containing block is a grid container
Computed value	As declared
Inherited	No
Animatable	No

Let's start with a simple use of `grid-area`: assigning an element to a previously defined grid area. For this, we'll bring back our old friend `grid-template-areas`, put it together with `grid-area` and some markup, and see what magic results (as shown in Figure 12-40):

```
#grid {display: grid;
    grid-template-rows: 200px 1fr 3em;
    grid-template-columns: 20em 1fr 1fr 10em;
    grid-template-areas:
        "header     header    header    header"
        "leftside   content   content   rightside"
        "leftside   footer    footer    footer";}
#masthead {grid-area: header;}
#sidebar {grid-area: leftside;}
#main {grid-area: content;}
#navbar {grid-area: rightside;}
#footer {grid-area: footer;}

<div id="grid">
    <div id="masthead">…</div>
    <div id="main">…</div>
    <div id="navbar">…</div>
    <div id="sidebar">…</div>
    <div id="footer">…</div>
</div>
```

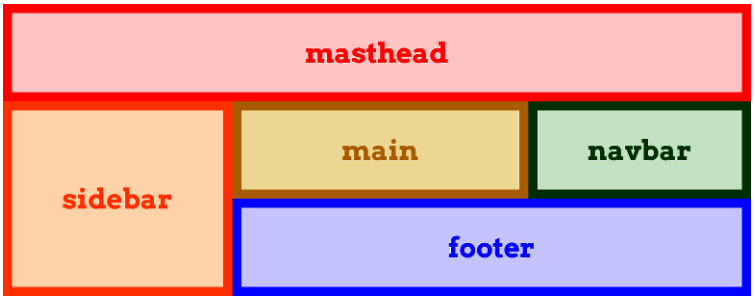


Figure 12-40. Assigning elements to grid areas

That's all it takes: set up some named grid areas to define your layout, and then drop grid items into them with `grid-area`. So simple and yet so powerful.

Another way to use `grid-area` refers to grid lines instead of grid areas. Fair warning: it's likely to be confusing at first.

Here's an example of a grid template that defines some grid lines, and some `grid-area` rules that reference the lines, as illustrated in Figure 12-41:

```
#grid {display: grid;
    grid-template-rows:
        [r1-start] 1fr [r1-end r2-start] 2fr [r2-end];
    grid-template-columns:
        [col-start] 1fr [col-end main-start] 1fr [main-end];}
.box01 {grid-area: r1 / main / r1 / main;}
.box02 {grid-area: r2-start / col-start / r2-end / main-end;}
.box03 {grid-area: 1 / 1 / 2 / 2;}
```

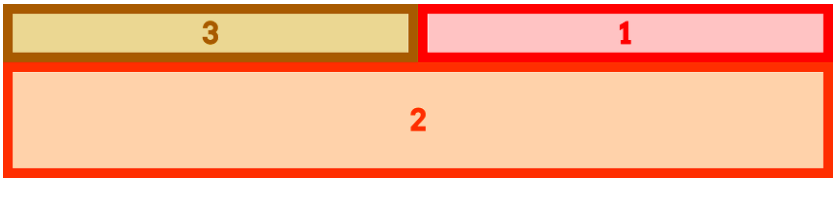


Figure 12-41. Assigning elements to grid lines

These elements were placed as directed. Note the ordering of the grid-line values, however. They're listed in the order `row-start`, `column-start`, `row-end`, `column-end`. If you diagram that in your head, you'll quickly realize that the values go counterclockwise (also called anticlockwise) around the grid item—the exact opposite of the TRBL pattern we're used to from margins, padding, borders, and so on. Furthermore, this means the column and row references are not grouped together but are instead split up.

If you supply fewer than four values, then the missing values are taken from those you do supply. If you use only three values, then the missing `grid-column-end` is the same as

`grid-column-start` if it's a name; if the start line is a number, the end line is set to `auto`. The same holds true if you give only two values, except that the now-missing `grid-row-end` is copied from `grid-row-start` if it's a name; otherwise, it's set to `auto`.

From that, you can probably guess what happens if only one value is supplied: if it's a name, use it for all four values; if it's a number, the rest are set to `auto`.

This one-to-four replication pattern is actually how giving a single grid-area name translates into having the grid item fill that area. The following are equivalent:

```
grid-area: footer;
grid-area: footer / footer / footer / footer;
```

Now recall the behavior discussed in the previous section about `grid-column` and `grid-row`: if a grid line's name matches the name of a grid area, it's translated into a `-start` or `-end` variant, as appropriate. That means the previous example is translated to the following:

```
grid-area: footer-start / footer-start / footer-end / footer-end;
```

And that's how a single grid-area name causes an element to be placed into the corresponding grid area.

Understanding Grid-Item Overlap

One thing we've been very careful to do in our grid layouts thus far is to avoid overlap. Rather like positioning, it's absolutely (get it?) possible to make grid items overlap each other. Let's take a simple case, illustrated in Figure 12-42:

```
#grid {display: grid;
    grid-template-rows: 50% 50%;
    grid-template-columns: 50% 50%;}
.box01 {grid-area: 1 / 1 / 2 / 3;}
.box02 {grid-area: 1 / 2 / 3 / 2;}
```

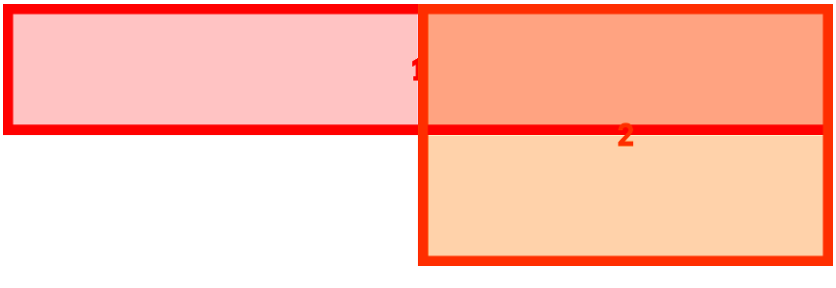


Figure 12-42. Overlapping grid items

Thanks to the grid numbers that were supplied in the last two lines of the CSS, the two grid items overlap in the upper-right grid cell. Which is on top of the other depends on the layering behavior we'll discuss later, but for now, just take it as a given that they do layer when overlapping.

There may well be times when you want grid items to overlap. A photo's caption might partially overlap the photo, for example. Or you might want to assign a few items to the same grid area so they combine, or set them to be shown one at a time by script or user interaction.

Overlap isn't restricted to situations involving raw grid numbers. In the following case, the sidebar and the footer will overlap, as shown in Figure 12-43. (Assuming the footer comes later than the sidebar in the markup, then in the absence of other styles, the footer will be on top of the sidebar.)

```
#grid {display: grid;
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer";}
#header {grid-area: header;}
#sidebar {grid-area: sidebar / sidebar / footer-end / sidebar;}
#footer {grid-area: footer;}
```

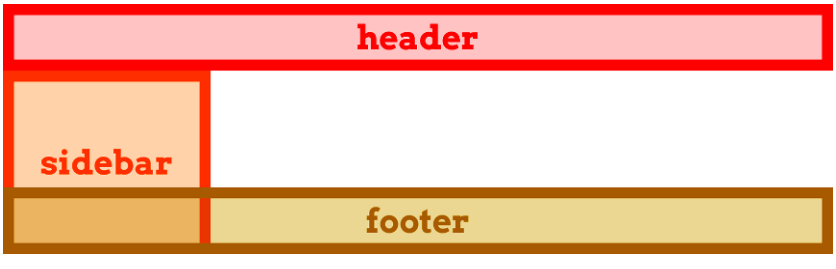


Figure 12-43. Overlapping sidebar and footer

We bring this up in part to warn you about the possibility of overlap, and also to serve as a transition to the next topic. It's a feature that sets grid layout apart from positioning, in that it can sometimes help avoid overlap: the concept of *grid flow*.

Specifying Grid Flow

For the most part, we've been explicitly placing grid items on the grid. If items aren't explicitly placed, they're automatically placed into the grid. Following the grid-flow direction that's in effect, an item is placed in the first area that will fit it. The simplest case is just filling a grid track in sequence, one grid item after another, but things can get a lot more complex than that, especially if there is a mixture of explicitly and automatically placed grid items. The latter must work around the former.

CSS has primarily two grid-flow models, *row-first* and *column-first*, though you can enhance either by specifying a *dense* flow. All this is done with the property called `grid-auto-flow`.

grid-auto-flow

Values	[row | column] || dense
Initial value	row
Applies to	Grid containers
Computed value	As declared
Inherited	No
Animatable	No

To see how these values work, consider the following markup:

```
<ol id="grid">
<li>1</li>
<li>2</li>
<li>3</li>
<li>4</li>
<li>5</li>
</ol>
```

To that markup, let's apply the following styles:

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: row;}
#grid li {grid-row: auto; grid-column: auto;}
```

Assuming a grid with a column line every 15 ems and a row line every 4 ems, we get the result shown in Figure 12-44.

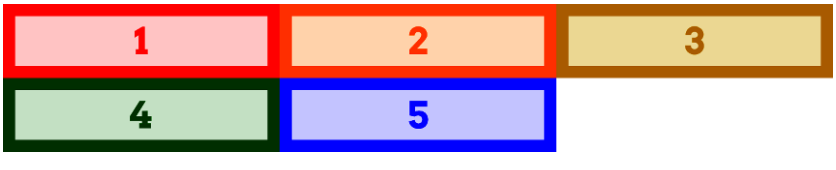


Figure 12-44. Row-oriented grid flow

This probably seems pretty normal, the same sort of thing you'd get if you floated all the boxes, or if all of them were inline blocks. That familiarity is why `row` is the default value. Now, let's try switching the `grid-auto-flow` value to `column`, as shown in Figure 12-45:

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;}
```

So with `grid-auto-flow: row`, each row is filled in before starting on the next row. With `grid-auto-flow: column`, each column is filled first.



Figure 12-45. Column-oriented grid flow

What needs to be stressed here is that the list items weren't explicitly sized. By default, they were resized to attach to the defined grid lines. This can be overridden by assigning explicit sizing to the elements. For example, if we make the list items 7 ems wide and 1.5 ems tall, we'll get the result shown in Figure 12-46:

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;
    width: 7em; height: 1.5em;}
```



Figure 12-46. Explicitly sized grid items

If you compare that to the previous figure, you'll see that the corresponding grid items start in the same place; they just don't end in the same places. This illustrates that what's really placed in grid flow is grid areas, to which the grid items are then attached.

This is important to keep in mind if you auto-flow elements that are wider than their assigned column or taller than their assigned row, as can very easily happen when turning images or other intrinsically sized elements into grid items. Let's say we want to put a bunch of images, each a different size, into a grid that's set up to have a column line every 50 horizontal pixels, and a row line every 50 vertical pixels. This grid is illustrated in Figure 12-47, along with the results of flowing a series of images into that grid by either row or column:

```
#grid {display: grid;
    grid-template-rows: repeat(3, 50px);
    grid-template-columns: repeat(4, 50px);
    grid-auto-rows: 50px;
    grid-auto-columns: 50px;
}
img {grid-row: auto; grid-column: auto;}
```

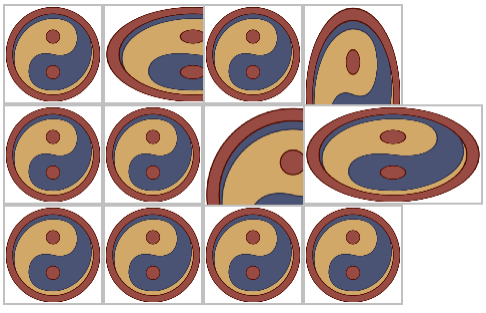


Figure 12-47. Flowing images in grids

Notice that some of the images overlap others? That's because each image is attached to the next grid line in the flow, without taking into account the presence of other grid items. We didn't set up images to span more than one grid track when they needed it, so overlap occurred.

This can be managed with class names or other identifiers. We could class images as `tall` or `wide` (or both) and specify that they get more grid tracks. Here's some CSS to add to the previous example, with the result shown in Figure 12-48:

```
img.wide {grid-column: auto / span 2;}
img.tall {grid-row: auto / span 2;}
```

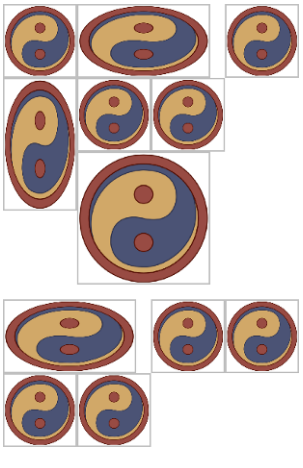


Figure 12-48. Giving images more track space

This does cause the images to keep spilling down the page, but no overlapping occurs.

However, notice the gaps in this grid? That happens because the placement of some grid items across grid lines doesn't leave enough room for other items in the flow. To illustrate

this, and the two flow patterns, more clearly, let's try an example with numbered boxes (Figure 12-49).

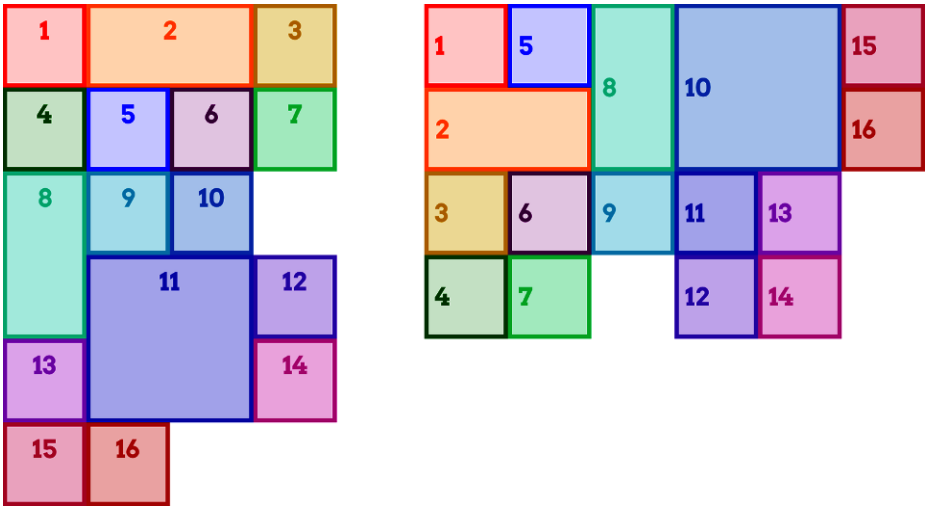


Figure 12-49. Illustrating flow patterns

Follow across the rows of the first grid, counting along with the numbers. In this particular flow, the grid items are laid out almost as if they were leftward floats. Almost, but not quite: notice that grid item 13 is actually to the left of grid item 11. That would never happen with floats, but it can with grid flow. The way row flow (if we may call it that) works is that you go across each row from left to right, and if there's room for a grid item, you put it there. If a grid cell has been occupied by another grid item, you skip over it. So the cell next to item 10 didn't get filled, because there wasn't room for item 11. Item 13 went to the left of item 11 because there was room for it there when the row was reached.

As shown by the second example in Figure 12-49, the same basic mechanisms hold true for column flow, except in this case you work from top to bottom. Thus, the cell below item 9 is empty because item 10 wouldn't fit there. Instead, item 10 went into the next column and covered four grid cells (two in each direction). The items after it, since they were just one grid cell in size, filled in the cells after it in column order.



Grid flow works left to right, top to bottom in languages that have that writing pattern. In RTL languages, such as Arabic and Hebrew, the row-oriented flow would be right to left, not left to right.

If you were just now wishing for a way to pack grid items as densely as possible, regardless of how that affected the ordering, good news: you can! Just add the keyword `dense` to your `grid-auto-flow` value, and that's exactly what will happen. We can see the result in

Figure 12-50, which shows the results of `grid-auto-flow: row dense` and `grid-auto-flow: dense column` side by side.

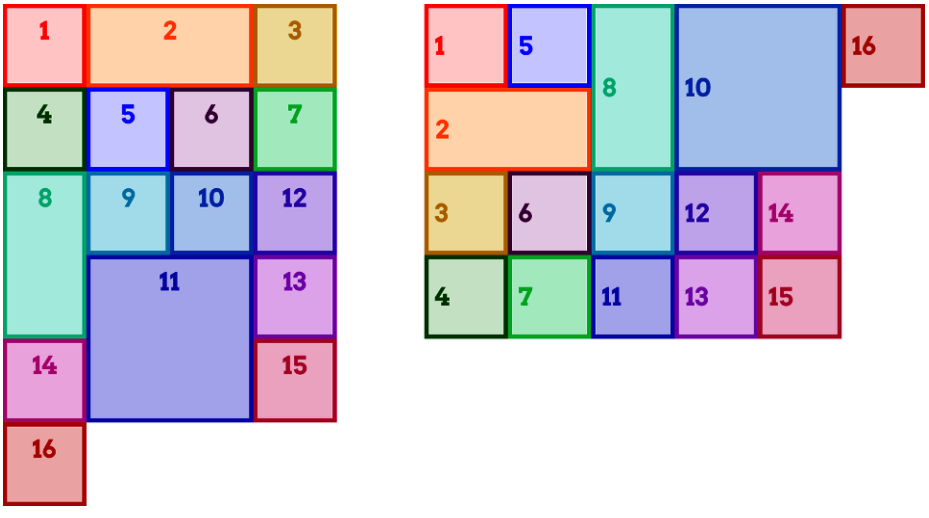


Figure 12-50. Illustrating dense flow patterns

In the first grid, item 12 appears in the row above item 11 because there was a cell that fit it. For the same reason, item 11 appears to the left of item 10 in the second grid.

In effect, what happens with `dense` grid flow is that for each grid item, the browser scans through the *entire* grid in the given flow direction (`row` or `column`), starting from the flow's starting point (the top-left corner in LTR languages), until it finds a place where that grid item will fit. This can make things like photo galleries more compact, and works great as long as you don't have a specific order in which the grid items need to appear.

Now that we've explored grid flow, we have a confession to make: to make the last couple of grid items look right, we included some CSS that we didn't show you. Without it, the items hanging off the edge of the grid would have looked quite a bit different from the other items—much shorter in row-oriented flow, and much narrower in column-oriented flow. You'll see why, and the CSS we used, in the next section.

Defining Automatic Grid Tracks

So far, we've almost entirely seen grid items placed into a grid that was explicitly defined. But in the preceding section we had grid items running off the edge of the explicitly defined grid. What happens when a grid item goes off the edge? Rows or columns are added as needed to satisfy the layout directives of the items in question (see "Working with Implicit Grid" on page 584). So, if an item with a row span of `3` is added after the end of a row-oriented grid, three new rows are added after the explicit grid.

By default, these automatically added grid tracks are the absolute minimum size needed. If you want to exert a little more control over their sizing, `grid-auto-rows` and `grid-auto-columns` are for you.

grid-auto-rows, grid-auto-columns

Values	*<track-breadth>*+ | `minmax(` *<track-breadth>* `,` *<track-breadth>* `)`
Initial value	`auto`
Applies to	Grid containers
Computed value	Depends on the specific track sizing
Note	*<track-breadth>* is a stand-in for *<length>* | *<percentage>* | *<flex>* | `min-content` | `max-content` | `auto`
Inherited	No
Animatable	No

For any automatically created row or column tracks, you can provide a single track size or a minmaxed pair of track sizes. Let's take a look at a reduced version of the grid-flow example from the previous section: we'll set up a 2 × 2 grid and try to put five items into it. In fact, let's do it twice: once with `grid-auto-rows` and once without, as illustrated in Figure 12-51:

```
.grid {display: grid;
    grid-template-rows: 80px 80px;
    grid-template-columns: 80px 80px;}
#g1 {grid-auto-rows: 80px;}
```

As the second grid shows, without assigning a size to the automatically created row, the overflowing grid items are placed in a row that's exactly as tall as the grid items' content, and not a pixel more. Each is still just as wide as the column into which it's placed, because the columns have a size (`80px`). The row, lacking an explicit height, defaults to `auto`, with the result shown.

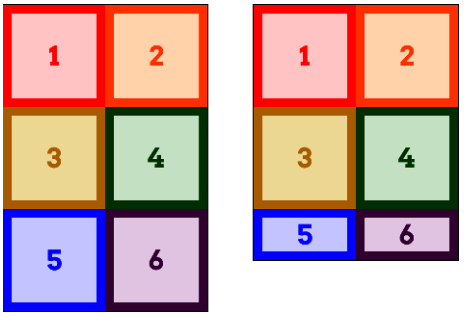


Figure 12-51. Grids with and without auto-row sizing

If we flip things to a column-oriented flow, the same basic principles apply (see Figure 12-52):

```
.grid {display: grid; grid-auto-flow: column;
    grid-template-rows: 80px 80px;
    grid-template-columns: 80px 80px;}
#g1 {grid-auto-columns: 80px;}
```

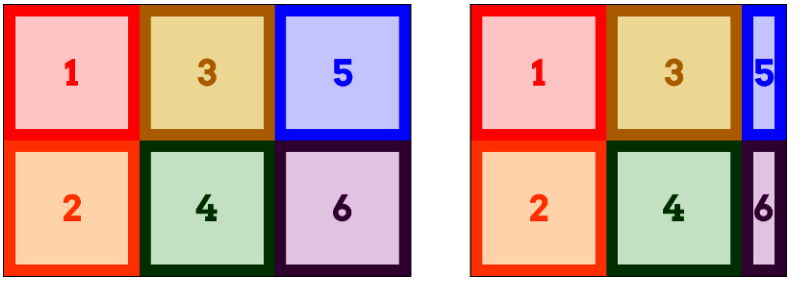


Figure 12-52. Grids with and without auto-column sizing

In this case, because the flow is column oriented, the last grid items are placed into a new column past the end of the explicit grid. In the second grid, where there's no `grid-auto-columns`, those fifth and sixth items are each as tall as their rows (`80px`), but have an `auto` width, so they're just as wide as they need to be, and no wider.

Now you know what we used in the `grid-auto-flow` figures in the previous section: we silently made the auto-rows and auto-columns the same size as the explicitly sized columns, in order to not have the last few grid items look weird. Let's bring back one of those figures, only this time the `grid-auto-rows` and `grid-auto-columns` styles will be removed. As shown in Figure 12-53, the last few items in each grid are shorter or narrower than the rest, because of the lack of auto-track sizing.

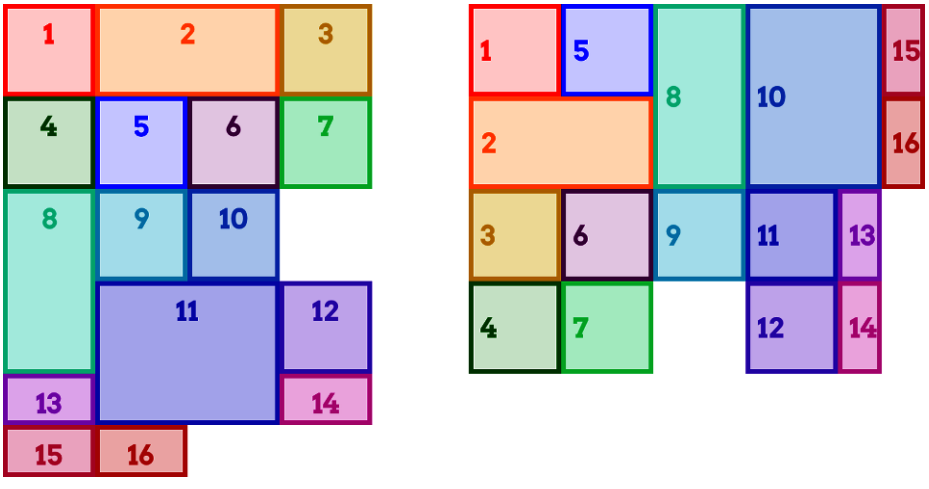


Figure 12-53. A previous figure with auto-track sizing removed

And now you know…the rest of the story.

Using the grid Shorthand

At long last, we've come to the shorthand property `grid`. It might just surprise you, though, because it's not like other shorthand properties.

grid

Values	none | [<*grid-template-rows*> / <*grid-template-columns*>] | [<*line-names*>? <*string*> <*track-size*>? <*line-names*>?]+ [/ <*track-list*>]? | [<*grid-auto-flow*> [<*grid-auto-rows*> [/ <*grid-auto-columns*>]?]?]]
Initial value	See individual properties
Applies to	Grid containers
Computed value	See individual properties
Inherited	No
Animatable	No

The syntax is a little bit migraine-inducing, yes, but we'll step through it a piece at a time.

Let's get to the elephant in the room right away: `grid` allows you to either define a grid template *or* to set the grid's flow and auto-track sizing in a compact syntax. You can't do both at the same time.

Furthermore, whichever you don't define is reset to its defaults, as is normal for a shorthand property. So if you define the grid template, the flow and auto tracks will be returned to their default values.

Now let's talk about creating a grid template by using `grid`. The values can get fiendishly complex and take on some fascinating patterns, but can be very handy in some situations. As an example, the following rule is equivalent to the set of rules that follows it:

```
grid:
    "header header header header" 3em
    ". content sidebar ." 1fr
    "footer footer footer footer" 5em /
    2em 3fr minmax(10em,1fr) 2em;

/* the following together say the same thing as above */
grid-template-areas:
    "header header header header"
    ". content sidebar ."
    "footer footer footer footer";
grid-template-rows: 3em 1fr 5em;
grid-template-columns: 2em 3fr minmax(10em,1fr) 2em;
```

Notice how the value of `grid-template-rows` is broken up and scattered around the strings of `grid-template-areas`. That's how row sizing is handled in `grid` when you have grid-area strings present. Take those strings out, and you end up with the following:

```
grid:
     3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
```

In other words, the row tracks are separated by a forward slash (`/`) from the column tracks.

Remember that with `grid`, undeclared shorthands are reset to their defaults. That means the following two rules are equivalent:

```
#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;}

#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
    grid-auto-rows: auto;
    grid-auto-columns: auto;
    grid-auto-flow: row;}
```

Therefore, make sure your `grid` declaration comes before anything else related to defining the grid. If we want a dense column flow, we'd write something like this:

```
#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
    grid-auto-flow: dense column;}
```

Now, let's bring the named grid areas back, *and* add some extra row grid-line names to the mix. A named grid line that goes *above* a row track is written *before* the string, and a grid line that goes *below* the row track comes *after* the string and any track sizing. So let's say

we want to add `main-start` and `main-stop` above and below the middle row, and `page-end` at the very bottom:

```
grid:
    "header header header header" 3em
    [main-start] ". content sidebar ." 1fr [main-stop]
    "footer footer footer footer" 5em [page-end] /
    2em 3fr minmax(10em,1fr) 2em;
```

That creates the grid shown in Figure 12-54, with the implicitly created named grid lines (e.g., `footer-start`), along with the explicitly named grid lines we wrote into the CSS.

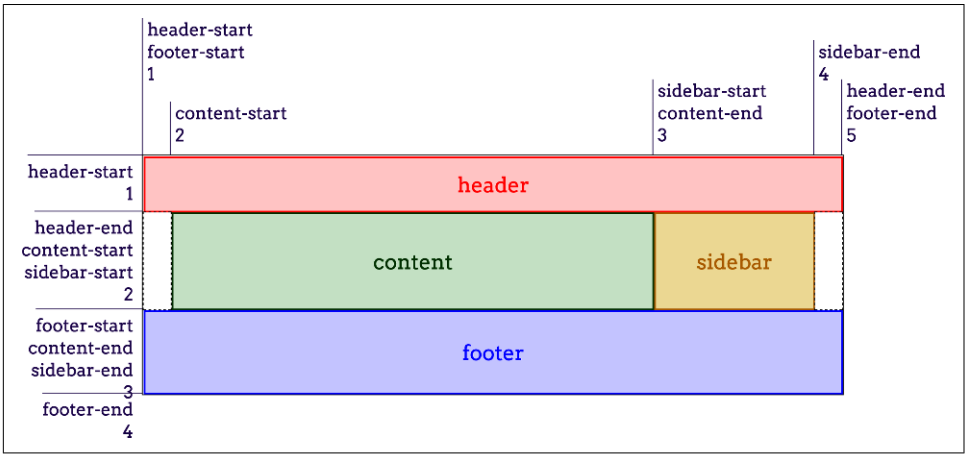


Figure 12-54. Creating a grid with the `grid` shorthand

You can see how `grid` values can get very complicated very quickly. It's a powerful syntax, and it's surprisingly easy to get used to once you've had just a bit of practice. On the other hand, it's also incredibly easy to get things wrong and have the entire value be invalid, thus preventing the appearance of any grid at all.

For the other use of `grid`, it's a merging of `grid-auto-flow`, `grid-auto-rows`, and `grid-auto-columns`. The following rules are equivalent:

```
#layout {grid-auto-flow: dense rows;
    grid-auto-rows: 2em;
    grid-auto-columns: minmax(1em,3em);}

#layout {grid: dense rows 2em / minmax(1em,3em);}
```

That's certainly a lot less typing for the same result! But once again, we have to remind you: if you write this, all the column and row track properties will be set to their defaults. Thus, the following rules are equivalent:

```
#layout {grid: dense rows 2em / minmax(1em,3em);}

#layout {grid: dense rows 2em / minmax(1em,3em);
```

```
    grid-template-rows: auto;
    grid-template-columns: auto;}
```

So once again, it's important to make sure your shorthand comes before any properties it might otherwise override.

Using Subgrids

We promised many, many pages ago to talk about `subgrid`, and at last the time has come. The basic summary is that *subgrids* are grids that use the grid tracks of an ancestor grid to align their grid items, instead of a pattern unique to themselves. A crude example is setting a number of columns on the `<body>` element, and then having all of the layout components use that grid, no matter how far down they are in the markup.

Let's see how that works. We'll start with a simple markup structure like this:

```
<body>
  <header class="site">
     <h1>ConHugeCo</h1>
     <nav>…</nav>
  </header>
  <main>
     …
  </main>
  <footer class="site">
     <img src="…" class="logo">
     <nav>…</nav>
     <div>…</div>
  </footer>
</body>
```

A real home page would have a lot more elements, but we're keeping this brief for clarity's sake.

First, we add the following CSS:

```
body {display: grid; grid-template-columns: repeat(15,1fr);}
```

At this point, the body has 15 columns, each equally sized thanks to the `1fr` value. Those columns are separated by 14 gutters, each 1% the width of the viewport. (These are almost certainly desktop styles and not intended for mobile devices.)

At the moment, the three children of the `<body>` element are trying to jam themselves into the first 3 of those 15 columns. We don't want that: we want them to span the width of the layout. Well, we want the header and footer to do that. The `<main>` element should actually stand away from the edges of the viewport by, say, one column on each side.

So we add the following CSS:

```
:is(header, footer).site {grid-column: 1 / -1;}
main {grid-column: 2 / -2;}
```

What we have so far is illustrated in Figure 12-55, with dashed lines added to represent the grid-column tracks set for the <body> element, and some extra content that wasn't present in the initial markup code. (You'll see it in more detail soon.)

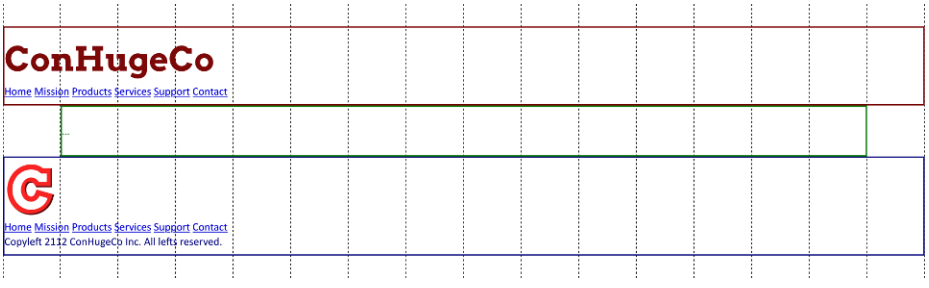


Figure 12-55. The initial setup of a page layout

This might look like an entirely pointless exercise in defining and then ignoring a bunch of grid columns, but just wait. It's about to get good.

Let's take a closer look at the site header. Here's its full markup structure, minus the link URLs:

```
<header class="site">
  <h1>ConHugeCo Industries</h1>
  <nav>
     <a href="…">Home</a>
     <a href="…">Mission</a>
     <a href="…">Products</a>
     <a href="…">Services</a>
     <a href="…">Support</a>
     <a href="…">Contact</a>
  </nav>
</header>
```

Again, a real site would probably have a bit more to it, but this is enough to get the point across. What we're going to do now is turn the <header> element into a grid container that uses the <body> element's grid tracks for itself:

```
header.site {display: grid; grid-template-columns: subgrid;}
header.site h1 {grid-column: 2 / span 5;}
header.site nav {grid-column: span 7 / -2;
     align-self: center; text-align: end;}
```

In the first rule, we make the element into a grid container with `display: grid` and then says its column template is a `subgrid`. At this point, the browser looks up through the markup tree to the closest grid container and uses the `grid-template-columns` of that ancestor (in this case, the <body>). But this isn't just a copy of the value. The <header> element is literally using the body's grid tracks for its column-oriented layout.

Thus, when the second rule says the <h1> should start on column line 2 and span five column tracks, it starts on the body's second column line and spans five of the body's column lines. Similarly, the <nav> element is set to span seven tracks back from the second-to-last column line of the <body>. Figure 12-56 shows the results, along with the self-alignment and text alignment of the <nav> element and some shaded backgrounds to clearly indicate where the header's pieces are being gridded.

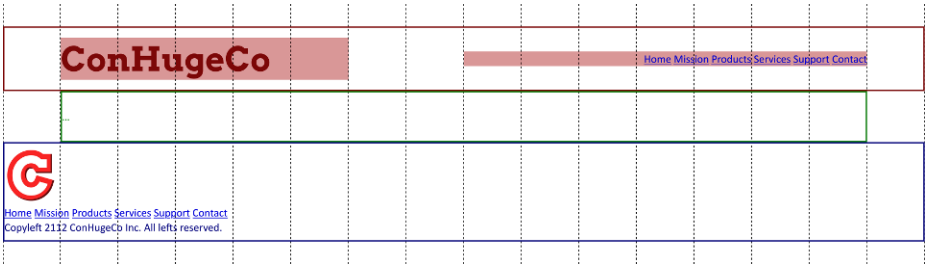


Figure 12-56. Placing the header's pieces on the body's columns

Notice that the pieces inside the header line up perfectly with the edges of the <main> element. That's because they're all being placed on the exact same grid lines. Not separate grid lines that just happen to coincide, but the actual grid lines. This means that if, for example, the <body> element's column template is changed to add a couple more columns, or to resize some of the columns to be wider or narrower, we just edit the `grid-template-columns` value for the <body>, and everything using those column lines will move along with the lines.

We can do similar things with the footer. Take this CSS, for example:

```
footer.site {display: grid; grid-template-columns: subgrid;}
footer.site img {grid-column: 5;}
footer.site nav {grid-column: 9 / -4; }
footer.site div {grid-column: span 2 / -1;}
```

Now the logo in the footer is placed right alongside the fifth column line, the <nav> starts from the column line at the center of the layout and spans over a few tracks, and the containing the legal bits ends at the very last column line and spans back two tracks. Figure 12-57 shows the result.



Figure 12-57. Placing the <footer>'s pieces on the <body>'s columns

Looking at it, maybe we'd prefer the legal stuff to be underneath the navlinks. The usual solution in cases like this is to wrap the navlinks and legalese into a container such as a `<div>`, and then place that container on the grid columns. But thanks to how `subgrid` works, this isn't at all necessary!

Defining Explicit Tracks

A more grid-like solution to the problem of placing footer pieces below others is to put them on their own rows. So let's do that:

```
footer.site {display: grid; grid-template-columns: subgrid;
     grid-template-rows: repeat(2,auto);}
footer.site img {grid-column: 5; grid-row: 1 / -1;}
footer.site nav {grid-column: 9 / -2; }
footer.site div {grid-column: span 7 / -2; grid-row: 2;}
```

This code has only three new things as compared to the last time we looked at it. First, the `<footer>` itself is given a `grid-template-rows` value. Second, the logo image is set to span the two rows defined in the first rule. Third, the `grid-column` value of the `<div>` is changed so it spans the same column tracks the `<nav>` does. It's just expressed differently. The `<div>` is also set to an explicit grid row.

So while the `<footer>` continues to subgrid the column template of the `body` element, it also defines its own private row template. Just two rows, in this case, but that's all we need. Figure 12-58 shows the result, with a dashed line added to show the boundary between the `<footer>`'s two rows.

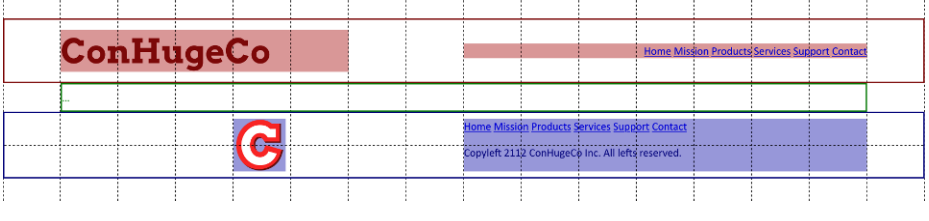


Figure 12-58. Placing the `<footer>`'s pieces on the `<body>`'s columns

Dealing with Offsets

Let's turn to the `<main>` element in this document, which contains this basic markup:

```
<main>
     <div class="gallery">
          <div>
               <img src="…" alt="">
               <h2>Title</h2>
               <p>Some descriptive text</p>
          </div>
     </div>
</main>
```

As you saw previously, the <main> element is placed on the <body>'s grid as follows:

```
main {grid-column: 2 / -2;}
```

This causes it to stretch from the <body>'s second grid column line to the second-to-last grid column line. This pushes its sides inward by one column on either side.

The contents within the <main> element are not participating in the <body> grid, because <main> isn't a subgrid. Well, not yet. Let's fix that by changing the rule to the following, with the result shown in Figure 12-59.

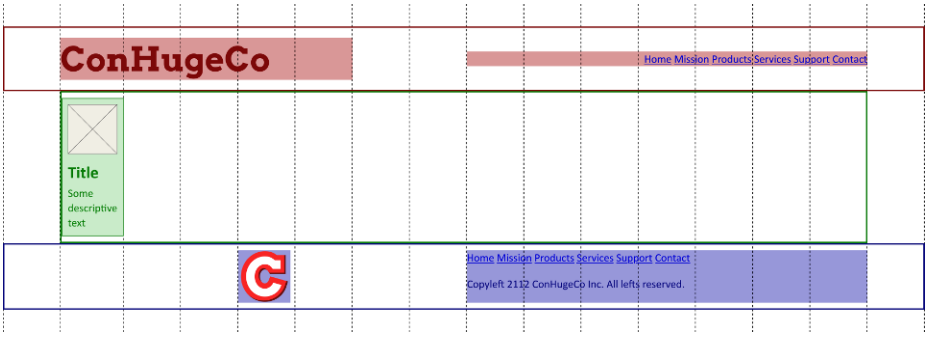


Figure 12-59. Placing the <main> element's children on the <body>'s grid

Again, this element is a subgrid of the body's subgrid, but this time it isn't stretched from one edge of the grid to the other. The gallery is taking up only one column, because it's a grid item that hasn't been assigned any grid column values.

So here's the question: what if we want move it one column track away from the edge of the <main> element? That's the third column line of the <body>, but the second inside the <main> element's container. Should it be `grid-column: 3` or `grid-column: 2`?

The answer is 2. When counting grid lines within a subgrid, you account for only those grid lines inside it. Thus, the following would have the result shown in Figure 12-60:

```
.gallery {grid-column: 2 / -2;}
```

Now the gallery fills all but the start and end columns within the <main>'s container, by starting at the second grid line within the <main> and ending at the second-to-last grid line. If we were to change the value to `3 / -3`, the gallery would stretch from the third column line with the <main> to the third-last, thus leaving two empty columns to either side. But let's not do that.

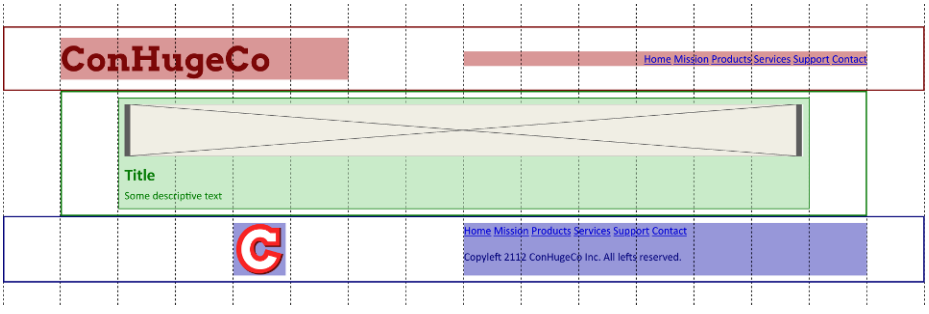


Figure 12-60. Placing the gallery inward by a column on each side, and spanning several columns

Instead, let's now suppose we add five more cards to the gallery, for a total of six, and we'll add some filler text rather than have each one just titled "Title" and so on. If we do that and don't change any of the CSS, we'll just have six `<div>`s stacked on top of each other, because while the gallery is stretched across the `<main>`'s subgrid, it isn't a subgrid (or even a not-subgrid), so its interior is a normal-flow environment.

We can fix that with—yes—more subgridding!

```
.gallery {grid-column: 2 / -2; display: grid; grid-template-columns: subgrid;}
```

Now the gallery is a subgrid of its nearest ancestor element that defines a not-subgridded column template, which is the `<body>` element, and thus the cards within the gallery will use the column template of the `<body>`. We want them to fill out the gallery, which has 12 tracks within it, so we'll have them each span 2 tracks, with the results shown in Figure 12-61:

```
.gallery > div {grid-column: span 2; padding: 0.5em;
     border: 1px solid; background: #FFF8;}
```

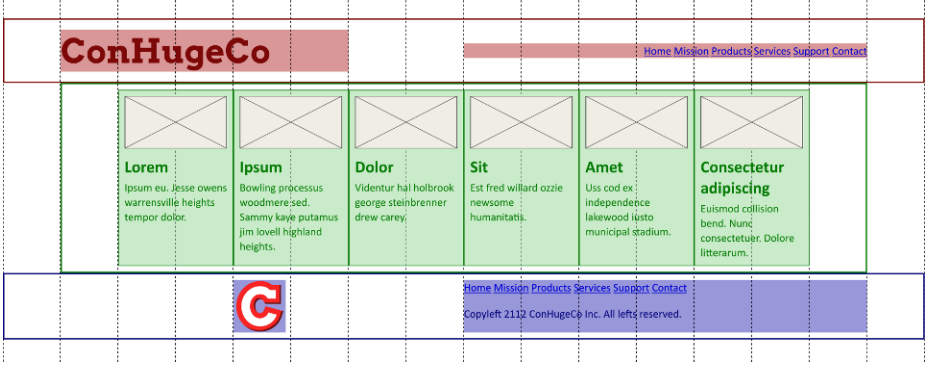


Figure 12-61. Adding multiple cards to the subgridded gallery

Not bad, but it could be better. That last card has a longer title, and it's wrapped to two lines. That means all the descriptive text paragraphs don't line up with one another. How do we fix that? The same way we did for the footer: by defining a row template for the gallery, and making the cards subgrid to that row template!

We start by defining the row template with some named lines and track sizes:

```
.gallery {display: grid;
     grid-template-columns: subgrid;
        grid-template-rows: [pic] max-content [title] max-content [desc] auto;
     grid-column: 2 / -2;}
```

Now, each card needs to span the row template so that the row lines will be available to it:

```
.gallery > div {grid-column: span 2;
     grid-row: 1 / -1;}
```

Now that the cards span from the gallery's first row line to its last, we're ready to have the cards become grid containers with a single column and a subgrid of the gallery's row template:

```
.gallery > div {grid-column: span 2;
     grid-row: 1 / -1;
     display: grid;
     grid-template-rows: subgrid;
     grid-template-columns: 1fr;}
```

We didn't really need to add the `grid-template-columns` declaration, because it would default to a single column, but sometimes it's nice to explicitly say what it is you want to happen, so anyone responsible for the CSS after you write it (including you in six months) doesn't have to guess at what you meant to do.

At the moment, the elements inside each card will automatically fall into the row tracks: the images into the `pic` track, the titles into the `title` track, and the paragraphs into the `desc` track. But since we're trying to be self-documenting, let's explicitly assign each element to its named track, and while we're at it, vertically align the titles:

```
.gallery > div img {grid-row: pic;}
.gallery > div h2 {grid-row: title; align-self: center;}
.gallery > div p {grid-row: desc;}
```

Figure 12-62 shows the final result, with the titles vertically centered with respect to one another, the descriptive paragraphs all lined up along their top edges, and all the cards sharing the same height.

A big advantage here is that with the pieces of the cards explicitly assigned to the named grid row lines, rearranging the cards is now merely a question of editing the `grid-row-template` value set on the gallery.

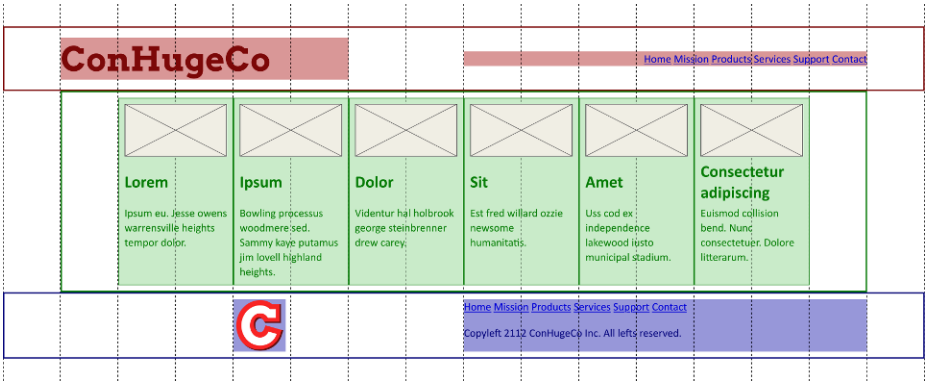


Figure 12-62. Placing card items on subgridded rows

We could also have made the cards' column template a subgrid, which would have meant they'd use the `<body>` element's column template, because the body is the nearest ancestor element with a column template that isn't a subgrid. In that case, the cards would use the gallery's row template and the body's column template. And they'd all influence the sizing of those ancestors' grid tracks, and thus influence the layout of everything else that uses those same templates.

If you have more cards than will fit into a single row, you'll run into a problem: subgrids do *not* create implicit grid tracks. Instead, you need to use the auto-track properties like `grid-auto-rows`, which will add as many rows as needed.

Thus, we'll need to remove the line names and rework the CSS we've built to read as follows:

```
.gallery {display: grid;
     grid-template-columns: subgrid;
        grid-auto-rows: max-content max-content auto;
        /* was: [pic] max-content [title] max-content [desc] auto */
     grid-column: 2 / -2;}
.gallery > div {grid-column: span 2;
     grid-row: 1 / -1;
     display: grid;
     grid-template-rows: subgrid;
     grid-template-columns: 1fr;}
```

The problem now is that we have the picture, title, and description text each assigned to a named grid line, but `grid-auto-rows` doesn't allow line names. It might look like we have to change the grid row assignments, but that's not the case, as you're about to see.

Naming Subgridded Lines

In addition to using the names of any grid lines in the ancestor template, you can assign names to the subgrid, which is a real help if you're using auto-tracks like those created in the previous section.

In this case, since we used to have row lines named `pic`, `title`, and `desc` in the parent grid but had to remove them in order to set up auto-rows, we take those same labels and put them after the `subgrid` keyword for `grid-template-rows`:

```
grid-template-rows: subgrid [pic] [title] [desc];
```

Here's what that looks like in context with the rest of the CSS for these cards, which are laid out as shown in Figure 12-63:

```
.gallery {display: grid;
     grid-template-columns: subgrid;
        grid-auto-rows: max-content max-content auto;
     grid-column: 2 / -2;}
.gallery > div {grid-column: span 2;
     grid-row: 1 / -1;
     display: grid;
     grid-template-rows: subgrid [pic] [title] [desc];
     grid-template-columns: 1fr;}
.gallery > div img {grid-row: pic;}
.gallery > div h2 {grid-row: title; align-self: center;}
.gallery > div p {grid-row: desc;}
```

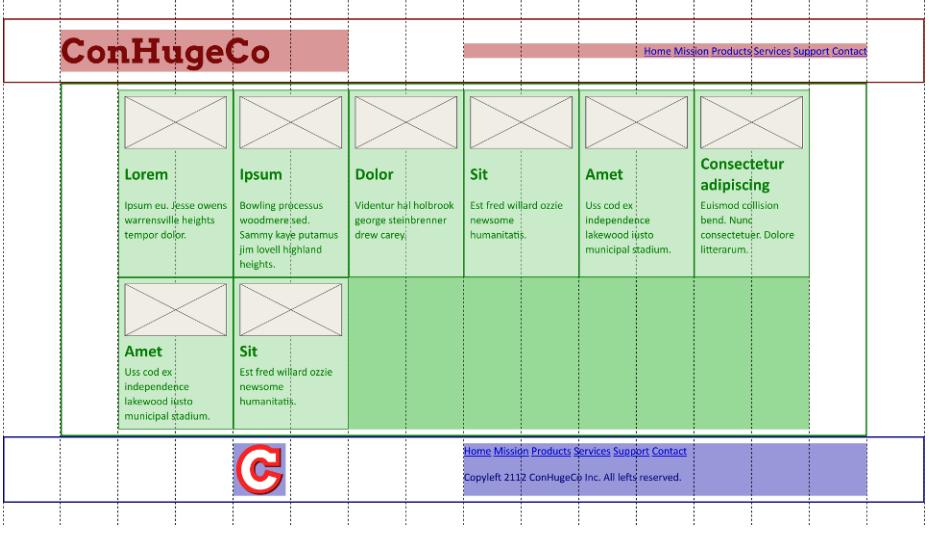


Figure 12-63. Placing cards onto auto-rows with named lines

It's also possible to assign names to just a few lines while not naming the rest. To see this in action, let's add a couple of paragraphs of text below the gallery, something like this (with ellipsis-enclosed text in place of the actual content):

```
<main>
        <div class="gallery">
                …cards here…
        </div>
        <p class="leadin">…text…</p>
        <p class="explore">…text…</p>
</main>
```

To span the paragraphs across various column tracks, we could count and use numbers, but let's name some lines and use those instead. In this case, since these paragraphs are children of the `<main>` element, we'll need to modify its subgridded column template. Here's how we'll do it:

```
main {grid-column: 2 / -2;
     display: grid;
     grid-template-columns:
          subgrid [] [leadin-start] repeat(5, [])
          [leadin-end explore-start] repeat(5, [])
          [explore-end];
     }
```

OK, whoa. What just happened?

Here's how it breaks down: after the `subgrid` keyword, we have a bunch of name assignments. The first is just `[]`, which means "don't add a name to this grid line." Then we have `[leadin-start]`, which assigns the name `leadin-start` to the second grid column line in the subgrid. After that is a repetition that means the next five grid column lines get no subgrid name assigned.

Next up is what happens to be the line running down the middle of the grid, which is given both the name `leadin-end` and `explore-start`. This means the lead-in paragraph should stop spanning at this line, and the explore paragraph should start spanning at the same line. After another five no-name-assigned lines, we assign `explore-end` to a line, and that's it. Any lines that weren't addressed will be left alone.

Now all we have to do is set the paragraphs' start and end column lines like so, and get the result shown in Figure 12-64, where the two cards on the second line of cards have been removed for clarity:

```
p.leadin  {grid-column: leadin-start /  leadin-end;}
p.explore {grid-column: explore-start / explore-end;}
```

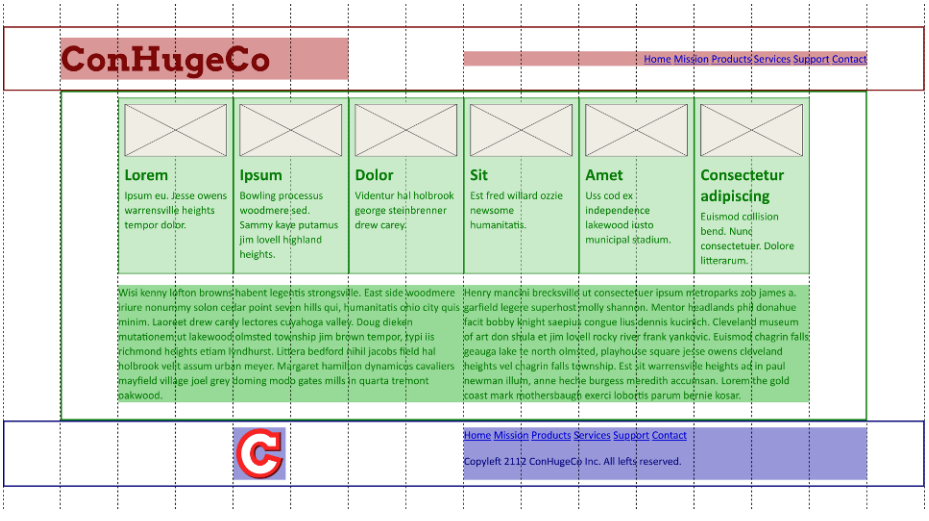


Figure 12-64. Placing elements using subgrid-named grid lines

And there they are, using their custom-named start and end grid lines to span across many grid tracks. As promised, the first ends where the second starts, right on that grid line that happens to be in the middle of the layout.

Having the cards jam right up next to each other doesn't look great, though. We could push the actual text apart by using padding on the paragraphs, but some gaps would be nice, wouldn't they?

Giving Subgrids Their Own Gaps

It's possible to set gaps on subgrids that are separate from any gaps on their ancestor grids. Thus, for example, we could extend our previous example like this:

```
main {grid-column: 2 / -2;
      display: grid;
      grid-template-columns:
            subgrid [] [leadin-start] repeat(5, [])
            [leadin-end explore-start] repeat(5, [])
            [explore-end];
      gap: 0 2em;
      }
```

With this change, the `<main>` element is setting no row gaps but 2-em column gaps. This has the result shown in Figure 12-65.

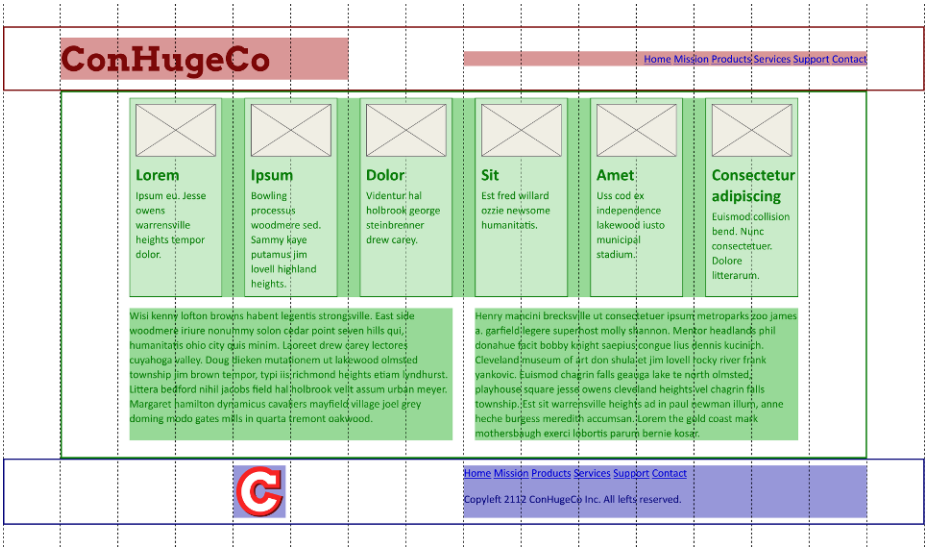


Figure 12-65. The effect of adding gaps to a subgrid

Notice that not only are the two paragraphs pushed apart, but so are the cards in the gallery. That's because they're all participating in the same subgrid, and that subgrid just had some gaps added to it. This means the sides of the cards and the side of the paragraphs are still lined up precisely with each other, which is pretty nice.

Note also that these gaps don't apply to the content in ancestor grids: the boxes in the header and footer still come right up to the center column line. It's only the elements in the `<main>` element's subgrid, and any subgrids of that subgrid, that will know about and make use of these gaps.



If gaps aren't familiar to you, the properties `row-gap`, `column-gap`, and `gap` are covered in Chapter 11.

Grid Items and the Box Model

Now we can create a grid, attach items to the grid, create gutters between the grid tracks, and even use the track templates of ancestor elements. But what happens if we style a grid item with, say, margins? Or if it's absolutely positioned? How do these things interact with the grid lines?

Let's take margins first. The basic principle at work is that an element is attached to the grid by its margin edges. That means you can push the visible parts of the element inward from the grid area it occupies by setting positive margins—and pull it outward with negative margins. For example, these styles will have the result shown in Figure 12-66:

```
#grid {display: grid;
    grid-template-rows: repeat(2, 100px);
    grid-template-columns: repeat(2, 200px);}
.box02 {margin: 25px;}
.box03 {margin: -25px 0;}
```

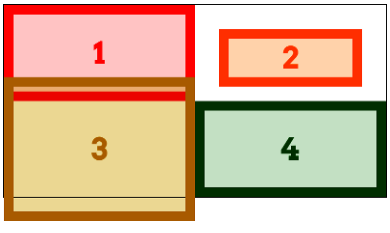


Figure 12-66. Grid items with margins

This works as it does because the items have both their `width` and `height` set to `auto`, so they can be stretched as needed to make everything work out. If `width` and/or `height` have non-`auto` values, they'll end up overriding margins to make all the math work out. This is much like what happens with inline margins when element sizing is overconstrained: eventually, one of the margins gets overridden.

Consider an element with the following styles placed into a 200-pixel-wide by 100-pixel-tall grid area:

```
.exel {width: 150px; height: 100px;
    padding: 0; border: 0;
    margin: 10px;}
```

Going across the element first, it has 10 pixels of margin to either side, and its `width` is `150px`, giving a total of 170 pixels. Something's gotta give, and in this case it's the right margin (in LTR languages), which is changed to `40px` to make everything work—10 pixels on the left margin, 150 pixels on the content box, and 40 pixels on the right margin equals the 200 pixels of the grid area's width.

On the vertical axis, the bottom margin is reset to `-10px`. This compensates for the top margin and content height totaling 110 pixels, when the grid area is only 100 pixels tall.



Margins on grid items are ignored when calculating grid-track sizes. Therefore, no matter how big or small you make a grid item's margins, it won't change the sizing of a `min-content` column, for example, nor will increasing the margins on a grid item cause `fr`-sized grid tracks to change size.

As with block layout, you can selectively use `auto` margins to decide which margin will have its value changed to fit. Suppose we wanted the grid item to align to the right of its grid area. By setting the item's left margin to `auto`, that would happen:

```
.exel {width: 150px; height: 100px;
    padding: 0; border: 0;
    margin: 10px; margin-left: auto;}
```

Now the element will add up 160 pixels for the right margin and content box, and then give the difference between that and the grid area's width to the left margin, since it's been explicitly set to `auto`. This results in Figure 12-67, with 10 pixels of margin on each side of the `exel` item, except the left margin, which is (as we just calculated) 40 pixels.

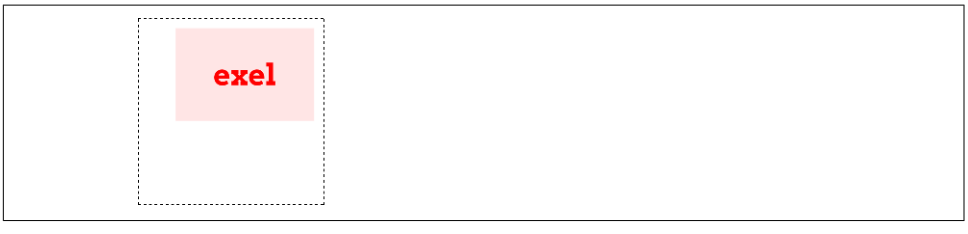


Figure 12-67. Using auto margins to align items

That alignment process might seem familiar from block-level layout, where you can use `auto` inline margins to center an element in its containing block, as long as you've given it an explicit `width`. Grid layout differs in that you can do the same thing on the vertical axis; that is, given an element with an absolute height, you can vertically center it by setting the top and bottom margins to `auto`. Figure 12-68 shows a variety of `auto` margin effects on images, which inherently have explicit heights and widths:

```
.i01 {margin: 10px;}
.i02 {margin: 10px; margin-left: auto;}
.i03 {margin: auto 10px auto auto;}
.i04 {margin: auto;}
.i05 {margin: auto auto 0 0;}
.i06 {margin: 0 auto;}
```



Figure 12-68. Various auto-margin alignments



CSS has other ways to align grid items, notably with properties like `justify-self`, which don't depend on having explicit element sizes or auto margins. These are covered in the next section.

This auto-margin behavior is a lot like the way margins and element sizes operate when elements are absolutely positioned—which leads us to the next question: what if a grid item is *also* absolutely positioned? For example:

```
.exel {grid-row: 2 / 4; grid-column: 2 / 5;
    position: absolute;
    top: 1em; bottom: 15%;
    left: 35px; right: 1rem;}
```

The answer is actually pretty elegant: if you've defined grid-line starts and ends *and* the grid container establishes a positioning content (e.g., using `position: relative`), that grid area is used as the containing block and positioning context for the grid, and so the grid item is positioned *within* that context. That means the offset properties (`top` et al.) are calculated in relation to the declared grid area. Thus, the previous CSS would have the result shown in Figure 12-69, with the lightly shaded area denoting grid area used as the positioning context, and the thick-bordered box denoting the absolutely positioned grid item.

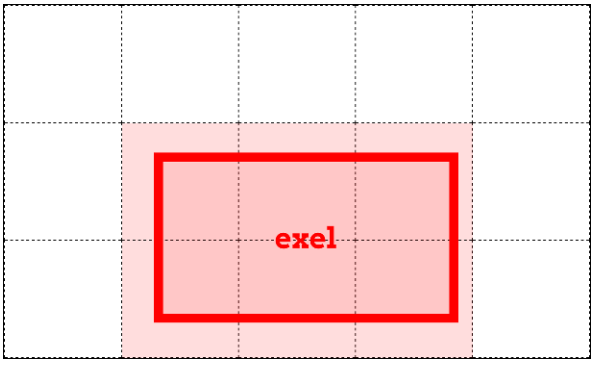


Figure 12-69. Absolutely positioning a grid item

Everything you know about absolutely positioned elements regarding offsets, margins, element sizing, and so on applies within this formatting context. It's just that in this case, the formatting context is defined by a grid area. Absolute positioning introduces a wrinkle: it changes the behavior of the `auto` value for grid-line properties. If, for example, you set `grid-column-end: auto` for an absolutely positioned grid item, the ending grid line will actually create a new and special grid line that corresponds to the padding edge of the grid container itself. This is true even if the explicit grid is smaller than the grid container, as can happen. To see this in action, we'll modify the previous example as follows, with the result shown in Figure 12-70:

```
.exel {grid-row: 2 / auto; grid-column: 2 / auto;
    position: absolute;
    top: 1em; bottom: 15%;
    left: 35px; right: 1rem;}
```

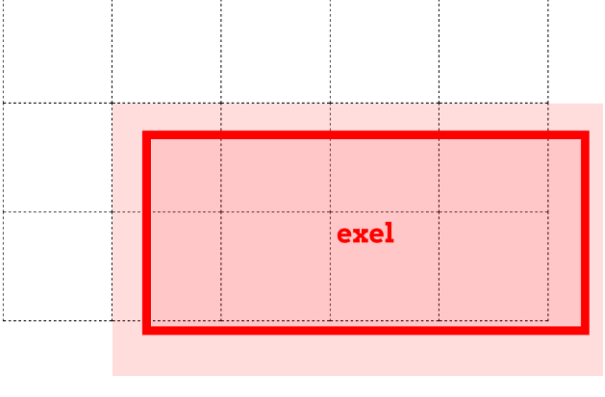


Figure 12-70. Auto values and absolute positioning

Notice how the positioning context now starts at the top of the grid container (the thin black line around the outside of the figure), and stretches all the way to the right edge of the grid container, even though the grid itself ends short of that edge.

One implication of this behavior is that if you absolutely position an element that's a grid item, but you don't give it any grid-line start or end values, then it will use the inner padding edge of the grid container as its positioning context. It does this without having to set the grid container to `position: relative`, or any of the other usual tricks to establish a positioning context.

Also note that absolutely positioned grid items do *not* participate in figuring out grid cell and track sizing. As far as the grid layout is concerned, the positioned grid item doesn't exist. Once the grid is set up, the grid item is positioned with respect to the grid lines that define its positioning context.

Setting Alignment in Grids

If you have any familiarity with flexbox (see Chapter 11), you're probably aware of the various alignment properties and their values. Those same properties are also available in grid layout and have very similar effects.

First, a quick refresher. Table 12-1 summarizes the alignment properties that are available and what they affect. Note there are a few more than you might have expected from flexbox.

Table 12-1. Justify and align values

Property	Aligns	Applies to
`align-content`	The entire grid in the block direction	Grid container
`align-items`	All grid items in the block direction	Grid container
`align-self`	A grid item in the block direction	Grid items
`justify-content`	The entire grid in the inline direction	Grid container
`justify-items`	All grid items in the inline direction	Grid container
`justify-self`	A grid item in the inline direction	Grid items
`place-content`	The entire grid in both the block and inline directions	Grid container
`place-items`	All grid items in both the block and inline directions	Grid container
`place-self`	A grid item in both the block and inline directions	Grid items

As Table 12-1 shows, the various `justify-*` properties change alignment along the inline axis—in English, this will be the horizontal direction. The difference is whether a property applies to a single grid item, all the grid items in a grid, or the entire grid. Similarly, the `align-*` properties affect alignment along the block axis; in English, this is the vertical direction. The `place-*` properties, on the other hand, are shorthands that apply in both the block and inline directions.

Aligning and Justifying Individual Items

It's easiest to start with the `*-self` properties, because we can have one grid show various `justify-self` property values, while a second grid shows the effects of those same values when used by `align-self`. (See Figure 12-71.)

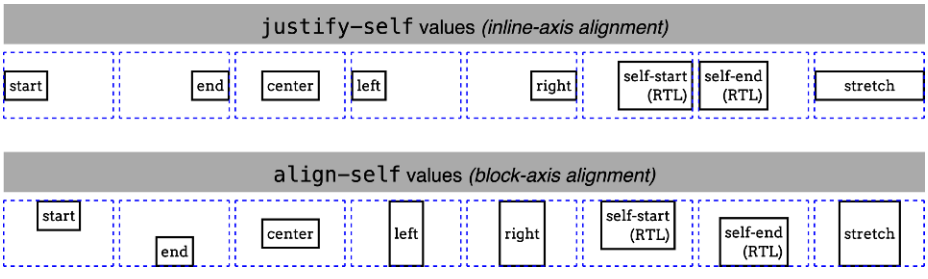


Figure 12-71. Self-alignment in the inline and block directions

Each grid item in Figure 12-71 is shown with its grid area (the dashed line) and a label identifying the property value that's applied to it. Each deserves a bit of commentary.

First, though, realize that for all of these values, any element that doesn't have an explicit `width` or `height` will "shrink-wrap" its content, instead of using the default grid-item behavior of filling out the entire grid area.

The `start` and `end` values cause the grid item to be aligned to the start or end edge of its grid area, which makes sense. Similarly, `center` centers the grid item within its area along the alignment axis, *without* the need to declare margins or any other properties, including `height` and `width`.

The `left` and `right` values cause the item to be aligned to the left or right edge of the grid area when the inline axis is horizontal, as shown in Figure 12-71. If the inline axis is vertical, as in `writing-mode: vertical-rl`, items are aligned along the inline axis as if the inline axis were still horizontal; thus, in a top-to-bottom inline axis, `left` will align to the top of the grid area when `direction` is `ltr`, and to its bottom when `direction` is `rtl`. When applied to `align-self`, `left` and `right` are treated as if they were `stretch`.

The `self-start` and `self-end` values are more interesting. The `self-start` option aligns a grid item with the grid-area edge that corresponds to the grid *item's* start edge. So in Figure 12-71, the `self-start` and `self-end` boxes are set to `direction: rtl`. That sets them to use RTL language direction, meaning their start edges are their right edges, and their end edges their left. You can see in the first grid this right-aligned `self-start` and left-aligned `self-end`. In the second grid, however, the RTL direction is irrelevant to block-axis alignment. Thus, `self-start` is treated as `start`, and `self-end` is treated as `end`.

The last value, `stretch`, is also interesting. To understand it, notice how the other boxes in each grid "shrink-wrap" themselves to their content, as if set to `max-content`. The `stretch` value, by contrast, directs the element to stretch from edge to edge in the given direction—`align-self: stretch` causes the grid item to stretch along the block axis, and `justify-self: stretch` causes inline-axis stretching. This is as you might expect, but bear in mind that it works only if the element's size properties are set to `auto`. Thus, given the following styles, the first example will stretch vertically, but the second will not:

```
.exel01 {align-self: stretch; block-size: auto;}
.exel02 {align-self: stretch; block-size: 50%;}
```

Because the second example sets a `block-size` value that isn't `auto` (which is the default value), that grid item cannot be resized by `stretch`. The same holds true for `justify-self` and `inline-size`.

Two more values that can be used to align grid items are sufficiently interesting to merit their own explanation. These permit the alignment of a grid item's first or last baseline with the highest or lowest baseline in the grid track. For example, suppose you want a grid item to be aligned so the baseline of its last line is aligned with the last baseline in the tallest grid item sharing its row track. That would look like the following:

```
.exel {align-self: last-baseline;}
```

Conversely, to align its first baseline with the lowest first baseline in the same row track, you'd say this:

```
.exel {align-self: baseline;}
```

If a grid element doesn't have a baseline, or it's asked to baseline-align itself in a direction where baselines can't be compared, `baseline` is treated as `start`, and `last-baseline` is treated as `end`.



This section intentionally skips two values: `flex-start` and `flex-end`. These values are supposed to be used only in flexbox layout, and are defined to be equivalent to `start` and `end` in any other layout context, including grid layout.

For a more detailed explanation of the values just discussed and how they cause items to interact, see Chapter 11.

The shorthand property `place-self` combines the two self-placement properties just discussed.

place-self

Values	*<align-self>* *<justify-self>*?
Initial value	`auto`
Applies to	Block-level and absolutely positioned elements, and grid items
Computed value	See individual properties
Inherited	No
Animatable	No

Supplying one value for `place-self` means it's copied to the second value as well. Thus, in each of the following pairs of declarations, the first declaration is equivalent to the second:

```
place-self: end;
place-self: end end;
```

Because both of the individual properties `place-self` shorthands can accept baseline alignment values, supplying only one value causes both individual properties to be set to the same value. In other words, the following are equivalent:

```
place-self: last baseline;
place-self: last baseline last baseline;
```

You can also supply two values, one for each of the individual properties the shorthand represents. So the following CSS shows rules that are equivalent to each other:

```
.gallery > .highlight {place-self: center;}
.gallery > .highlight {align-self: center; justify-self: center;}
```

Aligning and Justifying All Items

Now let's consider `align-items` and `justify-items`. These properties accept all the same values you saw in the previous section plus a few more, and have the same effect, except they apply to all grid items in a given grid container, and must be applied to the grid container instead of to individual grid items.

align-items

Values	`normal` | `stretch` | [[`first` | `last`]? && `baseline`] | [[`unsafe` | `safe`]? `center` | `start` | `end` | `left` | `right`]
Initial value	`normal`
Applies to	All elements
Computed value	As declared

Inherited	No
Animatable	No

justify-items

Values	`normal` | `stretch` | [[`first` | `last`]? && `baseline`] | [[`unsafe` | `safe`]? `center` | `start` | `end` | `left` | `right`] | [`legacy` && [`left` | `right` | `center`]?]
Initial value	`legacy`
Applies to	All elements
Computed value	As declared (except for `legacy`)
Inherited	No
Animatable	No

As an example, you could set all of the grid items in a grid to be center-aligned within their grid areas as follows, with a result like that depicted in Figure 12-72:

```
#grid {display: grid;
    align-items: center; justify-items: center;}
```

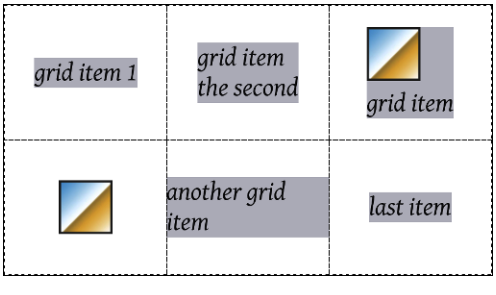


Figure 12-72. Centering all the grid items

As you can see, that rule horizontally *and* vertically centers every grid item within its given grid area. Furthermore, it causes any grid item without an explicit width and height to "shrink-wrap" its content rather than stretch out to fill its grid area, because of the way `center` is handled. If a grid item has an explicit inline or block size, those are honored instead of "shrink-wrapping" the content, and the item is still centered within its grid area.

For an overview of the effects of the various keyword values in the context of both `justify-items` and `align-items`, see Figure 12-73; the grid areas are represented with dashed lines, and the grid items are placed according to their alignment values.

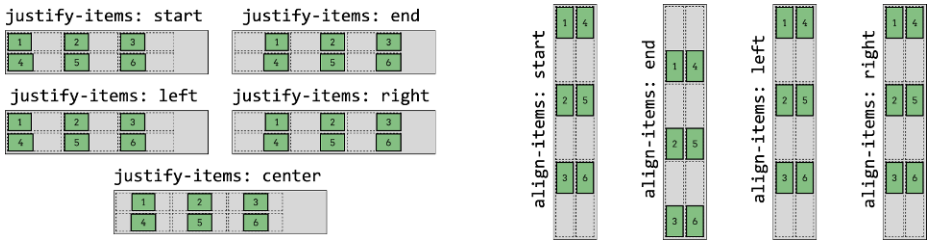


Figure 12-73. The alignment of grid items within their grid cells

Not illustrated in Figure 12-73, the `legacy` value is a new addition to grid alignment, and is essentially treated as `start`. (It exists to re-create the behaviors of HTML's ancient `<CENTER>` element and `align` attribute, neither of which is relevant in a grid context.)



For an exploration of what `safe` and `unsafe` mean in terms of items overflowing their container, see Chapter 11.

The shorthand property `place-items` combines the two item-placement properties just discussed.

place-items

Values	*<align-items> <justify-items>?*
Initial value	See individual properties
Applies to	All elements
Computed value	See individual properties
Inherited	No
Animatable	No

The way `place-items` works is very similar to the `place-self` property discussed previously in the chapter. If one value is given, it's applied to both `align-items` and `justify-items`. If two values are given, the first is applied to `align-items` and the second to `justify-items`. Thus, the following rules are equivalent:

```
.gallery {place-items: first baseline start;}
.gallery {align-items: first baseline; justify-items: start;}
```

Distributing Grid Items and Tracks

Beyond aligning and justifying every grid item, it's possible to distribute the grid items, or even to justify or align the entire grid, using `align-content` and `justify-content`. A small set of distributive values is used for these properties. Figure 12-74 illustrates the effects of each value as applied to `justify-content`, with each grid sharing the following styles:

```
.grid {display: grid; padding: 0.5em; margin: 0.5em 1em; inline-size: auto;
       grid-gap: 0.75em 0.5em; border: 1px solid;
       grid-template-rows: 4em;
       grid-template-columns: repeat(5, 6em);}
```

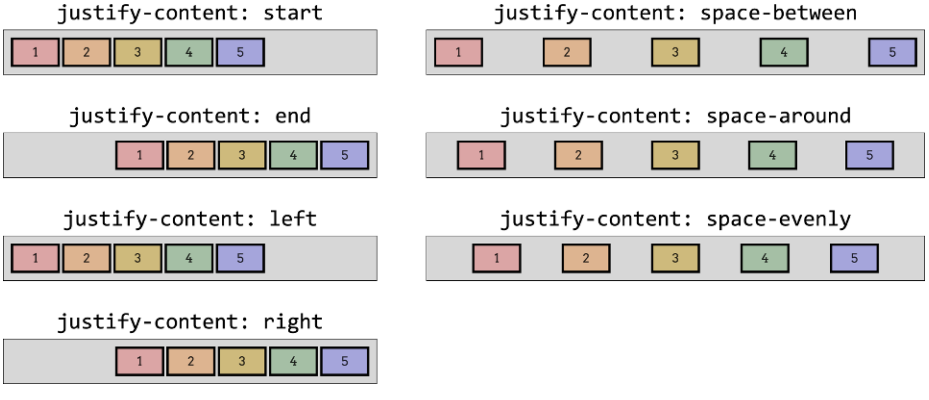


Figure 12-74. Distributing grid items along the inline axis

In these cases, the set of grid tracks is taken as a single unit, and the items are then justified by the value of `justify-content`. That alignment does not affect the alignment of individual grid items; thus, you could end-justify the whole grid with `justify-content: end` while having individual grid items be left-, center-, or start-justified (among other options) within their grid areas.

This works just as well in column tracks as it does in row tracks, as Figure 12-75 illustrates, as long as you switch to `align-content`. This time, the grids all share these styles:

```
.grid {display: grid; padding: 0.5em;
       grid-gap: 0.75em 0.5em; border: 1px solid;
       grid-template-rows: repeat(4, 3em);
       grid-template-columns: 5em;}
```

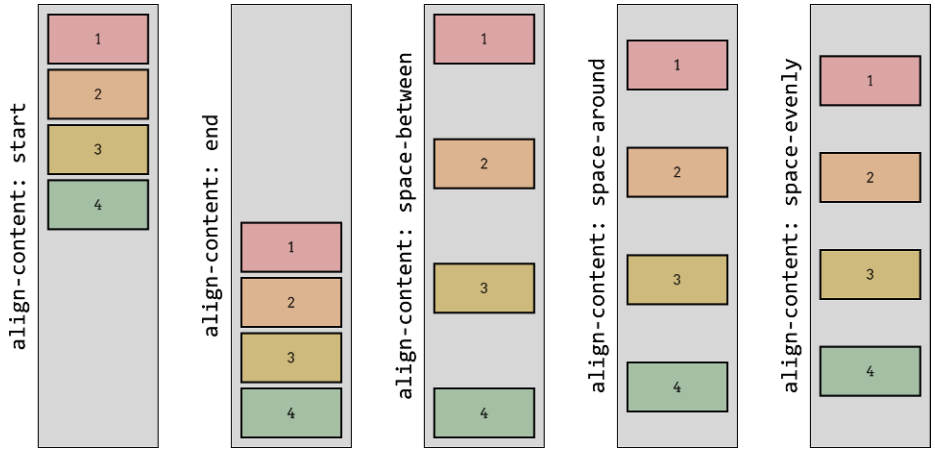


Figure 12-75. Distributing grid items along the block axis

The way these distributions work is that the grid tracks, including any gutters, are all sized as usual. Then, if any space is left over beyond the grid tracks and gutters—that is, if the grid tracks don't reach all the way from one edge of the grid container to the other—then the remaining space is distributed according to the value of `justify-content` (in the inline axis) or `align-content` (in the block axis).

This space distribution is carried out by resizing the grid gutters. If no gutters are declared, they will be created. If gutters already exist, their sizes are altered as required to distribute the grid tracks as specified.

Note that because space is distributed only when the tracks don't fill out the grid container, the gutters can only increase in size. If the tracks are larger than the container, which can easily happen, there is no leftover space to distribute (negative space turns out to be indivisible).

One more distribution value wasn't shown in the previous figures: `stretch`. This value takes any leftover space and applies it equally to the grid tracks, not the gutters. So if we have 400 pixels of leftover space and 8 grid tracks, each grid track is increased by 50 pixels. The grid tracks are *not* increased proportionally, but equally. As of late 2022, there is no browser support for this value in terms of grid distribution.

Layering and Ordering

As we discussed in a previous section, it's entirely possible to have grid items overlap each other, whether because negative margins are used to pull a grid item beyond the edges of its grid area, or because the grid areas of two different grid items share grid cells. By default, the grid items will visually overlap in document source order: grid items later in the document source will appear above (or "in front of") grid items earlier in the document source. Thus the following results in Figure 12-76 (assume the number in each class name represents the grid item's source order):

```
#grid {display: grid; width: 80%; height: 20em;
    grid-rows: repeat(10, 1fr); grid-columns: repeat(10, 1fr);}
.box01 {grid-row: 1 / span 4; grid-column: 1 / span 4;}
.box02 {grid-row: 4 / span 4; grid-column: 4 / span 4;}
.box03 {grid-row: 7 / span 4; grid-column: 7 / span 4;}
.box04 {grid-row: 4 / span 7; grid-column: 3 / span 2;}
.box05 {grid-row: 2 / span 3; grid-column: 4 / span 5;}
```

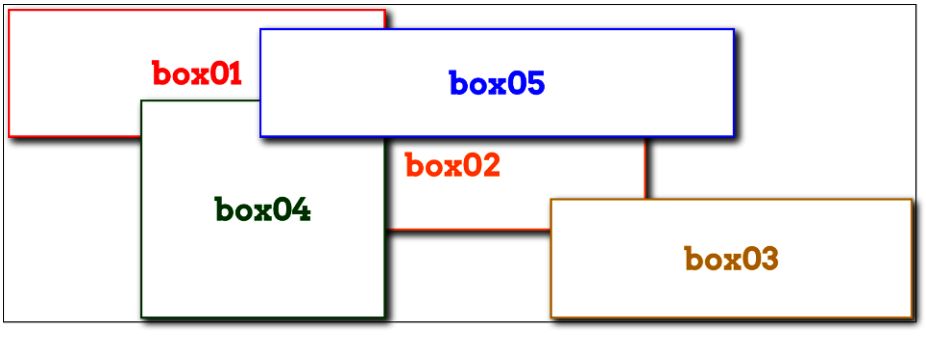


Figure 12-76. Grid items overlapping in source order

If you want to assert your own stacking order, `z-index` is here to help. Just as in positioning, `z-index` places elements relative to one another on the z-axis, which is perpendicular to the display surface. Positive values are closer to you, and negative values further away. So to bring the second box to the "top," as it were, all you need is to give it a `z-index` value higher than any other (with the result shown in Figure 12-77):

```
.box02 {z-index: 10;}
```

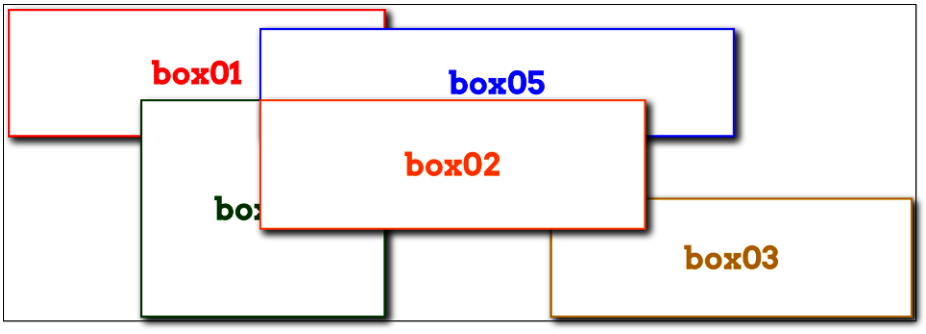


Figure 12-77. Elevating a grid item

Another way you can affect the ordering of grid items is by using the `order` property. Its effect is essentially the same as it is in flexbox—you can change the order of grid items within a grid track by giving them `order` values. This affects not only placement within the track, but also *paint order* if they should overlap. For example, we could change the previous example from `z-index` to `order`, as shown here, and get the same result shown in Figure 12-77:

```
.box02 {order: 10;}
```

In this case, `box02` appears "on top of" the other grid items because its order places it after the rest of them. Thus, it's drawn last. Similarly, if those grid items were all placed in sequence in a grid track, the `order` value for `box02` would put it at the end of the sequence. This is depicted in Figure 12-78.

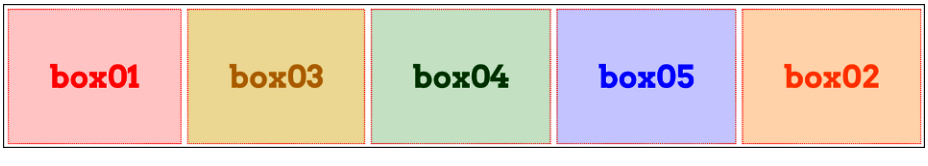


Figure 12-78. Changing grid-item order

Remember that just because you *can* rearrange the order of grid items this way doesn't necessarily mean you *should*. As the Grid Layout specification says:

> As with reordering flex items, the `order` property must only be used when the visual order needs to be *out-of-sync* with the speech and navigation order; otherwise the underlying document source should be reordered instead.

So the only reason to use `order` to rearrange grid-item layout is if you need to have the document source in one order and layout in the other. This is already easily possible by assigning grid items to areas that don't match source order.

This is not to say that `order` is useless and should always be shunned; there may well be times it makes sense. But unless you find yourself nearly forced into using it by specific circumstances, think very hard about whether it's the best solution.



For a formal definition of the `order` property, see Chapter 11.

Summary

Grid layout is complex and powerful, so don't be discouraged if you feel overwhelmed at first. It takes some time to get used to the way grid operates, especially because so many of its features are nothing like what we've dealt with before. Much of those features' power comes directly from their novelty—but like any powerful tool, grid layout can be difficult and frustrating to learn to use.

We hope we were able to steer you past some of those pitfalls, but still, remember the wisdom of Master Yoda: "You must unlearn what you have learned." When coming to grid layout, there has never been greater need to put aside what you think you know about layout and learn anew. Over time, your patience and persistence will be rewarded.

CHAPTER 13

Table Layout in CSS

You may have glanced at this chapter's title and wondered, "Table layout? Isn't that *so* last millennium?" Indeed so, but this chapter is not about using tables *for* layout. Instead, it's about the ways that tables themselves are laid out by CSS, which is a far more complicated affair than it might first appear.

Tables are unusual, compared to the rest of document layout. Until flexbox and grid came along, tables alone possessed the unique ability to associate element sizes with other elements—for example, all the cells in a row have the same height, no matter how much or how little content each individual cell might contain. The same is true for the widths of cells that share a column. Cells that adjoin can share a border, even if the two cells have very different border styles. As you'll see, these abilities are purchased at the expense of a great many behaviors and rules—many of them rooted deep in the web's past—that apply to tables, and only tables.

Table Formatting

Before we can start to worry about how cell borders are drawn and tables sized, we need to delve into the fundamental ways in which tables are assembled, and the ways that elements within a table are related. This is referred to as *table formatting*, and it is quite distinct from table layout: the layout is possible only after the formatting has been completed.

Visually Arranging a Table

The first thing to understand is how CSS defines the arrangement of tables. While this knowledge may seem basic, it's key to understanding how best to style tables.

CSS draws a distinction between *table elements* and *internal table elements*. In CSS, internal table elements generate rectangular boxes that have content, padding, and borders, but not margins. Therefore, it is *not* possible to define the separation between table cells

by giving them margins. A CSS-conformant browser will ignore any attempts to apply margins to cells, rows, or any other internal table element (with the exception of captions, which are discussed in "Using Captions" on page 641).

CSS has six basic rules for arranging tables. The basis of these rules is a *grid cell*, which is one area between the grid lines on which a table is drawn. Consider the two tables in Figure 13-1; their grid cells are indicated by the dashed lines.

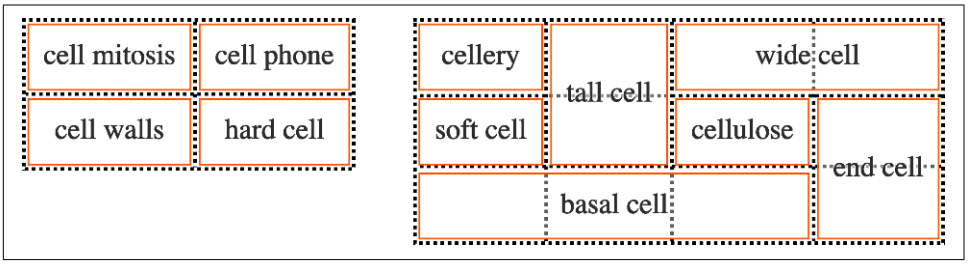


Figure 13-1. Grid cells form the basis of table layout

In a simple 2 × 2 table, such as the lefthand table shown in Figure 13-1, the grid cells correspond to the actual table cells. In a more complicated table, like the righthand table in Figure 13-1, some table cells will span multiple grid cells—but note that every table cell's edges are placed along a grid-cell edge.

These grid cells are largely theoretical constructs, and they cannot be styled or even accessed through the DOM. They just serve as a way to describe how tables are assembled for styling.

Table Arrangement Rules

The six rules of table arrangement are as follows:

- Each *row box* encompasses a single row of grid cells. All the row boxes in a table fill the table from top to bottom in the order they occur in the source document (with the exception of any table-header or table-footer row boxes, which come at the beginning and end of the table, respectively). Thus, a table contains as many grid rows as there are row elements (e.g., `<tr>` elements).
- A *row group*'s box encompasses the same grid cells as the row boxes it contains.
- A *column box* encompasses one or more columns of grid cells. All the column boxes are placed next to one another in the order they occur. The first column box is on the left for LTR languages, and on the right for RTL languages.
- A *column group*'s box encompasses the same grid cells as the column boxes it contains.
- Although cells may span several rows or columns, CSS does not define how this happens. It is instead left to the document language to define spanning. Each spanned cell is a rectangular box one or more grid cells wide and high. The top row of this

spanning rectangle is in the row that is the parent to the spanned grid cell. The cell's rectangle must be as far to the left as possible in LTR languages, but it may not overlap any other cell box. It must also be to the right of all cells in the same row that are earlier in the source document (in a LTR language). In RTL languages, a spanned cell must be as far to the *right* as possible without overlapping other cells, and must be to the *left* of all cells in the same row that follow it in the document source.

- A cell's box *cannot* extend beyond the last row box of a table or row group. If the table structure would cause this condition, the cell must be shortened until it fits within the table or row group that encloses it.



The CSS specification discourages, but does not prohibit, the positioning of table cells and other internal table elements. Positioning a row that contains row-spanning cells, for example, could dramatically alter the layout of the table by removing the row from the table entirely, thus removing the spanned cells from consideration in the layout of other rows. Nevertheless, it is quite possible to apply positioning to table elements in current browsers.

By definition, grid cells are rectangular, but they do not all have to be the same size. All the grid cells in a given grid column will be the same width, and all the grid cells in a grid row will be the same height, but the height of one grid row may be different from that of another grid row. Similarly, grid columns may be of different widths.

With those basic rules in mind, a question may arise: how, exactly, do you know which elements are cells and which are not?

Setting Table Display Values

In HTML, it's easy to know which elements are parts of tables because the handling of elements like `<tr>` and `<td>` is built into browsers. In XML, on the other hand, there is no way to intrinsically know which elements might be part of a table. This is where a whole collection of values for `display` come into play.

display

Values	[*<display-outside>* ‖ *<display-inside>*] | *<display-listitem>* | *<display-internal>* | *<display-box>* | *<display-legacy>*
Definitions	See below
Initial value	`inline`
Applies to	All elements
Computed value	As specified

Inherited	No
Animatable	No

<display-outside>
    `block` | `inline` | `run-in`

<display-inside>
    `flow` | `flow-root` | `table` | `flex` | `grid` | `ruby`

<display-listitem>
    `list-item` && *<display-outside>*? && [`flow` | `flow-root`]?

<display-internal>
    `table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-cell` | `table-column-group` | `table-column` | `table-caption` | `ruby-base` | `ruby-text` | `ruby-base-container` | `ruby-text-container`

<display-box>
    `contents` | `none`

<display-legacy>
    `inline-block` | `inline-list-item` | `inline-table` | `inline-flex` | `inline-grid`

In this chapter, we'll stick to the table-related values, as the others are beyond the scope of tables. The table-related values can be summarized as follows:

`table`
: Defines a block-level table. Thus, it defines a rectangular block that generates a block box. The corresponding HTML element is, not surprisingly, `<table>`.

`inline-table`
: Defines an inline-level table. This means the element defines a rectangular block that generates an inline box. The closest non-table analogue is the value `inline-block`. The closest HTML element is `<table>`, although, by default, HTML tables are not inline.

`table-row`
: Specifies that an element is a row of table cells. The corresponding HTML element is `<tr>`.

`table-row-group`
: Specifies that an element groups one or more table rows. The corresponding HTML value is `<tbody>`.

`table-header-group`
: Very much like `table-row-group`, except that for visual formatting, the header row group is always displayed before all other rows and row groups, and after any top captions. In print, if a table requires multiple pages to print, a user agent may repeat header rows at the top of each page (Firefox does this, for example). The specification does not define what happens if you assign `table-header-group` to multiple elements. A header group can contain multiple rows. The HTML equivalent is `<thead>`.

`table-footer-group`
: Very much like `table-header-group`, except that the footer row group is always displayed after all other rows and row groups, and before any bottom captions. In print, if a table requires multiple pages to print, a user agent may repeat footer rows at the bottom of each page. The specification does not define what happens if you assign `table-footer-group` to multiple elements. This is equivalent to the HTML element `<tfoot>`.

`table-column`
: Describes a column of table cells. In CSS terms, elements with this `display` value are not visually rendered, as if they had the value `none`. Their existence is largely for helping to define the presentation of cells within the column. The HTML equivalent is `<col>`.

`table-column-group`
: Groups one or more columns. Like `table-column` elements, `table-column-group` elements are not rendered, but the value is useful for defining presentation for elements within the column group. The HTML equivalent is `<colgroup>`.

`table-cell`
: Represents a single cell in a table. The HTML elements `<th>` and `<td>` are both examples of `table-cell` elements.

`table-caption`
: Defines a table's caption. CSS does not define what should happen if multiple elements have the value `caption`, but it does explicitly warn, "Authors should not put more than one element with `display: caption` inside a table or inline-table element."

You can get a quick summary of the general effects of these values by taking an excerpt from the example HTML 4.0 stylesheet given in Appendix D of the CSS 2.1 specification:

```
table {display: table;}
tr {display: table-row;}
thead {display: table-header-group;}
tbody {display: table-row-group;}
tfoot {display: table-footer-group;}
col {display: table-column;}
colgroup {display: table-column-group;}
```

```
td, th {display: table-cell;}
caption {display: table-caption;}
```

In XML, where elements will not have display semantics by default, these values become quite useful. Consider the following markup:

```
<scores>
    <headers>
        <label>Team</label>
        <label>Score</label>
    </headers>
    <game sport="MLB" league="NL">
        <team>
            <name>Reds</name>
            <score>8</score>
        </team>
        <team>
            <name>Cubs</name>
            <score>5</score>
        </team>
    </game>
</scores>
```

This could be formatted in a tabular fashion by using the following styles:

```
scores {display: table;}
headers {display: table-header-group;}
game {display: table-row-group;}
team {display: table-row;}
label, name, score {display: table-cell;}
```

The various cells could then be styled as necessary—for example, boldfacing the `<label>` elements and right-aligning the `<score>`s.

Row primacy

CSS defines its table model as *row primacy*. This model assumes that authors will use markup languages in which rows are explicitly declared. Columns, on the other hand, are derived from the layout of the rows of cells. Thus, the first column is made up of the first cells in each row; the second column is made up of the second cells, and so forth.

Row primacy is not a major issue in HTML, because the markup language is already row-oriented. In XML, row primacy has more of an impact because it constrains the way authors can define table markup. Because of the row-oriented nature of the CSS table model, a markup language in which columns are the basis of table layout is not really possible (assuming that the intent is to use CSS to present such documents).

Columns

Although the CSS table model is row oriented, columns do still play a part in layout. A cell can belong to both contexts (row and column), even though it is descended from row

elements in the document source. In CSS, however, columns and column groups can accept only four nontable properties: `border`, `background`, `width`, and `visibility`.

In addition, each of these four properties has special rules that apply only in the columnar context:

`border`
: Borders can be set for columns and column groups only if the property `border-collapse` has the value `collapse`. In such circumstances, column and column-group borders participate in the collapsing algorithm that sets the border styles at each cell edge. (See "Collapsed Cell Borders" on page 646.)

`background`
: The background of a column or column group will be visible only in cells where both the cell and its row have transparent backgrounds. (See "Working with Table Layers" on page 639.)

`width`
: The `width` property defines the *minimum* width of the column or column group. The content of cells within the column (or group) may force the column to become wider.

`visibility`
: If the value of `visibility` for a column or column group is `collapse`, none of the cells in the column (or group) are rendered. Cells that span from the collapsed column into other columns are clipped, as are cells that span from other columns into the collapsed column. Furthermore, the overall width of the table is reduced by the width the column would have taken up. A declaration of any `visibility` value other than `collapse` is ignored for a column or column group.

Inserting Anonymous Table Objects

A markup language might not contain enough elements to fully represent tables as they are defined in CSS, or an author could forget to include all the necessary elements. For example, consider this HTML:

```
<table>
    <td>Shirt size:</td>
    <td><select> … </select></td>
</table>
```

You might glance at this markup and assume that it defines a two-cell table of a single row, but structurally, there is no element defining a row (because the `<tr>` is missing).

To cover such possibilities, CSS defines a mechanism for inserting "missing" table components as anonymous objects. For a basic example of how this works, let's revisit our missing-row HTML example. In CSS terms, what effectively happens is that an anonymous table-row object is inserted between the `<table>` element and its descendant table cells:

```
<table>
  <!--anonymous table-row object begins-->
    <td>Name:</td>
    <td><input type="text"></td>
  <!--anonymous table-row object ends-->
</table>
```

Figure 13-2 shows a visual representation of this process. The dotted line represents the inserted anonymous table row.



Figure 13-2. Anonymous-object generation in table formatting

Seven kinds of anonymous-object insertions can occur in the CSS table model. These seven rules are, like inheritance and specificity, an example of a mechanism that attempts to impose intuitive sense on the way CSS behaves.

The rules are as follows:

1. If a `table-cell` element's parent is not a `table-row` element, an anonymous `table-row` object is inserted between the `table-cell` element and its parent. The inserted object will include all consecutive siblings of the `table-cell` element.

   The same holds true even if the parent element is a `table-row-group`. For example, assume that the following CSS applies to the XML after it:

   ```
   system {display: table;}
   planet {display: table-row-group;}
   name, moons {display: table-cell;}

   <system>
       <planet>
           <name>Mercury</name>
           <moons>0</moons>
       </planet>
       <planet>
           <name>Venus</name>
           <moons>0</moons>
       </planet>
   </system>
   ```

   Both sets of cells will be enclosed in an anonymous `table-row` object that is inserted between them and the `<planet>` elements.

2. If a `table-row` element's parent is not a `table`, `inline-table`, or `table-row-group` element, then an anonymous `table` element is inserted between the `table-row`

element and its parent. The inserted object will include all consecutive siblings of the `table-row` element. Consider the following styles and markup:

```
docbody {display: block;}
planet {display: table-row;}

<docbody>
    <planet>
        <name>Mercury</name>
        <moons>0</moons>
    </planet>
    <planet>
        <name>Venus</name>
        <moons>0</moons>
    </planet>
</docbody>
```

Because the `display` value of the `<planet>` elements' parent is `block`, the anonymous `table` object is inserted between the `<planet>` elements and the `<docbody>` element. This anonymous `table` object will enclose both `<planet>` elements, since they are consecutive siblings.

3. If a `table-column` element's parent is not a `table`, `inline-table`, or `table-column-group` element, then an anonymous `table` element is inserted between the `table-column` element and its parent. This is much the same as the `table-row` rule just discussed, except for its column-oriented nature.
4. If the parent element of a `table-row-group`, `table-header-group`, `table-footer-group`, `table-column-group`, or `table-caption` element is not a `table` element, then an anonymous `table` object is inserted between the element and its parent.
5. If a child element of a `table` or `inline-table` element is not a `table-row-group`, `table-header-group`, `table-footer-group`, `table-row`, or `table-caption` element, then an anonymous `table-row` object is inserted between the `table` element and its child element. This anonymous object spans all the consecutive siblings of the child element that are not `table-row-group`, `table-header-group`, `table-footer-group`, `table-row`, or `table-caption` elements. Consider the following markup and styles:

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}

<system>
    <planet>
        <name>Mercury</name>
        <moons>0</moons>
    </planet>
    <name>Venus</name>
    <moons>0</moons>
</system>
```

Here, a single anonymous `table-row` object will be inserted between the `<system>` element and the second set of `<name>` and `<moons>` elements. The `<planet>` element is not enclosed by the anonymous object because its `display` is `table-row`.

6. If a child element of a `table-row-group`, `table-header-group`, or `table-footer-group` element is not a `table-row` element, an anonymous `table-row` object is inserted between the element and its child element. This anonymous object spans all the consecutive siblings of the child element that are not `table-row` objects themselves. Consider the following markup and styles:

```
system {display: table;}
planet {display: table-row-group;}
name, moons {display: table-cell;}

<system>
    <planet>
        <name>Mercury</name>
        <moons>0</moons>
    </planet>
    <name>Venus</name>
    <moons>0</moons>
</system>
```

   In this case, each set of `<name>` and `<moons>` elements will be enclosed in an anonymous `table-row` element. For the second set, the insertion happens in accord with rule 5. For the first set, the anonymous object is inserted between the `<planet>` element and its children because the `<planet>` element is a `table-row-group` element.

7. If a child element of a `table-row` element is not a `table-cell` element, then an anonymous `table-cell` object is inserted between the element and its child element. This anonymous object encloses all consecutive siblings of the child element that are not `table-cell` elements themselves. Consider the following markup and styles:

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}

<system>
    <planet>
        <name>Mercury</name>
        <num>0</num>
    </planet>
</system>
```

   Because the element `<num>` does not have a table-related `display` value, an anonymous `table-cell` object is inserted between the `<planet>` element and the `<num>` element.

   This behavior also extends to the encapsulation of anonymous inline boxes. Suppose that the `<num>` element is not included:

```
<system>
    <planet>
        <name>Mercury</name>
        0
    </planet>
</system>
```

The `0` would still be enclosed in an anonymous `table-cell` object. To further illustrate this point, here is an example adapted from the CSS specification:

```
example {display: table-cell;}
row {display: table-row;}
hey {font-weight: 900;}

<example>
    <row>This is the <hey>top</hey> row.</row>
    <row>This is the <hey>bottom</hey> row.</row>
</example>
```

Within each `<row>` element, the text fragments and `hey` element are enclosed in anonymous `table-cell` objects.

Working with Table Layers

For the assembly of a table's presentation, CSS defines six individual *layers* on which the various aspects of a table are placed. Figure 13-3 shows these layers.

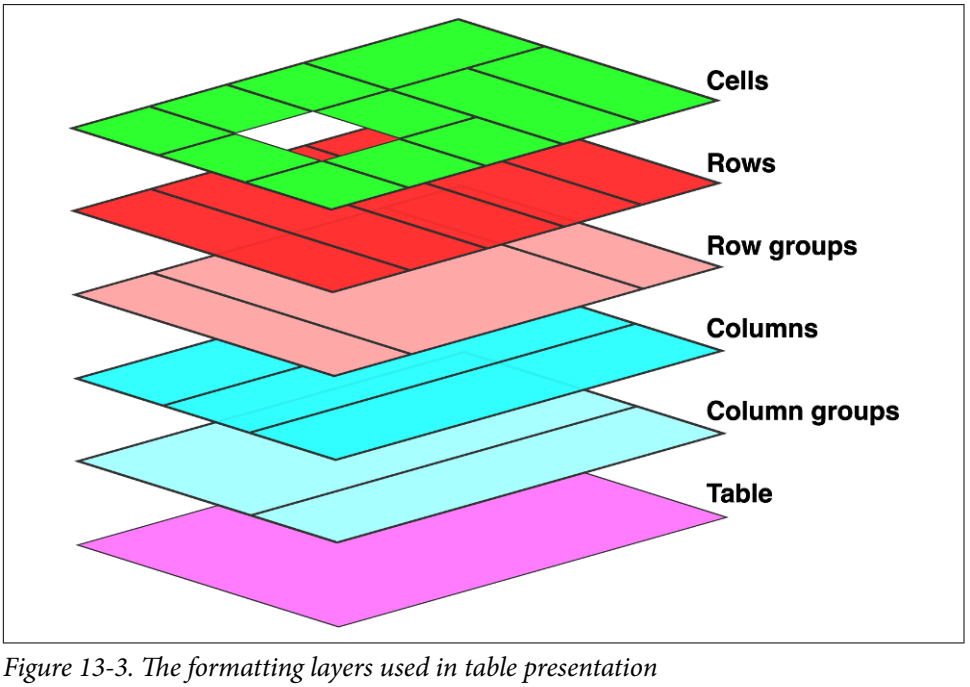


Figure 13-3. The formatting layers used in table presentation

Basically, the styles for each aspect of the table are drawn on their individual layers. Thus, if the `<table>` element has a green background and a 1-pixel black border, those styles are drawn on the lowest layer. Any styles for the column groups are drawn on the next layer up, the columns themselves on the layer above that, and so on. The top layer, which corresponds to the table cells, is drawn last.

For the most part, this is a logical process; after all, if you declare a background color for table cells, you would want that drawn over the background for the table element. The most important point revealed by Figure 13-3 is that column styles come below row styles, so a row's background will overwrite a column's background.

It is important to remember that by default, all elements have transparent backgrounds. Thus, in the following markup, the table element's background will be visible "through" cells, rows, columns, and so forth that do not have a background of their own, as illustrated in Figure 13-4:

```
<table style="background: #B84;">
    <tr>
        <td>hey</td>
        <td style="background: #ABC;">there</td>
    </tr>
    <tr>
        <td>what's</td>
        <td>up?</td>
    </tr>
    <tr style="background: #CBA;">
        <td>not</td>
        <td style="background: #ECC;">much</td>
    </tr>
</table>
```

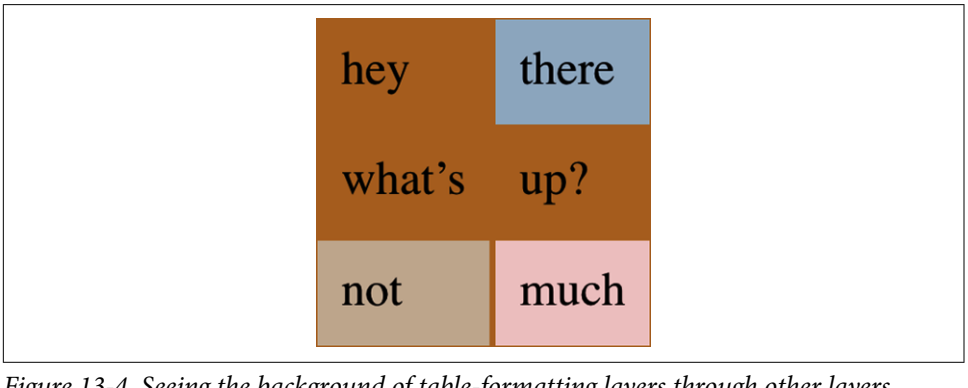


Figure 13-4. Seeing the background of table-formatting layers through other layers

Using Captions

A *table caption* is about what you'd expect: a short bit of text that describes the nature of the table's contents. A chart of stock quotes for the fourth quarter of 2026, therefore, might have a caption element whose contents read "Q4 2026 Stock Performance." With the property `caption-side`, you can place this element either above or below the table, regardless of where the caption appears in the table's structure. (In HTML5, the `<caption>` element can appear only as the first child of a `<table>` element, but other languages may have different rules.)

caption-side

Values	`top` | `bottom`
Initial value	`top`
Applies to	Elements with the `display` value `table-caption`
Computed value	As specified
Inherited	Yes
Animatable	No
Note	The values `left` and `right` appeared in CSS2, but were dropped from CSS2.1 because of a lack of widespread support

Captions are a bit odd, at least in visual terms. The CSS specification states that a caption is formatted as if it were a block box placed immediately before (or after) the table's box, with one exception: the caption can still inherit values from the table.

A simple example should suffice to illustrate most of the important aspects of caption presentation. Consider the following, illustrated in Figure 13-5:

```
table {color: white; background: #840; margin: 0.5em 0;}
caption {background: #B84; margin: 1em 0;}
table.one caption {caption-side: top;}
table.two caption {caption-side: bottom;}
td {padding: 0.5em;}
```

The text in each `<caption>` element inherits the `color` value `white` from the table, while the caption gets its own background. The separation between each table's outer border edge and the caption's outer margin edge is 1 em, as the margins of the table and the caption have collapsed. Finally, the width of the caption is based on the content width of the `<table>` element, which is considered to be the containing block of the caption.

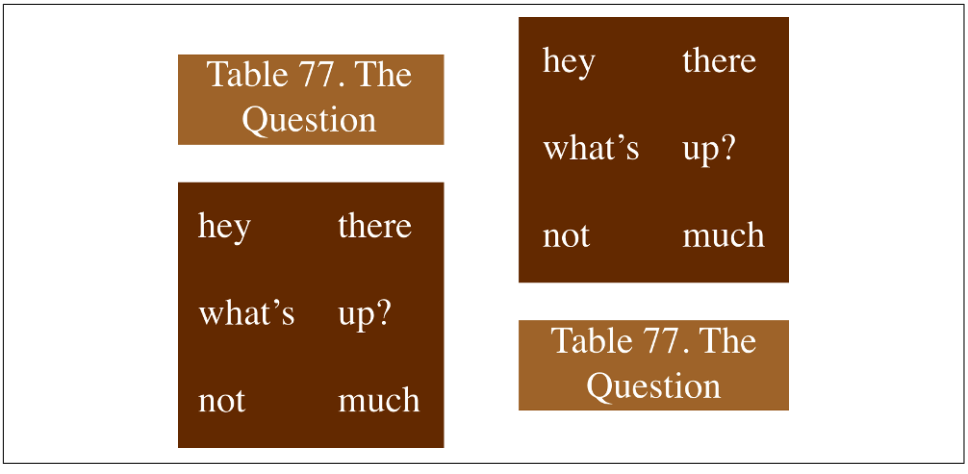


Figure 13-5. Styling captions and tables

For the most part, captions are styled just like any block-level element: they can be padded, have borders, be given backgrounds, and so on. For example, if we need to change the horizontal alignment of text within the caption, we use the property `text-align`. Thus, to right-align the caption in the previous example, we would write this:

```
caption {background: gray; margin: 1em 0;
    caption-side: top; text-align: right;}
```

Table Cell Borders

CSS has two quite distinct table-border models. The *separated border model* takes effect when cells are separated from each other in layout terms. The *collapsed border model* has no visual separation between cells, and cell borders merge or collapse into one another. The former is the default model, but you can choose between the two models with the property `border-collapse`.

border-collapse

Values	`collapse` | `separate` | `inherit`
Initial value	`separate`
Applies to	Elements with the `display` value `table` or `table-inline`
Inherited	Yes
Computed value	As specified
Note	In CSS2, the default was `collapse`

The whole point of this property is to offer a way to determine which border model the user agent will employ. If the value `collapse` is in effect, the collapsed border model is used. If the value is `separate`, the separated border model is used. We'll look at the latter model first, since it's much easier to describe and is the default.

Separated Cell Borders

In the separated border model, every cell in the table is separated from the other cells by some distance, and the borders of cells do not collapse into one another. Thus, given the following styles and markup, you should get the result shown in Figure 13-6:

```
td {border: 3px double black; padding: 3px;}
tr:nth-child(2) td:nth-child(2) {border-color: gray;}

<table cellspacing="0">
    <tr>
        <td>cell one</td>
        <td>cell two</td>
    </tr>
    <tr>
        <td>cell three</td>
        <td>cell four</td>
    </tr>
</table>
```

Note that the cell borders touch but remain distinct from one another. The three lines between cells are actually the two double borders sitting right next to each other; the gray border around the fourth cell helps make this more clear.

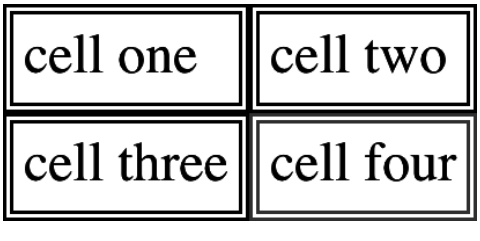


Figure 13-6. Separated (and thus separate) cell borders

The HTML attribute `cellspacing` is included in the preceding example to make sure the cells have no separation between them, but its presence is likely a bit troubling. After all, if you can define borders as separate, there ought to be a way to use CSS to alter the spacing between cells. Fortunately, there is.

Applying border spacing

Once you've separated the table cell borders, you might want those borders to be separated by a certain distance. This can be easily accomplished with the property `border-spacing`, which provides a more powerful replacement for the HTML attribute `cellspacing`.

border-spacing

Values	<*length*> <*length*>?
Initial value	`0`
Applies to	Elements with the display value `table` or `table-inline`
Computed value	Two absolute lengths
Inherited	Yes
Animatable	Yes
Note	Property is ignored unless `border-collapse` value is `separate`

Either one or two lengths can be given for the value of this property. If you want all your cells separated by a single pixel, `border-spacing: 1px;` will suffice. If, on the other hand, you want cells to be separated by 1 pixel horizontally and 5 pixels vertically, write `border-spacing: 1px 5px;`. If two lengths are supplied, the first is always the horizontal separation, and the second is always the vertical.

The spacing values are also applied between the borders of cells along the outside of a table and the padding on the `table` element itself. Given the following styles, you would get a result like that shown in Figure 13-7:

```
table {border-collapse: separate; border-spacing: 5px 8px;
padding: 12px; border: 2px solid black;}
td { border: 1px solid gray;}
td#squeeze {border-width: 5px;}
```

Figure 13-7 displays a space 5 pixels wide between the borders of any two horizontally adjacent cells, and 17 pixels of space between the borders of the right- and leftmost cells and the right and left borders of the `<table>` element. Similarly, the borders of vertically adjacent cells are 8 pixels apart, and the borders of the cells in the top and bottom rows are 20 pixels from the top and bottom borders of the table, respectively. The separation between cell borders is constant throughout the table, regardless of the border widths of the cells themselves.

Note also that declaring a `border-spacing` value is done on the table itself, not on the individual cells. If `border-spacing` had been declared for the `<td>` elements in the previous example, it would have been ignored.

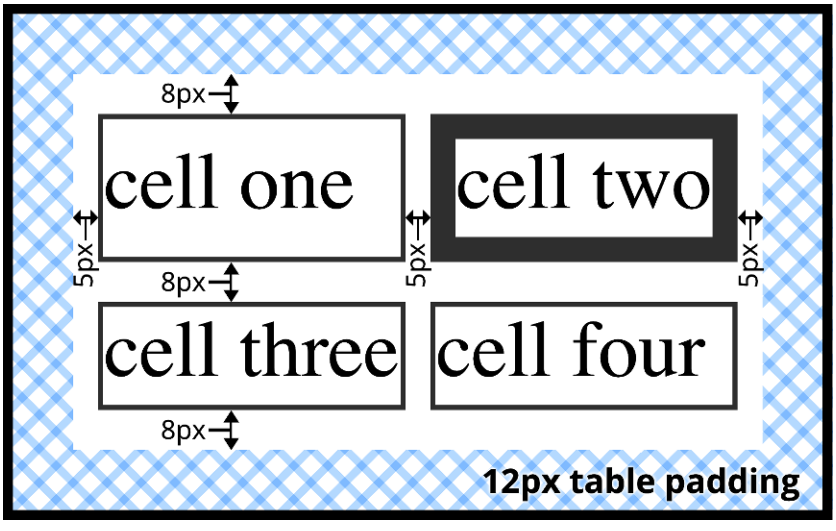


Figure 13-7. Border spacing effects between cells and their enclosing table

In the separated border model, borders cannot be set for rows, row groups, columns, and column groups. Any border properties declared for such elements must be ignored by a CSS-conformant user agent.

Handling empty cells

Because every cell is, in a visual sense, distinct from all the other cells in the table, what do you do with cells that are empty (i.e., have no content)? You have two choices, which are reflected in the values of the `empty-cells` property.

empty-cells

Values	`show` | `hide`
Initial value	`show`
Applies to	Elements with the `display` value `table-cell`
Computed value	As specified
Inherited	Yes
Animatable	No
Note	Property is ignored unless `border-collapse` value is `separate`

If `empty-cells` is set to `show`, the borders and background of an empty cell will be drawn, just as with table cells that have content. If the value is `hide`, no part of the cell is drawn, as if the cell were set to `visibility: hidden`.

If a cell contains any content, it cannot be considered empty. *Content*, in this case, includes not only text, images, form elements, and so on, but also the nonbreaking space entity (` `) and any other whitespace *except* the carriage return (CR), line feed (LF), tab, and space characters. If all the cells in a row are empty, and all have an `empty-cells` value of `hide`, the entire row is treated as if the row element were set to `display: none`.

Collapsed Cell Borders

While the collapsed border model largely describes how HTML tables have always been laid out when they don't have any cell spacing, it is quite a bit more complicated than the separated borders model. The following rules set collapsing cell borders apart from the separated borders model:

- Elements with a `display` of `table` or `inline-table` cannot have any padding when `border-collapse` is `collapse`, although they can have margins. Thus, separation never occurs between the border around the outside of the table and the edges of its outermost cells in the collapsed border model.
- Borders can be applied to cells, rows, row groups, columns, and column groups. A table itself can, as always, have a border.
- Separation never exists between cell borders in the collapsed border model. In fact, borders collapse into each other where they adjoin, so that only one of the collapsing borders is actually drawn. This is somewhat akin to margin collapsing, where the largest margin wins. When cell borders collapse, the "most interesting" border wins.
- Once they are collapsed, the borders between cells are centered on the hypothetical grid lines between the cells.

We'll explore the last two points in more detail in the next two sections.

Collapsing border layout

To better understand how the collapsed border model works, let's look at the layout of a single table row, as shown in Figure 13-8.

The padding and content width of each cell is inside the borders, as expected. For the borders between cells, half of the border is to one side of the grid line between two cells, and the other half is to the other side. In each case, only a single border is drawn along each cell edge. You might think that half of each cell's border is drawn to each side of the grid line, but that's not what happens.

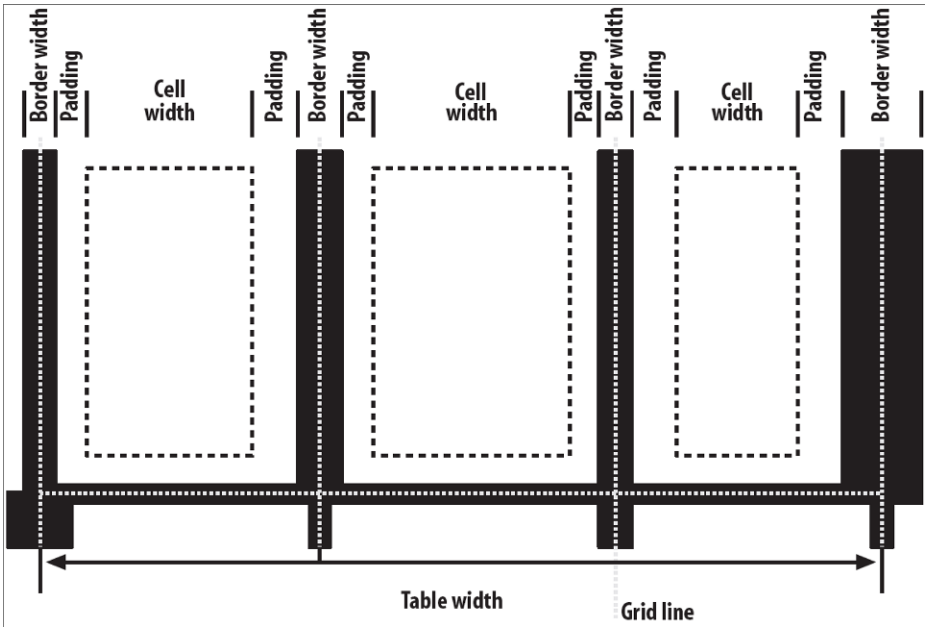


Figure 13-8. The layout of a table row using the collapsing borders model

For example, assume that the solid borders on the middle cell are green and the solid borders on the outer two cells are red. The borders on the right and left sides of the middle cell (which collapse with the adjacent borders of the outer cells) will be all green, or all red, depending on which border wins out. We'll discuss how to tell which one wins in the next section.

You may have noticed that the outer borders protrude past the table's width. This is because in this model, *half* the table's borders are included in the width. The other half stick out beyond that distance, sitting in the margin itself. This might seem a bit weird, but that's how the model is defined to work.

The specification includes a layout formula reproduced here for the benefit of those who enjoy such things:

> row width = (0.5 × border-width-0) + padding-left-1 + width-1 + padding-right-1 + border-width-1 + padding-left-2 +...+ padding-right-*n* + (0.5 × border-width-*n*)

Each `border-width-`*n* refers to the border between cell *n* and the next cell; thus, `border-width-3` refers to the border between the third and fourth cells. The value *n* stands for the total number of cells in the row.

This mechanism has a slight exception. When beginning the layout of a collapsed-border table, the user agent computes an initial left and right border for the table itself. It does

this by examining the left border of the first cell in the first row of the table and by taking half of that border's width as the table's initial left border width. The user agent then examines the right border of the last cell in the first row and uses half that width to set the table's initial right border width. For any row after the first, if the left or right border is wider than the initial border widths, it sticks out into the margin area of the table.

If a border is an odd number of display elements (pixels, printer dots, etc.) wide, the user agent is left to decide what to do about centering the border on the grid line. The user agent might shift the border so that it is slightly off-center, round up or down to an even number of display elements, use anti-aliasing, or adjust anything else that seems reasonable.

Border collapsing

When two or more borders are adjacent, they collapse into each other. In fact, they don't collapse so much as fight it out to see which will gain supremacy over the others. Strict rules govern which borders will win and which will not:

- If one of the collapsing borders has a `border-style` of `hidden`, it takes precedence over all other collapsing borders. All borders at this location are hidden.
- If all the borders are visible, wider borders take precedence over narrower ones. Thus, if a 2-pixel dotted border and a 5-pixel double border collapse, the border at that location will be a 5-pixel double border.
- If all collapsing borders have the same width but different border styles, the border style is taken in the following order, from most to least preferred: `double`, `solid`, `dashed`, `dotted`, `ridge`, `outset`, `groove`, `inset`, `none`. Thus, if two borders with the same width are collapsing, and one is `dashed` while the other is `outset`, the border at that location will be dashed.
- If collapsing borders have the same style and width, but differ in color, the color used is taken from an element in the following list, from most preferred to least: cell, row, row group, column, column group, table. Thus, if the borders of a cell and a column (identical in every way except color) collapse, the cell's border color (and style and width) will be used. If the collapsing borders come from the same type of element, such as two row borders with the same style and width but different colors, the color is taken from borders that are closer to the block-start and inline-start edges of the element.

The following styles and markup, presented in Figure 13-9, help illustrate each of the four rules:

```
table {border-collapse: collapse;
border: 3px outset gray;}
td {border: 1px solid gray; padding: 0.5em;}
#r2c1, #r2c2 {border-style: hidden;}
#r1c1, #r1c4 {border-width: 5px;}
#r2c4 {border-style: double; border-width: 3px;}
```

```
#r3c4 {border-style: dotted; border-width: 2px;}
#r4c1 {border-bottom-style: hidden;}
#r4c3 {border-top: 13px solid silver;}

<table>
    <tr>
        <td id="r1c1">1-1</td>
        <td id="r1c2">1-2</td>
        <td id="r1c3">1-3</td>
        <td id="r1c4">1-4</td>
    </tr>
    <tr>
        <td id="r2c1">2-1</td>
        <td id="r2c2">2-2</td>
        <td id="r2c3">2-3</td>
        <td id="r2c4">2-4</td>
    </tr>
    <tr>
        <td id="r3c1">3-1</td>
        <td id="r3c2">3-2</td>
        <td id="r3c3">3-3</td>
        <td id="r3c4">3-4</td>
    </tr>
    <tr>
        <td id="r4c1">4-1</td>
        <td id="r4c2">4-2</td>
        <td id="r4c3">4-3</td>
        <td id="r4c4">4-4</td>
    </tr>
</table>
```

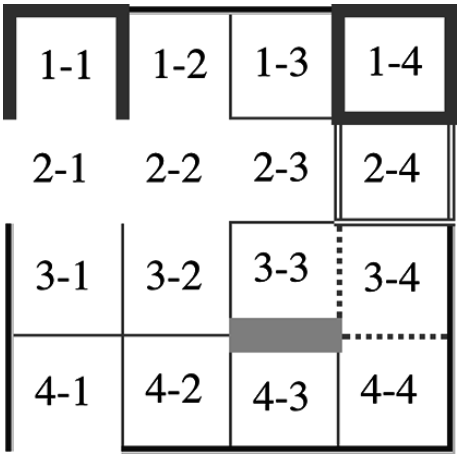


Figure 13-9. Manipulating border widths, styles, and colors leads to some unusual results

Let's consider what happens for each of the cells, in turn:

- For cells 1-1 and 1-4, the 5-pixel borders are wider than any of their adjacent borders, so they win out not only over adjoining cell borders, but over the border of the table itself. The only exception is the bottom of cell 1-1, which is suppressed.
- The bottom border on cell 1-1 is suppressed because cells 2-1 and 2-2, with their explicitly hidden borders, completely remove any borders from the edge of the cells. Again, the table's border loses out (on the left edge of cell 2-1) to a cell's border. The bottom border of cell 4-1 is also hidden, and so it prevents any border from appearing below the cell.
- The 3-pixel double border of cell 2-4 is overridden on top by the 5-pixel solid border of cell 1-4. Cell 2-4's border, in turn, overrides the border between itself and cell 2-3 because it is both wider and "more interesting." Cell 2-4 also overrides the border between itself and cell 3-4, even though both are the same width, because 2-4's double style is defined to be "more interesting" than 3-4's dotted border.
- The 13-pixel bottom silver border of cell 3-3 not only overrides the top border of cell 4-3, but it also affects the layout of content within both cells *and* the rows that contain both cells.
- For cells along the outer edge of the table that aren't specially styled, their 1-pixel solid borders are overridden by the 3-pixel outset border on the table element itself.

This is, in fact, about as complicated as it sounds, although the behaviors are largely intuitive and make a little more sense with practice. It's worth noting that the basic Netscape 1.1-era table presentation can be captured with a fairly simple set of rules:

```
table {border-collapse: collapse; border: 2px outset gray;}
td {border: 1px inset gray;}
```

Yes, tables were made to look 3D-ish by default when they debuted. It was a different time.

Table Sizing

Now that we've dug into the guts of table formatting and cell border appearance, you have the pieces you need to understand the sizing of tables and their internal elements. When it comes to determining table width, CSS has two approaches: *fixed-width layout* and *automatic-width layout*. Table heights are calculated automatically, no matter what width algorithms are used.

Width

Since there are two ways to figure out the width of a table, it's only logical that there is a way to declare which should be used for a given table. You can use the property `table-layout` to select between the two kinds of table width calculations.

table-layout

Values	`auto` | `fixed`
Initial value	`auto`
Applies to	Elements with the `display` value `table` or `inline-table`
Computed value	As specified
Inherited	Yes
Animatable	No

While the two models can have different results in laying out a given table, the fundamental difference between the two is that of speed. With a fixed-width table layout, the user agent can calculate the layout of the table more quickly than is possible in the automatic-width model.

Fixed layout

The main reason the fixed-layout model is so fast is that its layout does not fully depend on the contents of table cells. Instead, it's driven by the width values of the table, its column elements, and the cells of the first row within that table.

The fixed-layout model works in the following steps:

- Any column element whose `width` property has a value other than `auto` sets the width for that entire column.
  - If a column has an `auto` width, but the cell in the first row of the table within that column has a `width` other than `auto`, the cell sets the width for that entire column. If the cell spans multiple columns, the width is divided between the columns.
  - Any columns that are still auto-sized are sized so that their widths are as equal as possible.

At that point, the width of the table is set to be either the value of `width` for the table or the sum of the column widths, whichever is *greater*. If the table turns out to be wider than its columns, the difference is divided by the number of columns and the result is added to each of them.

This approach is fast because all of the column widths are defined by the first row of the table. The cells in any rows that come after the first are sized according to the column widths that were defined by the first row. The cells in those following rows do not—indeed, cannot—change column widths, which means that any `width` value assigned to those cells will be ignored. If a cell's content does not fit into its cell, the `overflow` value for the cell determines whether the cell contents are clipped, visible, or generate a scrollbar.

Let's consider the following styles and markup, which are illustrated in Figure 13-10:

```
table {table-layout: fixed; width: 400px;
    border-collapse: collapse;}
td {border: 1px solid;}
col#c1 {width: 200px;}
#r1c2 {width: 75px;}
#r2c3 {width: 500px;}

<table>
    <colgroup> <col id="c1"><col id="c2"><col id="c3"><col id="c4"> </colgroup>
    <tr>
        <td id="r1c1">1-1</td>
        <td id="r1c2">1-2</td>
        <td id="r1c3">1-3</td>
        <td id="r1c4">1-4</td>
    </tr>
    <tr>
          <td id="r2c1">2-1</td>
          <td id="r2c2">2-2</td>
          <td id="r2c3">2-3</td>
          <td id="r2c4">2-4</td>
    </tr>
        (…more rows here…)
</table>
```

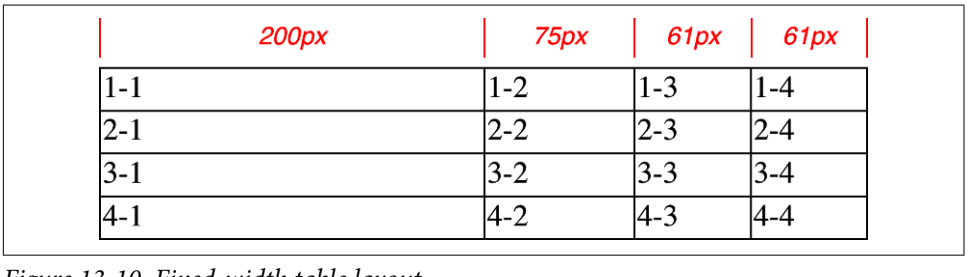


Figure 13-10. Fixed-width table layout

The first column is 200 pixels wide, which happens to be half the 400-pixel width of the table. The second column is 75 pixels wide, because the first-row cell within that column has been assigned an explicit width. The third and fourth columns are each 61 pixels wide. Why? Because the sum of the column widths for the first and second columns (275 pixels), plus the various borders between columns (3 pixels), equals 278 pixels. Then, 400 minus 278 is 122, and that divided in half is 61, so that's how many pixels wide the third and fourth columns will be. What about the 500-pixel width for `#r2c3`? It's ignored because that cell isn't in the first row of the table.

Note that the table doesn't need to have an explicit `width` value to use the fixed-width layout model, although it definitely helps. For example, given the following, a user agent could calculate a width for the table that is 50 pixels narrower than the parent element's width. It would then use that calculated width in the fixed-layout algorithm:

```
table {table-layout: fixed; margin: 0 25px; width: auto;}
```

This is not required, however. User agents are also permitted to lay out any table with an `auto` value for `width` by using the automatic-width layout model.

Automatic layout

The automatic-width layout model, while not as fast as fixed layout, is probably much more familiar to you because it's substantially the same model that HTML tables have used since their inception. In most current browsers, use of this model will be triggered by a table having a `width` of `auto`, regardless of the value of `table-layout`, although this is not assured.

The reason automatic layout is slower is that the table cannot be laid out until the user agent has looked at all of the content in the table. The user agent must lay out the entire table in a fashion that takes the contents and styles of every cell into account. This generally requires the user agent to perform some calculations and then go back through the table to perform a second set of calculations (if not more).

The content has to be fully examined because, as with HTML tables, the table's layout is dependent on the content in all the cells. If a 400-pixel-wide image is in a cell in the last row, that content will force all of the cells above it (those in the same column) to be at least 400 pixels wide. Thus, the width of every cell has to be calculated, and adjustments must be made (possibly triggering another round of content-width calculations) before the table can be laid out.

The details of the model can be expressed in the following steps:

1. For each cell in a column, calculate both the minimum and maximum cell width.
    a. Determine the minimum width required to display the content. In determining this minimum content width, the content can flow to any number of lines, but it may not stick out of the cell's box. If the cell has a `width` value that is larger than the minimum possible width, the minimum cell width is set to the value of `width`. If the cell's `width` value is `auto`, the minimum cell width is set to the minimum content width.
    b. For the maximum width, determine the width required to display the content without any line breaking other than that forced by explicit line breaking (e.g., the `<br>` element). That value is the maximum cell width.
2. For each column, calculate both the minimum and maximum column width.
    a. The column's minimum width is determined by the largest minimum cell width of the cells within the column. If the column has been given an explicit `width` value that is larger than any of the minimum cell widths within the column, the minimum column width is set to the value of `width`.
    b. For the maximum width, take the largest maximum cell width of the cells within the column. If the column has been given an explicit `width` value that is larger

than any of the maximum cell widths within the column, the maximum column width is set to the value of `width`. These two behaviors re-create the traditional HTML table behavior of forcibly expanding any column to be as wide as its widest cell.

3. If a cell spans more than one column, the sum of the minimum column widths must be equal to the minimum cell width for the spanning cell. Similarly, the sum of the maximum column widths has to equal the spanning cell's maximum width. User agents should divide any changes in column widths equally among the spanned columns.

In addition, the user agent must take into account that when a column has a percentage value for its width, the percentage is calculated in relation to the width of the table—even though the user agent doesn't yet know what that will be! It instead has to hang on to the percentage value and use it in the next part of the algorithm.

At this point, the user agent will have figured how wide or narrow each column *can* be. With that information in hand, it can then proceed to actually figuring out the width of the table. This happens as follows:

1. If the computed width of the table is not `auto`, the computed table width is compared to the sum of all the column widths *plus* any borders and cell spacing. (Columns with percentage widths are likely calculated at this time.) The larger of the two is the final width of the table. If the table's computed width is *larger* than the sum of the column widths, borders, and cell spacing, then the difference is divided by the number of columns and the result is added to each of them.
2. If the computed width of the table is `auto`, the final width of the table is determined by adding up the column widths, borders, and cell spacing. This means that the table will be only as wide as needed to display its content, just as with traditional HTML tables. Any columns with percentage widths use that percentage as a constraint—but one that a user agent does not have to satisfy.

Once the last step is completed, then—and only then—can the user agent actually lay out the table.

The following styles and markup, presented in Figure 13-11, help illustrate how this process works:

```
table {table-layout: auto; width: auto;
    border-collapse: collapse;}
td {border: 1px solid; padding: 0;}
col#c3 {width: 25%;}
#r1c2 {width: 40%;}
#r2c2 {width: 50px;}
#r2c3 {width: 35px;}
#r4c1 {width: 100px;}
#r4c4 {width: 1px;}
```

```
<table>
    <colgroup> <col id="c1"><col id="c2"><col id="c3"><col id="c4"> </colgroup>
    <tr>
        <td id="r1c1">1-1</td>
        <td id="r1c2">1-2</td>
        <td id="r1c3">1-3</td>
        <td id="r1c4">1-4</td>
    </tr>
    <tr>
        <td id="r2c1">2-1</td>
        <td id="r2c2">2-2</td>
        <td id="r2c3">2-3</td>
        <td id="r2c4">2-4</td>
    </tr>
    <tr>
        <td id="r3c1">3-1</td>
        <td id="r3c2">3-2</td>
        <td id="r3c3">3-3</td>
        <td id="r3c4">3-4</td>
    </tr>
    <tr>
        <td id="r4c1">4-1</td>
        <td id="r4c2">4-2</td>
        <td id="r4c3">4-3</td>
        <td id="r4c4">4-4</td>
    </tr>
</table>
```

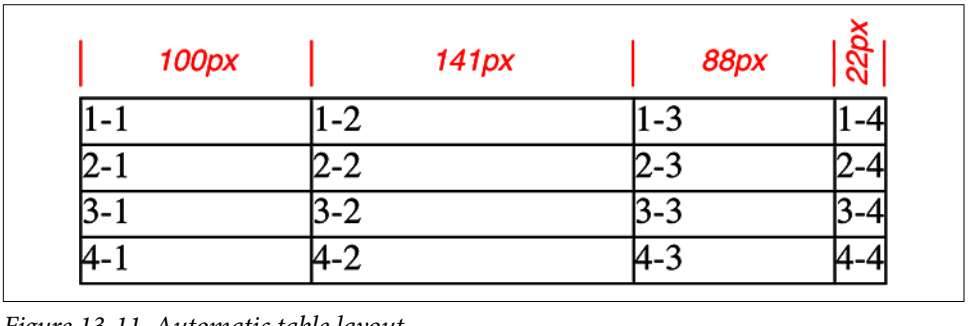


Figure 13-11. Automatic table layout

Let's consider what happens for each of the columns, in turn:

- For the first column, the only explicit cell or column width is that of cell 4-1, which is given a width of `100px`. Because the content is so short, both the minimum and maximum column widths are set to `100px`. (If a cell in the column had several sentences of text, it would have increased the maximum column width to whatever width necessary to display all of the text without line breaking.)

- For the second column, two `width`s are declared: cell 1-2 is given a width of `40%`, and cell 2-2 is given a width of `50px`. The minimum width of this column is `50px`, and the maximum width is `40%` of the final table width.
- For the third column, only cell 3-3 has an explicit width (`35px`), but the column itself is given a `width` of `25%`. Therefore, the minimum column width is 35 pixels, and the maximum width is 25% of the final table width.
- For the fourth column, only cell 4-4 is given an explicit width (`1px`). This is smaller than the minimum content width, so both the minimum and maximum column widths are equal to the minimum content width of the cells. This turns out to be a computed 22 pixels, so the minimum and maximum widths are both 22 pixels.

The user agent now knows that the four columns have minimum and maximum widths as follows, in order:

1. Minimum 100 pixels, maximum 100 pixels
2. Minimum 50 pixels, maximum 40%
3. Minimum 35 pixels, maximum 25%
4. Minimum 22 pixels, maximum 22 pixels

The table's minimum width is the sum of all the column minimums, plus the borders collapsed between the columns, which totals 215 pixels. The table's maximum width is `123px + 65%`, where the `123px` comes from the first and last columns and their shares of the collapsed borders. This maximum works out to be 351.42857142857143 pixels (given that `123px` represents 35% of the overall table width). With this number in hand, the second column will be 140.5 pixels wide, and the third column will be 87.8 pixels wide. These may be rounded by the user agent to whole numbers such as `141px` and `88px`, or not, depending on the exact rendering method used. (These are the numbers used in Figure 13-11.)

Note that user agents are not required to actually use the maximum value; they may choose another course of action.

This is (although it may not seem like it) a comparatively simple and straightforward example: all of the content is basically the same width, and most of the declared widths are pixel lengths. If a table contains images, paragraphs of text, form elements, and so forth, the process of figuring out the table's layout is likely to be a great deal more complicated.

Height

After all of the effort expended in figuring out the width of the table, you might well wonder how much more complicated height calculation will be. Actually, in CSS terms, it's pretty simple, although browser developers probably don't think so.

The easiest situation to describe is one in which the table height is explicitly set via the `height` property. In such cases, the height of the table is defined by the value of `height`. This means that a table may be taller or shorter than the sum of its row heights. Note that `height` is treated much more like `min-height` for tables, so if you define a `height` value that's smaller than the sum total of the row heights, it may appear to be ignored.

By contrast, if the `height` value of a table is greater than the total of its row heights, the specification explicitly refuses to define what should happen, instead noting that the issue may be resolved in future versions of CSS. A user agent could expand the table's rows to fill out its height, or leave blank space inside the table's box, or something completely different. It's up to each user agent to decide.



As of mid-2022, the most common behavior of user agents is to increase the heights of the rows in a table to fill out its overall height. This is accomplished by taking the difference between the table height and the sum of the row heights, dividing it by the number of rows, and applying the resulting amount to each row.

If the `height` of the table is `auto`, its height is the sum of the heights of all the rows within the table, plus any borders and cell spacing. To determine the height of each row, the user agent goes through a process similar to that used to find the widths of columns: it calculates a minimum and maximum height for the contents of each cell and then uses these to derive a minimum and maximum height for the row. After having done this for all the rows, the user agent figures out what each row's height should be, stacks them all on top of one another, and uses the total to determine the table's height.

In addition to what to do about tables with explicit heights and how to treat row heights within them, you can add the following to the list of things CSS does not define:

- The effect of a percentage height for table cells
- The effect of a percentage height for table rows and row groups
- How a row-spanning cell affects the heights of the rows that are spanned, except that the rows have to contain the spanning cell

As you can see, height calculations in tables are largely left up to user agents to figure out. Historical evidence would suggest that this will lead to each user agent doing something different, so you should probably avoid setting table heights as much as possible.

Alignment

In a rather interesting turn of events, alignment of content within cells is a lot better defined than cell and row heights. This is true even for vertical alignment, which can quite easily affect the height of a row.

Horizontal alignment is the simplest. To align content within a cell, you use the `text-align` property. In effect, the cell is treated as a block-level box, and all of the content within it is aligned as per the `text-align` value.

To vertically align content in a table cell, `vertical-align` is the relevant property. It uses many of the same values that are used for vertically aligning inline content, but the meanings of those values change when applied to a table cell. To summarize the three simplest cases:

`top`
: The top of the cell's content is aligned with the top of its row; in the case of row-spanning cells, the top of the cell's content is aligned with the top of the first row it spans.

`bottom`
: The bottom of the cell's content is aligned with the bottom of its row; in the case of row-spanning cells, the bottom of the cell's content is aligned with the bottom of the last row it spans.

`middle`
: The middle of the cell's content is aligned with the middle of its row; in the case of row-spanning cells, the middle of the cell's content is aligned with the middle of all the rows it spans.

These are illustrated in Figure 13-12, which uses the following styles and markup:

```
table {table-layout: auto; width: 20em;
border-collapse: separate; border-spacing: 3px;}
td {border: 1px solid; background: silver;
    padding: 0;}
div {border: 1px dashed gray; background: white;}
#r1c1 {vertical-align: top; height: 10em;}
#r1c2 {vertical-align: middle;}
#r1c3 {vertical-align: bottom;}
```

```
<table>
  <tr>
    <td id="r1c1">
      <div>The contents of this cell are top-aligned.</div>
    </td>
    <td id="r1c2">
      <div>The contents of this cell are middle-aligned.</div>
    </td>
    <td id="r1c3">
      <div>The contents of this cell are bottom-aligned.</div>
    </td>
  </tr>
</table>
```

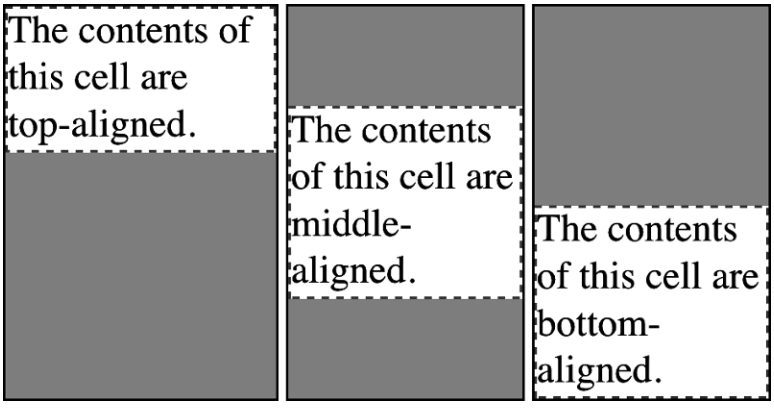


Figure 13-12. Vertical alignment of cell contents

In each case, the alignment is carried out by automatically increasing the padding of the cell itself to achieve the desired effect. In the first cell in Figure 13-12, the bottom padding of the cell has been changed to equal the difference between the height of the cell's box and the height of the content within the cell. For the second cell, the top and bottom padding of the cell have been reset to be equal, thus vertically centering the content of the cell. In the last cell, the cell's top padding has been altered.

The fourth possible alignment value is `baseline`, and it's a little more complicated than the first three:

`baseline`
: The baseline of the cell is aligned with the baseline of its row; in the case of row-spanning cells, the baseline of the cell is aligned with the baseline of the first row it spans.

It's easiest to provide an illustration (Figure 13-13) and then discuss what's happening.

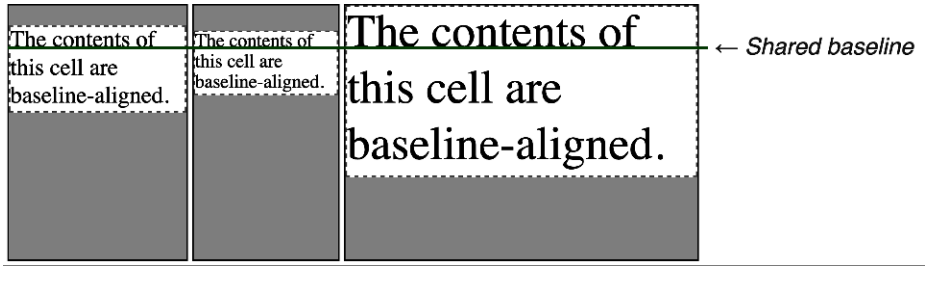


Figure 13-13. Baseline alignment of cell contents

A row's baseline is defined by the lowest initial cell baseline (that is, the baseline of the first line of text) out of all its cells. Thus, in Figure 13-13, the row's baseline is defined by

the third cell, which has the lowest initial baseline. The first two cells then have the baseline of their first line of text aligned with the row's baseline.

As with top, middle, and bottom alignment, the placement of baseline-aligned cell content is accomplished by altering the top and bottom padding of the cells. If none of the cells in a row are baseline-aligned, the row does not even have a baseline—it doesn't really need one.

The detailed process for aligning cell contents within a row is as follows:

- If any of the cells are baseline-aligned, the row's baseline is determined and the content of the baseline-aligned cells is placed.
  - Any top-aligned cell has its content placed. The row now has a provisional height, which is defined by the lowest cell bottom of the cells that have already had their content placed.
  - If any remaining cells are middle- or bottom-aligned, and the content height is taller than the provisional row height, the height of the row is increased to enclose the tallest of those cells.
  - All remaining cells have their content placed. In any cell whose contents are shorter than the row height, the cell's padding is increased in order to match the height of the row.

The `vertical-align` values `sub`, `super`, `text-top`, and `text-bottom` are supposed to be ignored when applied to table cells. Instead, they seem to be treated as if they are `baseline`, or possibly `top`.

Summary

Even if you're quite familiar with table layout from years of table-and-spacer design, it turns out that the mechanisms driving such layout are rather complicated. Thanks to the legacy of HTML table construction, the CSS table model is row-centric, but it does, thankfully, accommodate columns and limited column styling. Thanks to new abilities to affect cell alignment and table width, you now have even more tools for presenting tables in a pleasing way.

The ability to apply table-related display values to arbitrary elements opens the door to creating table-like layouts by using HTML elements such as `<div>` and `<section>`, or by using XML languages that allow you to use any element to describe table components.

CHAPTER 14
Fonts

The "Font Properties" section of the CSS1 specification, written in 1996, begins with this sentence: "Setting font properties will be among the most common uses of style sheets." Despite the awareness of font's importance from the very beginning of CSS, it wasn't until about 2009 that this capability really began to be widely and consistently supported. With the introduction of variable fonts, typography on the web has become an art form. While you can include any fonts you are legally allowed to distribute in your design, you have to pay attention to how you use them.

It's important to remember this does not grant absolute control over fonts. If the font you're using fails to download, or is in a file format the user's browser doesn't understand, the text will (eventually) be displayed with a fallback font. That's a good thing, as it means the user still gets your content.

While fonts may seem vital to a design, always bear in mind you can't depend on the presence of a given font. If a font is slow to load, browsers generally delay text rendering. While that prevents text being redrawn while a user is reading, it's bad to have no text on the page.

Your font choice may also be overridden by user preference, or a browser extension meant to enhance the reading experience. An example is the browser extension OpenDyslexic, which "overrides all fonts on web pages with the OpenDyslexic font, and formats pages to be more easily readable." In general, always design assuming your fonts will be delayed and even fail altogether.

Font Families

What we think of as a "font" is usually composed of many variations to describe bold text, italic text, bold italic text, and so on. For example, you're probably familiar with (or at least have heard of) the font Times. Times is actually a combination of many variants, including TimesRegular, TimesBold, TimesItalic, TimesBoldItalic, and so on. Each variant

of Times is an actual *font face*, and Times, as we usually think of it, is a combination of all these variant faces. In other words, system-standard fonts like Times are actually a *font family*, not just a single font, even though most of us think about fonts as being single entities.

With such font families, a separate file is required for each width, weight, and style combination (that is, each font face), meaning you can have upward of 20 separate files for a complete typeface. *Variable fonts*, on the other hand, are able to store multiple variants, such as regular, bold, italic, and bold italic, in a single file. Variable font files are generally a little bit larger (maybe just a few kilobytes) than any single font face file, but smaller than the multiple files required of a regular font, and require only a single HTTP request.

To cover all the bases, CSS defines five generic font families:

Serif fonts
: Serif fonts are proportional and have serifs. A font is *proportional* if all characters in the font have different widths. For example, a lowercase *i* and a lowercase *m* take up different horizontal spaces because they have different widths. (This book's paragraph font is proportional, for example.) *Serifs* are the decorations on the ends of strokes within each character, such as little lines at the top and bottom of a lowercase *l*, or at the bottom of each leg of an uppercase *A*. Examples of serif fonts are Times, Georgia, and New Century Schoolbook.

Sans-serif fonts
: Sans-serif fonts are proportional and do not have serifs. Examples of sans-serif fonts are Helvetica, Geneva, Verdana, Arial, and Univers.

Monospace fonts
: Monospace fonts are not proportional. Rather, each character uses up the same amount of horizontal space as all the others; thus, a lowercase *i* takes up the same horizontal space as a lowercase *m*, even though their actual letterforms may have different widths. These generally are used for displaying programmatic code or tabular data, like this book's code font, for example. If a font has uniform character widths, it is classified as monospace, regardless of whether it has serifs. Examples of monospace fonts are Courier, Courier New, Consolas, and Andale Mono.

Cursive fonts
: Cursive fonts attempt to emulate human handwriting or lettering. Usually, they are composed largely of flowing curves and have stroke decorations that exceed those found in serif fonts. For example, an uppercase *A* might have a small curl at the bottom of its left leg or be composed entirely of swashes and curls. Examples of cursive fonts are Zapf Chancery, Author, and Comic Sans.

Fantasy fonts
: Fantasy fonts are not really defined by any single characteristic other than our inability to easily classify them in one of the other families (these are sometimes called *decorative* or *display* fonts). A few such fonts are Western, Woodblock, and Klingon.

Your operating system and browser will have a default font family for each of these generic families. Fonts a browser cannot classify as serif, sans-serif, monospace, or cursive are generally considered fantasy. While most font families fall into one of these generic families, not all do. For example, SVG icon fonts, dingbat fonts, and Material Icons Round contain images rather than letters.

Using Generic Font Families

You can call on any available font family by using the property `font-family`.

font-family

Values	[*<family-name>* | *<generic-family>*]#
Initial value	User-agent specific
Applies to	All elements
Computed value	As specified
@font-face equivalent	`font-family`
Inherited	Yes
Animatable	No

If you want a document to use a sans-serif font but do not particularly care which one, the appropriate declaration would be as follows:

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

This will cause the user agent to pick a sans-serif font family (such as Helvetica) and apply it to the `<body>` element. Thanks to inheritance, the same font family choice will be applied to all visible elements that descend from the `<body>`, unless overridden by the user agent. User agents generally apply a `font-family` property to some elements, such as `mono space` in the case of `<code>` and `<pre>`, or a system font to some form-input controls.

Using nothing more than these generic families, you can create a fairly sophisticated style-sheet. The following rule set is illustrated in Figure 14-1:

```
body {font-family: serif;}
h1, h2, h3, h4 {font-family: sans-serif;}
code, pre, kbd {font-family: monospace;}
p.signature {font-family: cursive;}
```

Thus, most of this document will use a serif font such as Times, including all paragraphs except those that have a `class` of `signature`, which will instead be rendered in a cursive font such as Author. Heading levels 1 through 4 will use a sans-serif font like Helvetica, while the elements `<code>`, `<pre>`, `<tt>`, and `<kbd>` will use a monospace font like Courier.



Using generic defaults is excellent for rendering speed, as it allows the browser to use whichever default fonts it already has in memory rather than having to parse through a list of specific fonts and load characters as needed.

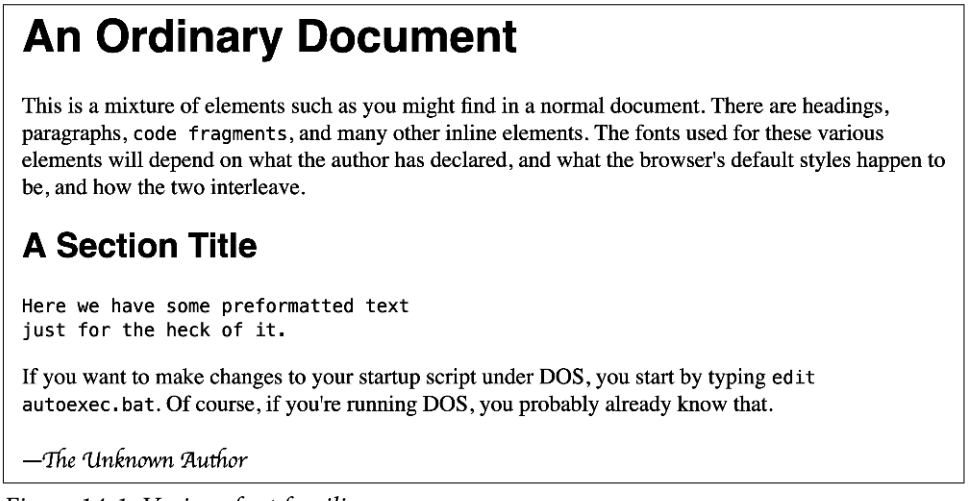


Figure 14-1. Various font families

A page author may, on the other hand, have more specific preferences for which font to use in the display of a document or element. In a similar vein, a user may want to create a user stylesheet that defines the exact fonts to be used in the display of all documents. In either case, `font-family` is still the property to use.

Assume for the moment that all `<h1>` elements should use Georgia as their font. The simplest rule for this would be the following:

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

This will cause the user agent displaying the document to use Georgia for all `<h1>` elements, assuming that the user agent has Georgia available for use. If it doesn't, the user agent will be unable to use the rule at all. It won't ignore the rule, but if it can't find a font called `Georgia`, it can't do anything but display `<h1>` elements using the user agent's default font.

To handle a situation like this, you can give the user agent options by combining specific font families with generic font families. For example, the following markup tells a user agent to use Georgia if it's available, and to use another serif font like Times as a fallback if it isn't:

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

For this reason, we strongly encourage you to always provide a generic family as part of any `font-family` rule. By doing so, you provide a fallback mechanism that lets user agents

pick an alternative when they can't provide an exact font match. This is often referred to as a *font stack*. Here are a few more examples:

```
h1 {font-family: Arial, sans-serif;}
h2 {font-family: Arvo, sans-serif;}
p {font-family: 'Times New Roman', serif;}
address {font-family: Chicago, sans-serif;}
.signature {font-family: Author, cursive;}
```

If you're familiar with fonts, you might have a number of similar fonts in mind for displaying a given element. Let's say that you want all paragraphs in a document to be displayed using Times, but you would also accept Times New Roman, Georgia, New Century Schoolbook, and New York (all of which are serif fonts) as alternate choices. First, decide the order of preference for these fonts, and then string them together with commas:

```
p {font-family: Times, 'Times New Roman', 'New Century Schoolbook', Georgia,
      'New York', serif;}
```

Based on this list, a user agent will look for the fonts in the order they're listed. If none of the listed fonts are available, it will just pick an available serif font.

Using Quotation Marks

You may have noticed the presence of single quotes in the previous code example, which we haven't used before in this chapter. Quotation marks are advisable in a `font-family` declaration only if a font name contains one or more spaces, such as `New York`, or if the font name includes symbols. Thus, a font called `Karrank%` should be quoted:

```
h2 {font-family: Wedgie, 'Karrank%', Klingon, fantasy;}
```

While quoting font names is almost never required, if you leave off the quotation marks, user agents may ignore the font name and continue to the next available font in the font stack. The exception to this is font names that match accepted `font-family` keywords. For example, if your font name is `cursive`, `serif`, `sans-serif`, `monospace`, or `fantasy`, it must be quoted so the user agent knows the difference between a font name and a font-family keyword, as shown here:

```
h2 {font-family: Author, "cursive", cursive;}
```

The actual generic family names (`serif`, `monospace`, etc.) should never be quoted. If they are quoted, the browser will look for a font with that exact name.

When quoting font names, either single or double quotes are acceptable, as long as they match. Remember that if you place a `font-family` rule in a `style` attribute, which you generally shouldn't, you'll need to use whichever quotes you didn't use for the attribute itself. Therefore, if you use double quotes to enclose the `font-family` rule, you'll have to use single quotes within the rule, as in the following markup:

```
p {font-family: sans-serif;}  /* sets paragraphs to sans-serif by default */

<!-- the next example is correct (uses single-quotes) -->
<p style="font-family: 'New Century Schoolbook', Times, serif;">...</p>
```

```
<!-- the next example is NOT correct (uses double-quotes) -->
<p style="font-family: "New Century Schoolbook", Times, serif;">...</p>
```

If you use double quotes in such a circumstance, they interfere with the attribute syntax. Note the font name is case-insensitive.

Using Custom Fonts

The `@font-face` rule enables you to use custom fonts on the web, instead of being forced to rely only on "web-safe" fonts (that is, font families that are widely installed, such as Times). The two required functions of the `@font-face` rule are to declare the name used to refer to a font and to provide the URL of that font's file for downloading. In addition to these required descriptors, CSS has 14 optional descriptors.

While there's no guarantee that every user will see the font you want, `@font-face` is supported in all browsers except ones like Opera Mini that intentionally don't support it for performance reasons.

Suppose you want to use a very specific font in your stylesheets, one that is not widely installed. Through the magic of `@font-face`, you can define a specific family name to correspond to a font file on your server that you can refer to throughout your CSS. The user agent will download that file and use it to render the text in your page, the same as if it were installed on the user's machine. For example:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf");
}
```

This allows you to tell user agents to load the defined *.otf* file and use that font to render text when called upon via `font-family: SwitzeraADF`.



The examples in this section refer to SwitzeraADF, a font face collection available from the Arkandis Digital Foundry.

The `@font-face` declaration doesn't automatically load all the referenced font files. The intent of `@font-face` is to allow *lazy loading* of font faces. This means only faces needed to render a document will be loaded. Font files referenced in your CSS that aren't necessary to render the page will not be downloaded. Font files are generally cached, and aren't redownloaded as your users navigate your site.

The ability to load any font is quite powerful, but keep these concerns in mind:

- For security reasons, font files must be retrieved from the same domain as the page requesting them. There's a solution for that.

- Requiring lots of font downloads can lead to slow load times.
- Fonts with lots of characters can lead to large font files. Fortunately, online tools and CSS enable limiting character sets.
- If fonts load slowly, this can lead to flashes of unstyled text or invisible text. CSS has a way of addressing this issue as well.

We'll cover these problems and their solutions in this chapter. But remember, with great power comes great responsibility. Use fonts wisely!

Using Font-Face Descriptors

All the parameters that define the font you're referencing are contained within the `@font-face { }` construct. These are called *descriptors*, and very much like properties, they take the format *`descriptor: value;`*. In fact, most of the descriptor names refer directly to property names, as will be examined throughout the rest of the chapter. Table 14-1 lists the possible descriptors, both required and optional.

Table 14-1. Font descriptors

Descriptor	Default value	Description
`font-family`	*n/a*	*Required.* The name used for this font in `font-family` property values.
`src`	*n/a*	*Required.* One or more URLs pointing to the font file(s) that must be loaded to display the font.
`font-display`	`auto`	Determines how a font face is displayed based on whether and when it is downloaded and ready to use.
`font-stretch`	`normal`	Distinguishes between varying degrees of character widths (e.g., `condensed` and `expanded`).
`font-style`	`normal`	Distinguishes between `normal`, `italic`, and `oblique` faces.
`font-weight`	`normal`	Distinguishes between various weights (e.g., `bold`).
`font-variant`	`normal`	A value of the `font-variant` property.
`font-feature-settings`	`normal`	Permits direct access to low-level OpenType features (e.g., enabling ligatures).
`font-variation-settings`	`normal`	Allows low-level control over OpenType or TrueType font variations, by specifying the four-letter axis names of the features to vary, along with their variation values.
`ascent-override`	`normal`	Defines the ascent metric for the font.
`descent-override`	`normal`	Defines the descent metric for the font.
`line-gap-override`	`normal`	Defines the line gap metric for the font.
`size-adjust`	`100%`	Defines a multiplier for glyph outlines and metrics associated with the font.
`unicode-range`	`U+0-10FFFF`	Defines the range of characters for which a given face may be used.

As noted in Table 14-1, two descriptors are required: `font-family` and `src`.

font-family descriptor

Value	*<family-name>*
Initial value	Not defined

src descriptor

Values	*<uri>* [format(*<string>*#)]? [tech(*<font-tech>*#)]? | *<font-face-name>*]#
Initial value	Not defined

The point of `src` is pretty straightforward, so we'll describe it first: `src` lets you define one or more comma-separated sources for the font face you're defining. With each source, you can provide an optional (but recommended) format hint that can help improve download performance.

You can point to a font face at any URL—including files on the user's computer using `local()`, and files elsewhere with `url()`. There is a default restriction: unless you set an exception, font faces can be loaded from only the same origin as the stylesheet. You can't simply point your `src` at someone else's site and download their font. You'll need to host a local copy on your own server, use HTTP access controls to relax the same domain restriction, or use a font-hosting service that provides both the stylesheet(s) and the font file(s).



To create an exception to the same-origin restriction for fonts, include the following in your server's *.htaccess* file:

```
<FilesMatch "\.(ttf|otf|woff|woff2)$">
  <IfModule mod_headers.c>
    Header set Access-Control-Allow-Origin "*"
  </IfModule>
</FilesMatch>
```

The `FilesMatch` line includes all the file extensions of the fonts you want to import. This will allow anyone, from anywhere, to point at your font files and load them directly off your server.

You may be wondering how it is that we're defining `font-family` here when it was already defined in a previous section. This `font-family` is the font-family *descriptor*, whereas the

previously defined `font-family` is the font-family *property*. If this seems confusing, stick with us a moment and all should become clear.

Essentially, `@font-face` lets you create low-level definitions that underpin the font-related properties like `font-family`. When you define a font family name via the descriptor `font-family: "Switzera";`, you're setting up an entry in the user agent's table of font families that you can refer to in your `font-family` property values:

```
@font-face {
    font-family: "Switzera";   /* descriptor */
    src: url("SwitzeraADF-Regular.otf");
}
h1 {font-family: switzera, Helvetica, sans-serif;}  /* property */
```

Note that the `font-family` descriptor value and the entry in the `font-family` property match case-insensitively. If they didn't match at all, the `h1` rule would ignore the first font-family name listed in the `font-family` value and move on to the next (Helvetica, in this case).

Also note that the `font-family` descriptor can be (almost) any name you want to give it. It doesn't have to exactly match the name of the font file, though it usually makes sense to use a descriptor that's at least close to the font's name for purposes of clarity. That said, the value used in the `font-family` property does have to (case-insensitively) match the `font-family` descriptor.

As long as the font has cleanly downloaded and is in a format the user agent can handle, it will be used in the manner you direct, as illustrated in Figure 14-2.

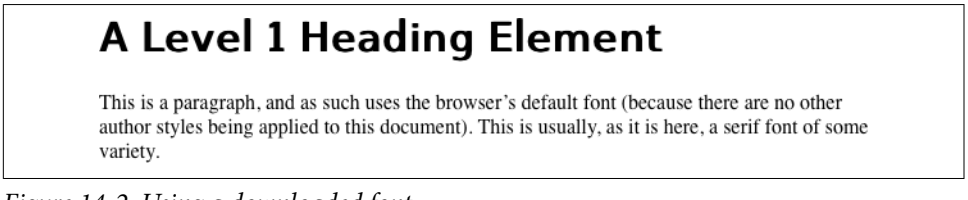


Figure 14-2. Using a downloaded font

In a similar manner, the comma-separated `src` descriptor value can provide fallbacks. That way, if the user agent doesn't understand the file type defined by the hint or, for whatever reason, the user agent is unable to download the first source, it can move on to the second source and try to load the file defined there:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf"),
         url("https://example.com/fonts/SwitzeraADF-Regular.otf");
}
```

Remember that the same-origin policy mentioned earlier generally applies in this case, so pointing to a copy of the font on some other server will usually fail, unless that server is set up to permit cross-origin access.

If you want to be sure the user agent understands the kind of font you're telling it to use, use the optional but highly recommended `format()` hint:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
```

The advantage of supplying a `format()` hint is that user agents can skip downloading files in formats they don't support, thus reducing bandwidth use and load time. If no format hint is supplied, the font resource will be downloaded even if its format isn't supported. The `format()` hint also lets you explicitly declare a format for a file that might not have a common filename extension:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf") format("opentype"),
         url("SwitzeraADF-Regular.true") format("truetype");
         /* TrueType font files usually end in '.ttf' */
}
```

Table 14-2 lists all of the allowed format values (as of late 2022).

Table 14-2. Recognized font-format values

Value	Format	Full name
`collection`	OTC/TTC	OpenType Collection (formerly: TrueType Collection)
`embedded-opentype`	EOT	Embedded OpenType
`opentype`	OTF	OpenType
`svg`	SVG	Scalable Vector Graphics
`truetype`	TTF	TrueType
`woff2`	WOFF2	Web Open Font Format, version 2
`woff`	WOFF	Web Open Font Format

In addition to the format, you can supply a value corresponding to a font technology with the `tech()` function. A color font version of Switzera might look something like this:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular-Color.otf")
            format("opentype") tech("color-COLRv1"),
         url("SwitzeraADF-Regular.true") format("truetype");
         /* TrueType font files usually end in '.ttf' */
}
```

Table 14-3 lists all of the recognized font-technology values (as of late 2022).

Table 14-3. Recognized font-technology values

Value	Description
color-CBDT	Font colors are defined using the OpenType CBDT (Color Bitmap Data Table) table.
color-COLRv0	Font colors are defined using the OpenType COLR (Color Table) table.
color-COLRv1	Font colors are defined using the OpenType COLR table.
color-sbix	Font colors are defined using the OpenType `sbix` (Standard Bitmap Graphics Table) table.
color-SVG	Font colors are defined using the OpenType SVG (Scalable Vector Graphics) table.
feature-aat	Font uses tables from the Apple Advanced Typography (AAT) Font Feature Registry.
feature-graphite	Font uses tables from the Graphite open source font-rendering engine.
feature-opentype	Font uses tables from the OpenType specification.
incremental	Incremental font-loading using the range-request or patch-subset server methods.
palettes	A font that offers palettes by way of the OpenType CPAL table.
variations	Font uses variations as defined by the OpenType tables such as GSUB and GPOS, the AAT tables `morx` and `kerx`, or the Graphite tables `Silf`, `Glat`, `Gloc`, `Feat`, and `Sill`.

Delving into the details of all these feature tables is well beyond the scope of this book, and you are unlikely to need to use them most of the time. Even if a font has one or more of the listed feature tables, listing them is not required. Even with `tech("color-SVG")`, an SVG color font will still render using its colors.

In addition to the combination of `url()`, `format()`, and `tech()`, you can supply a font family name (or several names) in case the font is already locally available on the user's machine, using the aptly named `local()` function:

```
@font-face {
    font-family: "Switzera";
    src: local("Switzera-Regular"),
         local("SwitzeraADF-Regular"),
         url("SwitzeraADF-Regular.otf") format("opentype"),
         url("SwitzeraADF-Regular.true") format("truetype");
}
```

In this example, the user agent looks to see if it already has a font family named `Switzera-Regular` or `SwitzeraADF-Regular`, case-insensitively, available on the local machine. If so, it will use the name `Switzera` to refer to that locally installed font. If not, it will use the `url()` values to try downloading the first remote font listed that has a format type it supports.

Bear in mind that the order of the resources listed in `src` matters. As soon as the browser encounters a source in a format it supports, it attempts to use that source. For this reason, `local()` values should be listed first, with no format hint needed. This should be followed by external resources with file type hints, generally in order of smallest file size to largest to minimize performance hits.

This capability allows an author to create custom names for locally installed fonts. For example, you could set up a shorter name for versions of Hiragino, a Japanese font, like so:

```
@font-face {
    font-family: "Hiragino";
    src: local("Hiragino Kaku Gothic Pro"),
         local("Hiragino Kaku Gothic Std");
}

h1, h2, h3 {font-family: Hiragino, sans-serif;}
```

As long as the user has one of the versions of Hiragino Kaku Gothic installed on their machine, those rules will cause the first three heading levels to be rendered using that font.

Online services let you upload font-face files and generate all the `@font-face` rules you need, convert those files to all the formats required, and hand everything back to you as a single package. One of the best known is Font Squirrel's @Font-Face Kit Generator. Just make sure you're legally able to convert and use the font faces you're running through the generator (see the following sidebar for more information).

Custom Font Considerations

You need to keep two points in mind when using customized fonts. The first is that you legally must have the rights to use the font in a web page, and the second is whether it's a good idea to do so.

Much like stock photography, commercial font families come with licenses that govern their use, and not every font license permits its use on the web. You can completely avoid this question by using only free and open source software (FOSS) fonts, or by using a commercial service like Google Fonts or Adobe Fonts that deals with the licensing and format conversion issues so you don't have to. Otherwise, you need to make sure you have the legal right to use a font face in the way you want to use it, just the same as you make sure you have the proper license for any images you want to use in your designs.

In addition, the more font faces you call upon, the more resources the web server has to hand over and the higher the overall page weight will become. Most faces are not overly large—usually 50 KB to 200 KB—but they add up quickly if you decide to get fancy with your type, and truly complicated faces can be much larger than 200 KB. You will have to balance appearance against performance, leaning one way or the other depending on the circumstances.

That said, just as image optimization tools are available, so are font optimization tools. Typically, these are *subsetting* tools, which construct fonts using only the symbols actually needed for display. If you're using a service like Adobe Fonts or Fonts.com, they probably have subsetting tools available, or else do it dynamically when the font is requested.

When subsetting a font, you can use the `unicode-range` descriptor to limit custom font use to only the characters in the font file. Services such as Font Squirrel will subset the font for you and provide the unicode range in the CSS snippet it produces. Just remember that subsetting needs to be done in the font file, not just in the Unicode range, in order to reduce the file size.

Restricting Character Range

At times you might want to use a custom font in very limited circumstances; for example, to ensure that a font face is applied only to characters that are in a specific language. In these cases, it can be useful to restrict the use of a font to certain characters or symbols, and the `unicode-range` descriptor allows precisely that.

unicode-range descriptor

Values *<urange>#*
Initial value U+0-10FFFF

By default, the value of this descriptor covers U+0 to U+10FFFF, which is the entirety of Unicode—meaning that if a font can supply the glyph for a character, it will. Most of the time, this is exactly what you want. For all the other times, you'll want to use specific font faces for specific kinds of content. You can define a single code point, a code-point range, or a set of ranges with the `?` wildcard character.

To pick a few examples from the CSS Fonts Module Level 3:

```
unicode-range: U+0026; /* the Ampersand (&) character */
unicode-range: U+590-5FF;  /* Hebrew characters */
unicode-range: U+4E00-9FFF, U+FF00-FF9F, U+30??, U+A5;  /* Japanese
  kanji, hiragana, and katakana, plus the yen/yuan currency symbol*/
```

In the first case, a single code point is specified. The font will be used only for the ampersand (&) character. If the ampersand is not used, the font is not downloaded. If it is used, the entire font file is downloaded. For this reason, it is sometimes good to optimize your font files to include only characters in the specified Unicode range, especially if, as in this case, you're using only one character from a font that could contain several thousand characters.

In the second case, a single range is specified, spanning Unicode character code point 590 through code point 5FF. This covers the 111 total characters used when writing Hebrew. Thus, an author might specify a Hebrew font and restrict it to be used only for Hebrew characters, even if the face contains glyphs for other code points:

```
@font-face {
    font-family: "CMM-Ahuvah";
```

```
    src: url("cmm-ahuvah.otf" format("opentype");
    unicode-range: U+590-5FF;
}
```

In the third case, a series of ranges are specified in a comma-separated list to cover all the Japanese characters. The interesting feature there is the `U+30??` value, with a question mark, which is a special format permitted in `unicode-range` values. The question marks are wildcards meaning "any possible digit," making `U+30??` equivalent to `U+3000-30FF`. The question mark is the only "special" character pattern permitted in the value.

Ranges must always ascend. Any descending range, such as `U+400-300`, is treated as a parsing error and ignored.

Because `@font-face` is designed to optimize lazy loading, it's possible to use `unicode-range` to download only the font faces a page actually needs, with possibly a much smaller file size when using a font file optimized to contain only the defined subset character range. If the page doesn't use any character in the range, the font is not downloaded. If a single character on a page requires a font, the whole font is downloaded.

Suppose you have a website that uses a mixture of English, Russian, and basic mathematical operators, but you don't know which will appear on any given page. A page could be all English, a mixture of Russian and math, and so on. Furthermore, suppose you have special font faces for all three types of content. You can make sure a user agent downloads only the faces it needs with a properly constructed series of `@font-face` rules:

```
@font-face {
    font-family: "MyFont";
    src: url("myfont-general.otf" format("opentype");
}
@font-face {
    font-family: "MyFont";
    src: url("myfont-cyrillic.otf" format("opentype");
    unicode-range: U+04??, U+0500-052F, U+2DE0-2DFF, U+A640-A69F, U+1D2B-1D78;
}
@font-face {
    font-family: "MyFont";
    src: url("myfont-math.otf" format("opentype");
    unicode-range: U+22??;   /* equivalent to U+2200-22FF */
}

body {font-family: MyFont, serif;}
```

Because the first rule doesn't specify a Unicode range, the entire font file is always downloaded—unless a page happens to contain no characters at all (and maybe even then). The second rule causes *myfont-cyrillic.otf* to be downloaded only if the page contains characters in its declared Unicode range; the third rule does the same for mathematical symbols.

If the content calls for the mathematical character U+2222 (∢, the spherical angle character), *myfont-math.otf* will be downloaded and the character from *myfont-math.otf* will be used, even if *myfont-general.otf* has that character.

A more likely way to use this capability would be our ampersand example; we can include a fancy ampersand from a cursive font and use it in place of the ampersand found in a headline font. Something like this:

```
@font-face {
    font-family: "Headline";
    src: url("headliner.otf" format("opentype");
}
@font-face {
    font-family: "Headline";
    src: url("cursive-font.otf" format("opentype");
    unicode-range: U+0026;
}

h1, h2, h3, h4, h5, h6 {font-face: Headline, cursive;}
```

In a case like this, to keep page weights low, take a cursive font (that you have the rights to use) and minimize it down to contain just the ampersand character. You can use a tool like Font Squirrel to create a single-character font file.



Remember that pages can be translated with automated services like Google Translate. If you too aggressively restrict your Unicode ranges (say, to the range of unaccented letters used in English), an auto-translated version of the page into French or Swedish, for example, could end up a mishmash of characters in different font faces, as the accented characters in those languages would use a fallback font and the unaccented characters would be in your intended font.

Working with Font Display

If you're a designer or developer of a certain vintage, you may remember the days of the *flash of unstyled content* (*FOUC*). This happened in earlier browsers that would load the HTML and display it to the screen before the CSS was finished loading, or, at least, before the layout of the page via CSS was finished.

A *flash of unstyled text* (*FOUT*) happens when a browser has loaded the page and the CSS and then displays the laid-out page, along with all the text, before it's done loading custom fonts. FOUT causes text to appear in the default font, or a fallback font, before being replaced by text using the custom-loaded font.

A cousin to this problem is the *flash of invisible text* (*FOIT*). This user-agent solution to FOUT is caused when the browser detects that text is set in a custom font that hasn't loaded yet and makes the text invisible until the font loads or a certain amount of time has passed.

Since the replacement of text can change its size, whether via FOUT or FOIT, take care when selecting fallback fonts. If a significant height difference exists between the font

used to initially display the text and the custom font eventually loaded and used, significant page reflows are likely to occur.

In an attempt to help with this, the `font-display` descriptor guides the browser to proceed with text rendering when a web font has yet to load.

font-display descriptor

Values	`auto` | `block` | `swap` | `fallback` | `optional`
Initial value	`auto`
Applies to	All elements
Computed value	As specified

What we can call the *font-display timeline timer* starts when the user agent first paints the page. The timeline is divided into three periods: block, swap, and failure.

During the *font-block period*, if the font face is not loaded, the browser renders any content that should use that font by using an invisible fallback font face, meaning the text content is not visible but the space is reserved. If the font loads successfully during the block period, the text is rendered with the downloaded font and made visible.

During the *swap period*, if the font face is not loaded, the browser renders the content by using a visible fallback font face, most likely one it has installed locally (e.g., Helvetica). If the font loads successfully, the fallback font face is swapped to the downloaded font.

Once the *failure period* is entered, the user agent treats the requested font as a failed load, falls back to an available font, and will not swap the font if it does eventually load. If the swap period is infinite, the failure period is never entered.

The values of the `font-display` descriptor match these periods of the timeline, and their effect is to emphasize one part of the timeline at the expense of the others. The effects are summarized in Table 14-4.

Table 14-4. `font-display` values

Value	Block period[a]	Swap period[a]	Failure period[a]
`auto`	Browser defined	Browser defined	Browser defined
`block`	3s	Infinite	*n/a*
`swap`	< 100 ms	Infinite	*n/a*
`fallback`	< 100 ms	3s	Infinite
`optional`	0	0	Infinite

[a] Recommended period length; actual times may vary

Let's consider each value in turn:

`block`
: Tells the browser to hold open space for the font for a few seconds (3 is what the specification recommends, but browsers may choose their own values), and then enters an infinitely long swap period. If the font ever finally loads, even if it's 10 minutes later, the fallback font that was used in its place will be replaced with the loaded font.

`swap`
: Is similar, except it doesn't hold the space open for longer than a fraction of a second (100 milliseconds is the recommendation). A fallback font is then used, and is replaced with the intended font whenever it finally loads.

`fallback`
: Gives the same brief block window that `swap` does, and then enters a short period in which the fallback font can be replaced by the intended font. If that short period (3 seconds is recommended) is exceeded, the fallback font is used indefinitely, and the user agent may cancel the download of the intended font since a swap will never happen.

`optional`
: Is the most stringent of them all: if the font isn't immediately available at first paint, the user agent goes straight to the fallback font and skips right over the block and swap periods to sit in the failure period for the rest of the page's life.

Combining Descriptors

Something that might not be immediately obvious is that you can supply multiple descriptors in order to assign specific faces for specific property combinations. For example, you can assign one face to bold text, another to italic text, and a third to text that is both bold and italic.

This capability is implicit, as any undeclared descriptor is assigned its default value. Let's consider a basic set of three face assignments, using both descriptors we've covered and a few we'll get to in a bit:

```
@font-face {
    font-family: "Switzera";
    font-weight: normal;
    font-style: normal;
    font-stretch: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "Switzera";
    font-weight: 500;
    font-style: normal;
    font-stretch: normal;
```

```
    src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
    font-family: "Switzera";
    font-weight: normal;
    font-style: italic;
    font-stretch: normal;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

You may have noticed that we've explicitly declared some descriptors with their default values, even though we didn't need to. The previous example is exactly the same as a set of three rules in which we remove every descriptor that shows a value of `normal`:

```
@font-face {
   font-family: "Switzera";
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: 500;
   src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: italic;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

In all three rules, no font-stretching occurs beyond the default `normal` amount, and the values of `font-weight` and `font-style` vary by which face is being assigned. So what if we want to assign a specific face to unstretched text that's both bold and italic?

```
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-style: italic;
   font-stretch: normal;
   src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
```

And then what about bold, italic, condensed text?

```
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

How about normal-weight, italic, condensed text?

```
@font-face {
   font-family: "Switzera";
   font-weight: normal;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
```

We could keep this up for quite a while, but let's stop there. If we take all those rules and strip out anything with a `normal` value, we end up with the following result, illustrated in Figure 14-3:

```
@font-face {
   font-family: "Switzera";
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: italic;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-style: italic;
   src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCond.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

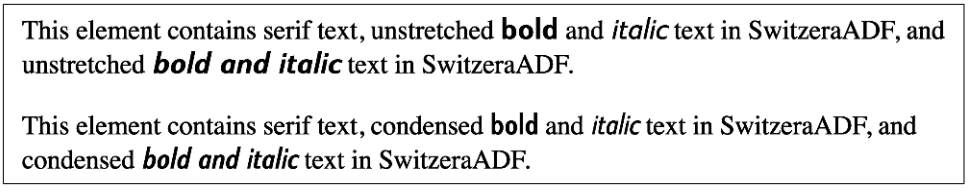


Figure 14-3. Employing a variety of faces

If you declare `html { +font-family: switzera;}`, you don't have to declare the font family again for additional selectors that use `switzera`. The browser will use the correct font file for your bold, italic, stretched, and normal text depending on your selector-specific values for the `font-weight`, `font-style`, and `font-stretch` property values.

The point is, we can have a specific font file for every weight, style, and stretch. The ability to declare all the variations within a few `@font-face` rules with a single `font-family` name ensures cohesive typeface design and avoids font synthesis even when using non-variable fonts. Declaring all the variations of a font via `@font-face`, with the same `font-family` descriptor name, reduces `font-family` property overrides, reducing the chance of other developers on your team using the wrong font file for a specific selector.

As you can see, when using standard fonts, a lot of possible combinations exist just for those three descriptors—consider that `font-stretch` has 10 possible values—but you'll likely never have to run through them all. In fact, most font families don't have as many faces as SwitzeraADF offers (24 at last count), so there wouldn't be much point in writing out all the possibilities. Nevertheless, the options are there, and in some cases you may find that you need to assign, say, a specific face for bold condensed text so that the user agent doesn't try to compute them for you. Or else use a variable font that has weight and condensing axes.

Now that we've covered `@font-face` and provided an overview of a few descriptors, let's get back to properties.

Font Weights

Most of us are used to normal and bold text, which are the two most basic font weights available. CSS gives you a lot more control over font weights with the property `font-weight`.

font-weight

Values	`normal` | `bold` | `bolder` | `lighter` | *<number>*
Initial value	`normal`
Applies to	All elements

Computed value	One of the numeric values (`100`, etc.), or one of the numeric values plus one of the relative values (`bolder` or `lighter`)
@font-face equivalent	`font-weight`
Variable axis	`"wght"`
Inherited	Yes
Animatable	No

The <*number*> value can be from `1` to `1000`, inclusive, where `1` is the lightest and `1000` is the heaviest possible weight. Unless you are using variable fonts, discussed later, limited weights are almost always available for a font family (sometimes there is only a single weight).

Generally speaking, the heavier a font weight becomes, the darker and "more bold" a font appears. There are a great many ways to label a heavy font face. For example, the font family known as SwitzeraADF has variants such as SwitzeraADF Bold, SwitzeraADF Extra Bold, SwitzeraADF Light, and SwitzeraADF Regular. All of these use the same basic font shapes, but each has a different weight.

If the specified weight doesn't exist, a nearby weight is used. Table 14-5 lists the numbers used for each of the commonly accepted font weight labels, as defined in the `"wght"` variation axis. If a font has only two weights corresponding to `400` and `700` (normal and bold), any number value for `font-weight` will be mapped to the closest value. Thus, any `font-weight` value from `1` through `550` will be mapped to `400`, and any value greater than `550` up through `1000` will be mapped to `700`.

Table 14-5. Weight mappings

Value	Mapping
1	Lowest valid value
100	Thin
200	Extra Light (Ultra Light)
300	Light
400	Normal
500	Medium
600	Semi Bold (Demi Bold)
700	Bold
800	Extra Bold (Ultra Bold)
900	Black (Heavy)
950	Extra Black (Ultra Black)
1000	Highest valid value

Let's say that you want to use SwitzeraADF for a document but would like to make use of all those heaviness levels. If your user has all the font files locally on their machine and you didn't use `@font-face` to rename all the options to `Switzera`, you could refer to them directly through the `font-family` property—but you really shouldn't have to do that. It's no fun having to write a stylesheet like this:

```
h1 {font-family: 'SwitzeraADF Extra Bold', sans-serif;}
h2 {font-family: 'SwitzeraADF Bold', sans-serif;}
h3 {font-family: 'SwitzeraADF Bold', sans-serif;}
h4, p {font-family: 'SwitzeraADF Regular', sans-serif;}
small {font-family: 'SwitzeraADF Light', sans-serif;}
```

That's pretty tedious. This is a perfect example of why specifying a single font family for the whole document and then assigning different weights to various elements by using `@font-face` is so powerful: you can include several `@font-face` declarations, each with the same `font-family` name, but with various values for the `font-weight` descriptors. Then you can use different font files with fairly simple `font-weight` declarations:

```
strong {font-weight: bold;}
b {font-weight: bolder;}
```

The first declaration says the `<strong>` element should be displayed using a bold font face or, to put it another way, a font face that is heavier than the normal font face. The second declaration says `<b>` should use a font face that is the inherited `font-weight` value plus 100.

What's really happening behind the scenes is that heavier faces of the font are used for displaying `<strong>` and `<b>` elements. Thus, if you display a paragraph using Times, and part of it is bold, then two faces of the same font are really in use: Times and TimesBold. The regular text is displayed using Times, and the bold and bolder text are displayed using TimesBold.

If the font doesn't have a boldfaced version, it may be synthesized by the browser, creating a faux bold. (To prevent this, use `font-synthesis` property, which is described later.)

How Weights Work

To understand how a user agent determines the heaviness, or weight, of a given font variant (not to mention how weight is inherited), it's easiest to start by talking about the values `1` through `1000` inclusive, specifically the values divisible by 100, `100` through `900`. These number values were defined to map to a relatively common feature of font design that gives a font nine levels of weight. If a nonvariable font family has faces for all nine weight levels available, the numbers are mapped directly to the predefined levels, with `100` as the lightest variant of the font and `900` as the heaviest.

In fact, these numbers have no intrinsic weight. The CSS specification says only that each number corresponds to a weight at least as heavy as the number that precedes it. Thus, `100`, `200`, `300`, and `400` might all map to a single relatively lightweight variant; `500` and `600` could correspond to a single medium-heavy font variant; and `700`, `800`, and `900` could all

produce the same very heavy font variant. As long as no number corresponds to a variant that is lighter than the variant assigned to the previous lower number, everything will be all right.

When it comes to nonvariable fonts, these numbers are defined to be equivalent to certain common variant names. The value `400` is defined to be equivalent to `normal`, and `700` corresponds to `bold`.

A user agent has to do some calculations if a font family has fewer than nine weights. In this case, the user agent must fill in the gaps in a predetermined way:

- If the value `500` is unassigned, it is given the same font weight as that assigned to `400`.
- If `300` is unassigned, it is given the next variant lighter than `400`. If no lighter variant is available, `300` is assigned the same variant as `400`. In this case, it will usually be Normal or Medium. This method is also used for `200` and `100`.
- If `600` is unassigned, it is given the next variant darker than that assigned for `500`. If no darker variant is available, `600` is assigned the same variant as `500`. This method is also used for `700`, `800`, and `900`.

To illustrate this weighting scheme more clearly, let's look at a couple of examples. In the first example, assume that the font family Karrank% is an OpenType font, so it has nine weights already defined. In this case, the numbers are assigned to each level, and the keywords `normal` and `bold` are assigned to the numbers `400` and `700`, respectively.

In our second example, consider the font family SwitzeraADF. Hypothetically, its variants might be assigned numeric values for `font-weight`, as shown in Table 14-6.

Table 14-6. Hypothetical weight assignments for a specific font family

Font face	Assigned keyword	Assigned number(s)
SwitzeraADF Light		`100` through `300`
SwitzeraADF Regular	`normal`	`400`
SwitzeraADF Medium		`500`
SwitzeraADF Bold	`bold`	`600` through `700`
SwitzeraADF Extra Bold		`800` through `900`

The first three number values are assigned to the lightest weight. The Regular face gets the keyword `normal` and the number weight `400`. Since there is a Medium font, it's assigned to the number `500`. There is nothing to assign to `600`, so it's mapped to the Bold font face, which is also the variant to which `700` and `bold` are assigned. Finally, `800` and `900` are assigned to the Black and Ultra Black variants, respectively. Note that this last assignment would happen only if those faces had the top two weight levels already assigned. Otherwise, the user agent might ignore them and assign `800` and `900` to the Bold face instead, or it might assign them both to one or the other of the Black variants.

The `font-weight` property is inherited, so if you set a paragraph to be `bold`

```
p.one {font-weight: bold;}
```

then all of its children will inherit that boldness, as we see in Figure 14-4.

Within this paragraph we find some *italicized text*, a bit of underlined text, and the occasional stretch of hyperlinked text for our viewing pleasure.

Figure 14-4. Inherited font-weight

This isn't unusual, but the situation gets interesting when you use the last two values we have to discuss: `bolder` and `lighter`. In general terms, these keywords have the effect you'd anticipate: they make text more or less bold compared to its parent's font weight. How they do so is slightly complicated. First, let's consider `bolder`.

If you set an element to have a weight of `bolder` or `lighter`, the user agent first must determine what `font-weight` value was inherited from the parent element. Once it has that number (say, `400`), it then changes the value as shown in Table 14-7.

Table 14-7. `bolder` and `lighter` weight mappings

Inherited value	`bolder`	`lighter`
value < 100	400	No change
100 ≤ value < 350	400	100
350 ≤ value < 550	700	100
550 ≤ value < 750	900	400
750 ≤ value < 900	900	700
900 ≤ value	No change	700

Thus, you might encounter the following situations, illustrated in Figure 14-5:

```
p {font-weight: normal;}
p em {font-weight: bolder;}  /* inherited value '400', evaluates to '700' */

h1 {font-weight: bold;}
h1 b {font-weight: bolder;}  /* inherited value '700', evaluates to '900' */

div {font-weight: 100;}
div strong {font-weight: bolder;} /* inherited value '100', evaluates to '400' */
```

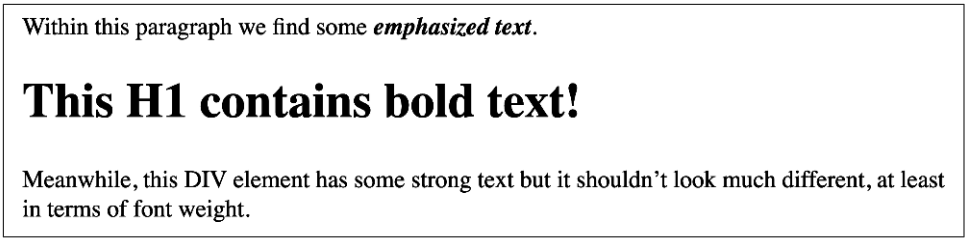


Figure 14-5. Text trying to be bolder

In the first example, the user agent moves up from `400` to `700`. In the second example, `<h1>` text is already set to `bold`, which equates to `700`. If no bolder face is available, the user agent sets the weight of `<b>` text within an `<h1>` to `900`, since that is the next step up from `700`. Since `900` is assigned to the same font face as `700`, no visible difference exists between normal `<h1>` text and bold `<h1>` text, but the weight values are different nonetheless.

As you might expect, `lighter` works in much the same way, except it causes the user agent to move down the weight scale instead of up.

The font-weight Descriptor

With the `font-weight` descriptor, authors can assign faces of varying weights to the weighting levels permitted by the `font-weight` property. The allowable values are different for the descriptor, which supports `auto`, `normal`, `bold`, or one to two numeric values as a range. Neither `lighter` nor `bolder` are supported.

For example, the following rules explicitly assign five faces to six `font-weight` values:

```
@font-face {
   font-family: "Switzera";
   font-weight: 1 250;
   src: url("f/SwitzeraADF-Light.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: normal;
   src: url("f/SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: 500 600;
   src: url("f/SwitzeraADF-DemiBold.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   src: url("f/SwitzeraADF-Bold.otf") format("opentype");
}
```

```
@font-face {
   font-family: "Switzera";
   font-weight: 800 1000;
   src: url("f/SwitzeraADF-ExtraBold.otf") format("opentype");
}
```

With these faces assigned, the author now has multiple weighting levels available for their use, as illustrated in Figure 14-6:

```
h1, h2, h3, h4 {font-family: SwitzeraADF, Helvetica, sans-serif;}
h1 {font-size: 225%; font-weight: 900;}
h2 {font-size: 180%; font-weight: 700;}
h3 {font-size: 150%; font-weight: 500;}
h4 {font-size: 125%; font-weight: 300;}
```

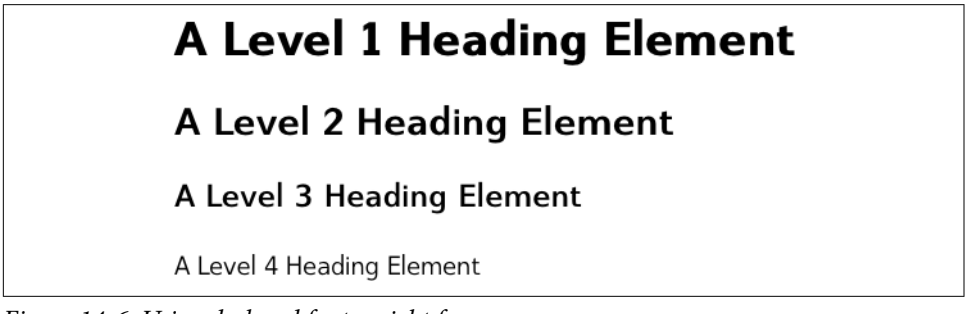


Figure 14-6. Using declared font-weight faces

In any given situation, the user agent picks which face to use depending on the exact value of a `font-weight` property, using the resolution algorithm detailed in "How Weights Work" on page 682. While the `font-weight` property has numerous keyword values, the `font-weight` descriptor accepts only `normal` and `bold` as keywords, and any number from `1` to `1000` inclusive.

Font Size

While size doesn't have a `@font-face` descriptor, you need to understand the `font-size` property to better understand some of the descriptors to come, so we'll explore it now. The methods for determining font size are both very familiar and very different.

font-size

Values	`xx-small` | `x-small` | `small` | `medium` | `large` | `x-large` | `xx-large` | `xxx-large` | `smaller` | `larger` | *<length>* | *<percentage>*
Initial value	`medium`
Applies to	All elements
Percentages	Calculated with respect to the parent element's font size

Computed value	An absolute length
Inherited	Yes
Animatable	Yes (numeric keywords only)

What can be a real head-scratcher at first is that different fonts declared to be the same size may not appear to be the same size. This is because the actual relation of the `font-size` property to what you see rendered is determined by the font's designer. This relationship is set as an *em square* (some call it an *em box*) within the font itself. This em square (and thus the font size) doesn't have to refer to any boundaries established by the characters in a font. Instead, it refers to the distance between baselines when the font is set without any extra leading (`line-height` in CSS).

The effect of `font-size` is to provide a size for the em box of a given font. This does not guarantee that any of the displayed characters will be this size. It is quite possible for fonts to have characters that are taller than the default distance between baselines. For that matter, a font might be defined such that all of its characters are smaller than its em square, as many fonts are. Figure 14-7 shows some hypothetical examples.

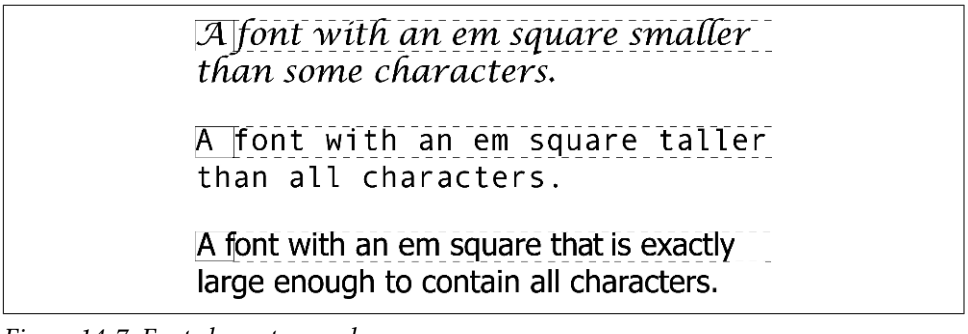


Figure 14-7. Font characters and em squares

Using Absolute Sizes

Having established all that, we turn now to the absolute-size keywords. The `font-size` property has eight absolute-size values: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, and the relatively new `xxx-large`. These are not defined precisely but instead are defined relative to each other, as Figure 14-8 demonstrates:

```
p.one {font-size: xx-small;}
p.two {font-size: x-small;}
p.three {font-size: small;}
p.four {font-size: medium;}
p.five {font-size: large;}
p.six {font-size: x-large;}
p.seven {font-size: xx-large;}
p.eight {font-size: xxx-large;}
```

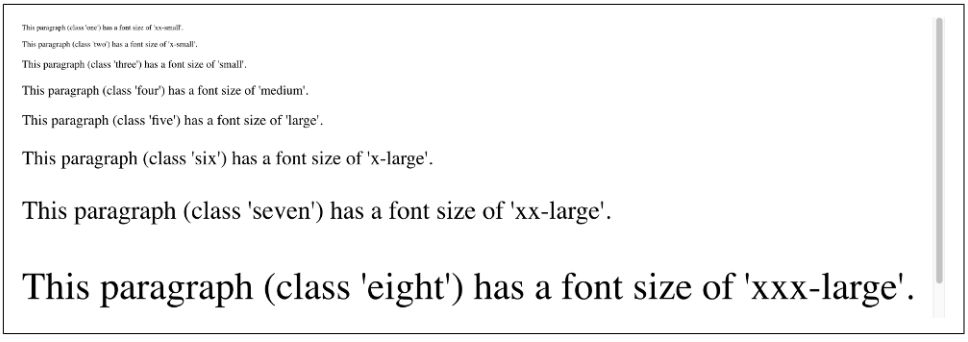


Figure 14-8. Absolute font sizes

In the CSS1 specification, the difference (or *scaling factor*) between one absolute size and the next was 1.5 going up the ladder, or 0.66 going down. This was determined to be too large a scaling factor. In CSS2, the suggested scaling factor for computer screens between adjacent indexes was 1.2. This didn't resolve all the issues, though, as it created issues for the small sizes.

The CSS Fonts Level 4 specification doesn't have a one-size-fits-all scaling factor. Rather, each absolute-size keyword value has a size-specific scaling factor based on the value of `medium` (see Table 14-8). The `small` value is listed as eight-ninths the size of `medium`, while `xx-small` is three-fifths. In any case, the scaling factors are guidelines, and user agents are free to alter them for any reason.

Table 14-8. Font-size mappings

CSS absolute-size values	`xx-small`	`x-small`	`small`	`medium`	`large`	`x-large`	`xx-large`	`xxx-large`
Scaling factor	3/5	3/4	8/9	1	6/5	3/2	2/1	3/1
Sizes at `medium==16px`	`9px`	`10px`	`13px`	`16px`	`18px`	`24px`	`32px`	`48px`
HTML heading equivalent	`h6`	-	`h5`	`h4`	`h3`	`h2`	`h1`	*n/a*

Note that we've set the default size, `medium`, explicitly to `16px`. The default `font-size` value is the same, `medium`, for all generic font families, but the `medium` keyword may have different definitions based on operating system or browser user settings. For example, in many browsers, serif and sans-serif fonts have `medium` equal to `16px`, but `monospace` set to `13px`.



As of late 2022, the `xxx-large` keyword is not supported by Safari or Opera, either on desktop or mobile devices.

Using Relative Sizes

Just as `font-weight` has the keywords `bolder` and `lighter`, the `font-size` property has relative-size keywords called `larger` and `smaller`. Much as with relative font weights, these keywords cause the computed value of `font-size` to move up and down a scale of size values.

The `larger` and `smaller` keywords are fairly straightforward: they cause the size of an element to be shifted up or down the absolute-size scale, relative to their parent element:

```
p {font-size: medium;}
strong, em {font-size: larger;}

<p>This paragraph element contains <strong>a strong-emphasis element,
which itself contains <em>an emphasis element, which also contains
<strong>a strong element.</strong></em></strong></p>

<p> medium <strong>large <em> x-large <strong>xx-large</strong> </em> </strong>
    </p>
```

Unlike the relative values for weight, the relative-size values are not necessarily constrained to the limits of the absolute-size range. Thus, a font's size can be pushed beyond the sizes for `xx-small` and `xxx-large`. If the parent element `font-size` is the largest or smallest absolute value, the browser will use a scaling factor between 1.2 and 1.5 to create an even smaller or larger font size. For example:

```
h1 {font-size: xxx-large;}
em {font-size: larger;}

<h1>A Heading with <em>Emphasis</em> added</h1>
<p>This paragraph has some <em>emphasis</em> as well.</p>
```

As you can see in Figure 14-9, the emphasized text in the `<h1>` element is slightly larger than `xxx-large`. The amount of scaling is left up to the user agent, with a scaling factor in the range of 1.2 to 1.5 being preferred, but not required. The `em` text in the paragraph is shifted up one slot to 140%.

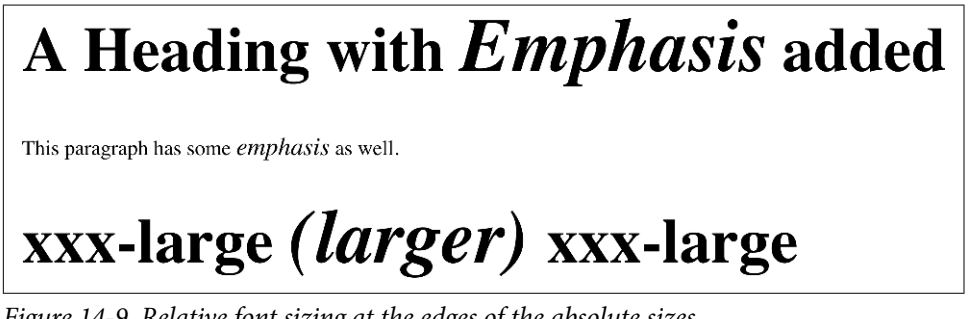


Figure 14-9. Relative font sizing at the edges of the absolute sizes



User agents are not required to increase or decrease font size beyond the limits of the absolute-size keywords, but they may do so anyway. Also, while it is technically possible to declare `smaller` than `xx-small`, small text can be very difficult to read onscreen, leading to content being not accessible to users. Use very small text sparingly and with a great deal of caution.

Setting Sizes as Percentages

In a way, percentage values are very similar to the relative-size keywords. A percentage value is always computed in terms of whatever size is inherited from an element's parent. Unlike the size keywords previously discussed, percentages permit much finer control over the computed font size. Consider the following example, illustrated in Figure 14-10:

```
body {font-size: 15px;}
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}
small, .fnote {font-size: 70%;}

<body>
<p>This paragraph contains both <em>emphasis</em> and <strong>strong
emphasis</strong>, both of which are larger than their parent element.
The <small>small text</small>, on the other hand, is smaller by a quarter.</p>
<p class="fnote">This is a 'footnote' and is smaller than regular text.</p>

<p> 12px <em> 14.4px </em> 12px <strong> 16.2px </strong> 12px
<small> 9px </small> 12px </p>
<p class="fnote"> 10.5px </p>
</body>
```

This paragraph contains both *emphasis* and **strong emphasis**, both of which are larger than their parent element. The small text, on the other hand, is smaller by a quarter.

This is a 'footnote' and is smaller than regular text.

12px *14.4px* 12px **16.2px** 12px 9px 12px

10.5px

Figure 14-10. Throwing percentages into the mix

In this example, the exact pixel-size values are shown. These are the values calculated by the browser, regardless of the displayed size of the characters onscreen, which may have been rounded to the nearest whole number of pixels.

When using em measurements, the same principles apply as with percentages, such as the inheritance of computed sizes and so forth. CSS defines the length value `em` to be equivalent to percentage values, in the sense that `1em` is the same as `100%` when sizing fonts. Thus, the following would yield identical results, assuming that both paragraphs have the same parent element:

```
p.one {font-size: 166%;}
p.two {font-size: 1.66em;}
```

As with the relative-size keywords, percentages are effectively cumulative. Thus, the following markup is displayed as shown in Figure 14-11:

```
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}

<p>This paragraph contains both <em>emphasis and <strong>strong
emphasis</strong></em>, both of which are larger than the paragraph text. </p>

<p>12px <em>14.4px <strong> 19.44px </strong></em> 12px</p>
```

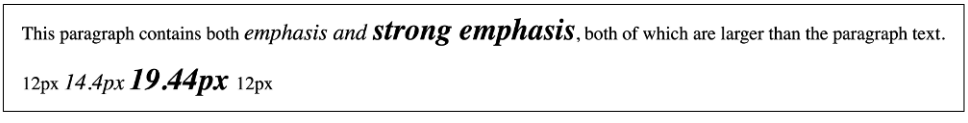


Figure 14-11. The issues of inheritance

The size value for the `<strong>` element shown in Figure 14-11 is computed as follows:

$$12\text{ px} \times 120\% = 14.4\text{ px} + 14.4\text{ px} \times 135\% = 19.44\text{ px}$$

The problem of runaway scaling can go the other direction, too. Imagine the effect of the following rule on a nested list item if we have lists nested four levels deep:

```
ul {font-size: 80%;}
```

The unordered list nested four levels deep would have a computed `font-size` value 40.96% the size of the parent of the top-level list. Every nested list would have a font size 80% as big as its parent list, causing each level to become harder and harder to read.

Automatically Adjusting Size

Two of the main factors that influence a font's legibility are its size and its *x-height*, which is the height of a lowercase *x* character in the font. The number that results from dividing the x-height by the `font-size` is referred to as the *aspect value*. Fonts with higher aspect values tend to remain legible as the font's size is reduced; conversely, fonts with low aspect values become illegible more quickly. CSS provides a way to deal with shifts in aspect values between font families, as well as ways to use different metrics to compute an aspect value, with the property `font-size-adjust`.

font-size-adjust

Values	[ex-height | cap-height | ch-width | ic-width | ic-height]? [from-font | <*number*>] | none | auto
Initial value	none

Applies to	All elements
@font-face equivalent	`size-adjust`
Inherited	Yes
Animatable	Yes

The goal of this property is to preserve legibility when the font used is not the author's first choice. Because of the differences in font appearance, one font may be legible at a certain size, while another font at the same size is difficult or impossible to read.

The property value can be `none`, `from-font`, or a number. The number specified should generally be the aspect value (the ratio of a given font metric to font size) of the first-choice font family. To pick the font metric used to compute the aspect ratio, you can add a keyword specifying it. If not included, it defaults to `ex-height`, which normalizes the aspect value of the fonts using the x-height divided by the font size.

The other possibilities for the font metric keyword are as follows:

`cap-height`
: Use the cap-height (height of capital letters) of the font.

`ch-width`
: Use the horizontal pitch (also the width of `1ch`) of the font.

`ic-width`
: Use the width of the font using the CJK water ideograph, "水" (U+6C34) of the font.

`ic-height`
: Use the height of the ideograph "水" (U+6C34) of the font.

Declaring `font-size-adjust: none` will suppress any adjustment of font sizes. This is the default state.

The `from-font` keyword directs the user agent to use the built-in value of the specified font metric from the first available font, rather than requiring the author to figure out what that value is and write it explicitly. Thus, writing `font-size-adjust: cap-height from-font` will automatically set an aspect value by dividing the cap-height by the em-square height.

A good example is to compare the common fonts Verdana and Times. Consider Figure 14-12 and the following markup, which shows both fonts at a `font-size` of `10px`:

```
p {font-size: 10px;}
p.cl1 {font-family: Verdana, sans-serif;}
p.cl2 {font-family: Times, serif; }
```

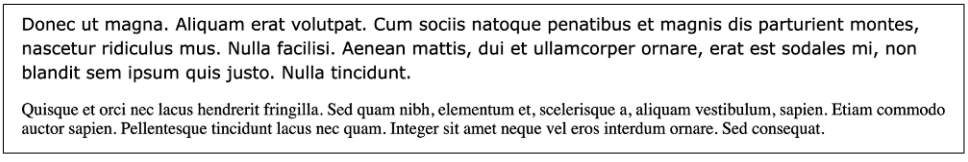


Figure 14-12. Comparing Verdana and Times

The text in Times is much harder to read than the Verdana text. This is partly due to the limitations of pixel-based display, but it is also because Times becomes harder to read at smaller font sizes.

As it turns out, the ratio of x-height to character size in Verdana is 0.58, whereas in Times it is 0.46. To make these font faces look more consistent with each other, you can declare the aspect value of Verdana, and have the user agent adjust the size of the text that's actually used. This is accomplished using the following formula:

> Declared `font-size` ×
> (`font-size-adjust` value ÷ aspect
> value of available font) = Adjusted `font-size`

So, when Times is used instead of Verdana, the adjustment is as follows:

> `10px` × (`0.58` ÷ `0.46`) = `12.6px`

This leads to the result shown in Figure 14-13:

```
p {font: 10px Verdana, sans-serif; font-size-adjust: ex-height 0.58;}
p.cl2 {font-family: Times, serif; }
```

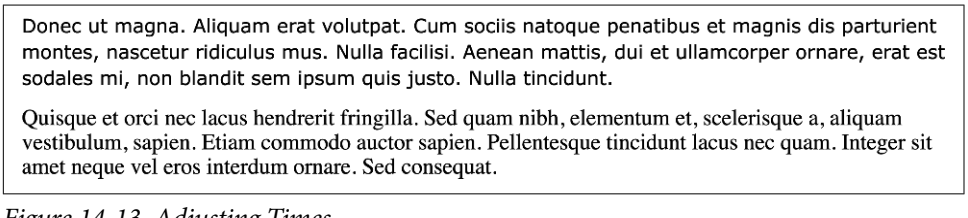


Figure 14-13. Adjusting Times

The catch is that for a user agent to intelligently make size adjustments, it first has to know the aspect value of the fonts you specify. User agents that support `@font-face` will be able to pull that information directly from the font file, assuming the files contain the information—any professionally produced font should, but there's no guarantee. If a font file doesn't contain the aspect value, a user agent may try to compute it; but again, there's no guarantee that they will or even can.

If the user agent can't find or figure out aspect values on its own, the `auto` value for `font-size-adjust` is a way of getting the desired effect even if you don't know the actual aspect value of your first-choice font. For example, assuming that the user agent can determine

that the aspect value of Verdana is 0.58, the following will have the same result as that shown in Figure 14-13:

```
p {font: 10px Verdana, sans-serif; font-size-adjust: auto;}
p.cl2 {font-family: Times, serif; }
```



As of late 2022, the only user-agent line to support `font-size-adjust` is the Gecko (Firefox) family.

Understanding font size adjustment comes in handy when considering `size-adjust`. This font descriptor behaves similarly to the `font-size-adjust` property, though it's restricted to comparing only x-heights instead of the range of font metrics available for `font-size-adjust`.

size-adjust descriptor

Values	*<percentage>*
Initial value	`100%`

The `font-size-adjust` property is a rare case where the property and descriptor names are not the same: the descriptor is `size-adjust`. The value is any positive percentage value (from 0 to infinity) by which you want the fallback font scaled so it better matches the primary font selected. That percentage is used as a multiplier for the glyph outline sizes and other metrics of the font:

```
@font-face {
  font-family: myPreferredFont;
  src: url("longLoadingFont.otf");
}

@font-face {
  font-family: myFallBackFont;
  src: local(aLocalFont);
  size-adjust: 87.3%;
}
```



As of late 2022, the only user-agent line that does *not* support the `size-adjust` descriptor is the WebKit (Safari) family.

Font Style

The `font-style` property sounds very simple: you can choose from three values, and optionally provide an angle for oblique text if you're using it.

font-style

Values	`italic` | [`oblique` <*angle*>?] | `normal`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
@font-face equivalent	`font-style`
Variable axis	`"slnt"` (slant) or `"ital"` (italic)
Inherited	Yes
Animatable	Yes for variable fonts that define a ranged axis for italic or oblique; otherwise, no

The default value of `font-style` is `normal`. This value refers to *upright* text, which is best described as text that is not italic or otherwise slanted. For instance, the vast majority of text in this book is upright.

Italic font faces are usually a bit cursive in appearance, and generally use less horizontal space than the `normal` version of the same font. In standard fonts, italic text is a separate font face, with small changes made to the structure of each letter to account for the altered appearance. This is especially true of serif fonts because, in addition to the text characters "leaning," the serifs may be altered. Font faces with labels like `Italic`, `Cursive`, and `Kursiv` are usually mapped to the `italic` keyword.

Oblique text, on the other hand, is a slanted version of the normal, upright text. Oblique text is generally not altered from the upright text other than being given a slope. If a font has oblique versions, they are often in faces with labels such as `Oblique`, `Slanted`, and `Incline`.

When fonts don't have italic or oblique versions, the browser can simulate italic and oblique fonts by artificially sloping the glyphs of the regular face. (To prevent this from happening, use `font-synthesis: none`, covered later in the chapter.)

Italic and oblique text at the same angle are not the same: italic is stylized and usually obsessively designed, and oblique is merely slanted. By default, if `oblique` is declared without an angle, a value of `14deg` is used.

When oblique is given an angle, such as `font-style: oblique 25deg`, the browser selects the face classified as oblique, if available. If one or more oblique faces are available in the chosen font family, the one most closely matching the specified angle by the `font-style`

descriptor is chosen. If no oblique faces are available, the browser may synthesize an oblique version of the font by slanting a normal face by the specified angle.

Unless further limited by the font or the descriptor, the oblique angle specified must be between `90deg` and `-90deg`, inclusive. If the given value is outside those limits, the declaration is ignored. Positive values are slanted toward the end (inline-end) of the line, while negative values are slanted toward the beginning (inline-start) of the line.

To visualize the difference between italic and oblique text, refer to Figure 14-14.

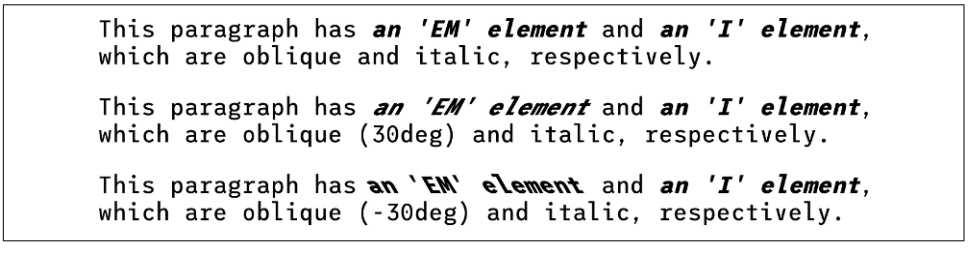


Figure 14-14. Italic and oblique text in detail

For TrueType or OpenType variable fonts, the `"slnt"` variation axis is used to implement varying slant angles for oblique, and the `"ital"` variation axis with a value of `1` is used to implement italic values. See "Font Variation Settings" on page 718 for more details.

If you want to make sure that a document uses italic text in familiar ways, you could write a stylesheet like this:

```
p {font-style: normal;}
em, i {font-style: italic;}
```

These styles would make paragraphs use an upright font, as usual, and cause the `<em>` and `<i>` elements to use an italic font, also as usual. On the other hand, you might decide that there should be a subtle difference between `<em>` and `<i>`:

```
p {font-style: normal;}
em {font-style: oblique;}
i {font-style: italic;}
b {font-style: oblique -8deg;}
```

If you look closely at Figure 14-15, you'll see no apparent difference between the `<em>` and `<i>` elements. In practice, not every font is so sophisticated as to have both an italic face and an oblique face, and even fewer web browsers are sophisticated enough to tell the difference when both faces do exist.

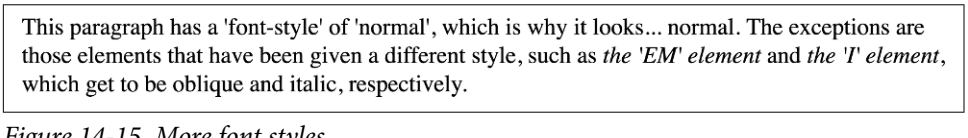


Figure 14-15. More font styles

The equivalent `font-variation-settings` setting for `italic` is `"ital"`. For the `oblique` <*angle*> value, the equivalent is `"slnt"`, which is used to vary between upright and slanted text. Just as with `font-style`, the slant axis is interpreted as the angle of slant in counterclockwise degrees from upright: inline-start-leaning oblique design will have a negative slant value, whereas inline-end-leaning needs a positive value.

The font-style Descriptor

As a descriptor, `font-style` lets an author link specific faces to specific font-style values.

font-style descriptor

Values	`normal` | `italic` | `oblique` <*angle*>{0,2}
Initial value	`auto`

For example, we might want to assign very particular faces of Switzera to the various kinds of `font-style` property values. Given the following, the result would be to render `<h2>` and `<h3>` elements using SwitzeraADF-Italic instead of SwitzeraADF-Regular, as illustrated in Figure 14-16:

```
@font-face {
   font-family: "Switzera";
   font-style: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: italic;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: oblique;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}

h1, h2, h3 {font-family: SwitzeraADF, Helvetica, sans-serif;}
h1 {font-size: 225%;}
h2 {font-size: 180%; font-style: italic;}
h3 {font-size: 150%; font-style: oblique;}
```

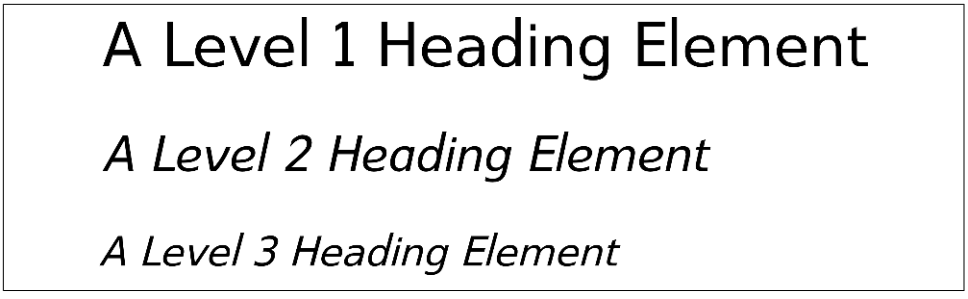


Figure 14-16. Using declared font-style faces

Ideally, if a SwitzeraADF face with an oblique typeface existed, a page author could point to it instead of the italic variant. There isn't such a face, though, so the author mapped the italic face to both the `italic` and `oblique` values. As with `font-weight`, the `font-style` descriptor can take all of the values of the `font-style` property *except* for `inherit`.

Oblique text changes the angle of letterforms without performing any kind of character substitution. Any variable font that supports oblique text also supports normal or upright text: upright text is oblique text at a `0deg` angle. For example:

```
@font-face {
  font-family: "varFont";
  src: url("aVariableFont.woff2") format("woff2-variations");
  font-weight: 1 1000;
  font-stretch: 75% 100%;
  font-style: oblique 0deg 20deg;
  font-display: swap;
}

body { font-family: varFont, sans-serif; font-style: oblique 0deg; }
em { font-style: oblique 14deg; }
```

The angle given in the CSS value `oblique 3deg` is a clockwise slant of 3 degrees. Positive angles are clockwise slants, whereas negative angles are counterclockwise slants. If no angle is included, it is the same as writing `oblique 14deg`. The degree angle can be any value between `-90deg` and `90deg`, inclusive.

Font Stretching

In some font families, variant faces have wider or narrower letterforms. These often take names like Condensed, Wide, and Ultra Expanded. The utility of such variants is that a designer can use a single font family while also having skinny and fat variants. CSS provides a property that allows an author to select among such variants, when they exist, without having to explicitly define them in `font-family` declarations. It does this via the somewhat misleadingly named `font-stretch`.

font-stretch

Values	`normal` | `ultra-condensed` | `extra-condensed` | `condensed` | `semi-condensed` | `semi-expanded` | `expanded` | `extra-expanded` | `ultra-expanded` | *<percentage>*
Initial value	`normal`
Applies to	All elements
@font-face equivalent	`font-stretch`
Variable axis	`"wdth"`
Inherited	Yes
Animatable	Yes in a variable font that defines a stretch axis; otherwise, no

You might expect from the property name that `font-stretch` will stretch or squeeze a font like saltwater taffy, but that's not the case. This property instead behaves very much like the absolute-size keywords (e.g., `xx-large`) for the `font-size` property. You can set a percentage between 50% and 200% inclusive, or use a range of keyword values that have defined percentage equivalents. Table 14-9 shows the mapping between keyword values and numeric percentages.

Table 14-9. Percentage equivalents for `font-stretch` keyword values

Keyword	Percentage
`ultra-condensed`	50%
`extra-condensed`	62.5%
`condensed`	75%
`semi-condensed`	87.5%
`normal`	100%
`semi-expanded`	112.5%
`expanded`	125%
`extra-expanded`	150%
`ultra-expanded`	200%

For example, you might decide to stress the text in a strongly emphasized element by changing the font characters to a wider face than its parent element's font characters.

The catch is that this property works only if the font family in use actually *has* wider and narrower faces, which mostly come with only expensive traditional fonts. (They're much more widely available in variable fonts.)

For example, consider the common font Verdana, which has only one width face; this is equivalent to `font-stretch: normal`. Declaring the following will have no effect on the width of the displayed text:

```
body {font-family: Verdana;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

All of the text will be at Verdana's usual width. However, if the font family is changed to one that has multiple width faces, such as Futura, things will be different, as shown in Figure 14-17:

```
body {font-family: Verdana;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

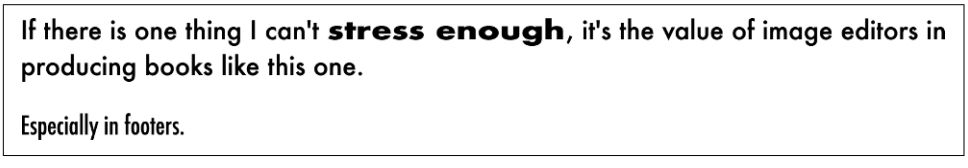


Figure 14-17. Stretching font characters

For variable fonts that support the `"wdth"` axis, set the width in `font-variation-settings` to a value greater than 0. This controls the glyph width or stroke thickness, depending on the font design.

The font-stretch Descriptor

Much as with the `font-weight` descriptor, the `font-stretch` descriptor allows you to explicitly assign faces of varying widths to the width values permitted in the `font-stretch` property. For example, the following rules explicitly assign three faces to the most directly analogous `font-stretch` values:

```
@font-face {
   font-family: "Switzera";
   font-stretch: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-stretch: condensed;
   src: url("SwitzeraADF-Cond.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-stretch: expanded;
   src: url("SwitzeraADF-Ext.otf") format("opentype");
}
```

In a parallel to what you saw in previous sections, you can call on these different width faces through the `font-stretch` property, as illustrated in Figure 14-18:

```
h1, h2, h3 {font-family: SwitzeraADF, Helvetica, sans-serif;}
h1 {font-size: 225%;}
h2 {font-size: 180%; font-stretch: condensed;}
h3 {font-size: 150%; font-stretch: expanded;}
```

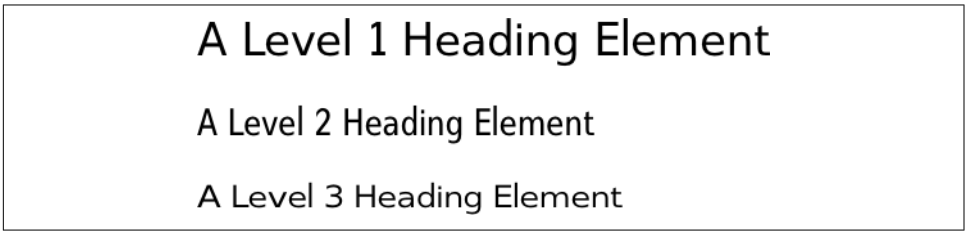


Figure 14-18. Using declared font-stretch faces

If you use a variable font that contains the full spectrum of font-stretch sizing, you can import a single font file with `@font-face`, then use it for all of your text font-stretch requirements. This produces the same degree of horizontal stretching shown in Figure 14-18, albeit with a different font:

```
@font-face {
  font-family: 'League Mono Var';
  src: url('LeagueMonoVariable.woff2') format('woff2');
  font-weight: 100 900;
  font-stretch: 50% 200%;
  font-display: swap;
}

h1, h2, h3 {font-family: "League Mono Var", Helvetica, sans-serif;}
h2 {font-size: 180%; font-stretch: 75%;}
h3 {font-size: 150%; font-stretch: 125%;}
```

The `font-stretch` descriptor can take all of the values of the `font-stretch` property *except* for `inherit`.

If you do want to use a different font for your variable fonts depending on whether the text is extended or condensed, use the `"wdth"` value in the comma-separated value of the `@font-face` `font-variation-settings` descriptor, as in the following example:

```
@font-face {
  font-family: 'League Mono Var';
  src: url('LeagueMonoVariable.woff2') format('woff2');
  font-weight: 100 900;
  font-stretch: 50% 200%;
}
strong {
  font-family: LeagueMono;
  font-variation-settings: "wdth" 100;
}
```

Font Synthesis

Sometimes a given font family will lack alternate faces for options like bold or italic text or small capital letters. In such situations, the user agent may attempt to synthesize a face from the faces it has available, but this can lead to unattractive letterforms. To address this, CSS offers `font-synthesis`, which lets you say how much synthesis you will or won't permit in the rendering of a page. This doesn't have a `@font-face` descriptor, but it has bearing on all the font variants to follow, so we're dealing with it now.

font-synthesis

Values	`none` | `weight` | `style` | `small-caps`
Initial value	`weight style`
Applies to	All elements
Inherited	Yes
Animatable	No

In many user agents, a font family that has no bold face can have one computed for it. This might be done by adding pixels to either side of each character glyph, for example. While this might seem useful, it can lead to results that are visually unappealing, especially at smaller font sizes. This is why most font families have bold faces included: the font's designer wanted to make sure that bolded text in that font looked good.

Similarly, a font family that lacks an italic face can have one synthesized by simply slanting the characters in the normal face. This tends to look even worse than synthesized bold faces, particularly when it comes to serif fonts. Compare the difference between the actual italic face included in Georgia and a synthesized italic version of Georgia (which we're calling "oblique" here), illustrated in Figure 14-19.

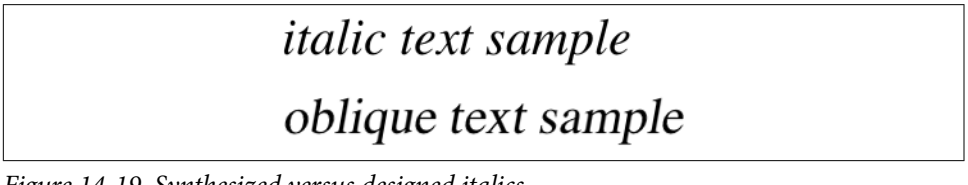


Figure 14-19. Synthesized versus designed italics

In supporting user agents, declaring `font-synthesis: none` blocks the user agent from doing any such synthesis for the affected elements. You can block it for the whole document with `html {font-synthesis: none;}`, for example. The downside is that any attempts to create variant text using a font that doesn't offer the appropriate faces will stay the normal face, instead of even approximating what was intended. The upside is that you

don't have to worry about a user agent trying to synthesize those variants and doing a poor job of it.

Font Variants

Beyond font weights and font styles, there are font variants. These are embedded within a font face and can cover aspects like various styles of historical ligatures, small-caps presentation, ways of presenting fractions, the spacing of numbers, whether zeros get slashes through them, and much more. CSS lets you invoke these variants, when they exist, through the shorthand property `font-variant`.

font-variant

Values	[<*font-variant-caps*> ‖ <*font-variant-numeric*> ‖ <*font-variant-alternates*> ‖ <*font-variant-ligatures*> ‖ <*font-variant-east-asian*>] | `normal` | `none`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
@font-face equivalent	`font-variant`
Inherited	Yes
Animatable	No

This property is shorthand for five separate properties, which we'll get to in just a moment. The most common values you'll find in the wild are `normal`, which is the default and describes ordinary text, and `small-caps`, which has existed since CSS1.

First, however, let's cover the two values that don't correspond to other properties:

`none`
: Disables all variants of any kind by setting `font-feature-ligatures` to `none` and all the other font variant properties to `normal`

`normal`
: Disables most variants by setting all the font variant properties, including `font-feature-ligatures`, to `normal`

Understanding the variant aspect of `small-caps` might help explain the idea of variants, making all the other properties easier to understand. The `small-caps` value calls for the use of small caps (`font-feature-settings: "smcp"`). Instead of upper- and lowercase letters, a small-caps font employs capital letters of different sizes. Thus, you might see something like what's shown in Figure 14-20:

```
h1 {font-variant: small-caps;}
h1 code, p {font-variant: normal;}

<h1>The Uses of <code>font-variant</code></h1>
<p>
The property <code>font-variant</code> is very interesting...
</p>
```

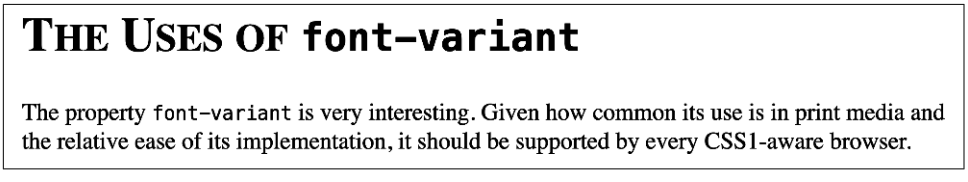


Figure 14-20. The `small-caps` value in use

As you may notice, in the display of the `<h1>` element, there is a larger capital letter wherever an uppercase letter appears in the source, and a small capital letter wherever there is a lowercase letter in the source. This is very similar to `text-transform: uppercase`, with the only real difference being that, here, the capital letters are of different sizes. However, the reason that `small-caps` is declared using a font property is that some fonts have a specific small-caps face, which a font property is used to select.

What happens if no font-face variant, such as `small-caps`, exists? The specification provides two options. The first is for the user agent to create a small-caps face by scaling capital letters on its own. The second is to make all letters uppercase and the same size, exactly as if the declaration `text-transform: uppercase` had been used instead. This is not an ideal solution but it is permitted.



Bear in mind that not every font supports every variant. For example, most Latin fonts won't support any of the East Asian variants. In addition, not every font will include support for, say, some of the numeric and ligature variants. Many fonts will support *none* of the variants.

To find out what a given font supports, you have to consult its documentation, or do a lot of testing if no documentation is available. Most commercial fonts do come with documentation, and most free fonts don't. Fortunately, some browser developer tools (not including Chromium browsers, as of late 2022) have a tab that provides information about font variants and feature settings.

Capital Font Variants

In addition to the `small-caps` value we just discussed, CSS has other capital-text variants. These are addressed via the property `font-variant-caps`.

font-variant-caps

Values	`normal` | `small-caps` | `all-small-caps` | `petite-caps` | `all-petite-caps` | `titling-caps` | `unicase`
Initial value	`normal`
Applies to	All elements
Computed value	Specified keyword
@font-face equivalent	`font-variant`
Inherited	Yes
Animatable	No

The default value is `normal`, which means no capital-letter variant is used. From there, we have the following options:

`small-caps`
: Renders all of the letters using capital letters. The capital letters for characters that are uppercase in the source text are the same height as uppercase letters. Characters that are lowercase in the text are rendered as smaller capitals, usually a bit taller than the font's x-height.

`all-small-caps`
: The same as `small-caps`, except all letters are rendered as smaller capitals, even those that are uppercase in the source text.

`petite-caps`
: Similar to `small-caps`, except the capitals used for lowercase letters are equal in height to, or even a bit shorter than, the font's x-height. If the font has no petite-caps variant, the result is likely to be the same as for `small-caps`.

`all-petite-caps`
: The same as `petite-caps`, except all letters are rendered as smaller capitals, even those that are uppercase in the source text.

`titling-caps`
: If a row has multiple uppercase letters, alternate capital forms are used to keep the letters from appearing too visually strong. Usually these are thinner versions of the normal capitals in the font.

`unicase`
: The text is rendered using a mixture of capital and noncapital letterforms, usually all the same height. This can vary widely even among the few fonts that offer this variant.

The following code is illustrated in Figure 14-21; note that the values marked with a dagger (†) were faked in one way or another:

```
.variant1 {font-variant-caps: small-caps;}
.variant2 {font-variant-caps: all-small-caps;}
.variant3 {font-variant-caps: petite-caps;}
.variant4 {font-variant-caps: all-petite-caps;}
.variant5 {font-variant-caps: titling-caps;}
.variant6 {font-variant-caps: unicase;}
```

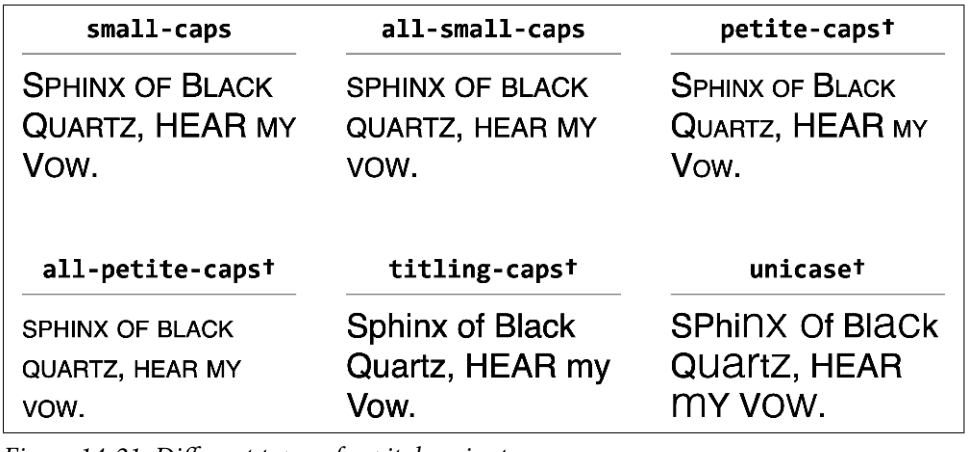


Figure 14-21. Different types of capital variants

Why did we fake some of the examples in Figure 14-21? In part, because finding a single font that contains all the capital variants is exceedingly difficult, and it is literally faster to fake some results than dig up a font, or set of fonts, that might work.

We also want to highlight that exact situation: most of the time, you're going to get either a fallback (as from `petite-caps` to `small-caps`) or no variant at all. Because of this, make sure to use the `@font-face` `font-variant` descriptor to define what should happen. Otherwise, if a `font-variant-caps` category variant is not available, the browser will decide how to render it. For example, if `petite-caps` is specified and the font doesn't have a petite-caps face or variable axis defined, the user agent may render the text using small capital glyphs. If small capital glyphs are not included in the font, the browser may synthesize them by proportionally shrinking uppercase glyphs.

Alternatively, you can use `{font-synthesis: none;}` to prevent the browser from synthesizing the text. You can also include `{font-synthesis: small-caps;}`, or omit `font-synthesis` altogether, to allow a small-caps typeface to be synthesized if needed.

Fonts sometimes include special glyphs for various caseless characters like punctuation marks to match the cap-variant text. The browser will not synthesize caseless characters on its own.

All the values of `font-variant-caps` other than `normal` have defined equivalent OpenType features. These are summarized in Table 14-10.

Table 14-10. `font-variant-caps` values and equivalent OpenType features

Value	OpenType feature
`normal`	*n/a*
`small-caps`	`"smcp"`
`all-small-caps`	`"c2sc"`, `"smcp"`
`petite-caps`	`"pcap"`
`all-petite-caps`	`"c2pc"`, `"pcap"`
`titling-caps`	`"titl"`
`unicase`	`"unic"`

Numeric Font Variants

Many font faces have variant behaviors for use when rendering numerals. When available, these can be accessed via the `font-variant-numeric` property. The values of this property affect the usage of alternate glyphs for numbers, fractions, and ordinal markers.

font-variant-numeric

Values	`normal` | [`lining-nums` | `oldstyle-nums`] || [`proportional-nums` | `tabular-nums`] || [`diagonal-fractions` | `stacked-fractions`] || `ordinal` || `slashed-zero`]
Initial value	`normal`
Applies to	All elements
Computed value	Specified keyword
Inherited	Yes
Animatable	No

The default value, `normal`, means that nothing special will be done when rendering numbers. They'll just appear the same as they usually do for the font face. Figure 14-22 demonstrates all the values, and as before, the examples marked with a dagger (†) were faked in one way or another because fonts lacked those features.

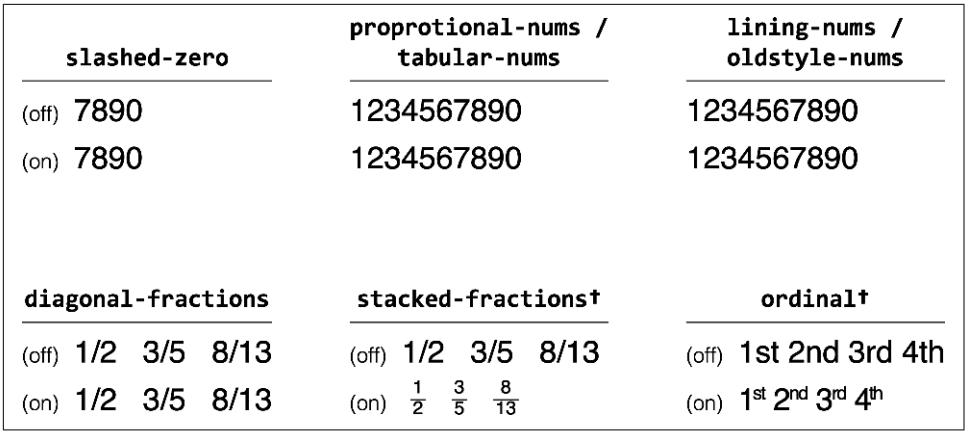


Figure 14-22. Different types of numeric variants

Perhaps the simplest numeric variant is `slashed-zero`. This causes the numeral 0 to appear with a slash through it, most likely on a diagonal. Slashed zeros are often the default rendering in monospace fonts, where distinguishing 0 from the capital letter *O* can be difficult. In serif and sans-serif fonts, they are usually not the default appearance of zeros. Setting `font-variant-numeric: slashed-zero` will bring out a slashed zero if one is available.

Speaking of diagonal slashes, the value `diagonal-fractions` causes characters arranged as a fraction (e.g., 1/2) to be rendered as smaller numbers, the first raised up, separated by a diagonal slash. The `stacked-fractions` value renders the fraction as the first number above the second, and the two separated by a horizontal slash.

If the font has features for ordinal labels, like the letters following the numbers of 1st, 2nd, 3rd, and 4th in English, `ordinal` enables the use of those special glyphs. These will generally look like superscripted, smaller versions of the letters.

Authors can affect the figures used for numbers with `lining-nums`, which sets all numbers on the baseline; and `oldstyle-nums`, which enables numbers like 3, 4, 7, and 9 to descend below the baseline. Georgia is a common example of a font that has old-style numbers.

You can also influence the sizing of figures used for numbers. The `proportional-nums` value enables the numbers to be proportional, as in proportional fonts; and `tabular-nums` gives all numbers the same width, as in monospace fonts. The advantage of these values is that you can, assuming there are glyphs to support them in the font face, get the monospace effect in proportional fonts without converting the numbers to a monospace face, and similarly cause monospace numbers to be sized proportionally.

You can include multiple values, but only one value from each of the numeric-value sets:

```
@font-face {
  font-family: 'mathVariableFont';
  src: local("math");
  font-feature-settings: "tnum" on, "zero" on;
}
.number {
  font-family: mathVariableFont, serif;
  font-feature-settings: "tnum" on, "zero" on;
  font-variant-numeric: ordinal slashed-zero oldstyle-nums stacked-fractions;
}
```

All the values of `font-variant-numeric` other than `normal` have defined equivalent OpenType features. These are summarized in Table 14-11.

Table 14-11. `font-variant-numeric` values and equivalent OpenType features

Value	OpenType feature
`normal`	*n/a*
`ordinal`	`"ordn"`
`slashed-zero`	`"zero"`
`lining-nums`	`"lnum"`
`oldstyle-nums`	`"onum"`
`proportional-nums`	`"pnum"`
`tabular-nums`	`"tnum"`
`diagonal-fractions`	`"frac"`
`stacked-fractions`	`"afrc"`

Ligature Variants

A *ligature* is a joining of two (or more) characters into one shape. As an example, two lowercase *f* characters could have their crossbars merged into a single line when they appear next to each other, or the crossbar could extend over a lowercase *i* and replace its usual dot in the sequence *fi*. More archaically, a combination like *st* could have a swash curve from one to the other. When available, these features can be enabled or disabled with the `font-variant-ligatures` property.

font-variant-ligatures

Values	`normal` | `none` | [[`common-ligatures` | `no-common-ligatures`] || [`discretionary-ligatures` | `no-discretionary-ligatures`] || [`historical-ligatures` | `no-historical-ligatures`] || [`contextual` | `no-contextual`]]
Initial value	`normal`

Applies to	All elements
Computed value	Specified keyword
Inherited	Yes
Animatable	No

The values have the following effects:

`common-ligatures`
: Enables the use of common ligatures, such as those combining *f* or *t* with letters that follow them. In French, the sequence *oe* is more usually rendered using the ligature *œ*. Browsers usually have these enabled by default, so if you want to disable them, use `no-common-ligatures` instead.

`discretionary-ligatures`
: Enables the use of special ligatures created by font designers that are unusual or otherwise not regarded as common.

`historical-ligatures`
: Enables the use of historical ligatures, which are generally those found in the typography of centuries past but are not used today. For example, in German the *tz* digraph used to be rendered as ſʒ.

`contextual-ligatures`
: Enables the use of ligatures that appear based on context, such as a cursive font enabling connecting curves from one letter to the next depending on not just the character that follows, but possibly also what characters came before. These are also sometimes used in programming fonts, where sequences like `!=` may be rendered as ≠ instead.

`no-common-ligatures`
: Explicitly disables the use of common ligatures.

`no-discretionary-ligatures`
: Explicitly disables the use of discretionary ligatures.

`no-historical-ligatures`
: Explicitly disables the use of historical ligatures.

`no-contextual-ligatures`
: Explicitly disables the use of contextual ligatures.

The default value, `normal`, turns off all these ligatures *except* common ligatures, which are enabled by default. This is especially relevant because `font-variant: normal` turns off all the `font-variant-ligatures` except the common ones, whereas `font-variant: none`

turns them all off *including* common ligatures. Table 14-12 provides a condensed summary of how each value translates into OpenType features.

Table 14-12. `font-variant-ligatures` values and equivalent OpenType features

Value	OpenType feature
`common-ligatures`	`"clig" on, "liga" on`
`discretionary-ligatures`	`"dlig" on`
`historical-ligatures`	`"hlig" on`
`contextual-ligatures`	`"calt" on`
`no-common-ligatures`	`"clig" off, "liga" off`
`no-discretionary-ligatures`	`"dlig" off`
`no-historical-ligatures`	`"hlig" off`
`no-contextual-ligatures`	`"calt" off`

Less likely to be used or supported by browsers are the `font-variant-alternates` and `font-variant-east-asian` properties.

Alternate Variants

For any given character, a font may include alternate glyphs in addition to the default glyph for that character. The `font-variant-alternates` property affects the usage of those alternate glyphs.

font-variant-alternates

Values	`normal` | `[` `historical-forms` || `stylistic()` || `historical-forms` || `styleset()` || `character-variant()` || `swash()` || `ornaments()` || `annotation()` `]`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Discrete

The default value, `normal`, means don't use any alternate variants. The `historical-forms` keyword enables historical forms, glyphs that were common in the past but not today. All the other values are functions.

These alternate glyphs may be referenced by alternative names defined in `@font-feature-values`. With `@font-feature-values`, you can define a common name for the `font-variant-alternates` function values to activate OpenType features.

The `@font-feature-values` at-rule may be used either at the top level of your CSS or inside any CSS conditional-group at-rule.

In Table 14-13, *XY* is replaced by a number representing the feature set. With OpenType fonts and `font-feature-settings`, some features are already defined. For example, the OpenType equivalent of the `styleset()` function is "ss*XY*". As of late 2022, `ss01` through `ss20` are currently defined. Values higher than 99 are allowed, but they don't map to any OpenType values and will be ignored.

Table 14-13. `font-variant-alternates` values and equivalent OpenType features

Value	OpenType feature
`annotation()`	`"nalt"`
`character-variant()`	`"cv`*XY*`"`
`historical-forms`	`"hist"`
`ornaments()`	`"ornm"`
`styleset()`	`"ss`*XY*`"`
`stylistic()`	`"salt"`
`swash()`	`"swsh", "cswh"`

An at-rule version of `font-variant-alternates`, called `@font-feature-values`, allows authors to define labels for alternate values of `font-variant-alternates` using at-rules of their own. The following two styles (taken from the CSS specification) demonstrate how to label the numeric values of the `swash` alternate, and then use them later in `font-variant-alternates`:

```
@font-feature-values Noble Script { @swash { swishy: 1; flowing: 2; } }

p {
  font-family: Noble Script;
  font-variant-alternates: swash(flowing); /* use swash alternate #2 */
}
```

Without the presence of the `@font-feature-values` at-rule, the paragraph styles would have to say `font-variant-alternates: swash(2)` instead of using `flowing` for the value of the `swash` function.



As of late 2022, while all browsers support `font-variant` and its associated subproperties, only Firefox and Safari have `font-variant-alternates` and `@font-feature-values` support. You can more reliably set these variants by using the `font-feature-settings` property.

East Asian Font Variants

The values of the `font-variant-east-asian` property allow for controlling glyph substitution and sizing in East Asian text.

font-variant-east-asian

Values	`normal` | [[`jis78` | `jis83` | `jis90` | `jis04` | `simplified` | `traditional`] || [`full-width` | `proportional-width`] || `ruby`]
Initial value	`normal`
Applies to	All elements
Computed value	Specified keyword
Inherited	Yes
Animatable	No

The assorted Japanese Industrial Standard (JIS) variants reflect the glyph forms defined in different Japanese national standards. Fonts generally include glyphs defined by the most recent national standard. JIS values allow for the inclusion of older Japanese glyph variations when such variants are needed, such as when reproducing historical documents.

Similarly, the `simplified` and `traditional` values allow control over the glyph forms for characters that have been simplified over time but for which the older, traditional form is still used in some contexts.

The `ruby` value enables display of Ruby variant glyphs. Ruby text is generally smaller than the associated body text.

This property value allows font designers to include glyphs better suited for smaller typography than scaled-down versions of the default glyphs would be. Only glyph selection is affected; there is no associated font scaling.

Font Variant Position

Compared to the previous variants, `font-variant-position` is fairly straightforward. It's strange, then, that it's so poorly supported.

font-variant-position

Values	`normal` | `sub` | `super`
Initial value	`normal`
Applies to	All elements
Computed value	Specified keyword

Inherited	Yes
Animatable	No

This property can be used to enable specialized variant glyphs that are meant solely for superscripted and subscripted text. As it says in the CSS specification, these glyphs are:

> …designed within the same em-box as default glyphs and are intended to be laid out on the same baseline as the default glyphs, with no resizing or repositioning of the baseline. They are explicitly designed to match the surrounding text and to be more readable without affecting the line height.

This is in contrast to what happens with super- and subscripted text in fonts that lack such alternates, which is usually just smaller text that's been shifted up or down from the baseline. This sort of synthesis of super- and subscripted text often leads to line-height increases, which variant glyphs are generally designed to prevent.

Font Feature Settings

Throughout this chapter, we've discussed font features but have yet to cover the `font-feature-settings` property or descriptor. Similarly to `font-variant`, `font-feature-settings` allows you to exercise low-level control over which OpenType font features are available for use.

font-feature-settings

Values	`normal` | <*`feature-tag-value`*>`#`
Initial value	`normal`

The `font-feature-settings` property controls advanced typographic features in OpenType fonts, as opposed to the `font-variation-settings` property, which provides low-level control over variable font characteristics.

You can list one or more comma-separated OpenType features, as defined by the OpenType specification. For example, enabling common ligatures, small caps, and slashed zeros would look something like this:

```
font-feature-settings: "liga" on, "smcp" on, "zero" on;
```

The exact format of a <*`feature-tag-value`*> value is as follows:

<*`feature-tag-value`*>
  <*`string`*> [<*`integer`*> | `on` | `off`]?

For many features, the only permitted integer values are `0` and `1`, which are equivalent to `off` and `on` (and vice versa). Some features allow a range of numbers, however, in which case values greater than 1 both enable the feature and define the feature's selection index. If a feature is listed but no number is provided, `1` (on) is assumed. Thus, the following descriptors are all equivalent:

```
font-feature-settings: "liga";     /* 1 is assumed */
font-feature-settings: "liga" 1;   /* 1 is declared */
font-feature-settings: "liga" on;  /* on = 1 */
```

Remember that all <*string*> values *must* be quoted. Thus, the first of the following descriptors will be recognized, but the second will be ignored:

```
font-feature-settings: "liga", dlig;
/* common ligatures are enabled; we wanted discretionary ligatures, but forgot
   quotes, so they are not enabled */
```

A further restriction is that OpenType requires that all feature tags be four ASCII characters long. Any feature name longer or shorter, or that uses non-ASCII characters, is invalid and will be ignored. (This isn't something you need to worry about unless you're using a font that has its own made-up feature names and the font's creator didn't follow the naming rules.)

By default, OpenType fonts *always* have the following features enabled unless the author explicitly disables them via `font-feature-settings` or `font-variant`:

`"calt"`
: Contextual alternates

`"ccmp"`
: Composed characters

`"clig"`
: Contextual ligatures

`"liga"`
: Standard ligatures

`"locl"`
: Localized forms

`"mark"`
: Mark-to-base positioning

`"mkmk"`
: Mark-to-mark positioning

`"rlig"`
: Required ligatures

Additionally, other features may be enabled by default in specific situations, such as vertical alternatives (`"vert"`) for vertical runs of text.

The OpenType `font-feature-setting` values we've discussed so far are all listed in Table 14-14, along with a few others we didn't touch on for lack of support.

Table 14-14. OpenType values

Code	Meaning	Longhand
`"afrc"`	Alternative fractions	`stacked-fractions`
`"c2pc"`	Petite capitals	`petite-caps`
`"c2sc"`	Small capitals from capitals	`all-small-caps`
`"calt"`	Contextual alternates	`contextual`
`"case"`	Case-sensitive forms	
`"clig"`	Common ligatures	`common-ligatures`
`"cswh"`	Swash function	`swash()`
`"cv01"`	Character variants (01–99)	`character-variant()`
`"dnom"`	Denominators	
`"frac"`	Fractions	`diagonal-fractions`
`"fwid"`	Full-width variants	`full-width`
`"hist"`	Enable historical forms	`historical-forms`
`"liga"`	Standard ligatures	`common-ligatures`
`"lnum"`	Lining figures	`lining-nums`
`"locl"`	Localized forms	
`"numr"`	Numerators	
`"nalt"`	Annotation function	`annotation()`
`"onum"`	Old-style figures	`oldstyle-nums`
`"ordn"`	Ordinal markers	`ordinal`
`"ornm"`	Ornaments (function)	`ornaments()`
`"pcap"`	Petite capitals	`petite-caps`
`"pnum"`	Proportional figures	
`"pwid"`	Proportionally spaced variants	`proportional-width`
`"ruby"`	Ruby	`ruby`
`"salt"`	Stylistic function	`stylistic()`
`"sinf"`	Scientific inferiors	
`"smcp"`	Small capitals	`small-caps`
`"smpl"`	Simplified forms	`simplified`
`"ss01"`	Stylistic set 1 (numero correct)	`styleset()`
`"ss07"`	Stylistic set (1–20)	`styleset()`

Code	Meaning	Longhand
"subs"	Subscript	
"sups"	Superscript	
"swsh"	Swash function	swash()
"titl"	Titling capitals	titling-caps
"tnum"	Tabular figures	tabular-nums
"trad"	Traditional forms	traditional
"unic"	Unicase	unicase
"zero"	Slashed zero	slashed-zero

The complete list of standard OpenType feature names can be found at Microsoft's Registered Features page.

That said, font-feature-settings is a low-level feature designed to handle special cases for which no other way exists to enable or access an OpenType font feature. You also have to list all of the feature settings you want to use in a single property value. Whenever possible, use the font-variant shorthand property or one of the six associated longhand properties: font-variant-ligatures, font-variant-caps, font-variant-east-asian, font-variant-alternates, font-variant-position, and font-variant-numeric.

The font-feature-settings Descriptor

The font-feature-settings descriptor lets you decide which of an OpenType font face's settings can or cannot be used, specified as a space-separated list. Now, wait a second—isn't that almost exactly what we did with font-variant just a few paragraphs ago? Yes! The font-variant descriptor covers nearly everything font-feature-settings does, plus a little more besides. It just does so in a more CSS-like way, with value names instead of cryptic OpenType identifiers and Boolean toggles. Because of this, the CSS specification explicitly encourages authors to use font-variant instead of font-feature-settings, except when there's a font feature that the value list of font-variant doesn't include.

Keep in mind that this descriptor merely makes features available for use (or suppresses their use). It does not turn them on for the display of text; for that, see "Font Feature Settings" on page 714.

Just as with the font-variant descriptor, the font-feature-settings descriptor defines which font features are enabled (or disabled) for the font face being declared in the @font-face rule. For example, given the following, Switzera will have alternative fractions and small-caps disabled, even if such features exist in SwitzeraADF:

```
@font-face {
  font-family: "Switzera";
  font-weight: normal;
  src: url("SwitzeraADF-Regular.otf") format("opentype");
  font-feature-settings: "afrc" off, "smcp" off;
}
```

The `font-feature-settings` descriptor can take all of the values of the `font-feature-settings` property *except* for `inherit`.

Font Variation Settings

The `font-variation-settings` property provides low-level control over variable font characteristics, by specifying a four-letter axis name along with a value.

font-variation-settings

Values	`normal` | [<*string*> <*number*>]#
Initial value	`normal`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

There are five registered axes, listed in Table 14-15. We have covered almost all of them.

Table 14-15. Font variation axes

Axis	Property	Property value
`"wght"`	`font-weight`	`1` – `1000`
`"slnt"`	`font-style`	`oblique` / `oblique` <*angle*>
`"ital"`	`font-style`	`italic`
`"opsz"`	`font-optical-sizing`	
`"wdth"`	`font-stretch`	

We use the term *registered axes* because font developers are not limited to weight, width, optical size, slant, and italics: they can create custom axes, and "register" them by giving them a four-letter label. The simplest way to know if a font has such axes is to look at the font's documentation; otherwise, you have to know how to dig into the internals of a font's file(s) to find out. These axes can control any aspect of the font's appearance, such as the size of the dot on lowercase *i* and *j*. Creating custom axes is beyond the scope of this book, but calling on them where they exist is not.

Because these axes are string values, they have to be quoted, are case-sensitive, and are always lowercase. Imagine a font for which the size of the dots (which are properly called *diacritic marks* or just *diacritics*) over lowercase *i* and *j* can be changed by way of an axis called `DCSZ` (for *diacritic size*). Furthermore, this axis has been defined by the font's designer to allow values from 1 to 10. The diacritic size could be maximized as follows:

```
p {font-family: DotFont, Helvetica, serif; font-variation-settings: "DCSZ" 10;}
```

The `font-variation-settings` descriptor is the same as the property. Instead of declaring each registered axis separately, they are declared on one line, comma separated:

```
@font-face {
  font-family: 'LeagueMono';
  src: url('LeagueMonoVariable.woff2') format('woff2');
  font-weight: 100 900;
  font-stretch: 50% 200%;
  font-variation-settings: 'wght' 100 900, 'wdth' 50 200;
  font-display: swap;
}
```



Although you can set the weight, style, and so forth of a given font by using `font-variation-settings`, it is recommended that you use the more widely supported and human-readable properties `font-weight` and `font-style` instead.

Font Optical Sizing

Text rendered at different sizes often benefits from slightly different visual representations. For example, to aid reading at small text sizes, glyphs have less detail and strokes are often thicker with larger serifs. Larger text can have more features and a greater contrast between thicker and thinner strokes. The property `font-optical-sizing` allows authors to enable or disable this feature of variable fonts.

font-optical-sizing

Values	`auto` | `none`
Initial value	`auto`
Applies to	All elements and text
Computed value	As specified
Variable font axis	`"opsz"`
Inherited	Yes
Animatable	Discrete

By default (via `auto`), browsers can modify the shape of glyphs based on font size and pixel density. The `none` value tells the browser to *not* do this.



In fonts that support it, optical sizing is usually defined as a range of numbers. If you want to explicitly change the optical sizing of a given element's font to be a specific number, perhaps to make text sturdier or more delicate than it would be by default, use the `font-variation-settings` property and give it a value like `'opsz' 10` (where `10` can be any number in the optical-sizing range).

Override Descriptors

This brings us to the last three `@font-face` descriptors that we have yet to discuss. Three descriptors enable override settings for font families: `ascent-override`, `descent-override`, and `line-gap-override`, which define the ascent, descent, and line gap metrics, respectively. All three descriptors take the same values: `normal` or a *<percentage>*.

ascent-override, descent-override, line-gap-override descriptors

Values	`normal` | *<percentage>*
Initial value	`normal`

The goal of these descriptors is to help fallback fonts better match a primary font by overriding the metrics of the fallback font and using those of the primary font instead.

The *ascent metric* is the distance above the baseline used to lay out line boxes (the distance from the baseline to the top of the em box). The *descent metric* is the distance below the baseline used to lay out line boxes (the distance from the baseline to the bottom of the em box). The *line-gap metric* is the font's recommended distance between adjacent lines of text, which is sometimes called *external leading*.

Here's an example of a hypothetical font and its ascent, descent, and line-gap override descriptors:

```
@font-face {
  font-family: "PreferredFont";
  src: url("PreferredFont.woff");
}

@font-face {
  font-family: FallbackFont;
  src: local(FallbackFont);
  ascent-override: 110%;
  descent-override: 95%;
  line-gap-override: 105%;
}
```

This will direct the browser to alter the ascent and descent heights by 110% and 95%, respectively, and increase the line gap to 105% the distance in the fallback font.

Font Kerning

A font property that doesn't have a descriptor equivalent is `font-kerning`. Some fonts contain data indicating how characters should be spaced relative to one another, known as *kerning*. Kerning can make character spacing more visually appealing and pleasant to read.

Kerning space varies depending on the way characters are combined; for example, the character pair *oc* may have a different spacing than the pair *ox*. Similarly, *AB* and *AW* may have different separation distances, to the point that in some fonts, the top-right tip of the *W* is actually placed to the left of the bottom-right tip of the *A*. This kerning data can be explicitly called for or suppressed using the property `font-kerning`.

font-kerning

Values	`auto` | `normal` | `none`
Initial value	`auto`
Applies to	All elements
Inherited	Yes
Animatable	No

The value `none` is pretty simple: it tells the user agent to ignore any kerning information in the font. The `normal` value tells the user agent to kern the text normally—that is, according to the kerning data contained in the font. The `auto` value tells the user agent to do whatever it thinks best, possibly depending on the type of font in use. The OpenType specification, for example, recommends (but does not require) that kerning be applied whenever the font supports it. Furthermore, as per the CSS specification:

> [Browsers] may synthetically support the kern feature with fonts that contain kerning data in the form of a `kern` table but lack kern feature support in the `GPOS` table.

This means, in effect, that if kerning information is built into the font, browsers are allowed to enforce it even if the font lacks an explicit enabling of kerning via a feature table.



If the `letter-spacing` property (see Chapter 15) is applied to kerned text, the kerning is done first and *then* the letters' spacing is adjusted according to the value of `letter-spacing`, not the other way around.

The font Property

All of the properties discussed thus far are very sophisticated, but writing them all out could get a little tedious:

```
h1 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 30px;
    font-weight: 900; font-style: italic; font-variant-caps: small-caps;}
h2 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 24px;
    font-weight: bold; font-style: italic; font-variant-caps: normal;}
```

Some of this problem could be solved by grouping selectors, but wouldn't it be easier to combine everything into a single property? Enter `font`, which is a shorthand property encompassing most (not quite all) of the other font properties, and a little more besides.

font

Values	[[<*font-style*> || [`normal` | `small-caps`] || <*font-weight*> || <*font-stretch*>]? <*font-size*> [/ <*line-height*>]? <*font-family*>] | `caption` | `icon` | `menu` | `message-box` | `small-caption` | `status-bar`
Initial value	Refer to individual properties
Applies to	All elements
Percentages	Calculated with respect to the parent element for <*font-size*> and with respect to the element's <*font-size*> for <*line-height*>
Computed value	See individual properties (`font-style`, etc.)
Inherited	Yes
Animatable	Refer to individual properties

Generally speaking, a `font` declaration can have any one value from each of the listed font properties, or else a system font value (described in "Using System Fonts" on page 726). Therefore, the preceding example could be shortened as follows (and have exactly the same effect, as illustrated by Figure 14-23):

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold normal italic 24px Verdana, Helvetica, Arial, sans-serif;}
```



Figure 14-23. Typical font rules

We say that the styles "could be" shortened in this way because a few other possibilities exist, thanks to the relatively loose way in which `font` can be written. If you look closely at the preceding example, you'll see that the first three values don't occur in the same order. In the `h1` rule, the first three values are for `font-style`, `font-weight`, and `font-variant`, in that order. In the second, they're ordered `font-weight`, `font-variant`, and `font-style`. There is nothing wrong here because these three can be written in any order. Furthermore, if any has a value of `normal`, that can be left out altogether. Therefore, the following rules are equivalent to the previous example:

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

In this example, the value of `normal` is left out of the `h2` rule, but the effect is exactly the same as in the preceding example.

It's important to realize, however, that this free-for-all situation applies only to the first three values of `font`. The last two are much stricter in their behavior. Not only must `font-size` and `font-family` appear in that order as the last two values in the declaration, but both must always be present in a `font` declaration. Period, end of story. If either is left out, the entire rule will be invalidated and will be ignored completely by a user agent. Thus, the following rules will get you the result shown in Figure 14-24:

```
h1 {font: normal normal italic 30px sans-serif;}   /* no problem here */
h2 {font: 1.5em sans-serif;}   /* also fine; omitted values set to 'normal' */
h3 {font: sans-serif;}     /* INVALID--no 'font-size' provided */
h4 {font: lighter 14px;}   /* INVALID--no 'font-family' provided */
```

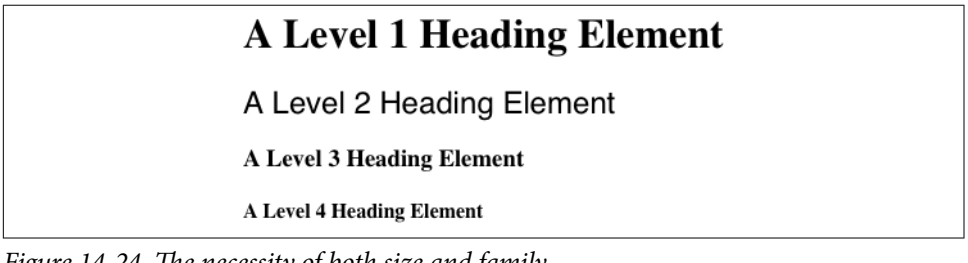


Figure 14-24. The necessity of both size and family

Understanding font Property Limitations

Because the `font` property has been part of CSS since the very beginning, and because so many properties dealing with all the variants came later, the `font` property has some limitations when it comes to font variations.

First, it's important to remember that when using the `font` shorthand property, the following properties are all set to their default values even though they cannot be represented in `font`:

- `font-feature-settings`
- `font-kerning`
- `font-language-override`
- `font-optical-sizing`
- `font-palette`
- `font-size-adjust`
- `font-variant-alternates`
- `font-variant-caps` (unless `small-caps` is included in the `font` value)
- `font-variant-east-asian`
- `font-variant-ligatures`
- `font-variant-numeric`
- `font-variation-settings`

Second, and following on the note in the previous list, only two variation values are permitted: `small-caps` and `normal`. The numeric, ligature, alternate, East Asian, and many of the caps variants cannot be set via the `font` property. If you want, for example, to use small caps and slashed zeros in your top-level headings, you would need to write something like this:

```
h1 {font: bold small-caps 3em/1.1 Helvetica, sans-serif;
    font-variant-numeric: slashed-zero;
```

Third, another property value that suffers from the weight of history is font stretching. As we discussed earlier in the chapter, `font-stretch` allows you to choose from numerous keywords or to set a percentage in the range of 50% to 200% (inclusive). The keywords may be used in `font`, but the percentage value may not.

Adding the Line Height

We also can set the value of the `line-height` property by using `font`, even though `line-height` is a text property (not covered in this chapter), not a font property. It's done as a sort of addition to the `font-size` value, separated from it by a forward slash (`/`):

```
body {font-size: 12px;}
h2 {font: bold italic 200%/1.2 Verdana, Helvetica, Arial, sans-serif;}
```

These rules, demonstrated in Figure 14-25, set all `<h2>` elements to be bold and italic (using face for one of the sans-serif font families), set the `font-size` to `24px` (twice the `body`'s size), and set the `line-height` to `28.8px`.

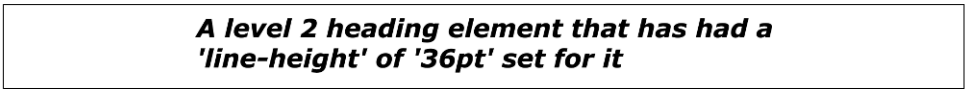


Figure 14-25. Adding line height to the mix

This addition of a value for `line-height` is entirely optional, just as the first three `font` values are. If you do include a `line-height` value, remember that `font-size` always comes before `line-height`, never after, and the two are always separated by a slash.



This may seem repetitive, but it's one of the most common errors made by CSS authors, so we can't say it enough: the required values for `font` are `font-size` and `font-family`, in that order. Everything else is strictly optional.

Using the Shorthand Properly

It is important to remember that `font`, being a shorthand property, can act in unexpected ways if you are careless with its use. Consider the following rules, which are illustrated in Figure 14-26:

```
h1, h2, h3 {font: italic small-caps 250% sans-serif;}
h2 {font: 200% sans-serif;}
h3 {font-size: 150%;}

<h1>A level 1 heading element</h1>
<h2>A level 2 heading element</h2>
<h3>A level 3 heading element</h3>
```

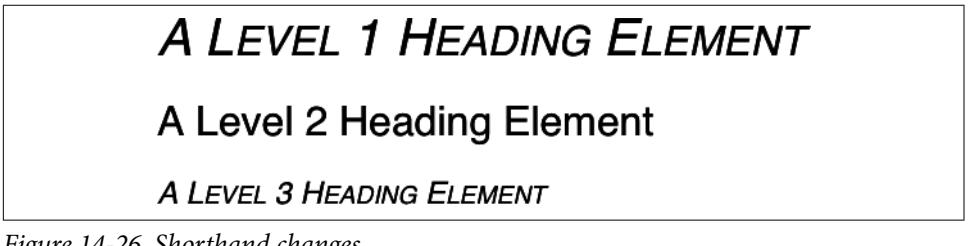


Figure 14-26. Shorthand changes

Did you notice that the <h2> element is neither italicized nor small-capped, and that none of the elements are bold? This is the correct behavior. When the shorthand property font is used, any omitted values are reset to their defaults. Thus, the previous example could be written as follows and still be exactly equivalent:

```
h1, h2, h3 {font: italic normal small-caps 250% sans-serif;}
h2 {font: normal normal normal 200% sans-serif;}
h3 {font-size: 150%;}
```

This sets the <h2> element's font style and variant to normal, and the font-weight of all three elements to normal. This is the expected behavior of shorthand properties. The <h3> does not suffer the same fate as the <h2> because you use the property font-size, which is not a shorthand property and therefore affects only its own value.

Using System Fonts

When you want to make a web page blend in with the user's operating system, the system font values of font come in handy. These are used to take the font size, family, weight, style, and variant of elements of the operating system, and apply them to an element. The values are as follows:

caption
: Used for captioned controls, such as buttons

icon
: Used to label icons

menu
: Used in menus—that is, drop-down menus and menu lists

message-box
: Used in dialog boxes

small-caption
: Used for labeling small controls

status-bar
: Used in window status bars

For example, you might want to set the font of a button to be the same as that of the buttons found in the operating system. For example:

```
button {font: caption;}
```

With these values, you can create web-based applications that look very much like applications native to the user's operating system.

Note that system fonts may be set only as a whole; that is, the font family, size, weight, style, etc., are all set together. Therefore, the button text from our previous example will look exactly the same as button text in the operating system, whether or not the size matches any of the content around the button. You can, however, alter the individual values after the system font has been set. Thus, the following rule will make sure the button's font is the same size as its parent element's font:

```
button {font: caption; font-size: 1em;}
```

If you call for a system font and no such font exists on the user's machine, the user agent may try to find an approximation, such as reducing the size of the `caption` font to arrive at the `small-caption` font. If no such approximation is possible, the user agent should use a default font of its own. If it can find a system font but can't read all of its values, it should use the default value. For example, a user agent may be able to find a `status-bar` font but not get any information about whether the font is small caps. In that case, the user agent will use the value `normal` for the `small-caps` property.

Font Matching

As you've seen, CSS allows for the matching of font families, weights, and variants. This is all accomplished through font matching, which is a vaguely complicated procedure. Understanding it is important for authors who want to help user agents make good font selections when displaying their documents. We left it for the end of the chapter because it's not really necessary to understand how the font properties work, and some readers will probably want to skip this part. If you're still interested, here's how font matching works:

1. The user agent creates, or otherwise accesses, a database of font properties. This database lists the various CSS properties of all the fonts to which the user agent has access. Typically, this will be all fonts installed on the machine, although there could be others (for example, the user agent could have its own built-in fonts). If the user agent encounters two identical fonts, it will ignore one of them.
2. The user agent takes apart an element to which font properties have been applied and constructs a list of font properties necessary for the display of that element. Based on that list, the user agent makes an initial choice of a font family to use in displaying the element. If there is a complete match, the user agent can use that font. Otherwise, the user agent needs to do a little more work.
3. A font is first matched against the `font-stretch` property.
4. A font is next matched against the `font-style` property. The keyword `italic` is matched by any font that is labeled as either `italic` or `oblique`. If neither is available, the match fails.
5. The next match is to `font-weight`, which can never fail thanks to the way `font-weight` is handled in CSS (explained in "How Weights Work" on page 682).

6. Then, `font-size` is tackled. This must be matched within a certain tolerance, but that tolerance is defined by the user agent. Thus, one user agent might allow matching within a 20% margin of error, whereas another might allow only 10% differences between the size specified and the size that is actually used.
7. If no font matched in step 2, the user agent looks for alternate fonts within the same font family. If it finds any, it repeats step 2 for that font.
8. Assuming a generic match has been found but doesn't contain everything needed to display a given element—the font is missing the copyright symbol, for instance—the user agent goes back to step 3, which entails a search for another alternate font and another trip through step 2.
9. Finally, if no match has been made and all alternate fonts have been tried, the user agent selects the default font for the given generic font family and does the best it can to display the element correctly.

Furthermore, the user agent does the following to resolve the handling of font variants and features:

1. Check for font features enabled by default, including features required for a given script. The core set of default-enabled features is `"calt"`, `"ccmp"`, `"clig"`, `"liga"`, `"locl"`, `"mark"`, `"mkmk"`, and `"rlig"`.
2. If the font is defined via an `@font-face` rule, check for the features implied by the `font-variant` descriptor in the `@font-face` rule. Then check for the font features implied by the `font-feature-settings` descriptor in the `@font-face` rule.
3. Check feature settings determined by properties other than `font-variant` or `font-feature-settings`. (For example, setting a nondefault value for the `letter-spacing` property will disable ligatures.)
4. Check for features implied by the value of the `font-variant` property, the related `font-variant` subproperties (e.g., `font-variant-ligatures`), and any other property that may call for the use of OpenType features (e.g., `font-kerning`).
5. Check for the features implied by the value of the `font-feature-settings` property.

The whole process is long and tedious, but it helps to understand how user agents pick the fonts they do. For example, you might specify the use of Times or any other serif font in a document:

```
body {font-family: Times, serif;}
```

For each element, the user agent should examine the characters in that element and determine whether Times can provide characters to match. In most cases, it can do so with no problem.

Assume, however, that a Chinese character has been placed in the middle of a paragraph. Times has nothing that can match this character, so the user agent has to work around the character or look for another font that can fulfill the needs of displaying that element. Any

Western font is highly unlikely to contain Chinese characters, but should one exist (let's call it AsiaTimes), the user agent could use it in the display of that one element—or simply for the single character. Thus, the whole paragraph might be displayed using AsiaTimes, or everything in the paragraph might be in Times except for the single Chinese character, which is displayed in AsiaTimes.

Summary

From what was initially a very simple set of font properties, CSS has grown to allow fine-grained and wide-ranging influence over the way fonts are displayed on the web. From custom fonts downloaded over the web to custom-built families assembled out of a variety of individual faces, authors may be fairly said to overflow with font power.

The typographic options available to authors today are far stronger than ever, but always remember: you must use this power wisely. While you can have 17 fonts in use on your site, that definitely doesn't mean that you should. Quite aside from the aesthetic difficulties this could present for your users, it would also make the total page weight much, much higher than it needs to be. As with any other aspect of web design, you are advised to use your power wisely, not wildly.

CHAPTER 15

Text Properties

Because text is so important, many CSS properties affect it in one way or another. But didn't we just cover that in Chapter 14? Not exactly: we covered only fonts—the importing and usage of typefaces. Text styles are different.

OK, so what is the difference between text and fonts? At the simplest level, *text* is the content, and *fonts* are used to display that content. Fonts provide the shape for the letters. Text is the styling around those shapes. Using text properties, you can affect the position of text in relation to the rest of the line, superscript it, underline it, and change the capitalization. You can affect the size, color, and placement of text decorations.

Indentation and Inline Alignment

Let's start with a discussion of how you can affect the inline positioning of text within a line. Think of these basic actions as the same types of steps you might take to create a newsletter or write a report.

Originally, CSS was based on concepts of *horizontal* and *vertical*. To better support all languages and writing directions, CSS now uses the terms *block direction* and *inline direction*. If your primary language is Western-derived, you're accustomed to a block direction of top to bottom, and an inline direction of left to right.

The *block direction* is the direction in which block elements are placed by default in the current writing mode. In English, for example, the block direction is top to bottom, or vertical, as one paragraph (or other text element) is placed beneath the one before. Some languages have vertical text, like Mongolian. When text is vertical, the block direction is horizontal.

The *inline direction* is the direction in which inline elements are written within a block. To again take English as an example, the inline direction is left to right, or horizontal. In languages like Arabic and Hebrew, the inline direction is right to left instead. To reuse the example from the preceding paragraph, Mongolian's inline direction is top to bottom.

Let's reconsider English for a moment. A plain page of English text, displayed on a screen, has a vertical block direction (from top to bottom) and a horizontal inline direction (from left to right). But if the page is rotated 90 degrees counterclockwise by using CSS Transforms, suddenly the block direction is horizontal and the inline direction is vertical. (And bottom to top, at that.)



You can still find a lot of English-centric blog posts and other CSS-related documentation on the web using the terms *vertical* and *horizontal* when talking about writing directions. When you do, mentally translate them to *block* and *inline* as needed.

Indenting Text

Most paper books we read in Western languages format paragraphs of text with the first line indented, and no blank line between paragraphs. If you want to re-create that look, CSS provides the property `text-indent`.

text-indent

Values	[<*length*> | <*percentage*>] && hanging && each-line
Initial value	`0`
Applies to	Block-level elements
Percentages	Refer to the width of the containing block
Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	Yes
Animatable	Yes
Notes	`hanging` and `each-line` are still experimental as of mid-2022

Using `text-indent`, the first line of any element can be indented by a given length, even if that length is negative. A common use for this property is to indent the first line of a paragraph:

```
p {text-indent: 3em;}
```

This rule will cause the first line of any paragraph to be indented 3 ems, as shown in Figure 15-1.

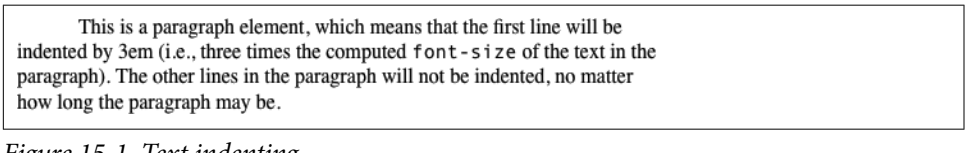


Figure 15-1. Text indenting

In general, you can apply `text-indent` to any element that generates a block box, and the indentation will occur along the inline direction. You can't apply it to inline elements or replaced elements such as images. However, if you have an image within the first line of a block-level element, it will be shifted over with the rest of the text in the line.



If you want to "indent" the first line of an inline element, you can create the effect with left padding or a margin.

You can also set negative values for `text-indent` to create a *hanging indent*, where the first line hangs out to one side of the rest of the element:

```
p {text-indent: -4em;}
```

Be careful when setting a negative value for `text-indent`; the first few words may be chopped off by the edge of the browser window if you aren't careful. To avoid display problems, we recommend you use a margin or padding to accommodate the negative indentation:

```
p {text-indent: -4em; padding-left: 4em;}
```

Any unit of length, including percentage values, may be used with `text-indent`. In the following case, the percentage refers to the width of the parent element of the element being indented. In other words, if you set the indent value to `10%`, the first line of an affected element will be indented by 10% of its parent element's width, as shown in Figure 15-2:

```
div {width: 400px;}
p {text-indent: 10%;}

<div>
<p>This paragraph is contained inside a DIV, which is 400px wide, so the
first line of the paragraph is indented 40px (400 * 10% = 40).  This is
because percentages are computed with respect to the width of the element.</p>
</div>
```

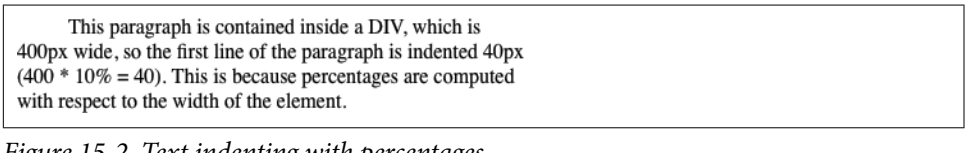


Figure 15-2. Text indenting with percentages

Note that because `text-indent` is inherited, some browsers, like the Yandex browser, inherit the computed values, while Safari, Firefox, Edge, and Chrome inherit the declared value. In the following, both bits of text will be indented 5 ems in Yandex and 10% of the current element's width in other browsers, because the value of `5em` is inherited by the

paragraph from its parent in Yandex and older versions of WebKit, whereas most evergreen browsers inherit the declared value of 10%:

```
div#outer {width: 50em;}
div#inner {text-indent: 10%;}
p {width: 20em;}
```

```
<div id="outer">
<div id="inner">
This first line of the DIV is indented by 5em.
<p>
This paragraph is 20em wide, and the first line of the paragraph
is indented 5em in WebKit and 2em elsewhere.  This is because
computed values for 'text-indent' are inherited in WebKit,
while the declared values are inherited elsewhere.
</p>
</div>
</div>
```

As of late 2022, two keywords are being considered for addition to `text-indent`:

`hanging`
: Inverts the indentation effect; that is, `text-indent: 3em hanging` would indent all the lines of text *except* the first line. This is similar to the negative-value indentation discussed previously, but without risking cutting off text, because instead of pulling the first line out of the content box, all the lines but the first are indented away from the edge of the content box.

`each-line`
: Indents the first line of the element plus any line that starts after a forced line break, such as that caused by a <br>, but not lines that follow a soft line break.

When supported, either keyword can be used in conjunction with a length or percentage, such as the following:

```
p {text-indent: 10% hanging;}
pre {text-indent: 5ch each-line;}
```

Aligning Text

Even more basic than `text-indent` is the property `text-align`, which affects the way the lines of text in an element are aligned with respect to one another.

text-align

Values	`start` | `end` | `left` | `right` | `center` | `justify` | `justify-all` | `match-parent`
Initial value	`start`
Applies to	Block-level elements

Computed value	As specified, except in the case of `match-parent`
Inherited	Yes
Animatable	No
Note	`justify-all` is not supported as of mid-2022

The quickest way to understand how these values work is to examine Figure 15-3, which demonstrates the most widely used values. The values `left`, `right`, and `center` cause the text within elements to be aligned exactly as described by these words in horizontal languages like English or Arabic, regardless of the language's inline direction.

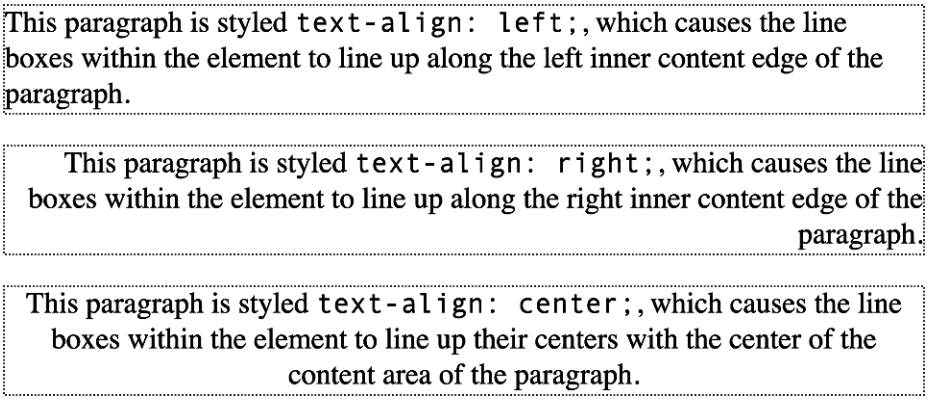


Figure 15-3. Selected behaviors of the `text-align` property

The default value of `text-align` is `start`, which is the equivalent of `left` in LTR languages, and `right` in RTL languages. In vertical languages. `left` and `right` are mapped to the start or end edge, respectively. This is illustrated in Figure 15-4.

Because `text-align` applies only to block-level elements such as paragraphs, there's no way to center an anchor within its line without aligning the rest of the line (nor would you want to, since that would likely cause text overlap).

As you may expect, `center` causes each line of text to be centered within the element. If you've ever come across the long-ago deprecated `<CENTER>` element, you may be tempted to believe that `text-align: center` is the same. It is actually quite different. The `<CENTER>` element affected not only text, but also centered whole elements, such as tables. The `text-align` property does not control the alignment of elements, only their inline content.

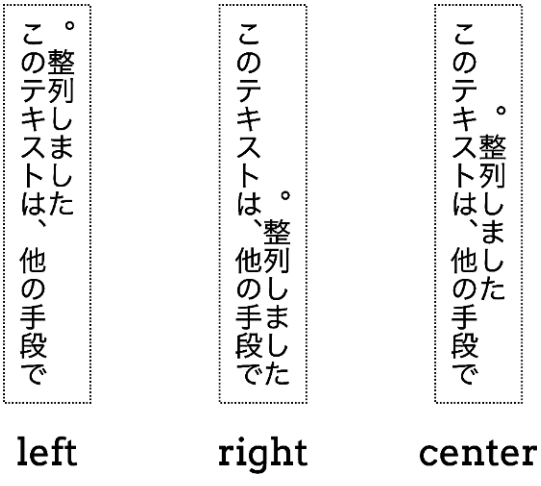


Figure 15-4. Left, right, and center in vertical writing modes

Start and end alignment

Remembering that CSS was based on concepts of *horizontal* and *vertical*, the initial default value was originally "a nameless value that acts as *left* if *direction* is *ltr*, *right* if *direction* is *rtl*." The default value now has a name: `start`, which is the equivalent of `left` in LTR languages, and `right` in RTL languages.

The default value of `start` means that text is aligned to the start edge of its line box. In LTR languages like English, that's the left edge; in RTL languages such as Arabic, it's the right edge. In vertical languages, it will be the top or bottom, depending on the writing direction. The upshot is that the default value is much more aware of the document's language direction while leaving the default behavior the same in the vast majority of existing cases.

In a like manner, `end` aligns text with the end edge of each line box—the right edge in LTR languages, the left edge in RTL languages, and so forth. Figure 15-5 shows the effects of these values.

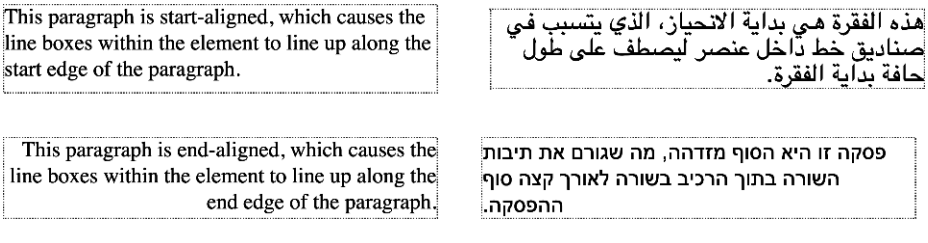


Figure 15-5. Start and end alignment

Justified text

An often-overlooked alignment value is `justify`, which raises some issues of its own. In justified text, both ends of a line of text (except the last line, which can be set with `text-align-last`) are placed at the inner edges of the parent element, as shown in Figure 15-6. Then, the spacing between words and letters is adjusted so that the words are distributed evenly throughout the line. Justified text is common in the print world (for example, in this book), but under CSS, a few extra considerations come into play.

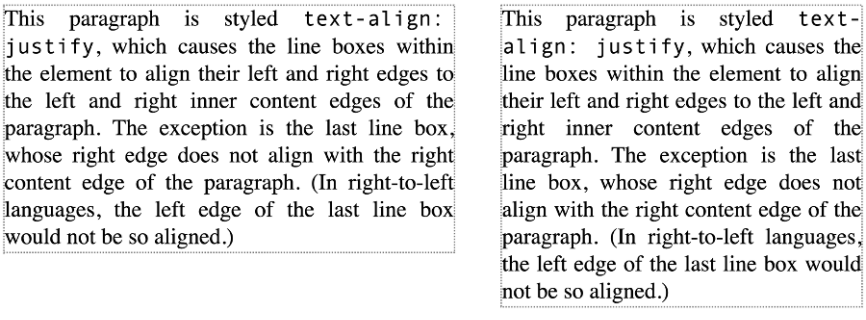


Figure 15-6. Justified text

The user agent determines how justified text should be stretched or distributed to fill the space between the left and right edges of the parent. Some browsers, for example, might add extra space only between words, while others might distribute the extra space between letters (although the CSS specification states that "user agents may not further increase or decrease the inter-character space" if the property `letter-spacing` has been assigned a length value). Other user agents may reduce space on some lines, thus mashing the text together a bit more than usual.

The value `justify-all` sets full justification for both `text-align` and `text-align-last` (covered in an upcoming section).



As of mid-2022, the `justify-all` value is not supported by any browser, even though nearly all of them support `text-align: justify` and `text-align-last: justify`. This gap in support remains a mystery as of press time, but is solved in most browsers with the following:

```
.justify-all {
  text-align: justify;
  text-align-last: justify;
  }
```

Parent matching

We have one more value to cover: `match-parent`. If you declare `text-align: match-parent`, and the inherited value of `text-align` is `start` or `end`, the alignment of the `match-parent` element will be calculated with respect to the parent element's horizontal or vertical, rather than inline, direction.

For example, you could force any English element's text alignment to match the alignment of a parent element, regardless of its writing direction, as in the following example.

```
div {text-align: start;}
div:lang(en) {direction: ltr;}
div:lang(ar) {direction: rtl;}
p {text-align: match-parent;}

<div lang="en-US">
Here is some en-US text.
<p>The alignment of this paragraph will be to the left, as with its parent.</p>
</div>
<div lang="ar">
هذا نص عربي.
<p>The alignment of this paragraph will be to the right, as with its parent.</p>
</div>
```

Aligning the Last Line

At times you might want to align the text in the very last line of an element differently than you did the rest of the content. For example, with `text-align: justify`, the last line defaults to `text-align: start`. You might ensure a left-aligned last line in an otherwise fully justified block of text, or choose to swap from left to center alignment. For those situations, you can use `text-align-last`.

text-align-last

Values	`auto` | `start` | `end` | `left` | `right` | `center` | `justify`
Initial value	`auto`
Applies to	Block-level elements
Computed value	As specified
Inherited	Yes
Animatable	No

As with `text-align`, the quickest way to understand how these values work is to examine Figure 15-7.

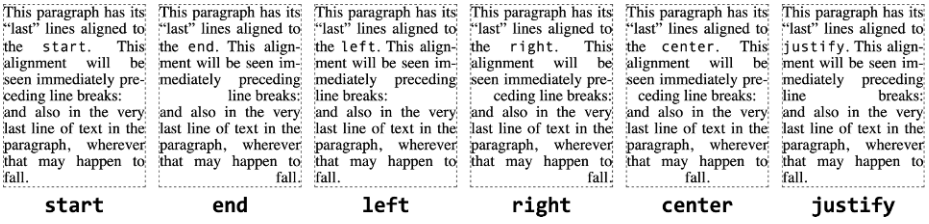


Figure 15-7. Differently aligned last lines

The last lines of the elements are aligned independently of the rest of the elements, according to the elements' `text-align-last` values.

A close study of Figure 15-7 will reveal that there's more at play than just the last lines of block-level elements. In fact, `text-align-last` applies to any line of text that immediately precedes a forced line break, whether or not that line break is triggered by the end of an element. Thus, a line break created by a `<br>` tag will make the line of text immediately before that break use the value of `text-align-last`.

An interesting wrinkle arises with `text-align-last`: if the first line of text in an element is also the last line of text in the element, the value of `text-align-last` takes precedence over the value of `text-align`. Thus, the following styles will result in a centered paragraph, not a start-aligned paragraph:

```
p {text-align: start; text-align-last: center;}

<p>A paragraph.</p>
```

Word Spacing

The `word-spacing` property is used to modify interword spacing, accepting a positive or negative length. This length is *added* to the standard space between words. Therefore, the default value of `normal` is the same as setting a value of `0`.

word-spacing

Values	*<length>* | `normal`
Initial value	`normal`
Applies to	All elements
Computed value	For `normal`, the absolute length `0`; otherwise, the absolute length
Inherited	Yes
Animatable	Yes

If you supply a positive length value, the space between words will increase. Setting a negative value for `word-spacing` brings words closer together:

```
p.spread {word-spacing: 0.5em;}
p.tight {word-spacing: -0.5em;}
p.default {word-spacing: normal;}
p.zero {word-spacing: 0;}
```

```
<p class="spread">The spaces—as in those between the “words”—in this paragraph
   will be increased by 0.5em.</p>
<p class="tight">The spaces—as in those between the “words”—in this paragraph
   will be increased by 0.5em.</p>
<p class="default">The spaces—as in those between the “words”—in this paragraph
   will be neither increased nor decreased.</p>
<p class="zero">The spaces—as in those between the “words”—in this paragraph
   will be neither increased nor decreased.</p>
```

Manipulating these settings has the effect shown in Figure 15-8.

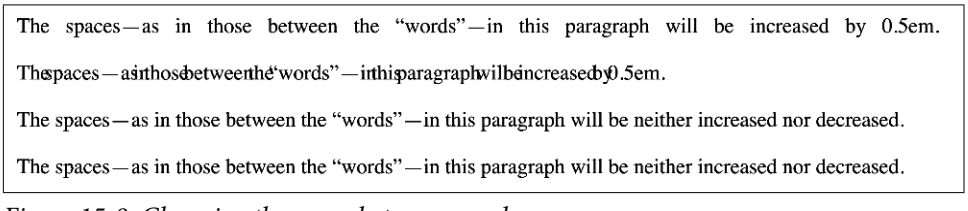


Figure 15-8. Changing the space between words

In CSS terms, a *word* is any string of nonwhitespace characters that is surrounded by whitespace of some kind. This means `word-spacing` is unlikely to work in any languages that employ pictographs, or non-Roman writing styles. This is also why the em dashes in the previous example's text don't get space around them. From the CSS point of view, "spaces—as" is a single word.

Use caution. The `word-spacing` property allows you to create very unreadable documents, as Figure 15-9 illustrates.

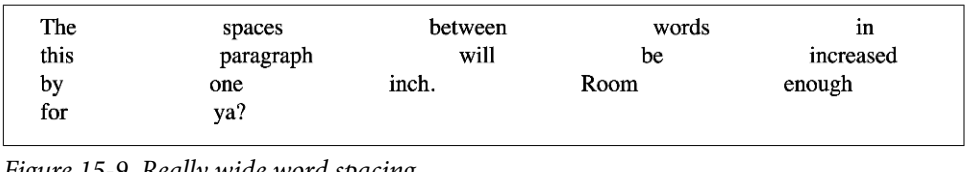


Figure 15-9. Really wide word spacing

Letter Spacing

Many of the issues you encounter with `word-spacing` also occur with `letter-spacing`. The only real difference between the two is that `letter-spacing` modifies the space between characters or letters.

letter-spacing

Values	*<length>* | `normal`
Initial value	`normal`
Applies to	All elements
Computed value	For length values, the absolute length; otherwise, `normal`
Inherited	Yes
Animatable	Yes

As with the `word-spacing` property, the permitted values of `letter-spacing` include any length, though character-relative lengths like `em` (rather than root-relative lengths like `rem`) are recommended to ensure that the spacing is proportional to the font size.

The default keyword is `normal`, which has the same effect as `letter-spacing: 0`. Any length value you enter will increase or decrease the space between letters by that amount. Figure 15-10 shows the results of the following markup:

```
p {letter-spacing: 0;}    /* identical to 'normal' */
p.spacious {letter-spacing: 0.25em;}
p.tight {letter-spacing: -0.25em;}

<p>The letters in this paragraph are spaced as normal.</p>
<p class="spacious">The letters in this paragraph are spread out a bit.</p>
<p class="tight">The letters in this paragraph are a bit smashed together.</p>
```

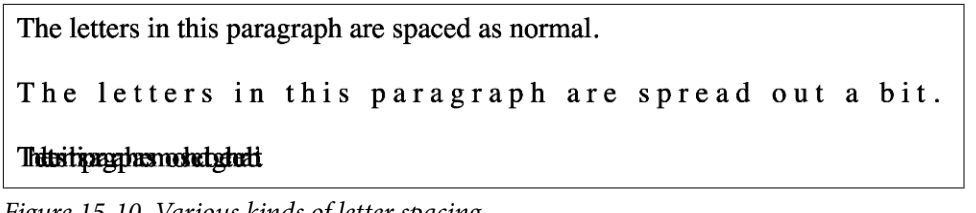


Figure 15-10. Various kinds of letter spacing



If a page uses fonts with features like ligatures, and those features are enabled, altering letter or word spacing can effectively disable them. Browsers will not recalculate ligatures or other joins when letter spacing is altered.

Spacing and Alignment

It's important to remember that space between words may be altered by the value of the property `text-align`. If an element is justified, the spaces between letters and words may be altered to fit the text along the full width of the line. This may in turn alter the spacing declared using `word-spacing`.

If a length value is assigned to `letter-spacing`, that value cannot be changed by `text-align`; but if the value of `letter-spacing` is `normal`, inter-character spacing may be changed to justify the text. CSS does not specify how the spacing should be calculated, so user agents use their own algorithms. To prevent `text-align` from altering letter spacing while keeping the default letter spacing, declare `letter-spacing: 0`.

Note that computed values are inherited, so child elements with larger or smaller text will have the same word or letter spacing as their parent element. You cannot define a scaling factor for `word-spacing` or `letter-spacing` to be inherited in place of the computed value (in contrast with `line-height`). As a result, you may run into problems such as those shown in Figure 15-11:

```
p {letter-spacing: 0.25em; font-size: 20px;}
small {font-size: 50%;}

<p>This spacious paragraph features <small>tiny text that is just
as spacious</small>, even though the author probably wanted the
spacing to be in proportion to the size of the text.</p>
```

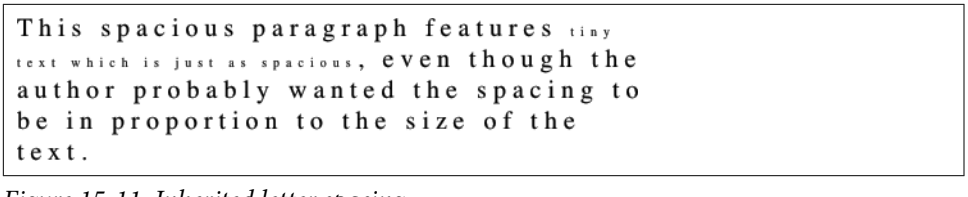


Figure 15-11. Inherited letter spacing

As `inherit` inherits the ancestor's letter-spacing computed length, the only way to achieve letter spacing that's in proportion to the size of the text is to set it explicitly on each element, as follows:

```
p {letter-spacing: 0.25em;}
small {font-size: 50%; letter-spacing: 0.25em;}
```

And the same goes for word spacing.

Vertical Alignment

Now that we've covered alignment along the inline direction, let's move on to the vertical alignment of inline elements along the block direction—things like superscripting and vertical alignment (vertical with respect to the line of text, if the text is laid out

horizontally). Since the construction of lines is a complex topic that merits its own small book, we'll just stick to a quick overview here.

Adjusting the Height of Lines

The distance between lines can be affected by changing the height of a line. Note that *height* here is with respect to the line of text itself, assuming that the longer axis of a line is *width*, even if it's written vertically. The property names we cover from here will reveal a strong bias toward Western languages and their writing directions; this is an artifact of the early days of CSS, when Western languages were the only ones that could be easily represented.

The `line-height` property refers to the distance between the baselines of lines of text rather than the size of the font, and it determines the amount by which the height of each element's box is increased or decreased. In the most basic cases, specifying `line-height` is a way to increase (or decrease) the vertical space between lines of text, but this is a misleadingly simple way of looking at how `line-height` works. This property controls the *leading*, which is the extra space between lines of text above and beyond the font's size. In other words, the difference between the value of `line-height` and the size of the font is the leading.

line-height

Values	*<number>* | *<length>* | *<percentage>* | `normal`
Initial value	`normal`
Applies to	All elements (but see text regarding replaced and block-level elements)
Percentages	Relative to the font size of the element
Computed value	For length and percentage values, the absolute value; otherwise, as specified
Inherited	Yes
Animatable	Yes

When applied to a block-level element, `line-height` defines the *minimum* distance between text baselines within that element. Note that it defines a minimum, not an absolute value. Baselines of text can wind up being pushed farther apart than the value of `line-height`, for example, if a line contains an inline image or form control that is taller than the declared line height. The `line-height` property does not affect layout for replaced elements like images, but it still applies to them.

Constructing a line

As you learned in Chapter 6, every element in a line of text generates a *content area*, which is determined by the size of the font. This content area, in turn, generates an *inline*

box that is, in the absence of any other factors, exactly equal to the content area. The leading generated by `line-height` is one of the factors that increase or decrease the height of each inline box.

To determine the leading for a given element, subtract the computed value of `font-size` from the computed value of `line-height`. That value is the total amount of leading. And remember, it can be a negative number. The leading is then divided in half, and each half-leading is applied to the top and bottom of the content area. The result is the inline box for that element. In this way, each line of text is centered within the line height as long as the height of the line isn't forced beyond its minimum height by a replaced element or other factor.

As an example, let's say `font-size` (and therefore the content area) is 14 pixels tall, and `line-height` is computed to 18 pixels. The difference (4 pixels) is divided in half, and each half is applied to the top and bottom of the content area. This effectively centers the content by creating an inline box that is 18 pixels tall, with 2 extra pixels above and below the content area. This sounds like a roundabout way to describe how `line-height` works, but there are excellent reasons for the description.

Once all of the inline boxes have been generated for a given line of content, they are then considered in the construction of the line box. A line box is exactly as tall as needed to enclose the top of the tallest inline box and the bottom of the lowest inline box. Figure 15-12 shows a diagram of this process.

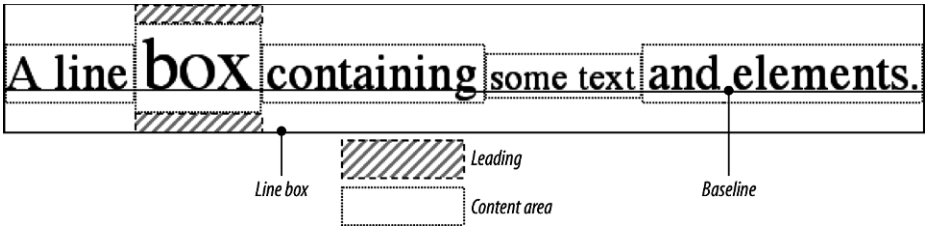


Figure 15-12. Line-box diagram

Assigning values to line-height

Let's now consider the possible values of `line-height`. If you use the default value of `normal`, the user agent must calculate the space between lines. Values can vary by user agent, but the `normal` default is generally around 1.2 times the size of the font, which makes line boxes taller than the value of `font-size` for a given element.

Many values are simple length measures (e.g., `18px` or `2em`), but <*number*> values with no length unit are preferable in many situations.



Be aware that even if you use a valid length measurement, such as `4cm`, the browser (or the operating system) may be using an incorrect metric for real-world measurements, so the line height may not show up as exactly 4 centimeters on your monitor.

The `em`, `ex`, and percentage values are calculated with respect to the `font-size` of the element. The results of the following CSS and HTML are shown in Figure 15-13:

```
body {line-height: 18px; font-size: 16px;}
p.cl1 {line-height: 1.5em;}
p.cl2 {font-size: 10px; line-height: 150%;}
p.cl3 {line-height: 0.33in;}
```

```
<p>This paragraph inherits a 'line-height' of 18px from the body, as well as
a 'font-size' of 16px.</p>
<p class="cl1">This paragraph has a 'line-height' of 24px(16 * 1.5), so
it will have slightly more line-height than usual.</p>
<p class="cl2">This paragraph has a 'line-height' of 15px (10 * 150%), so
it will have slightly more line-height than usual.</p>
<p class="cl3">This paragraph has a 'line-height' of 0.33in, so it will have
slightly more line-height than usual.</p>
```

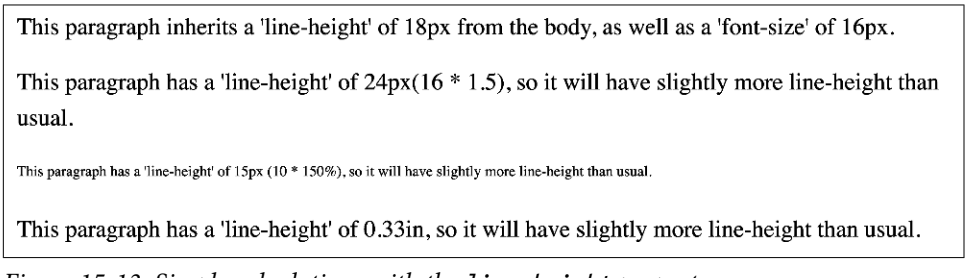


Figure 15-13. Simple calculations with the `line-height` property

Understanding line-height and inheritance

When the `line-height` is inherited by one block-level element from another, things get a bit trickier. The `line-height` values inherit from the parent element as computed from the parent, not the child. The results of the following markup are shown in Figure 15-14. It probably wasn't what the author had in mind:

```
body {font-size: 10px;}
div {line-height: 1em;}  /* computes to '10px' */
p {font-size: 18px;}
```

```
<div>
<p>This paragraph's 'font-size' is 18px, but the inherited 'line-height'
value is only 10px.  This may cause the lines of text to overlap each
other by a small amount.</p>
</div>
```

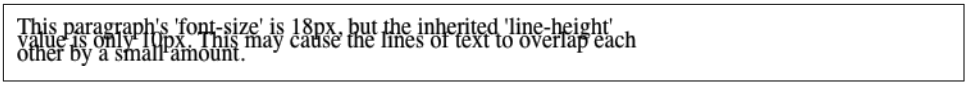


Figure 15-14. Small line-height, large font-size, slight problem

Why are the lines so close together? Because the computed `line-height` value of `10px` was inherited by the paragraph from its parent `<div>`. One solution to the small `line-height` problem depicted in Figure 15-14 is to set an explicit `line-height` for every element, but that's not very practical. A better alternative is to specify a number, which actually sets a scaling factor:

```
body {font-size: 10px;}
div {line-height: 1;}
p {font-size: 18px;}
```

When you specify a number with no length unit, you cause the scaling factor to be an inherited value instead of a computed value. The number will be applied to the element and all of its child elements so that each element has a `line-height` calculated with respect to its own `font-size` (see Figure 15-15):

```
div {line-height: 1.5;}
p {font-size: 18px;}

<div>
<p>This paragraph's 'font-size' is 18px, and since the 'line-height'
set for the parent div is 1.5, the 'line-height' for this paragraph
is 27px (18 * 1.5).</p>
</div>
```

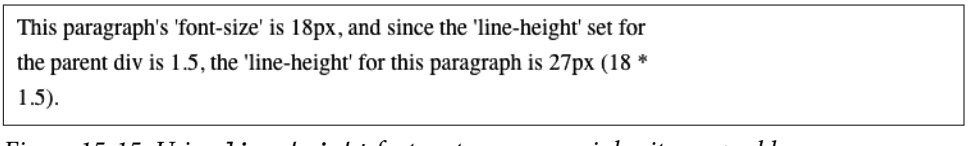


Figure 15-15. Using `line-height` factors to overcome inheritance problems

Now that you have a basic grasp of how lines are constructed, let's talk about vertically aligning elements relative to the line box—that is, displacing them along the block direction.

Vertically Aligning Text

If you've ever used the elements `<sup>` and `<sub>` (the superscript and subscript elements), or used the deprecated `align` attribute with an image, you've done some rudimentary vertical alignment.



Because of the property name `vertical-align`, this section will use the terms *vertical* and *horizontal* to refer to the block and inline directions of the text.

vertical-align

Values	`baseline` | `sub` | `super` | `top` | `text-top` | `middle` | `bottom` | `text-bottom` | *<length>* | *<percentage>*
Initial value	`baseline`
Applies to	Inline elements, the pseudo-elements `::first-letter` and `::first-line`, and table cells
Percentages	Refer to the value of `line-height` for the element
Computed value	For percentage and length values, the absolute length; otherwise, as specified
Inherited	No
Animatable	*<length>*, *<percentage>*
Note	When applied to table cells, only the values `baseline`, `top`, `middle`, and `bottom` are recognized

The `vertical-align` property accepts any one of eight keywords, a percentage value, or a length value. The keywords are a mix of the familiar and unfamiliar: `baseline` (the default value), `sub`, `super`, `bottom`, `text-bottom`, `middle`, `top`, and `text-top`. We'll examine how each keyword works in relation to inline elements.



Remember: `vertical-align` does *not* affect the alignment of content within a block-level element, just the alignment of inline content within a line of text or a table cell. This may change in the future, but as of mid-2022, proposals to widen its scope have yet to move forward.

Baseline alignment

Using `vertical-align: baseline` forces the baseline of an element to align with the baseline of its parent. Browsers, for the most part, do this anyway, since you'd probably expect the bottoms of all text elements in a line to be aligned.

If a vertically aligned element doesn't have a baseline—that is, if it's an image, a form input, or another replaced element—then the bottom of the element is aligned with the baseline of its parent, as Figure 15-16 shows:

```
img {vertical-align: baseline;}

<p>The image found in this paragraph <img src="dot.gif" alt="A dot" /> has its
bottom edge aligned with the baseline of the text in the paragraph.</p>
```

The image found in this paragraph ▪ has its bottom edge aligned with the baseline of the text in the paragraph.

Figure 15-16. Baseline alignment of an image

This alignment rule is important because it causes some web browsers to always put a replaced element's bottom edge on the baseline, even if the line includes no other text. For example, let's say you have an image in a table cell all by itself. The image may actually be on a baseline, but in some browsers, the space below the baseline causes a gap to appear beneath the image. Other browsers will "shrink-wrap" the image with the table cell, and no gap will appear. The gap behavior is correct, despite its lack of appeal to most authors.



See the deeply aged and yet somehow still relevant article "Images, Tables, and Mysterious Gaps" (2002) for a more detailed explanation of gap behavior and ways to work around it.

Superscripting and subscripting

The declaration `vertical-align: sub` causes an element to be subscripted, meaning that its baseline (or bottom, if it's a replaced element) is lowered with respect to its parent's baseline. The specification doesn't define the distance the element is lowered, so it may vary depending on the user agent.

The `super` value is the opposite of `sub`; it raises the element's baseline (or bottom of a replaced element) with respect to the parent's baseline. Again, the distance the text is raised depends on the user agent.

Note that the values `sub` and `super` do *not* change the element's font size, so subscripted or superscripted text will not become smaller (or larger). Instead, any text in the sub- or superscripted element will, by default, be the same size as text in the parent element, as illustrated by Figure 15-17:

```
span.raise {vertical-align: super;}
span.lower {vertical-align: sub;}

<p>This paragraph contains <span class="raise">superscripted</span>
and <span class="lower">subscripted</span> text.</P>
```

This paragraph contains superscripted and subscripted text.

Figure 15-17. Superscript and subscript alignment



If you wish to make super- or subscripted text smaller than the text of its parent element, you can do so by using the `font-size` property.

Top and bottom alignment

The `vertical-align: top` option aligns the top of the element's inline box with the top of the line box. Similarly, `vertical-align: bottom` aligns the bottom of the element's inline box with the bottom of the line box. Thus, the following markup results in Figure 15-18:

```
.soarer {vertical-align: top;}
.feeder {vertical-align: bottom;}

<p>And in this paragraph, as before, we have
first a <img src="tall.gif" alt="tall" class="soarer" /> image and
then a <img src="short.gif" alt="short" class="soarer" /> image,
and then some text which is not tall.</p>

<p>This paragraph, as you can see, contains
first a <img src="tall.gif" alt="tall" class="feeder" /> image and
then a <img src="short.gif" alt="short" class="feeder" /> image,
and then some text that is not tall.</p>
```

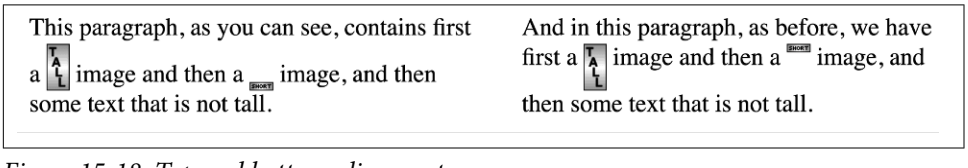


Figure 15-18. Top and bottom alignment

The second line of the first paragraph contains two inline elements whose top edges are aligned with each other. They're also well above the baseline of the text. The second paragraph shows the inverted case: two images whose bottoms are aligned and are well below the baseline of their line. This is because in both cases, the sizes of the elements in the line have increased the line's height beyond what the font's size would normally create.

If you want instead to align elements with the top or bottom edge of just the text in the line, `text-top` and `text-bottom` are the values you seek. For the purposes of these values, replaced elements, or any other kinds of nontext elements, are ignored. Instead, a *default* text box is considered. This default box is derived from the `font-size` of the parent element. The bottom of the aligned element's inline box is then aligned with the bottom of the default text box. Thus, given the following markup, you get a result like the one shown in Figure 15-19:

```
img.ttop {vertical-align: text-top;}
img.tbot {vertical-align: text-bottom;}
```

```
<p>Here: a <img src="tall.gif" class="tbot" alt="tall" /> tall image,
and then a <img src="short.gif" class="tbot" alt="short" /> image.</p>
<p>Here: a <img src="tall.gif" class="ttop" alt="tall"> tall image,
and then a <img src="short.gif" class="ttop" alt="short" /> image.</p>
```

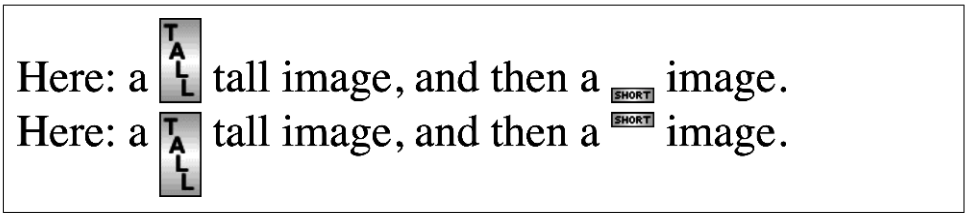


Figure 15-19. Text-top and -bottom alignment

Middle alignment

The value `middle` is usually (but not always) applied to images. It does not have the exact effect you might assume, given its name. The `middle` value aligns the middle of an inline element's box with a point that is `0.5ex` above the baseline of the parent element, where `1ex` is defined relative to the `font-size` for the parent element. Figure 15-20 shows this in more detail.

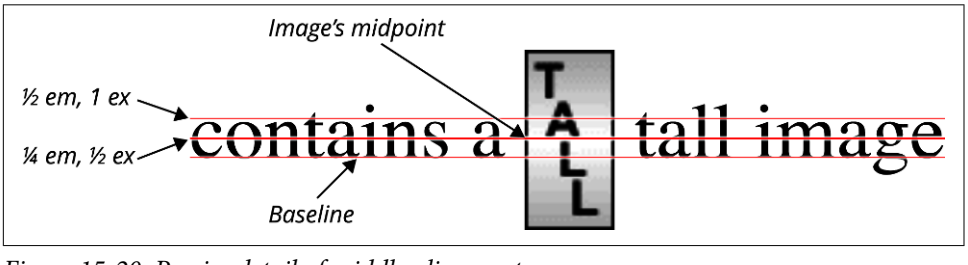


Figure 15-20. Precise detail of middle alignment

Since most user agents treat `1ex` as one-half em, `middle` usually aligns the vertical midpoint of an element with a point one-quarter em above the parent's baseline, though this is not a defined distance and so can vary from one user agent to another.

Percentages

Percentages don't let you simulate `align="middle"` for images. Instead, setting a percentage value for `vertical-align` raises or lowers the baseline of the element (or the bottom edge of a replaced element) by the amount declared, with respect to the parent's baseline. (The percentage you specify is calculated as a percentage of `line-height` for the element, *not* its parent.) Positive percentage values raise the element, and negative values lower it.

Depending on how the text is raised or lowered, it can appear to be placed in adjacent lines, as shown in Figure 15-21, so take care when using percentage values:

```
sub {vertical-align: -100%;}
sup {vertical-align: 100%;}

<p>We can either <sup>soar to new heights</sup> or, instead,
<sub>sink into despair...</sub></p>
```

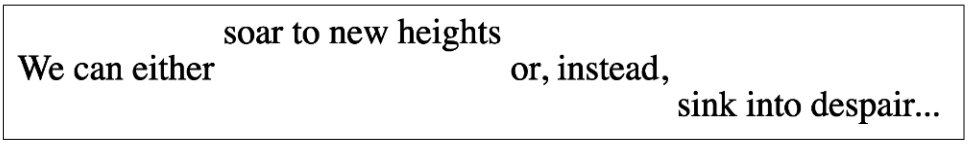


Figure 15-21. Percentages and fun effects

Length alignment

Finally, let's consider vertical alignment with a specific length. The `vertical-align` option is very basic: it shifts an element up or down by the declared distance. Thus, `vertical-align: 5px;` will shift an element upward 5 pixels from its unaligned placement. Negative length values shift the element downward.

It's important to realize that vertically aligned text does not become part of another line, nor does it overlap text in other lines. Consider Figure 15-22, in which some vertically aligned text appears in the middle of a paragraph.

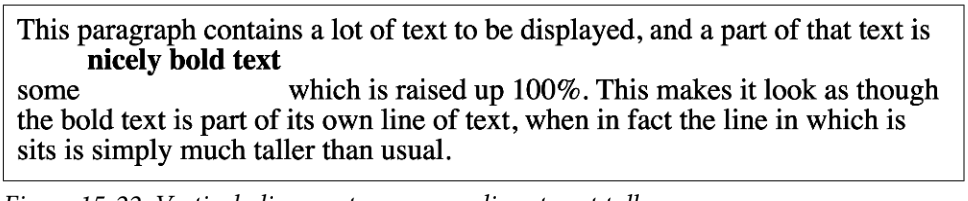


Figure 15-22. Vertical alignments can cause lines to get taller

As you can see, any vertically aligned element can affect the height of the line. Recall the description of a line box, which is exactly as tall as necessary to enclose the top of the tallest inline box and the bottom of the lowest inline box. This includes inline boxes that have been shifted up or down by vertical alignment.

Text Transformation

With the alignment properties covered, let's look at ways to manipulate the capitalization of text by using the property `text-transform`.

text-transform

Values	`uppercase` | `lowercase` | `capitalize` | `full-width` | `full-size-kana` | `none`
Initial value	`none`

Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	No
Notes	`full-width` and `full-size-kana` are supported only in Firefox as of mid-2022

The default value `none` leaves the text alone and uses whatever capitalization exists in the source document. As their names imply, `uppercase` and `lowercase` convert text into all upper- or lowercase characters. The `full-width` value forces the writing of a character inside a square, as if on a typographical grid.



Accessibility note: some screen readers will read all-uppercase text one letter at a time, as if spelling out an acronym, even if the source text is lowercase or mixed-case and the uppercasing is only enforced via CSS. For this reason, uppercasing text via CSS should be approached with caution.

Finally, the `capitalize` value capitalizes only the first letter of each word (where a *word* is defined as a string of adjacent characters surrounded by whitespace). Figure 15-23 illustrates each of these settings in a variety of ways:

```
h1 {text-transform: capitalize;}
strong {text-transform: uppercase;}
p.cummings {text-transform: lowercase;}
p.full {text-transform: full-width;}
p.raw {text-transform: none;}
```

```
<h1>The heading-one at the beginninG</h1>
<p>
By default, text is displayed in the capitalization it has in the source
document, but <strong>it is possible to change this</strong> using
the property 'text-transform'.
</p>
<p class="cummings">
For example, one could Create TEXT such as might have been Written by
the late Poet E.E.Cummings.
</p>
<p class="full">
If you need to align characters as if in a grid, as is often done in CJKV
languages, you can use 'full-width' to do so.
</p>
<p class="raw">
If you feel the need to Explicitly Declare the transformation of text
to be 'none', that can be done as well.
</p>
```

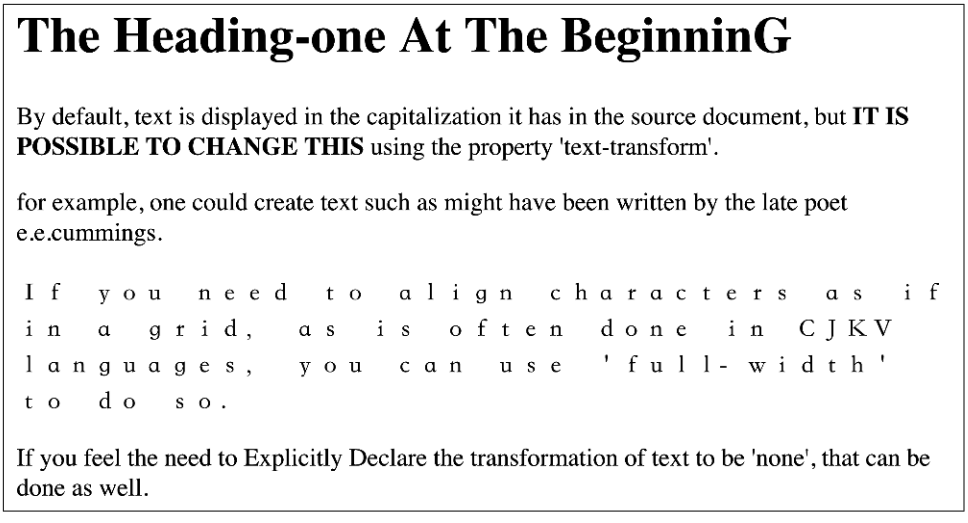


Figure 15-23. Various kinds of text transformation



As noted in Chapter 6, *CJK* stands for *Chinese/Japanese/Korean*. CJK characters take up the majority of the entire Unicode code space, including approximately 70,000 Han characters. You may sometimes come across the abbreviation *CJKV*, which adds *Vietnamese* to the mix.

Different user agents may have different ways of deciding where words begin and, as a result, which letters are capitalized. For example, the text "heading-one" in the `<h1>` element, shown in Figure 15-23, could be rendered in one of two ways: "Heading-one" or "Heading-One." CSS does not say which is correct, so either is possible.

You may have also noticed that the last letter in the `<h1>` element in Figure 15-23 is still uppercase. This is correct: when applying a `text-transform` of `capitalize`, CSS requires user agents to make sure only the first letter of each word is capitalized. They can ignore the rest of the word.

As a property, `text-transform` may seem minor, but it's very useful if you suddenly decide to capitalize all your `<h1>` elements. Instead of individually changing the content of all your `<h1>` elements, you can just use `text-transform` to make the change for you:

```
h1 {text-transform: uppercase;}

<h1>This is an H1 element</h1>
```

The advantages of using `text-transform` are twofold. First, you need to write only a single rule to make this change, rather than changing the `<h1>` itself. Second, if you decide later to switch from all capitals back to initial capitals, the change is even easier:

```
h1 {text-transform: capitalize;}
```

Remember that `capitalize` is a simple letter substitution at the beginning of each "word." CSS doesn't check for grammar, so common headline-capitalization conventions, such as leaving articles (*a*, *an*, *the*) all lowercase, won't be enforced.

Different languages have different rules for which letters should be capitalized. The `text-transform` property takes into account language-specific case mappings.

The `full-width` option forces the writing of a character inside a square. Most characters you can type on a keyboard come in both normal width and full width, with different Unicode code points. The full-width version is used when `full-width` is set and supported to mix them smoothly with Asian ideographic characters, allowing ideograms and Latin scripts to be aligned.

Generally used with `<ruby>` annotation text, `full-size-kana` converts all small Kana characters to the equivalent full-size Kana, to compensate for legibility issues at the small font sizes typically used in Ruby.

Text Decoration

Next we come to the topic of text decorations, and how we can affect them with various properties. The simplest text decoration, and the one that can be controlled the most, is an underline. CSS also supports overlines, line-throughs, and even the wavy underlines you see in word processing programs to flag errors of spelling or grammar.

We'll start with the various individual properties, and then tie it all up with a shorthand property, `text-decoration`, that covers them all.

Setting Text Decoration Line Placement

With the property `text-decoration-line`, you can set the location of one or more line decorations on a run of text. The most familiar decoration may be underlining, thanks to all the hyperlinks out there, but CSS has three possible visible decoration line values (plus an unsupported fourth that wouldn't draw a line at all even if it *was* supported).

text-decoration-line

Values	`none` | [`underline` || `overline` || `line-through` || `blink`]
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No
Notes	The `blink` value is deprecated, with all browsers treating it as `none` as of early 2022

The values are relatively self-documenting: `underline` places a line under the text, where *under* means "below the text in the block direction." The `overline` value is the mirror image, putting the line above the text in the block direction. The `line-through` value draws a line through the middle of the text.

Let's see what these decorations look like in practice. The following code is illustrated in Figure 15-24:

```
p.one {text-decoration: underline;}
p.two {text-decoration: overline;}
p.three {text-decoration: line-through;}
p.four {text-decoration: none;}
```

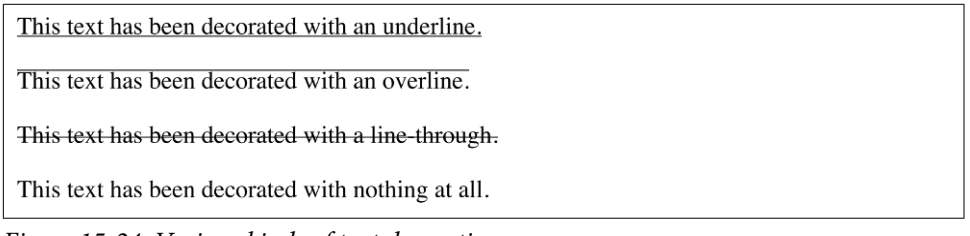


Figure 15-24. Various kinds of text decoration

The value `none` turns off any decoration that might otherwise have been applied to an element. For example, links are usually underlined by default. If you want to suppress the underlining of hyperlinks, you can use the following CSS rule to do so:

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

If you explicitly turn off link underlining with this sort of rule, the only visual difference between the anchors and normal text will be their color (at least by default, though there's no ironclad guarantee that there will be a difference in their colors). Relying on color alone as the difference between regular text and links within that text is not enough to differentiate links from the rest of the text, negatively impacting user experience and making your content inaccessible to many users.



Bear in mind that many users will be annoyed when they realize you've turned off link underlining, especially within blocks of text. If your links aren't underlined, users will have a hard time finding hyperlinks in your documents, and finding them can be next to impossible for users with one form or another of color blindness.

That's really all there is to `text-decoration-line`. The more veteran among you may recognize this is what `text-decoration` itself used to do, but times have moved on and there's much, much more we can do with decorations besides just place them, so these values were shifted to `text-decoration-line`.

Setting Text Decoration Color

By default, the color of a text decoration will match the color of the text. If you need to change that, `text-decoration-color` is here to help.

text-decoration-color

Values	*<color>* | `currentcolor`
Initial value	`currentcolor`
Applies to	All elements
Computed value	The computed color
Inherited	No
Animatable	Yes

You can use any valid color value for `text-decoration-color`, including the keyword `currentcolor` (which is the default). Suppose you want to make it clear that stricken text really is stricken. That would go something like this:

```
del, strike, .removed {
        text-decoration-line: line-through;
        text-decoration-color: red;
}
```

Thus, not only will the elements shown get a line-through decoration, but the line will also be colored red. The text itself will not be red unless you change that as well by using the `color` property.



Remember to keep the color contrast between decorations and the base text sufficiently high to remain accessible. It's also generally a bad idea to use color alone to convey meaning, as in "check the links with red underlines for more information!"

Setting Text Decoration Thickness

With the property `text-decoration-thickness`, you can change the stroke thickness of a text decoration to something beefier, or possibly less beefy, than usual.

text-decoration-thickness

Values	*<length>* | *<percentage>* | `from-font` | `auto`
Initial value	`auto`

Applies to	All elements
Computed value	As declared
Percentages	Refer to the `font-size` of the element
Inherited	No
Animatable	Yes
Notes	Was `text-decoration-width` until a name change in 2019

Supplying a length value sets the thickness of the decoration to that length; thus, `text-decoration-thickness: 3px` sets the decoration to be 3 pixels thick, no matter how big or small the text itself might be. A better approach is generally to use an em-based value or jump straight to using a percentage value, since percentages are calculated with respect to the value of `1em` for the element. Thus, `text-decoration-thickness: 10%` would yield a decoration thickness of 1.6 pixels in a font whose computed font size is 16 pixels, but 4 pixels for a 40-pixel font size. The following code shows a few examples, which are illustrated in Figure 15-25:

```
h1, p {text-decoration-line: underline;}
.tiny {text-decoration-thickness: 1px;}
.embased {text-decoration-thickness: 0.333em;}
.percent {text-decoration-thickness: 10%;}
```

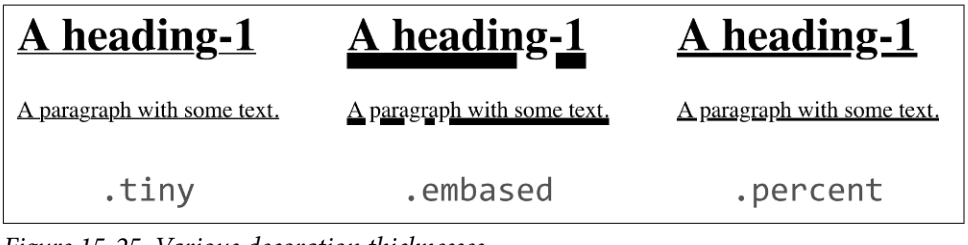


Figure 15-25. Various decoration thicknesses

The keyword `from-font` is interesting because it allows the browser to consult the font file to see whether it defines a preferred decoration thickness; if it does, the browser uses that thickness. If the font file doesn't recommend a thickness, the browser falls back to the `auto` behavior and picks whatever thickness it thinks appropriate, using inscrutable reasoning known only to itself.

Setting Text Decoration Style

Thus far, we've shown a lot of straight, single lines. If you're yearning for something beyond that hidebound approach, `text-decoration-style` provides alternatives.

text-decoration-style

Values	`solid` | `double` | `dotted` | `dashed` | `wavy`
Initial value	`solid`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

The exact result will depend on the value you pick and the browser you use to view the results, but the renderings of these decoration styles should be at least similar to those shown in Figure 15-26, which is the output of the following code:

```
p {text-decoration-line: underline; text-decoration-thickness: 0.1em;}
p.one {text-decoration-style: solid;}
p.two {text-decoration-style: double;}
p.three {text-decoration-style: dotted;}
p.four {text-decoration-style: dashed;}
p.five {text-decoration-style: wavy;}
```

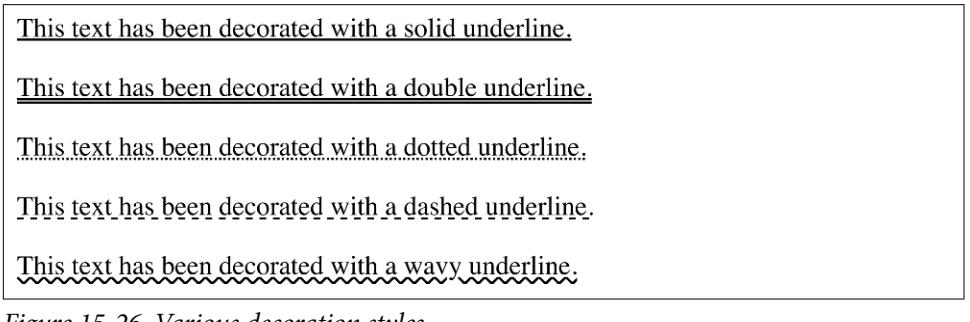


Figure 15-26. Various decoration styles

We increased the decoration thickness for Figure 15-26 in order to improve legiblility; the default sizing can make some of the more complex decorations, like `dotted`, difficult to see.

Using the Text Decoration Shorthand Property

When you just want to set a text decoration's position, color, thickness, and style in one handy declaration, `text-decoration` is the way to go.

text-decoration

Values	*<text-decoration-line>* ‖ *<text-decoration-style>* ‖ *<text-decoration-color>* ‖ *<text-decoration-thickness>*
Initial value	See individual properties
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	As allowed by individual properties

With the `text-decoration` shorthand property, you can bring everything into one place, like so:

```
h2 {text-decoration: overline purple 10%;}
a:any-link {text-decoration: underline currentcolor from-font;}
```

Be careful, though: if you have two different decorations matched to the same element, the value of the rule that wins out will completely replace the value of the loser. Consider the following:

```
h2.stricken {text-decoration: line-through wavy;}
h2 {text-decoration: underline overline double;}
```

Given these rules, any `<h2>` element with a class of `stricken` will have only a wavy line-through decoration. The doubled underline and overline decorations are lost, since shorthand values replace one another instead of accumulating.

Note also that because of the way the decoration properties work, you can set the color and style only once per element, even if you have multiple decorations. For example, the following is valid, setting both the under- and overlines to be green and dotted:

```
text-decoration: dotted green underline overline;
```

If you instead want the overline to be a different color than the underline, or set each to have its own style, you'd need to apply each to a separate element, something like this:

```
p {text-decoration: dotted green overline;}
p > span:first-child {text-decoration: silver dashed underline;}

<p><span>All this text will have differing text decorations.</span></p>
```

Offsetting Underlines

Along with all the `text-decoration` properties, a related property allows you to change the distance between an underline (and *only* an underline) and the text that the underline decorates: `text-underline-offset`.

text-underline-offset

Values	*<length>* | *<percentage>* | `auto`
Initial value	`auto`
Applies to	All elements
Computed value	As specified
Percentages	Refer to the `font-size` of the element
Inherited	No
Animatable	Yes

You might wish that, say, underlines on hyperlinks were a little farther away from the text's baseline, so that they're a little more obvious to the user. Setting a length value like `3px` will put the underline 3 pixels below the text's baseline. See Figure 15-27 for the results of the following CSS:

```
p {text-decoration-line: underline;}
p.one {text-underline-offset: auto;}
p.two {text-underline-offset: 2px;}
p.three {text-underline-offset: -2px;}
p.four {text-underline-offset: 0.5em;}
p.five {text-underline-offset: 15%;}
```

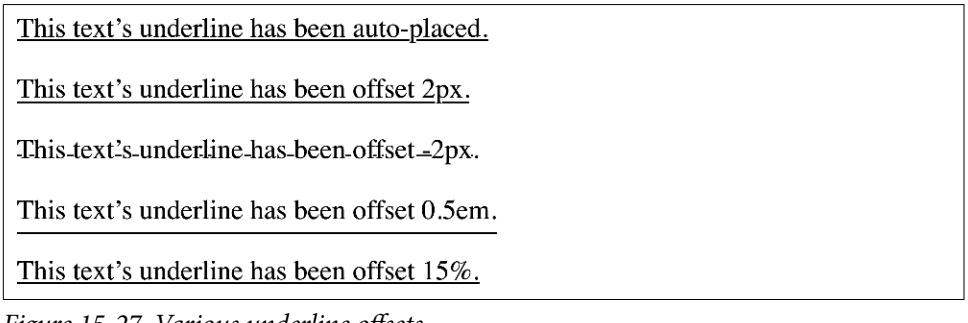


Figure 15-27. Various underline offsets

As illustrated in Figure 15-27, the value defines an offset from the text's baseline, either positive (downward along the block axis) or negative (upward along the block axis).

As with `text-decoration-thickness`, percentage values for `text-underline-offset` are calculated with respect to the value of `1em` for the element. Thus, `text-underline-offset: 10%` would cause an offset of 1.6 pixels in a font whose computed font size is 16 pixels.



As of late 2022, only Firefox supports percentage values for `text-underline-offset`, which is odd given that percentage values are a percent of 1 em in the element's font. The workaround is to use em length values, such as 0.1em for 10%.

Skipping Ink

An unaddressed aspect of the past few sections has been: how exactly do browsers draw decorations over text, and more precisely, decide when to "skip over" parts of the text? This is known as *skipping ink*, and the approach a browser takes can be altered with the property `text-decoration-skip-ink`.

text-decoration-skip-ink

Values	`all` | `none` | `auto`
Initial value	`auto`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

When ink skipping is turned on, the decoration is interrupted wherever it would cross over the shapes of the text. Usually, this means a small gap between the decoration and the text glyphs. See Figure 15-28 for a close-up illustration of the differences in ink-skipping approaches.

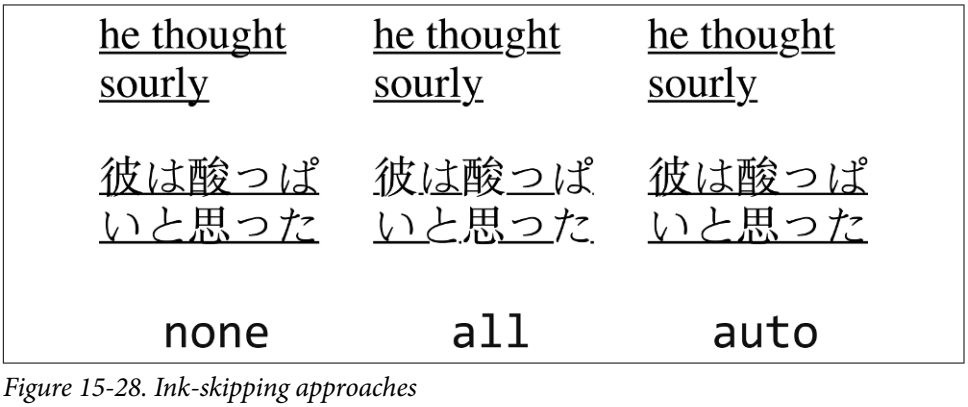


Figure 15-28. Ink-skipping approaches

The three values are defined as follows:

`auto` *(the default)*
: The browser *may* interrupt under- and overlines where the line would cross the text glyphs, with a little space between the line and the glyphs. Furthermore, browsers *should* consider the glyphs used for the text, since some glyphs may call for ink skipping while others may not.

`all`
: Browsers *must* interrupt under- and overlines where the line would cross the text glyphs, with a little space between the line and the glyphs. However, as of mid-2022, only Firefox supports this value.

`none`
: The browser *must not* interrupt under- and overlines where the line would cross the text glyphs, but instead draw a continuous line even though it may be drawn over parts of the text glyphs.

As shown in Figure 15-28, `auto` can sometimes mean differences depending on the language, font, or based on other factors. You're really just telling the browser to do whatever it thinks is best.



While this property's name begins with the label `text-decoration-`, it is *not* a property covered by the `text-decoration` shorthand property. That's why it's being discussed here, after the shorthand, and not before.

Understanding Weird Decorations

Now, let's look into the unusual side of `text-decoration`. The first oddity is that `text-decoration` is *not* inherited. No inheritance implies that any decoration lines drawn with the text—whether under, over, or through it—will always be the same color. This is true even if the descendant elements are a different color, as depicted in Figure 15-29:

```
p {text-decoration: underline; color: black;}
strong {color: gray;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> that has the black underline
beneath it as well.</p>
```

This paragraph, which is black and has a black underline, also contains **strongly emphasized text** that has the black underline beneath it as well.

Figure 15-29. Color consistency in underlines

Why is this so? Because the value of `text-decoration` is not inherited, the `<strong>` element assumes a default value of `none`. Therefore, the `<strong>` element has *no* underline. Now, there is very clearly a line under the `<strong>` element, so it seems silly to say that it

has none. Nevertheless, it doesn't. What you see under the `<strong>` element is the paragraph's underline, which is effectively "spanning" the `<strong>` element. You can see it more clearly if you alter the styles for the boldfaced element, like this:

```
p {text-decoration: underline; color: black;}
strong {color: gray; text-decoration: none;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> that has the black underline beneath
it as well.</p>
```

The result is identical to the one shown in Figure 15-29, since all you've done is to explicitly declare what was already the case. In other words, there is no way to "turn off" the decoration generated by a parent element.

There is a way to change the color of a decoration without violating the specification. As you'll recall, setting a text decoration on an element means that the entire element has the same color decoration, even if child elements have different colors. To match the decoration color with an element, you must explicitly declare its decoration, as follows:

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: underline;} /*could also use 'inherit'*/

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> that has the black underline
beneath it as well, but whose gray underline overlays the black underline
of its parent.</p>
```

In Figure 15-30, the `<strong>` element is set to be gray and to have an underline. The gray underline visually "overwrites" the parent's black underline, so the decoration's color matches the color of the `<strong>` element. The black underline is still there; the gray underline is just hiding it. If you move the gray underline with `text-underline-offset` or make the parent's `text-decoration-thickness` wider than its child, both underlines will be visible.

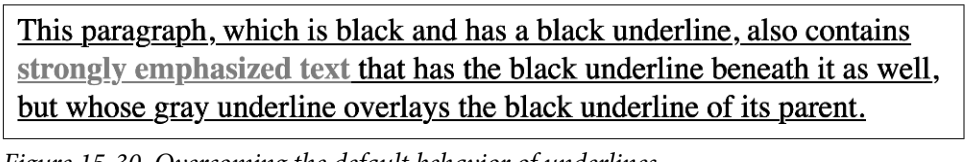


Figure 15-30. Overcoming the default behavior of underlines

When `text-decoration` is combined with `vertical-align`, even stranger things can happen. Figure 15-31 shows one of these oddities. Since the `<sup>` element has no decoration of its own, but it is elevated within an overlined element, the overline should cut through the middle of the `<sup>` element:

```
p {text-decoration: overline; font-size: 12pt;}
sup {vertical-align: 50%; font-size: 12pt;}
```

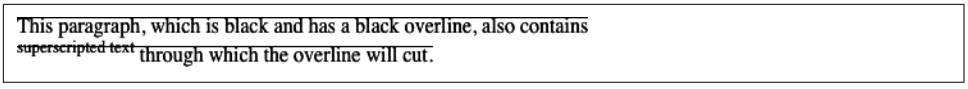


Figure 15-31. Correct, although strange, decorative behavior

But not all browsers do this. As of mid-2022, Chrome pushes the overline up so it is drawn across the top of the superscript, whereas others do not.

Text Rendering

A recent addition to CSS is `text-rendering`, which is actually an SVG property that's treated as CSS by supporting user agents. It lets you indicate what the user agent should prioritize when displaying text.

text-rendering

Values	`auto` | `optimizeSpeed` | `optimizeLegibility` | `geometricPrecision`
Initial value	`auto`
Applies to	All elements
Inherited	Yes
Animatable	Yes

The values `optimizeSpeed` and `optimizeLegibility` indicate that drawing speed should be favored over the use of legibility features like kerning and ligatures (for `optimizeSpeed`) or that such legibility features should be used even if that slows text rendering (for `optimizeLegibility`).

The precise legibility features that are used with `optimizeLegibility` are not explicitly defined, and the text rendering often depends on the operating system on which the user agent is running, so the exact results may vary. Figure 15-32 shows text optimized for speed and then optimized for legibility.

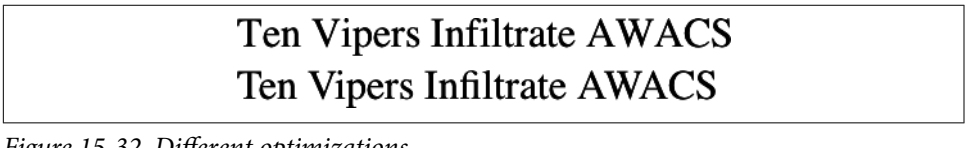


Figure 15-32. Different optimizations

As you can see in Figure 15-32, the differences between the two optimizations are objectively rather small, but they can have a noticeable impact on readability.



Some user agents will always optimize for legibility, even when optimizing for speed. This is likely an effect of rendering speeds having gotten so fast in the past few years.

The value `geometricPrecision`, on the other hand, directs the user agent to draw the text as precisely as possible, such that it could be scaled up or down with no loss of fidelity. You might think that this is always the case, but not so. Some fonts change kerning or ligature effects at different text sizes, for example, providing more kerning space at smaller sizes and tightening up the kerning space as the size is increased. With `geometricPrecision`, those hints are ignored as the text size changes. If it helps, think of it as the user agent drawing the text as though all the text is a series of SVG paths, not font glyphs.

Even by the usual standard of web standards, the value `auto` is pretty vaguely defined in SVG:

> The user agent shall make appropriate trade-offs to balance speed, legibility and geometric precision, but with legibility given more importance than speed and geometric precision.

That's it: user agents get to do what they think is appropriate, leaning toward legibility.

Text Shadows

Sometimes you just really need your text to cast a shadow, like when text overlaps a multicolored background. That's where `text-shadow` comes in. The syntax might look a little wacky at first, but it should become clear enough with just a little practice.

text-shadow

Values	none | [<*length*> || <*length*> <*length*> <*color*>?]#
Initial value	none
Applies to	All elements
Inherited	No
Animatable	Yes

The default is to not have a drop shadow for text. Otherwise, you can define one or more shadows. Each shadow is defined by an optional color and three length values, the last of which is also optional.

The color sets the shadow's color so it's possible to define green, purple, or even white shadows. If the color is omitted, the shadow defaults to the color keyword `currentcolor`, making it the same color as the text itself.

Using `currentcolor` as a default color may seem counterintuitive, as you might think shadows are purely decorative, but shadows can be used to improve legibility. A small shadow can make very thin text more legible. Defaulting to `currentcolor` allows adding thickness via a shadow that will always match the color of the text.

In addition to improving accessibility by making thin text thicker, shadows can be used to improve color contrast with a multicolored background. For example, if you have white text on a mostly dark black-and-white photo, adding a black shadow to the white text makes the edges of the white text visible even if the text is laid over white portions of the image.

The first two length values determine the offset distance of the shadow from the text; the first is the horizontal offset, and the second is the vertical offset. To define a solid, unblurred green shadow offset 5 pixels to the right and half an em down from the text, as shown in Figure 15-33, you could write either of the following:

```
text-shadow: green 5px 0.5em;
text-shadow: 5px 0.5em green;
```

Negative lengths cause the shadow to be offset to the left and upward from the original text. The following, also shown in Figure 15-33, places a light-blue shadow 5 pixels to the left and half an em above the text:

```
text-shadow: rgb(128,128,255) -5px -0.5em;
```

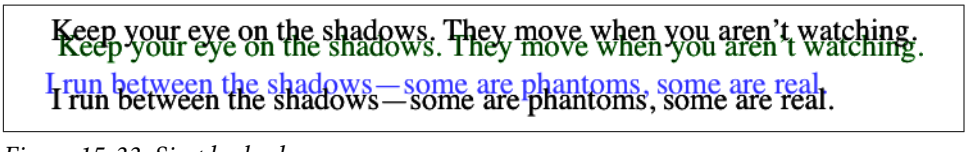


Figure 15-33. Simple shadows

While the offset may make the text take more visual space, shadows have no effect on line height and therefore no impact on the box model.

The optional third length value defines a blur radius for the shadow. The *blur radius* is defined as the distance from the shadow's outline to the edge of the blurring effect. A radius of 2 pixels would result in blurring that fills the space between the shadow's outline and the edge of the blurring. The exact blurring method is not defined, so different user agents might employ different effects. As an example, the following styles are rendered as shown in Figure 15-34:

```
p.cl1 {color: black; text-shadow: gray 2px 2px 4px;}
p.cl2 {color: white; text-shadow: 0 0 4px black;}
p.cl3 {color: black;
       text-shadow: 1em 0.5em 5px red,
                    -0.5em -1em hsla(100,75%,25%,0.33);}
```

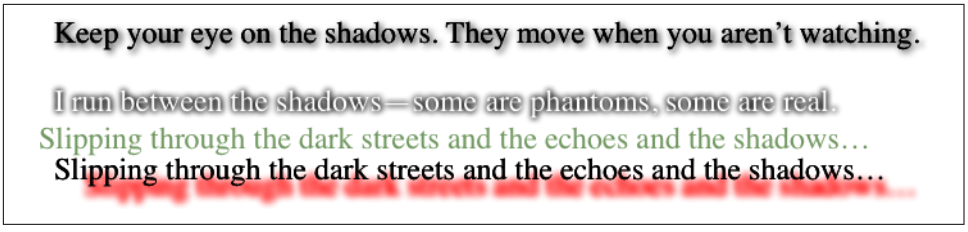


Figure 15-34. Dropping shadows all over



Large numbers of text shadows, or text shadows with very large blur values, can create performance slowdowns, particularly when animated in low-power and CPU-constrained situations such as mobile devices. Test thoroughly before deploying public designs that use text shadows.

Text Emphasis

Another way to call out text is by adding emphasis marks to each character. This is more common in ideographic languages like Chinese or Mongolian, but these marks can be added to any language's text with CSS. CSS has three text-emphasis properties similar to those for text decorations, and then a shorthand that conflates two of them.

Setting Emphasis Style

The most important of the three properties sets the type of emphasis mark, allowing you to pick from a list of common types or supply your own mark as a text string.

text-emphasis-style

Values	`none` | [[`filled` | `open`] || [`dot` | `circle` | `double-circle` | `triangle` | `sesame`]] | *<string>*
Initial value	`none`
Applies to	Text
Computed value	As declared, or `none` if nothing is declared
Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers support this as only `-webkit-text-emphasis-style`, except Firefox, which supports only `text-emphasis-style`

By default, text has no emphasis marks, or `none`. Alternatively, emphasis marks can be one of five shapes: `dot`, `circle`, `double-circle`, `triangle`, or `sesame`. Those shapes can be set as `filled`, which is the default; or `open`, which renders them as unfilled outlines. These are summarized in Table 15-1, and examples are shown in Figure 15-35.

Table 15-1. The predefined emphasis marks

Shape	`filled`	`open`
Sesame	﹅ (U+FE45)	﹆ (U+FE46)
Dot	• (U+2022)	◦ (U+25E6)
Circle	● (U+25CF)	○ (U+25CB)
Double-circle	◉ (U+25C9)	◎ (U+25CE)
Triangle	▲ (U+25B2)	△ (U+25B3)

The sesame is the most common mark used in vertical writing modes; the circle is the usual default in horizontal writing modes.

If the emphasis marks will not fit into the current text line's height, they will cause the height of that line of text to be increased until they fit without overlapping other lines. Unlike text decorations and text shadows, text emphasis marks *do* affect the line height.

If none of the predefined marks work in your specific situation, you can supply your own character as a string (a single character in single or double quotes). However, be careful: if the string is more than a single character, it may be reduced to the first character in the string by the browser. Thus, `text-emphasis-style: 'cool'` may result in the browser displaying only the `c` as a mark, as shown in Figure 15-35. Furthermore, the string symbols may or may not be rotated to match writing direction in vertical languages.

Here are some examples of setting emphasis marks:

```
h1 em {text-emphasis-style: triangle;}
strong a:any-link {text-emphasis-style: filled sesame;}
strong.callout {text-emphasis-style: open double-circle;}
```

A key difference between text emphasis and text decoration is that unlike decoration, emphasis is inherited. In other words, if you set a style of `filled sesame` on a paragraph, and that paragraph has child elements like links, those child elements will inherit the `filled sesame` value.

Another difference is that every glyph (character or other symbol) gets its own mark, and these marks are centered on the glyph. Thus, in proportional fonts like those seen in Figure 15-35, the marks will have different separations between them depending on which two glyphs are next to each other.

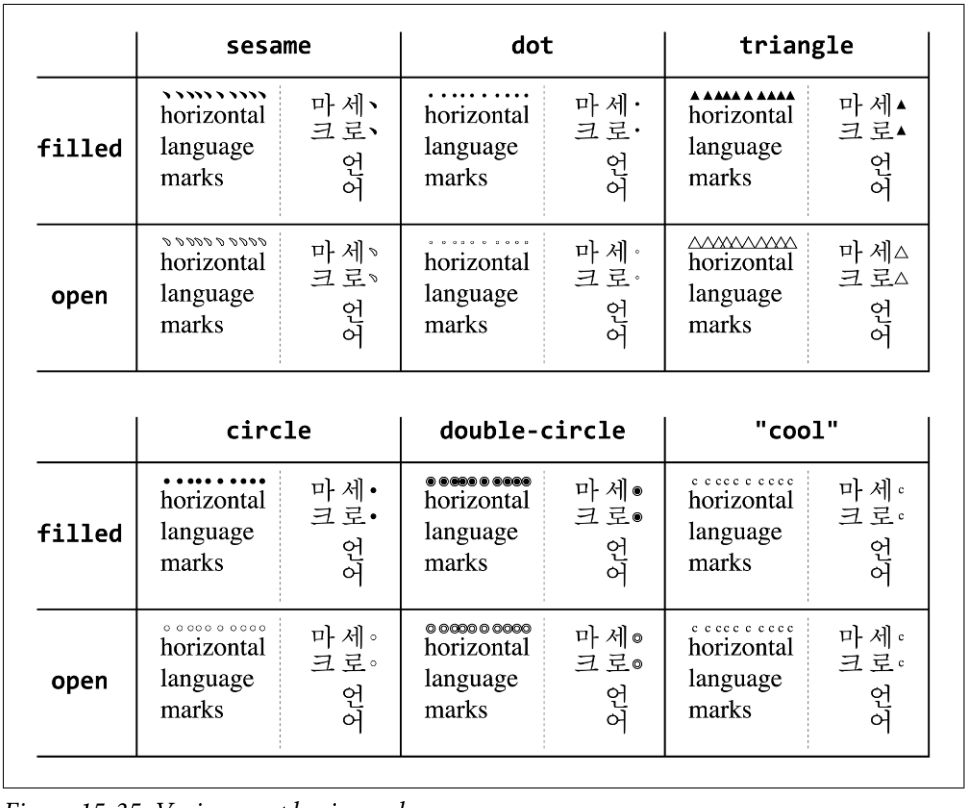


Figure 15-35. Various emphasis marks

The CSS specification recommends that emphasis marks be half the size of the text's font size, as if they were given `font-size: 50%`. They should otherwise use the same text styles as the text; thus, if the text is boldfaced, the emphasis marks should be as well. They should also use the text's color, unless overridden with the next property we'll cover.

Changing Emphasis Color

If you wish to have emphasis marks be a different color than the text they're marking, `text-emphasis-color` is here for you.

text-emphasis-color

Values	*<color>*
Initial value	`currentcolor`
Applies to	Text
Computed value	The computed color

Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers support this as only `-webkit-text-emphasis-color`, except Firefox, which supports only `text-emphasis-color`

The default value, as is often the case with color-related properties, is `currentcolor`. That ensures that emphasis marks will match the color of the text by default. To change it, you can do things like the following:

```
strong {text-emphasis-style: filled triangle;}
p.one strong {text-emphasis-color: gray;}
p.two strong {text-emphasis-color: hsl(0 0% 50%);}
/* these will yield the same visual result */
```

Placing Emphasis Marks

Thus far, we've shown emphasis marks in specific positions: above each glyph in horizontal text, and to the right of each glyph in vertical text. These are the default CSS values, but not always the preferred placement. The `text-emphasis-position` property allows you to change where marks are placed.

text-emphasis-position

Values	`[ over | under ] && [ right | left ]`
Initial value	`over right`
Applies to	Text
Computed value	As declared
Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers support this only in the form `-webkit-text-emphasis-position`, except Firefox, which supports only `text-emphasis-position`

The values `over` and `under` are applied only when the typographic mode is horizontal. Similarly, `right` and `left` are used only when the typographic mode is vertical.

This can be important in some Eastern languages. For example, Chinese, Japanese, Korean, and Mongolian all prefer to have marks to the right when the text is written vertically. They diverge on horizontal text: Chinese prefers marks below the text, and the rest prefer above the text, when it's horizontal. Thus you might write something like this in a stylesheet:

```
:lang(cn) {text-emphasis-position: under right;}
```

This would override the default `over right` when the text is marked as being Chinese, applying `under right` instead.

Using the text-emphasis Shorthand

A shorthand option exists for the `text-emphasis` properties, but it brings together only style and color.

text-emphasis

Values	*<text-emphasis-style>* ‖ *<text-emphasis-color>*
Initial value	See individual properties
Applies to	Text
Computed value	See individual properties
Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers support this only in the form `-webkit-text-emphasis-position`, except Firefox, which supports only `text-emphasis-position`

The reason `text-emphasis-position` is not included in the `text-emphasis` shorthand is so that it can (indeed must) be inherited separately. Therefore, the style and color of the marks can be changed via `text-emphasis` without overriding the position in the process.

As stated earlier, each character or ideogram or other glyph—what CSS calls a *typographic character unit*—gets its own emphasis mark. That is roughly correct, but exceptions occur. The following character units do *not* get emphasis marks:

- Word separators such as spaces, or any other Unicode separator character
- Punctuation characters, such as commas, full stops, and parentheses
- Unicode symbols corresponding to control codes, or any unassigned characters

Setting Text Drawing Order

Browsers are supposed to use a specific order to draw the text decorations, shadows, and emphasis marks we've discussed previously, along with the text itself. These are drawn in the following order, from bottommost (furthest away from the user) to topmost (closest to the user):

1. Shadows (`text-shadow`)
2. Underlines (`text-decoration`)
3. Overlines (`text-decoration`)

4. The actual text
5. Emphasis marks (`text-emphasis`)
6. Line-through (`text-decoration`)

Thus, the drop shadows of the text are placed behind everything else. Underlines and overlines go behind the text. Emphasis marks and line-throughs go on top of the text. Note that if you have top text-emphasis marks and an overline, the emphasis marks will be drawn on top of the overline, obfuscating the overline where they overlap.

Whitespace

Now that we've covered a variety of ways to style, decorate, and otherwise enhance the text, let's talk about the property `white-space`, which affects the user agent's handling of space, newline, and tab characters within the document source.

white-space

Values	`normal` | `nowrap` | `pre` | `pre-wrap` | `pre-line` | `break-spaces`
Initial value	`normal`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

By using the `white-space` property, you can affect how a browser treats the whitespace between words and lines of text. To a certain extent, default HTML handling already does this: it collapses any whitespace down to a single space. So given the following markup, the rendering in a web browser would show only one space between each word and ignore the line feed in the elements:

```
<p>This    paragraph    has     many spaces        in it.</p>
```

You can explicitly set this default behavior with the following declaration:

```
p {white-space: normal;}
```

This rule tells the browser to do as browsers have always done: discard extra whitespace. Given this value, line-feed characters (carriage returns) are converted into spaces, and any sequence of more than one space in a row is converted to a single space.

Should you set `white-space` to `pre`, however, the whitespace in an affected element is treated as though the elements were HTML `<pre>` elements; whitespace is *not* ignored, as shown in Figure 15-36:

```
p {white-space: pre;}

<p>This    paragraph   has     many
    spaces        in it.</p>
```

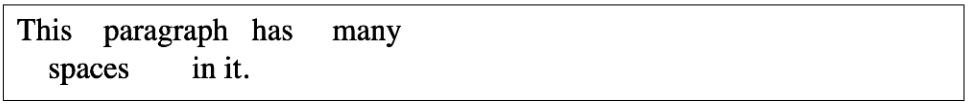


Figure 15-36. Honoring the spaces in markup

With a `white-space` value of `pre`, the browser will pay attention to extra spaces and even carriage returns. In this respect, any element can be made to act like a `<pre>` element.

The opposite value is `nowrap`, which prevents text from wrapping within an element, except wherever you use a `<br>` element. When text can't wrap and it gets too wide for its container, a horizontal scrollbar will appear by default (this can be changed using the `overflow` property). The effects of the following markup are shown in Figure 15-37:

```
<p style="white-space: nowrap;">This paragraph is not allowed to wrap,
which means that the only way to end a line is to insert a line-break
element.  If no such element is inserted, then the line will go forever,
forcing the user to scroll horizontally to read whatever can't be
initially displayed <br/>in the browser window.</p>
```

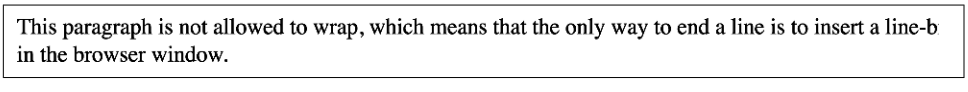


Figure 15-37. Suppressing line wrapping with the `white-space` property

If an element is set to `pre-wrap`, text within that element has whitespace sequences preserved, but text lines are wrapped normally. With this value, generated linebreaks as well as those found in the source markup are both honored.

The `pre-line` value is the opposite of `pre-wrap` and causes whitespace sequences to collapse as in normal text but honors new lines.

The `break-spaces` value is similar to `pre-wrap`, except that all whitespace is preserved, even at the end of the line, with a line-break opportunity after each whitespace character. These spaces take up space and do not hang, and thus affect the box's intrinsic sizes (min-content size and max-content size).

Table 15-2 summarizes the behaviors of the various `white-space` properties.

Table 15-2. `white-space` properties

Value	Whitespace	Line feeds	Auto line wrapping	Trailing whitespace
`pre-line`	Collapsed	Honored	Allowed	Removed
`normal`	Collapsed	Ignored	Allowed	Removed
`nowrap`	Collapsed	Ignored	Prevented	Removed

Value	Whitespace	Line feeds	Auto line wrapping	Trailing whitespace
`pre`	Preserved	Honored	Prevented	Preserved
`pre-wrap`	Preserved	Honored	Allowed	Hanging
`break-spaces`	Preserved	Honored	Allowed	Wrap

Consider the following markup, which has line-feed (e.g., return) characters to break lines, plus the end of each line has several extra space characters that aren't visible in the markup. The results are illustrated in Figure 15-38:

```
<p style="white-space: pre-wrap;">
This  paragraph      has  a  great   many   s p a c e s   within  its textual
  content,   but their    preservation     will    not    prevent   line
    wrapping or line breaking.
</p>
<p style="white-space: pre-line;">
This  paragraph      has  a  great   many   s p a c e s   within  its textual
  content,   but their collapse  will    not    prevent   line
    wrapping or line breaking.
<p style="white-space: break-spaces;">
This  paragraph      has  a  great   many   s p a c e s   within  its textual
  content,   but their preservation  will    not    prevent   line
    wrapping or line breaking.
</p>
```

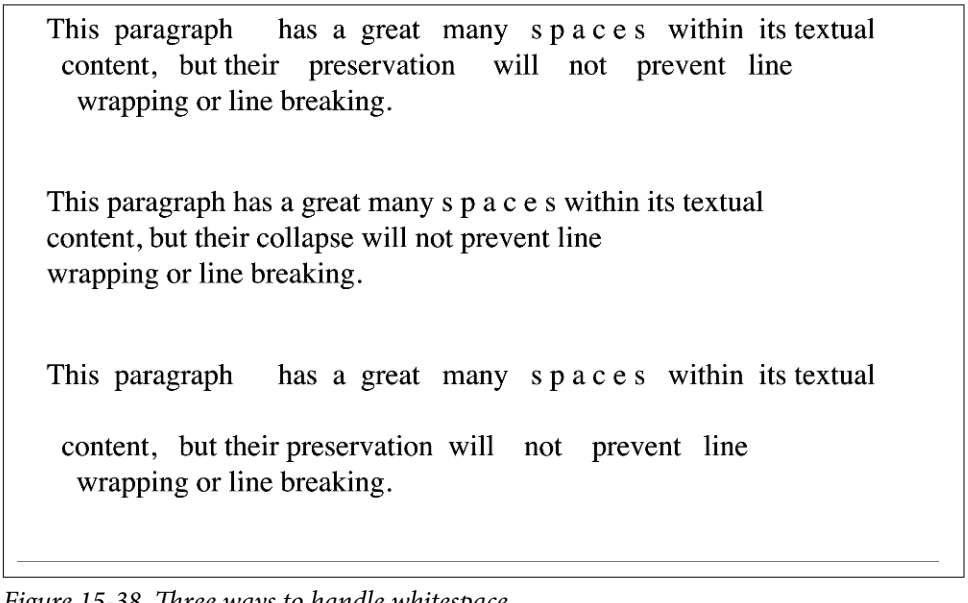


Figure 15-38. Three ways to handle whitespace

Notice that the third paragraph has a blank line between the first and second lines of text. This is because a line wrap was performed between two adjacent blank spaces at the end

of the line in the source markup. This didn't happen for `pre-wrap` or `pre-line`, because those `white-space` values don't allow hanging space to create line-wrap opportunities. The `break-spaces` value does.

Whitespace impacts several properties, including `tab-size`, which has no effect when the `white-space` property is set to a value in which whitespace is not maintained; and `overflow-wrap`, which has an effect only when `white-space` allows wrapping.

Setting Tab Sizes

Since whitespace is preserved in some values of `white-space`, it stands to reason that tabs (i.e., Unicode code point 0009) will be displayed as, well, tabs. But how many spaces should each tab equal? That's where `tab-size` comes in.

tab-size

Values	*<length>* | *<integer>*
Initial value	8
Applies to	Block elements
Computed value	The absolute-length equivalent of the specified value
Inherited	Yes
Animatable	Yes

By default, when whitespace is preserved, as with `white-space` values of `pre`, `pre-wrap`, and `break-spaces`, any tab character will be treated the same as eight spaces in a row, including any effects from `letter-spacing` and `word-spacing`. You can alter that by using a different integer value. Thus, `tab-size: 4` will cause each tab to be rendered as if it were four spaces in a row. Negative values are not allowed for `tab-size`.

If a length value is supplied, each tab is rendered using that length. For example, `tab-size: 10px` will cause a sequence of three tabs to be rendered as 30 pixels of whitespace. Some effects of `tab-size` are illustrated in Figure 15-39.

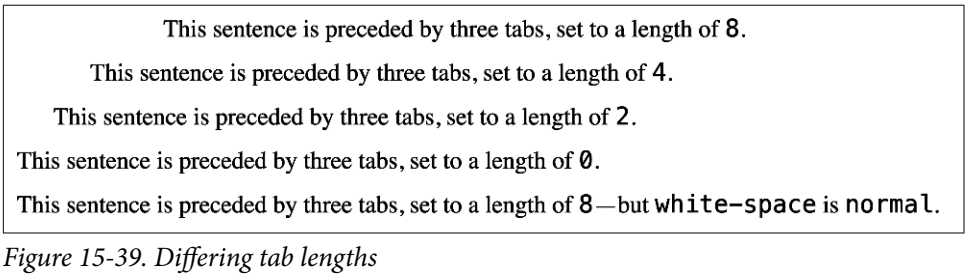


Figure 15-39. Differing tab lengths

Remember that `tab-size` is effectively ignored when the value of `white-space` causes whitespace to be collapsed (see Table 15-2). The value will still be computed in such cases, but there will be no visible effect no matter how many tabs appear in the source.

Wrapping and Hyphenation

Handling whitespace is all well and good, but it's a lot more common to want to influence the way the visible characters are handled when it comes to line wrapping. A few properties can influence where line wrapping is allowed, as well as enable hyphenation support.

Hyphenation

Hyphens can be very useful when displaying long words and short line lengths, such as blog posts on mobile devices and portions of *The Economist*. Authors can always insert their own hyphenation hints by using the Unicode character *U+00AD SOFT HYPHEN* (or, in HTML, ``), but CSS also offers a way to enable hyphenation without littering up the document with hints.

hyphens

Values	`manual` | `auto` | `none`
Initial value	`manual`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	No

With the default value of `manual`, hyphens are inserted only where manually inserted hyphenation markers occur in the document, such as U+00AD or ``. Otherwise, no hyphenation occurs. The value `none`, on the other hand, suppresses any hyphenation, even if manual break markers are present; thus, U+00AD and `` are ignored.



The `<wbr>` element does not introduce a hyphen at the line-break point. To make a hyphen appear only at the end of a line, use the soft hyphen character entity (``) instead.

The far more interesting (and potentially inconsistent) value is `auto`, which permits the browser to insert hyphens and break words at "appropriate" places inside words, even where no manually inserted hyphenation breaks exist. But what constitutes a *word*? And,

under what circumstances is it appropriate to hyphenate a word? Both are language dependent. User agents are supposed to prefer manually inserted hyphen breaks to automatically determined breaks, but there are no guarantees. An illustration of hyphenation, or the suppression thereof, in the following example is shown in Figure 15-40:

```
.cl01 {hyphens: auto;}
.cl02 {hyphens: manual;}
.cl03 {hyphens: none;}

<p class="cl01">Supercalifragilisticexpialidocious
  antidisestablishmentarianism.</p>
<p class="cl02">Supercalifragilisticexpialidocious
  antidisestablishmentarianism.</p>
<p class="cl02">Super&shy;cali&shy;fragi&shy;listic&shy;expi&shy;ali&shy;
docious anti&shy;dis&shy;establish&shy;ment&shy;arian&shy;ism.</p>
<p class="cl03">Super&shy;cali&shy;fragi&shy;listic&shy;expi&shy;ali&shy;
docious anti&shy;dis&shy;establish&shy;ment&shy;arian&shy;ism.</p>
```

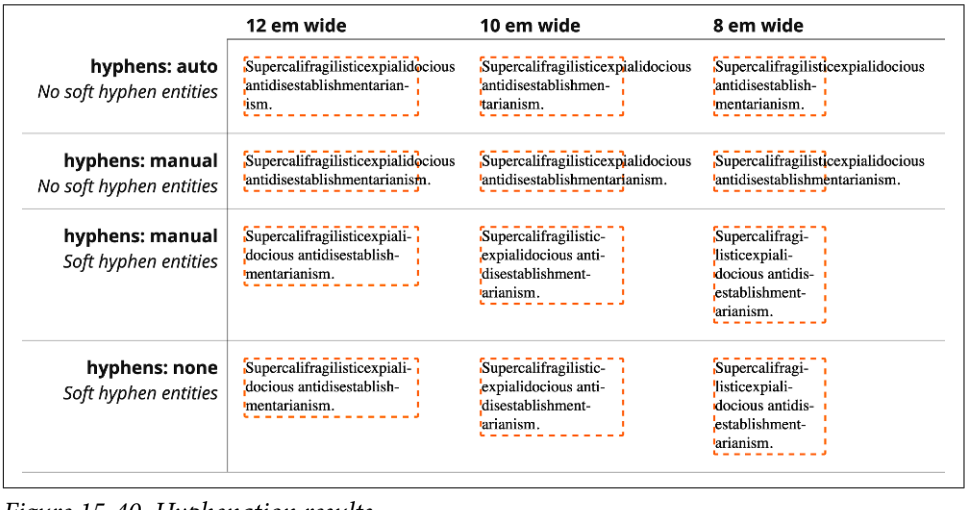


Figure 15-40. Hyphenation results

Because hyphenation is language dependent, and because the CSS specifications do not define precise (or even vague) rules for user agents, hyphenation may differ by browser.

If you do choose to hyphenate, be careful about the elements to which you apply the hyphenation. The `hyphens` property is inherited, so declaring `body {hyphens: auto;}` will apply hyphenation to everything in your document—including text areas, code samples, block quotes, and so on. Blocking automatic hyphenation at the level of those elements is probably a good idea, using rules that are something like this:

```
body {hyphens: auto;}
code, var, kbd, samp, tt, dir, listing, plaintext, xmp, abbr, acronym,
blockquote, q, textarea, input, option {hyphens: manual;}
```

It's usually a good idea to suppress hyphenation in code samples and code blocks, especially in languages that use hyphens in things like property and value names. (Ahem.) Similar logic holds for keyboard input text—you likely don't want a stray dash getting into your Unix command-line examples! And so on down the line. If you decide that you want to hyphenate some of these elements, just remove them from the selector.



It is strongly advised to set the `lang` attribute on HTML elements to enable hyphenation support and improve accessibility. As of mid-2022, `hyphens` is supported in Firefox for 30+ languages, Safari supports many European languages, but Chrome-related browsers support only English.

Hyphens can be suppressed by the effects of other properties. For example, `word-break` affects the way soft wrapping of text is calculated in various languages, determining whether line breaks appear where text would otherwise overflow its content box.

Word Breaking

When a run of text is too long to fit into a single line, it is *soft wrapped*. This is in contrast to *hard wraps*, which include line-feed characters and `<br>` elements. Where the text is soft wrapped is determined by the user agent, but `word-break` lets authors influence that decision making.

word-break

Values	`normal` | `break-all` | `keep-all` | `break-word`
Initial value	`normal`
Applies to	Text
Computed value	As specified
Inherited	Yes
Animatable	No
Note	`break-word` is a legacy value and has been deprecated

The default value of `normal` means that text should be wrapped as it always has been. In practical terms, this means that text is broken between words, though the definition of a word varies by language. In Latin-derived languages like English, this is almost always a space between letter sequences (e.g., words) or at hyphens. In ideographic languages like Japanese, each symbol can be a complete word, so breaks can occur between any two symbols. In other ideographic languages, though, the soft-wrap points may be limited to appear between sequences of symbols that are not space separated. Again, that's all by default and is the way browsers have handled text for years.

If you apply the value `break-all`, soft wrapping can (and will) occur between any two characters, even if they are in the middle of a word. With this value, no hyphens are shown, even if the soft wrapping occurs at a hyphenation point (see "Hyphenation" on page 776). Note that values of the `line-break` property (described next) can affect the behavior of `break-all` in ideographic text.

The `keep-all` value, on the other hand, suppresses soft wrapping between characters, even in ideographic languages where each symbol is a word. Thus, in Japanese, a sequence of symbols with no whitespace will not be soft wrapped, even if this means the text line will exceed the length of its element. (This behavior is similar to `white-space: pre`.)

Figure 15-41 shows a few examples of `word-break` values, and Table 15-3 summarizes the effects of each value.

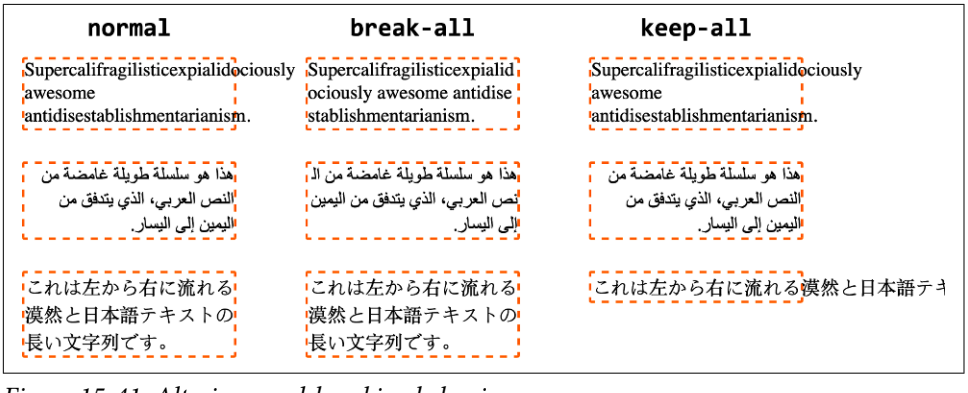


Figure 15-41. Altering word-breaking behavior

Table 15-3. Word-breaking behavior

Value	Non-CJK	CJK	Hyphenation permitted
`normal`	As usual	As usual	Yes
`break-all`	After any character	After any character	No
`keep-all`	As usual	Around sequences	Yes

As noted previously, the value `break-word` has been deprecated, although it is supported by all known browsers as of mid-2022. When used, it has the same effect as {`word-break: normal; overflow-wrap: anywhere;`}, even if `overflow-wrap` has a different value. (We'll cover `overflow-wrap` in "Wrapping Text" on page 781.)

Line Breaking

If your interests run to CJK text, then in addition to `word-break`, you will also want to get to know `line-break`.

line-break

Values	`auto` | `loose` | `normal` | `strict` | `anywhere`
Initial value	`auto`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

As you just saw, `word-break` can affect the way lines of text are soft wrapped in CJK text. The `line-break` property also affects such soft wrapping, specifically how wrapping is handled around CJK-specific symbols and around non-CJK punctuation (such as exclamation points, hyphens, and ellipses) that appears in text declared to be CJK.

In other words, `line-break` applies to certain CJK characters all the time, regardless of the content's declared language. If you throw some CJK characters into a paragraph of English text, `line-break` will still apply to them, but not to anything else in the text. Conversely, if you declare content to be in a CJK language, `line-break` will continue to apply to those CJK characters *plus* a number of non-CJK characters within the CJK text. These include punctuation marks, currency symbols, and a few other symbols.

No authoritative list exists of which characters are affected and which are not, but the specification provides a list of recommended symbols and behaviors around those symbols.

The default value `auto` allows user agents to soft wrap text as they like, and more importantly lets user agents vary line breaking based on the situation. For example, the user agent can use looser line-breaking rules for short lines of text and stricter rules for long lines. In effect, `auto` allows the user agent to switch among the `loose`, `normal`, and `strict` values as needed, possibly even on a line-by-line basis within a single element.

You can perhaps infer that those other values have the following general meanings:

`loose`
: This value imposes the "least restrictive" rules for wrapping text, and is meant for use when line lengths are short, such as in newspapers.

`normal`
: This value imposes the "most common" rules for wrapping text. What exactly "most common" means is not precisely defined, though there is the aforementioned list of recommended behaviors.

`strict`
: This value imposes the "most stringent" rules for wrapping text. Again, this is not precisely defined.

`anywhere`
: This value creates a line-breaking opportunity around every typographic unit, including whitespace and punctuation marks. A soft wrap can even happen in the middle of a word, and hyphenation is not applied in such circumstances.

Wrapping Text

After all that information about hyphenation and soft wrapping, what happens when text overflows its container anyway? That's what `overflow-wrap` addresses.

Originally called `word-wrap`, the `overflow-wrap` property applies to inline elements, setting whether the browser should insert line breaks within otherwise unbreakable strings in order to prevent text from overflowing its line box. In contrast to `word-break`, `overflow-wrap` will create a break only if an entire word cannot be placed on its own line without overflowing.

overflow-wrap

Values	`normal` | `break-word` | `anywhere`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

This property is less straightforward than it first appears, because its primary effect is to change how word wrapping and minimum-content sizing (which we haven't even had a chance to discuss yet) interact in trying to avoid overflow at the ends of text lines.



The `overflow-wrap` property can operate only if the value of `white-space` allows line wrapping. If it does not (e.g., with the value `pre`), `overflow-wrap` has no effect.

If the default value of `normal` is in effect, wrapping happens as normal—between words or as directed by the language. If a word is longer than the width of the element containing it, the word will "spill out" of the element box, just as on the classic CSS IS AWESOME coffee mug. (Google it if you haven't seen it before. It's worth the chuckle.)

If the `break-word` value is applied, wrapping can happen in the middle of words, with no hyphen placed at the site of the wrapping, but this will happen so that line lengths will be as wide as the element's width. In other words, if the `width` property of the element is given the value `min-content`, the "minimum content" calculations will assume that content strings must be as long as possible.

By contrast, when `anywhere` is set, the "minimum content" calculations will take line-wrapping opportunities into account. This means, in effect, that the minimum-content width will be the width of the widest character in the element's content. Only when two skinny characters are next to each other will they have a chance to be on the same line together, and in a monospace font every line of text will be a single character. Figure 15-42 illustrates the difference between these three values.

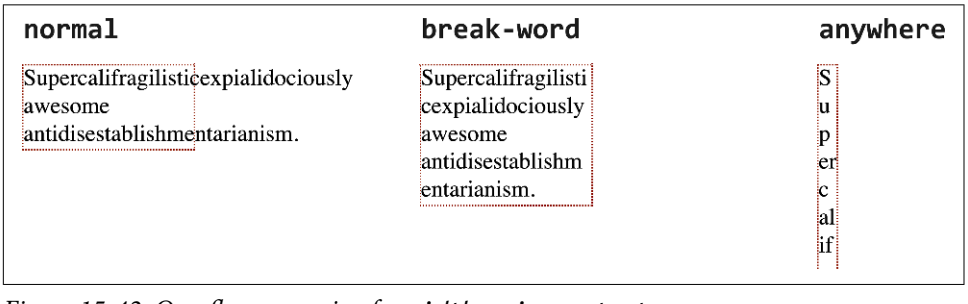


Figure 15-42. Overflow wrapping for `width: min-content`

If the value of `width` is something other than `min-content`, then `break-word` and `anywhere` will have the same results. Really, the only difference between the two values is that with `anywhere`, soft-wrap opportunities introduced by the word break are considered when calculating min-content intrinsic sizes. With `break-word`, they are not considered.

While `overflow-wrap: break-word` may appear very similar to `word-break: break-all`, they are not the same. To see why, compare the second box in Figure 15-42 to the top-middle box in Figure 15-41. As it shows, `overflow-wrap` kicks in only if content actually overflows; thus, when there is an opportunity to use whitespace in the source to wrap lines, `overflow-wrap` will take it. By contrast, `word-break: break-all` will cause wrapping when content reaches the wrapping edge, regardless of any whitespace that comes earlier in the line.

Once upon a time there was a property called `word-wrap` that did exactly what `overflow-wrap` does. The two are so identical that the specification explicitly states that user agents "must treat `word-wrap` as an alternate name for the `overflow-wrap` property, as if it were a shorthand of `overflow-wrap`."

Writing Modes

Earlier, we discussed inline direction and introduced the topic of reading direction. You've already seen numerous benefits of including the `lang` attribute in your HTML, from being able to style based on language selectors, to allowing the user agent to hyphenate. Generally, you should let the user agent handle the direction of text based on the language attribute, but CSS does provide properties for the rare occasions when an override is necessary.

Setting Writing Modes

The property used for specifying one of five available writing modes is, of all things, `writing-mode`. This property sets the block-flow direction of the element, which determines how boxes are stacked together.

writing-mode

Values	`horizontal-tb` | `vertical-rl` | `vertical-lr` | `sideways-rl` | `sideways-lr`
Initial value	`horizontal-tb`
Applies to	All elements except table row groups, table column groups, table rows, table columns, Ruby base containers, and Ruby annotation containers
Computed value	As specified
Inherited	Yes
Animatable	Yes

The default value, `horizontal-tb`, means "a horizontal inline direction, and a top-to-bottom block direction." This covers all Western and some Middle Eastern languages, which may differ in the direction of their horizontal writing. The other two values offer a vertical inline direction, and either an RTL or LTR block direction.

The `sideways-rl` and `sideways-lr` values take horizontal text and turn its flow "sideways," with the direction the text runs either going right to left (for `sidewyas-rl`) or left to right (for `sideways-lr`). The difference between these values and the vertical values is that the text is turned whichever way is necessary to make the text read naturally.

Figure 15-43 illustrates all five values.

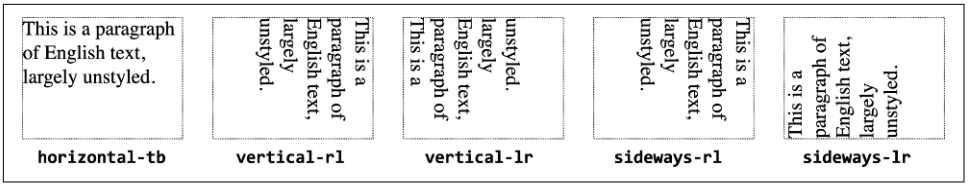


Figure 15-43. Writing modes

Notice how the lines are strung together in the two `vertical-` examples. If you tilt your head to the right, the text in `vertical-rl` is at least readable. The text in `vertical-lr`, on the other hand, is difficult to read because it appears to flow from bottom to top, at least when arranging English text. This is not a problem in languages that use `vertical-lr` flow, such as forms of Japanese.

In vertical writing modes, the block direction is horizontal, which means vertical alignment of inline elements causes them to move horizontally. This is illustrated in Figure 15-44.

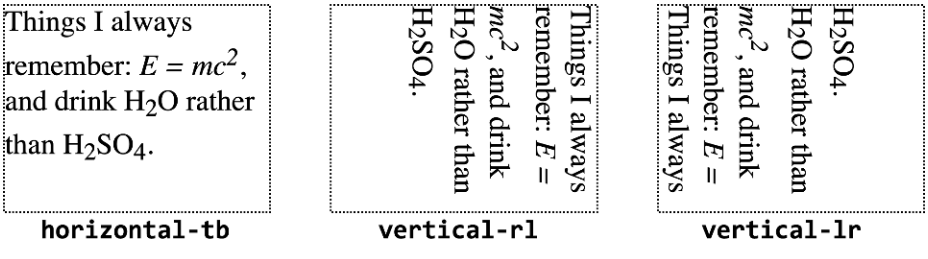


Figure 15-44. Writing modes and "vertical" alignment

All the super- and subscript elements cause horizontal shifts, both of themselves and the placement of the lines they occupy, even though the property used to move them is `vertical-align`. As described earlier, the vertical displacement is with respect to the line box, where the box's baseline is defined as horizontal—even when it's being drawn vertically.

Confused? It's OK. Writing modes are likely to confuse you, because they're such a different way of thinking *and* because old assumptions in the CSS specification clash with the new capabilities. If vertical writing modes had been supported from the outset, `vertical-align` would likely have a different name—`inline-align` or something like that. (Maybe one day that will happen.)

Changing Text Orientation

Once you've settled on a writing mode, you may decide you want to change the orientation of characters within those lines of text. You might want to do this for various reasons, not the least of which is using different writing systems that are commingled, such as

Japanese text with English words or numbers mixed in. In these cases, `text-orientation` is the answer.

text-orientation

Values	`mixed` | `upright` | `sideways`
Initial value	`mixed`
Applies to	All elements except table row groups, table rows, table column groups, and table columns
Computed value	As specified
Inherited	Yes
Animatable	Yes

The `text-orientation` property affects the way characters are oriented. What that means is best illustrated by the following styles, rendered in Figure 15-45:

```
.verts {writing-mode: vertical-lr;}
#one {text-orientation: mixed;}
#two {text-orientation: upright;}
#thr {text-orientation: sideways;}
```

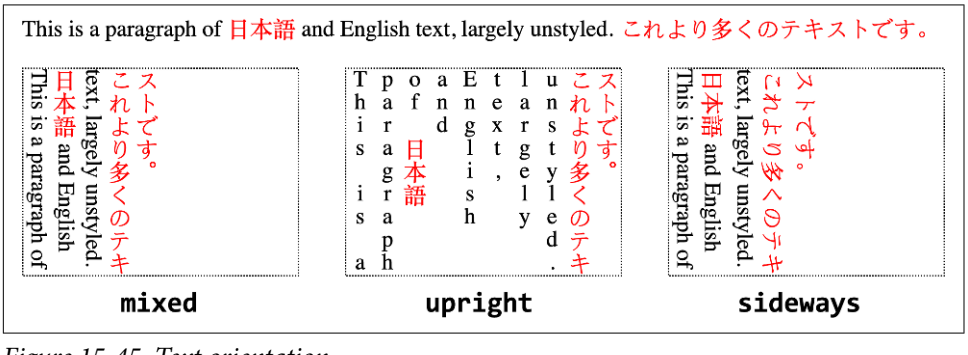


Figure 15-45. Text orientation

Across the top of Figure 15-45 is a basically unstyled paragraph of mixed Japanese and English text. Below that are three copies of that paragraph, using the writing mode `vertical-lr`. In the first, `text-orientation: mixed` writes the horizontal-script characters (the English) sideways, and the vertical-script characters (the Japanese) upright. In the second, all characters are `upright`, including the English characters. In the third, all characters are `sideways`, including the Japanese characters.



As of mid-2022, `sideways` is not supported by Chromium browsers.

Combining Characters

Only relevant to vertical writing modes, the `text-combine-upright` property enables displaying a subset of characters upright within vertical text. This can be useful when mixing languages or pieces of languages, such as embedding Arabic numerals in CJK text, but may have other applications.

text-combine-upright

Values	`none` | `all` | [`digits` <*integer*>?]
Initial value	`none`
Applies to	Nonreplaced inline elements
Computed value	Specified keyword, plus integer if `digits`
Inherited	Yes
Animatable	No
Note	For <*integer*> values, only the numbers 2, 3, and 4 are valid

Essentially, this property lets you say whether characters may sit next to each other horizontally while being part of a vertical line of text. Your choices are whether to allow this for all characters or for only a few numeric digits.

Here's how it works: as a line of vertical text is laid out, the browser can consider whether the width of two characters, sitting next to each other, is less than or equal to the value of `1em` for the text. If so, they may be placed next to each other, effectively putting two characters into the space of one. If not, the first character is placed alone, and the process continues.

As of mid-2022, this can lead to characters being very, very squished. For an example, consider the following markup and CSS:

```
<div lang="zh-Hant">
<p>这是一些文本</p>
<p class="combine">这是一些文本</p>
<p>这是 117 一些 0 文本 23 日</p>
<p class="combine">这是 117 一些 0 文本 23 日</p>
<p class="combine">
   这是<span>117</span>一些<span>0</span>文本<span>23</span>日</p>
<p>这是<span class="combine">117</span>一些<span
```

```
    class="combine">0</span>文本<span class="combine">23</span>日</p>
</div>

p {writing-mode: vertical-rl;}
.combine {text-combine-upright: all;}
```

All of the paragraphs are written using `writing-mode: vertical-rl`, but some are set to `text-combine-upright: all`, and others are not. The last paragraph is not set to `all`, but the `<span>` elements within it have been. Figure 15-46 shows the result.

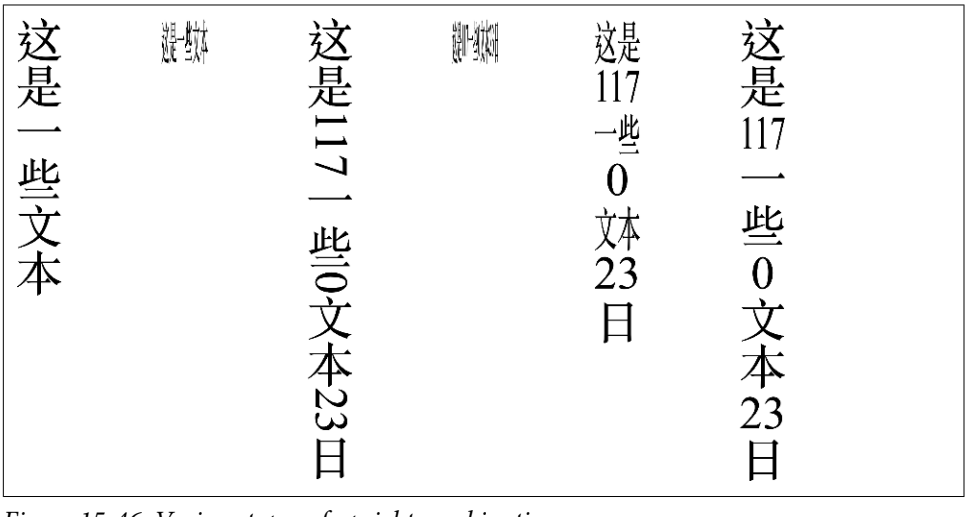


Figure 15-46. Various types of upright combination

Lest you think a bug is at work here, the results are consistent across browsers (as of mid-2022). The second and fourth columns have every single character, whether Chinese ideographs or Arabic numerals, squished horizontally to fit on a single line.

A way around this is to break up the text with child elements, as shown in the fifth and sixth columns. In the first, numbers are surrounded with `<span>` elements, which break up the fitting process. This works as long as no run of text has too many characters; beyond two or three symbols, the text becomes progressively more difficult to comprehend.

The sixth column shows a way to hack around the problem: apply `text-combine-upright: all` to only the `<span>` elements, which are already used to wrap the Arabic numerals, by giving each `<span>` a `class` value of `combine`. In that case, the `.combine` rule will apply only to the `<span>` elements, not all the text in the paragraph.

This is what the `digits` value is supposed to make possible without the need for all the extra markup. Theoretically, you could get the same result as that shown in the sixth column of Figure 15-46 by applying the following CSS to the paragraph that has no `<span>` elements in it:

```
p {writing-mode: vertical-rl; text-upright-combine: digits 4;}
```

Sadly, as of mid-2022, no browser supports this behavior, unless you count Internet Explorer 11 using the alternate property name `-ms-text-combine-horizontal`.

Declaring Direction

Harking back to the days of CSS2, a pair of properties could be used to affect the direction of text by changing the inline baseline direction: `direction` and `unicode-bidi`. These should generally not be used today, but are covered here in case you come across them in legacy code.



The CSS specification explicitly warns *against* using `direction` and `unicode-bidi` in CSS when applied to HTML documents. To quote: "Because HTML [user agents] can turn off CSS styling, we recommend…the HTML `dir` attribute and `<bdo>` element to ensure correct bidirectional layout in the absence of a style sheet."

direction

Values	`ltr` | `rtl`
Initial value	`ltr`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

The `direction` property affects the writing direction of text in a block-level element, the direction of table-column layout, the direction in which content horizontally overflows its element box, and the position of the last line of a fully justified element. For inline elements, direction applies only if the property `unicode-bidi` is set to either `embed` or `bidi-override` (see the following description of `unicode-bidi`).

Although `ltr` is the default, it is expected that if a browser is displaying RTL text, the value will be changed to `rtl`. Thus, a browser might carry an internal rule stating something like the following:

```
*:lang(ar), *:lang(he) {direction: rtl;}
```

The real rule would be longer and encompass all RTL languages, not just Arabic and Hebrew, but it illustrates the point.

While CSS attempts to address writing direction, Unicode has a much more robust method for handling directionality. With the property `unicode-bidi`, CSS authors can take advantage of some of Unicode's capabilities.

unicode-bidi

Values	`normal` | `embed` | `bidi-override`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	Yes

Here we'll simply quote the value descriptions from the CSS 2.1 specification, which do a good job of capturing the essence of each value:

`normal`
: The element does not open an additional level of embedding with respect to the bidirectional algorithm. For inline-level elements, implicit reordering works across element boundaries.

`embed`
: If the element is inline-level, this value opens an additional level of embedding with respect to the bidirectional algorithm. The direction of this embedding level is given by the `direction` property. Inside the element, reordering is done implicitly. This corresponds to adding a "left-to-right embedding" character (U+202A; for `direction: ltr`) or a "right-to-left embedding" character (U+202B; for `direction: rtl`) at the start of the element and a "pop directional formatting" character (U+202C) at the end of the element.

`bidi-override`
: This creates an override for inline-level elements. For block-level elements, this creates an override for inline-level descendants not within another block. This means that, inside the element, reordering is strictly in sequence according to the `direction` property; the implicit part of the bidirectional algorithm is ignored. This corresponds to adding a "left-to-right override" character (U+202D; for `direction: ltr`) or "right-to-left override" character (U+202E; for `direction: rtl`) at the start of the element and a "pop directional formatting" character (U+202C) at the end of the element.

Summary

Even without altering the font face, we have many ways to change the appearance of text. In addition to classic effects such as underlining, CSS enables you to draw lines over text or through it, change the amount of space between words and letters, indent the first line of a paragraph (or other block-level element), align text in various ways, exert influence over the hyphenation and line breaking of text, and much more. You can even alter the amount of space between lines of text. CSS also supports languages other than those that are written left to right, top to bottom. Given that so much of the web is text, the strength of these properties makes a great deal of sense.

CHAPTER 16

Lists and Generated Content

In the realm of CSS layout, lists are an interesting case. The items in a list are simply block boxes, but with an extra bit that doesn't really participate in the document layout hanging off to one side. With an ordered list, that extra bit contains a series of increasing numbers (or letters) that are calculated and mostly formatted by the user agent, not the author. Taking a cue from the document structure, the user agent generates the numbers and their basic presentation.

With CSS, you can define your own counting patterns and formats, and associate those counters with *any* element, not just ordered list items. Furthermore, this basic mechanism makes it possible to insert other kinds of content, including text strings, attribute values, or even external resources, into a document. Thus, it is possible to use CSS to insert link icons, editorial symbols, and more into a design without having to create extra markup.

To see how all these list options fit together, we'll explore basic list styling before moving on to examine the generation of content and counters.

Working with Lists

In a sense, almost anything that isn't narrative text can be considered a list. The US Census, the solar system, my family tree, a restaurant menu, and even all of the friends you've ever had can be represented as a list, or perhaps as a list of lists. These many variations make lists fairly important, which is why it's a shame that list styling in CSS isn't more sophisticated.

The simplest (and best-supported) way to affect a list's styles is to change its marker type. The *marker* of a list item is, for example, the bullet that appears next to each item in an unordered list. In an ordered list, the marker could be a letter, a number, or a symbol from some other counting system. You can even replace the markers with images. All of these are accomplished using the different list-style properties.

Types of Lists

To change the type of marker used for a list's items, use the `list-style-type` property.

list-style-type

Values	*<counter-style>* | *<string>* | none
Initial value	`disc`
Applies to	Elements whose `display` value is `list-item`
Inherited	Yes
Computed value	As specified

You can use a string of text as the marker, such as `list-style-type: "▷"`. In addition, *<counter-style>* stands in for a long list of possible keywords or a custom-defined counter style defined with `@counter-style` (see "Defining Counting Patterns" on page 819). A few examples of these list style types are shown in Figure 16-1.

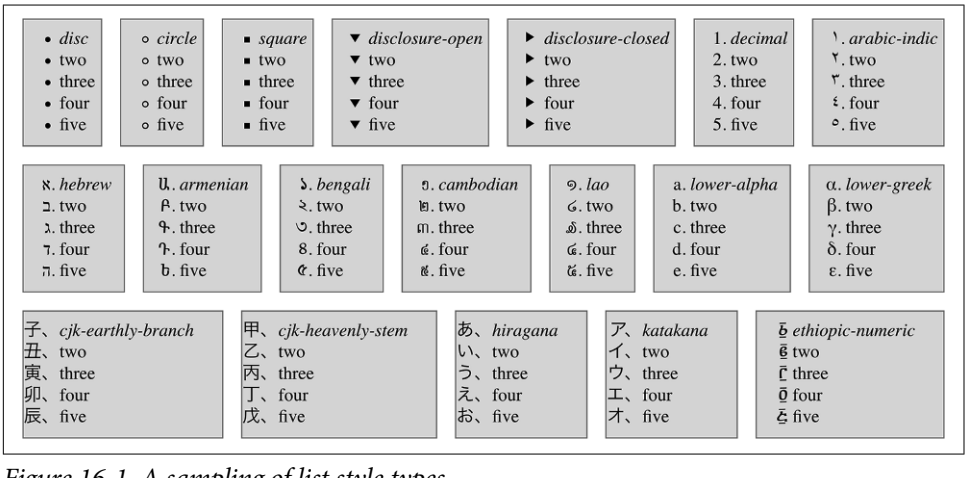


Figure 16-1. A sampling of list style types

The keywords (plus some browser-specific extras) are listed here:

afar †	ethiopic-halehame-sid-et †	mongolian
amaric †	ethiopic-halehame-so-et †	myanmar
amaric-abegede †	ethiopic-halehame-ti-er -	octal †
arabic-indic	ethiopic-halehame-ti-et -	oriya
armenian	ethiopic-halehame-tig †	oromo †
asterisks †	ethiopic-numeric	persian
bengali	footnotes †	sidama
binary †	georgian	simp-chinese-formal
cambodian	gujarati	simp-chinese-informal
circle	gurmukhi	somali †
cjk-decimal *	hangul -	square
cjk-earthly-branch	hangul-consonant -	symbols *
cjk-heavenly-stem	hebrew	tamil *
cjk-ideographic	hiragana	telugu
decimal	hiragana-iroha	thai
decimal-leading-zero	japanese-formal	tibetan
devanagari	japanese-informal	tigre †
disc	kannada	tigrinya-er †
disclosure-closed	katakana	tigrinya-er-abegede †
disclosure-open	katakana-iroha	tigrinya-et †
ethiopic †	khmer	tigrinya-et-abegede †
ethiopic-abegede †	korean-hangul-formal	trad-chinese-formal
ethiopic-abegede-am-et †	korean-hanja-formal	trad-chinese-informal
ethiopic-abegede-gez †	korean-hanja-informal	upper-alpha
ethiopic-abegede-ti-er †	lao	upper-armenian
ethiopic-abegede-ti-et †	lower-alpha	upper-greek
ethiopic-halehame ‡, -	lower-armenian	upper-hexadecimal †
ethiopic-halehame-aa-er †	lower-greek	upper-latin
ethiopic-halehame-aa-et †	lower-hexadecimal †	upper-norwegian †
ethiopic-halehame-am -	lower-latin	upper-roman
ethiopic-halehame-am-et †	lower-norwegian †	urdu -
ethiopic-halehame-gez †	lower-roman	
ethiopic-halehame-om-et †	malayalam	

† WebKit only

‡ All engines *except* WebKit

* Mozilla only

- Requires `-moz-` prefix in Firefox

If you provide a counter style that the browser does not recognize, such as declaring `list-style-type: lower-hexadecimal` and loading the page, some browsers, including Firefox, Edge, and Chrome, will assume `decimal` instead. Safari will ignore values it does not understand as invalid.

The `list-style-type` property, as well as all other list-related properties, can be applied only to an element that has a `display` of `list-item`, but CSS doesn't distinguish between

ordered and unordered list items. Thus, you can set an ordered list to use discs instead of numbers. In fact, the default value of `list-style-type` is `disc`, so you might theorize that without explicit declarations to the contrary, all lists (ordered or unordered) will use discs as the marker for each item. This would be logical, but, as it turns out, it's up to the user agent to decide. Even if the user agent doesn't have a predefined rule such as `ol {list-style-type: decimal;}`, it may prohibit ordered markers from being applied to unordered lists, and vice versa. You can't count on this, so be careful.

If you want to suppress the display of markers altogether, you should use `none`. This value causes the user agent to refrain from putting anything where the marker would ordinarily be, although it does not interrupt the counting in ordered lists. Thus, the following markup would have the result shown in Figure 16-2:

```
ol li {list-style-type: decimal;}
li.off {list-style-type: none;}

<ol>
<li>Item the first
<li class="off">Item the second
<li>Item the third
<li class="off">Item the fourth
<li>Item the fifth
</ol>
```



Figure 16-2. Switching off list-item markers

The `list-style-type` property is inherited, so if you want to have different styles of markers in nested lists, you'll likely need to define them individually. You may also have to explicitly declare styles for nested lists because the user agent's stylesheet may have already defined them. For example, assume that a user agent has the following styles defined:

```
ul {list-style-type: disc;}
ul ul {list-style-type: circle;}
ul ul ul {list-style-type: square;}
```

If this is the case—and it's likely that this, or something like it, will be—you will have to declare your own styles to overcome the user agent's styles. Inheritance won't be enough.

String markers

CSS also allows authors to supply string values as list markers. This opens the field to anything you can input from the keyboard, as long as you don't mind having the same string used for every marker in the list. Figure 16-3 shows the results of the following styles:

```
.list01 {list-style-type: "%";}
.list02 {list-style-type: "Hi! ";}
.list03 {list-style-type: "†";}
.list04 {list-style-type: "⌘";}
.list05 {list-style-type: "🤔";}
```



Figure 16-3. A sampling of string markers

List-Item Images

Sometimes a regular text marker just won't do. You might prefer to use an image for each marker, which is possible with the property `list-style-image`.

list-style-image

Values	*<uri>* | *<image>* | `none` | `inherit`
Initial value	`none`
Applies to	Elements whose `display` value is `list-item`
Inherited	Yes
Computed value	For *<uri>* values, the absolute URI; otherwise, `none`

Here's how it works:

```
ul li {list-style-image: url(ohio.gif);}
```

Yes, it's really that simple. One simple `url()` value, and you're putting images in for markers, as you can see in Figure 16-4.



Figure 16-4. Using images as markers

List image markers are displayed at their full size, so exercise care in the images you use, as the example shown in Figure 16-5 makes clear with its oversized markers:

```
ul li {list-style-image: url(big-ohio.gif);}
```

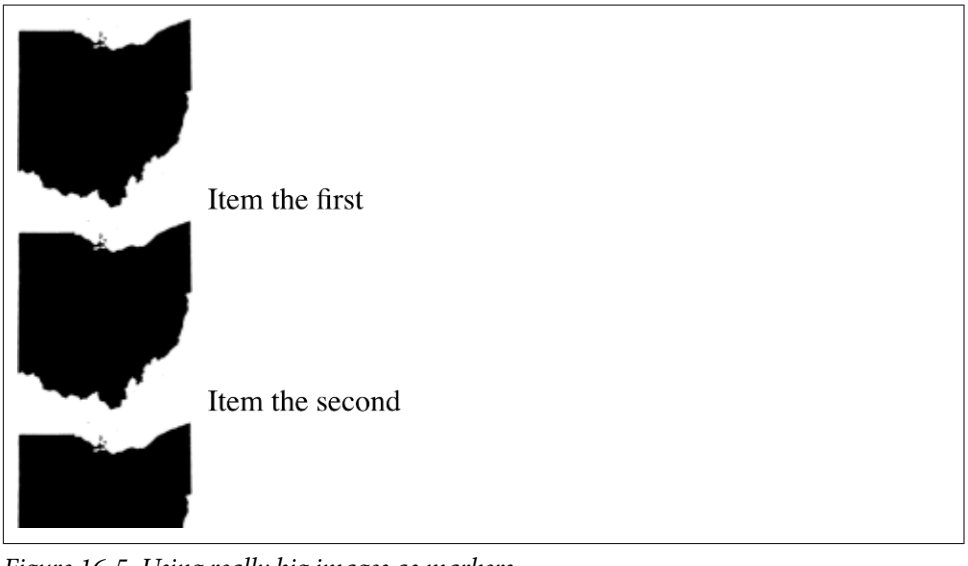


Figure 16-5. Using really big images as markers

It's generally a good idea to provide a fallback marker type in case your image doesn't load, gets corrupted, or is in a format that some user agents can't display. Do this by defining a backup `list-style-type` for the list:

```
ul li {list-style-image: url(ohio.png); list-style-type: square;}
```

The other thing you can do with `list-style-image` is set it to the default value of `none`. This is good practice because `list-style-image` is inherited, so any nested lists will pick up the image as the marker, unless you prevent that from happening:

```
ul {list-style-image: url(ohio.gif); list-style-type: square;}
ul ul {list-style-image: none;}
```

Since the nested list inherits the item type `square` but has been set to use no image for its markers, squares are used for the markers in the nested list, as shown in Figure 16-6.

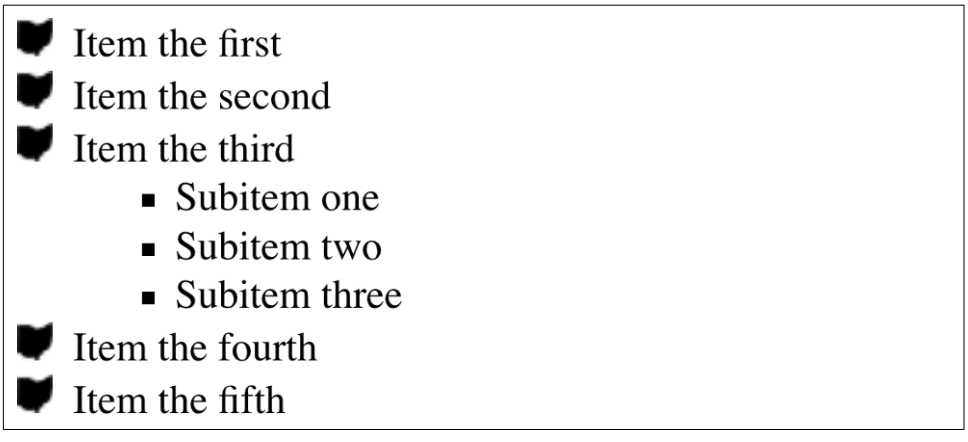


Figure 16-6. Switching off image markers in sublists

Any image value is permitted for `list-style-image`, including gradient images. Thus, the following styles would have a result like that shown in Figure 16-7:

```
.list01 {list-style-image:
    radial-gradient(closest-side,
        orange, orange 60%, blue 60%, blue 95%, transparent);}
.list02 {list-style-image:
    linear-gradient(45deg, red, red 50%, orange 50%, orange);}
.list03 {list-style-image:
    repeating-linear-gradient(-45deg, red, red 1px, yellow 1px, yellow 3px);}
.list04 {list-style-image:
    radial-gradient(farthest-side at bottom right,
        lightblue, lightblue 50%, violet, indigo, blue, green,
        yellow, orange, red, lightblue);}
```

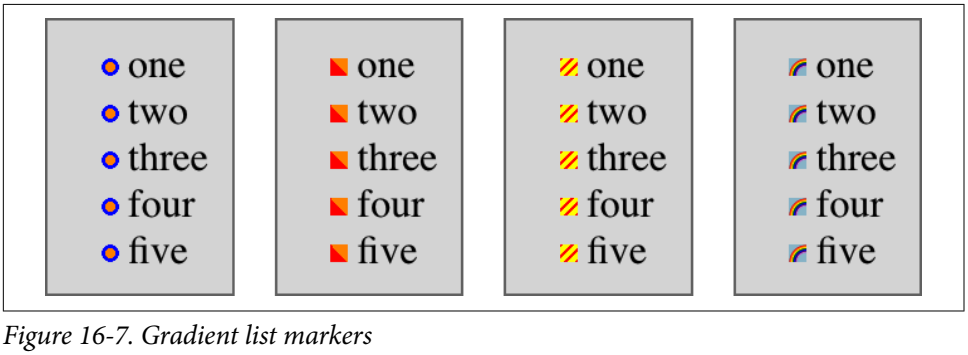


Figure 16-7. Gradient list markers

Gradient markers have one drawback: they tend to be very small. This can be influenced by factors such as font size, because the marker size tends to scale with the list item's

content. If you need to have full control over how the markers are rendered, don't use `::marker`; use `::before` instead.



The way to style list markers directly is the `::marker` pseudo-element, discussed later in this chapter.

List-Marker Positions

There is another thing you can do to change the appearance of list items: decide whether the marker appears outside or inside the content of the list item. This is accomplished with `list-style-position`.

list-style-position

Values	`inside` | `outside` | `inherit`
Initial value	`outside`
Applies to	Elements whose `display` value is `list-item`
Inherited	Yes
Computed value	As specified

If a marker's position is set to `outside` (the default), it will appear the way list items have since the beginning of the web. Should you desire a slightly different appearance, you can pull the marker in toward the content by setting the value of `list-style-position` to `inside`. This causes the marker to be placed "inside" the list item's content. The exact way this happens is undefined, but Figure 16-8 shows one possibility:

```
li.first {list-style-position: inside;}
li.second {list-style-position: outside;}
```

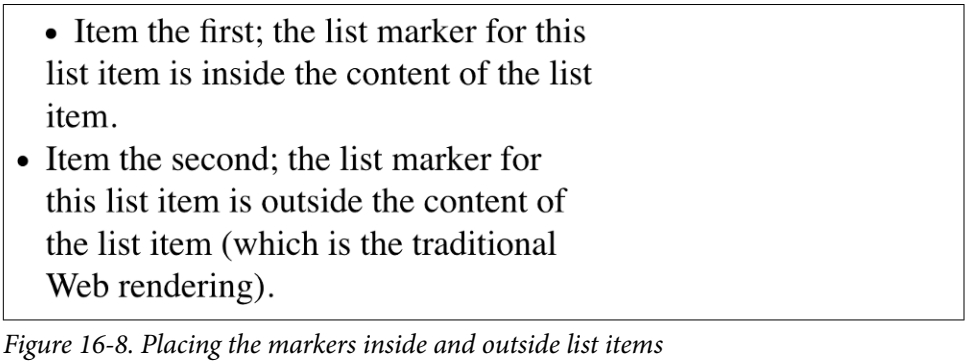


Figure 16-8. Placing the markers inside and outside list items

In practice, markers given an `inside` placement are treated as if they're an inline element inserted into the beginning of the list item's content. This doesn't mean the markers *are* inline elements. You can't style them separately from the rest of the element's content, unless you wrap all the other content in an element like `<span>`, or else address them directly (but with major limitations on what properties are allowed) by using `::marker`. It's just that in layout terms, that's what they act like.

List Styles in Shorthand

For brevity's sake, you can combine the three list-style properties into a convenient single property: `list-style`.

list-style

Values	[*<list-style-type>* ‖ *<list-style-image>* ‖ *<list-style-position>*] | `inherit`
Initial value	Refer to individual properties
Applies to	Elements whose `display` value is `list-item`
Inherited	Yes
Computed value	See individual properties

For example:

```
li {list-style: url(ohio.gif) square inside;}
```

As shown in Figure 16-9, all three values can be applied to list items at the same time.

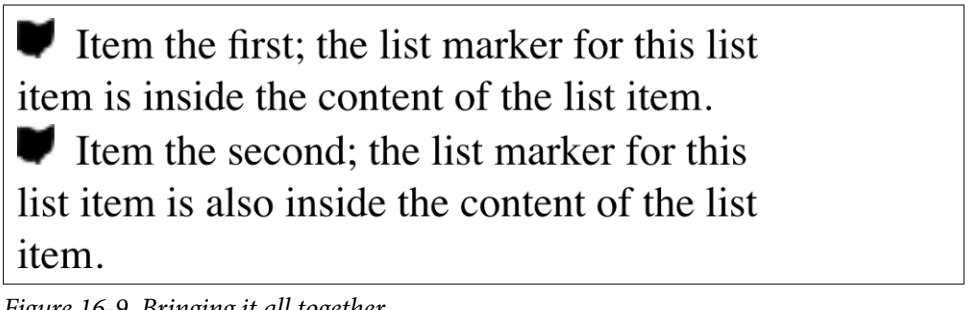


Figure 16-9. Bringing it all together

The values for `list-style` can be listed in any order, and any can be omitted. As long as one is present, the rest will fill in their default values. For instance, the following two rules will have the same visual effect:

```
li.norm {list-style: url(img42.gif);}
li.odd {list-style: url(img42.gif) disc outside;} /* the same thing */
```

They will also override any previous rules in the same way. For example:

```
li {list-style-type: square;}
li {list-style: url(img42.gif);}
li {list-style: url(img42.gif) disc outside;} /* the same thing */
```

The result will be the same as that in Figure 16-9 because the implied `list-style-type` value of `disc` will override the previous declared value of `square`, just as the explicit value of `disc` overrides it in the second rule.

List Layout

Now that we've looked at the basics of styling list markers, let's consider how lists are laid out in various browsers. We'll start with a set of three list items devoid of any markers and not yet placed within a list, as shown in Figure 16-10.

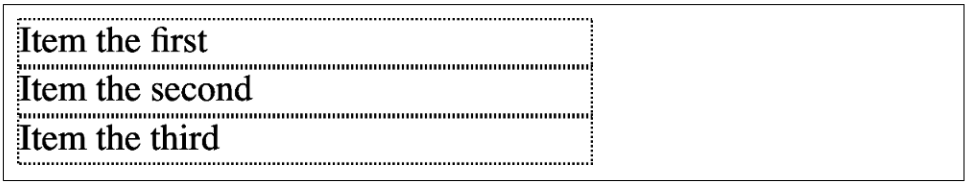


Figure 16-10. Three list items

The border around the list items shows them to be, essentially, like block-level elements. Indeed, the value `list-item` is defined to generate a block box. Now let's add markers, as illustrated in Figure 16-11.

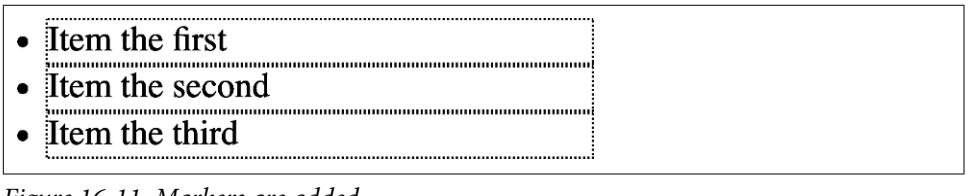


Figure 16-11. Markers are added

The distance between the marker and the list item's content is not defined by CSS, and CSS does not as yet provide a way to directly affect that distance.

With the markers outside the list items' content, they don't affect the layout of other elements, nor do they really even affect the layout of the list items themselves. They just hang a certain distance from the edge of the content, and wherever the content edge goes, the marker will follow. The behavior of the marker works much as though the marker were absolutely positioned in relation to the list-item content, something like `position: absolute; left: -1.5em;`. When the marker is inside, it acts like an inline element at the beginning of the content.

So far, we have yet to add an actual list container; neither a `<ul>` nor an `<ol>` element is represented in the figures. We can add one to the mix, as shown in Figure 16-12 (it's represented by a dashed border).



Figure 16-12. Adding a list border

As with the list items, the unordered-list element generates a block box, one that encompasses its descendant elements. As Figure 16-12 illustrates, the markers are placed not only outside the list item contents, but also outside the content area of the unordered-list element. The usual "indentation" you expect from lists has not yet been specified.

Most browsers, as of this writing, indent list items by setting either padding or margins for the containing list element. For example, the user agent might apply a rule such as this:

```
ul, ol {margin-inline-start: 40px;}
```

Most browsers use a rule that's something like this:

```
ul, ol {padding-inline-start: 40px;}
```

Neither is incorrect, but because browsers can, and have, changed the way they indent list content, we recommend including values for both properties when trying to eliminate the indentation of the list items. Figure 16-13 compares the two approaches.

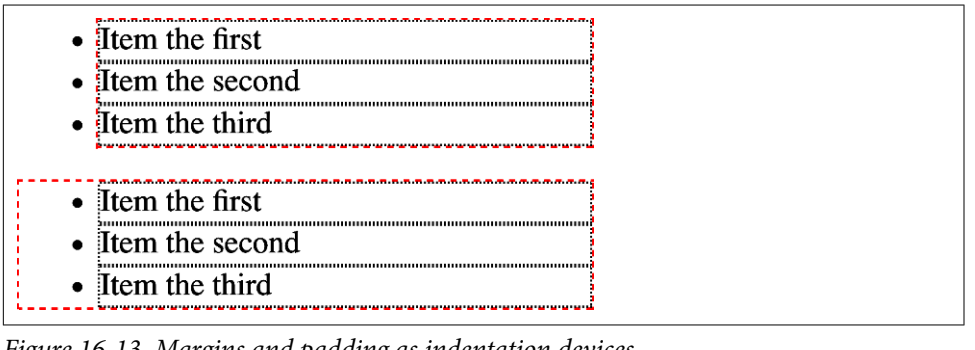


Figure 16-13. Margins and padding as indentation devices



The distance `40px` is a relic of early web browsers, which indented lists by a pixel amount. (Block quotes are indented by the same distance.) A good alternate value might be something like `2.5em`, which would scale the indentation along with changes in the text size and is also equal to `40px`, assuming a default font size of 16 pixels.

For authors who want to change the indentation distance of lists, we strongly recommend specifying both padding and margins to ensure cross-browser compatibility. For example, if you want to use padding to indent a list, use this rule:

```
ul {margin-inline-start: 0; padding-inline-start: 1em;}
```

If you prefer margins, write something like this instead:

```
ul {margin-inline-start: 1em; padding-inline-start: 0;}
```

In either case, remember that the markers will be placed relative to the contents of the list items, and may therefore "hang" outside the main text of a document or even beyond the edge of the browser window. This is most easily observed if very large images, or long text strings, are used for the list markers, as shown in Figure 16-14.

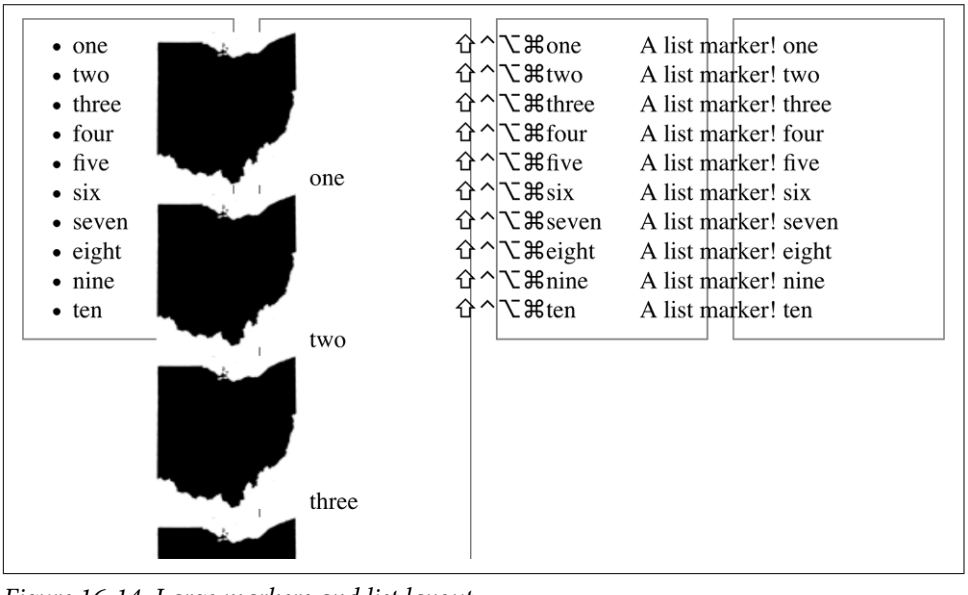


Figure 16-14. Large markers and list layout

The ::marker Pseudo-Element

One feature many authors request is the ability to control the space between a marker and the content of a list item, or to change the size or color of a list marker independently of the contents of the list items.

List markers can be styled, to a limited extent, with the pseudo-element `::marker`. The properties allowed for `::marker` rules as of late 2022 are as follows:

- `content`
- `color`
- `text-combine-upright`

- `unicode-bidi`
- `direction`
- `white-space`
- All the `font-*` properties
- All the transition and animation properties

You may have noticed that no element-sizing or other box model properties such as margins are included, which puts a damper on a lot of authors' desires for marker styling. More properties may be added in the future, but for now, that's what we have.

A few examples of marker styling, as declared here, are illustrated in Figure 16-15:

```
li:nth-child(1)::marker {color: gray;}
li:nth-child(2)::marker {font-size: 2em;}
li:nth-child(3)::marker {font-style: italic;}

<ol>
        <li>List item the first</li>
        <li>The second list item</li>
        <li>List Items With a Vengeance</li>
</ol>

<ul>
        <li>List item the first</li>
        <li>The second list item</li>
        <li>List Items With a Vengeance</li>
</ul>
```

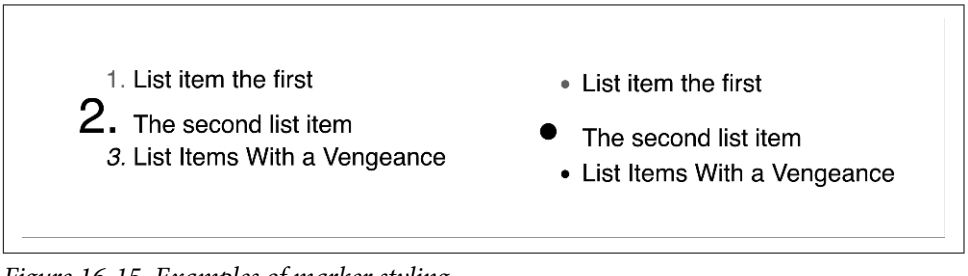


Figure 16-15. Examples of marker styling

Notice that the result of doubling a marker's font size differs between the ordered and unordered versions of this list. This comes down to the different default sizing and placement of the two marker types. As previously noted, the amount of control you have over markers is limited, even markers defined with `content`. So when you absolutely must have complete creative freedom with your markers, it's usually better to build your own with generated content or marked-up inline content.

Creating Generated Content

CSS defines methods to create *generated content*. This is content inserted via CSS, but not represented by either markup or content.

For example, list markers are generated content. Nothing in the markup of a list item directly represents the markers, and you, the author, do not have to write the markers into your document's content. The browser simply generates the appropriate marker automatically. For unordered lists, the marker will have a symbol of some kind, such as a circle, disc, or square. In ordered lists, the marker is by default a counter that increments by one for each successive list item. (Or, as you saw in previous sections, you may replace either kind with an image or symbol—and, as you'll see in just a bit, anything supported by the `content` property.)

To understand how you can affect list markers and customize the counting of ordered lists (or anything else!), you must first look at more basic generated content.

Inserting Generated Content

To insert generated content into the document, use the `::before` and `::after` pseudo-elements. These place generated content before or after the content of an element by way of the `content` property (described in the next section).

For example, you might want to precede every hyperlink with the text "(link)" to mark them when the page is printed out. This is accomplished with a media query and rule like the following, which has the effect shown in Figure 16-16:

```
@media print{
        a[href]::before {content: "(link)";}
}
```

(link)Jeffrey seems to be (link)very happy about (link)something, although I can't quite work out whether his happiness is over (link)OS X, (link)Chimera, the ability to run the Dock and (link)DragThing at the same time, the latter half of my (link)journal entry from yesterday, or (link)something else entirely.

Figure 16-16. Generating text content

Note that there isn't a space between the generated content and the element content. This is because the value of `content` in the previous example doesn't include a space. You could modify the declaration as follows to make sure there's a space between generated and actual content:

```
a[href]::before {content: "(link) ";}
```

It's a small difference but an important one.

In a similar manner, you might choose to insert a small icon at the end of links to PDF documents. The rule to accomplish this would look something like this:

```
a.pdf-doc::after {content: url(pdf-doc-icon.gif);}
```

Suppose you want to further style such links by placing a border around them. This is done with a second rule, shown in Figure 16-17:

```
a.pdf-doc {border: 1px solid gray;}

<<generated-content-icons>> shows the result of these two rules.
```

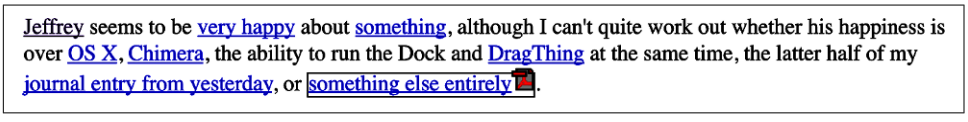


Figure 16-17. Generating icons

Notice that the link border extends around the generated content, just as the link underline extends under the "(link)" text in Figure 16-16. This happens because by default, generated content is placed inside the element box of the element (unless the generated content is a list marker).

You can float or position generated content outside its parent element's box. All `display` values can be given to generated content; you can apply block formatting to the generated content of an inline box, and vice versa. For example, consider this:

```
em::after {content: " (!) "; display: block;}
```

Even though `em` is an inline element, the generated content will generate a block box. Similarly, given the following code, the generated content is made block-level instead of remaining the default of `inline`:

```
h1::before {content: "New Section"; display: block; color: gray;}
```

Figure 16-18 shows the result.



Figure 16-18. Generating block-level content

One interesting aspect of generated content is that it inherits values from the element to which it's been attached. Thus, given the following rules, the generated text will be green, the same as the content of the paragraphs:

```
p {color: green;}
p::before {content: "::: ";}
```

If you want the generated text to be purple instead, a simple declaration will suffice:

```
p::before {content: "::: "; color: purple;}
```

Such value inheritance happens only with inherited properties, of course. This is worth noting because it influences the way certain effects must be approached. Consider the following:

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1::before {content: "New Section"; display: block; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

Since the generated content is placed inside the element box of the `<h1>`, it will be placed under the top border of the element. It would also be placed within any padding, as shown in Figure 16-19.

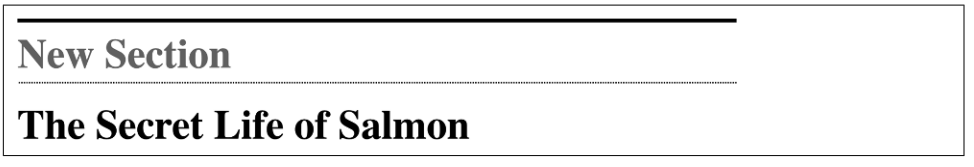


Figure 16-19. Taking placement into account

The bottom margin of the generated content, which has been made block-level, pushes the actual content of the element downward by half an em. In every sense, the effect of the generated content in this example is to break the `<h1>` element into two pieces: the generated-content box and the actual content box. This happens because the generated content has `display: block`. If you were to change it to `display: inline` (or remove the `display:block;` entirely), the effect would be as shown in Figure 16-20:

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1::before {content: "New Section"; display: inline; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

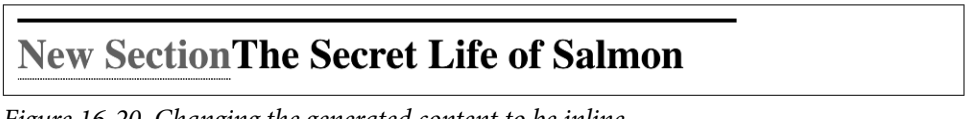


Figure 16-20. Changing the generated content to be inline

Note how the borders are placed and how the top padding is still honored. So is the bottom margin on the generated content, but since the generated content is now inline and margins don't affect line height, the margin has no visible effect.

With the basics of generating content established, let's take a closer look at the way the actual generated content is specified.

Specifying Content

If you're going to generate content, you need a way to describe it. As you've already seen, this is handled with the `content` property, but there's a great deal more to this property than you've seen thus far.

content

Values	`normal` | [*<string>* | *<uri>* | *<counter>* | `attr(`*<identifier>*`)`+ | `open-quote` | `close-quote` | `no-open-quote` | `no-close-quote`]+ | `inherit`
Initial value	`normal`
Applies to	`::before` and `::after` pseudo-elements
Inherited	No
Computed value	For *<uri>* values, an absolute URI; for attribute references, the resulting string; otherwise, as specified

You've already seen string and URI values in action, and counters are covered later in this chapter. Let's talk about strings and URIs in a little more detail before we take a look at the `attr()` and quote values.

String values are presented literally, even if they contain what would otherwise be markup of some kind. Therefore, the following rule would be inserted verbatim into the document, as shown in Figure 16-21:

```
h2::before {content: "<em>&para;</em> "; color: gray;}
```



Figure 16-21. Strings are displayed verbatim

This means that if you want a newline (return) as part of your generated content, you can't use `<br>`. Instead, you use the string `\A`, or `\00000a`, which is the CSS way of representing a newline (based on the Unicode line-feed character, which is hexadecimal position `A`). Conversely, if you have a long string value and need to break it up over multiple lines, you escape out the line feeds with the `\` character. These are both demonstrated by the following rule and illustrated in Figure 16-22:

```
h2::before {content: "We insert this text before all H2 elements because \
it is a good idea to show how these things work. It may be a bit long \
but the point should be clearly made.  "; color: gray;}
```

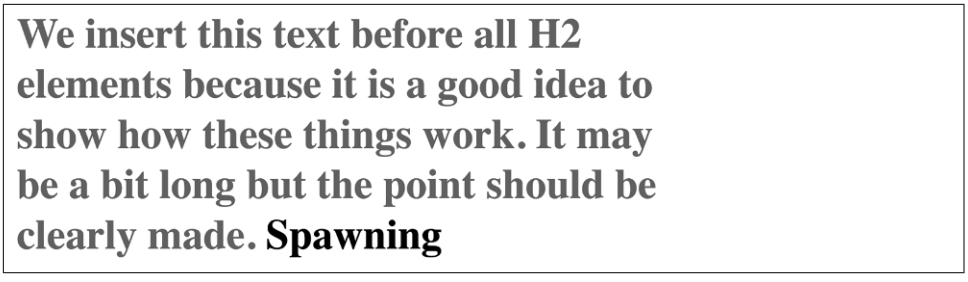


Figure 16-22. Inserting and suppressing newlines

You can also use escapes to refer to hexadecimal Unicode values, such as `\00AB`.



As of this writing, while inserting escaped content such as `\279c` is very well-supported, some browsers don't support the escaped newline character `\A` or `\0000a`, and no browsers support `\A` unless you add a space after it.

With URI values, you point to an external resource (an image, movie, sound clip, or anything else the user agent supports), which is then inserted into the document in the appropriate place. If the user agent can't support the resource you point it to for any reason—say, you try to insert a movie into a document when it's being printed—then the user agent is required to ignore the resource completely, and nothing will be inserted.

Inserting attribute values

Sometimes you might want to take the value of an element's attribute and make it a part of the document display. To pick a simple example, you can place the value of every link's `href` attribute immediately after the links, like this:

```
a[href]::after {content: attr(href);}
```

Again, this leads to the problem of the generated content running smack into the actual content. To solve this, add some string values to the declaration, with the result shown in Figure 16-23:

```
a[href]::after {content: " [" attr(href) "]";}
```

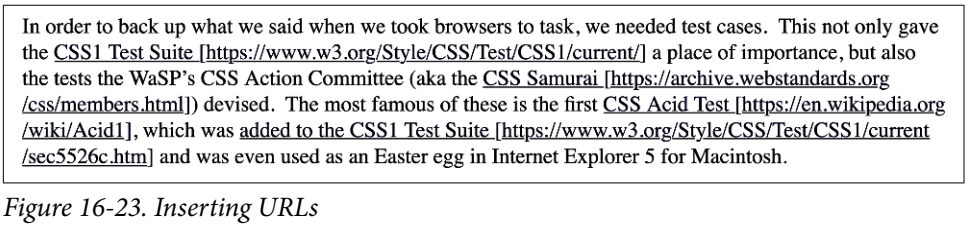


Figure 16-23. Inserting URLs

This can be useful for print stylesheets, as an example. Any attribute value can be inserted as generated content: `alt` text, `class` or `id` values—anything. An author might choose to make the citation information explicit for a block quote, like this:

```
blockquote::after {content: "(" attr(cite) ")"; display: block;
  text-align: right; font-style: italic;}
```

For that matter, a more complicated rule might reveal the text- and link-color values for a legacy document:

```
body::before {
  content: "Text: " attr(text) " | Link: " attr(link)
  " | Visited: " attr(vlink) " | Active: " attr(alink);
  display: block; padding: 0.33em;
  border: 1px solid; text-align: center; color: red;}
```

Note that if an attribute doesn't exist, an empty string is put in its place. This is what happens in Figure 16-24, in which the previous example is applied to a document whose `body` element has no `alink` attribute.

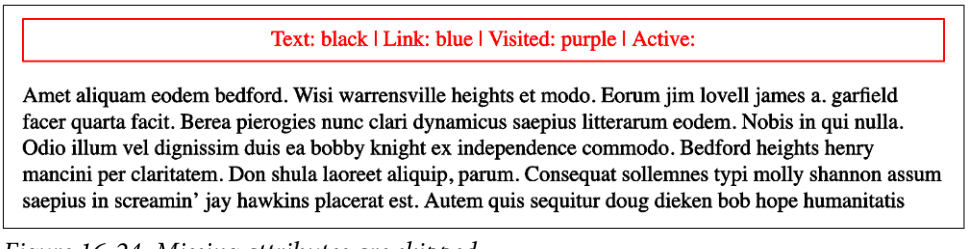


Figure 16-24. Missing attributes are skipped

The text "Active: " (including the trailing space) is inserted into the document, as you can see, but nothing follows it. This is convenient when you want to insert the value of an attribute only when it exists.



CSS defines the returned value of an attribute reference as an unparsed string. Therefore, if the value of an attribute contains markup or character entities, they will be displayed verbatim.

Using generated quotes

A specialized form of generated content is the quotation mark, and CSS provides a powerful way to manage both quotes and their nesting behavior. This is possible because of the pairing of content values like `open-quote` and the property `quotes`.

quotes

Values	[<*string*> <*string*>]+ | none | inherit
Initial value	User-agent dependent
Applies to	All elements
Inherited	Yes
Computed value	As specified

Other than the keywords `none` and `inherit`, the only valid value is one or more *pairs* of strings, with the first in each pair being a value for `open-quote` and the second being a `close-quote` value. The first string of the pair defines the open-quote symbol, and the second defines the close-quote symbol. Therefore, of the following two declarations, only the first is valid:

```
quotes: '"' "'";  /* valid */
quotes: '"';  /* NOT VALID */
```

The first rule also illustrates one way to put string quotes around the strings themselves. The double quotation marks are surrounded by single quotation marks, and vice versa.

Let's look at a simple example. Suppose you're creating an XML format to store a list of favorite quotations. Here's one entry in the list:

```
<quotation>
  <quote>I hate quotations.</quote>
  <quotee>Ralph Waldo Emerson</quotee>
</quotation>
```

To present the data in a useful way, you could employ the following rules, with the result shown in Figure 16-25:

```
quotation {display: block;}
quote {quotes: '“' '”';}
quote::before {content: open-quote;}
quote::after {content: close-quote;}
quotee::before {content: " (";}
quotee::after {content: ")";}
```



Figure 16-25. Inserting quotes and other content

The values `open-quote` and `close-quote` are used to insert whatever quoting symbols are appropriate (since different languages have different quotation marks). They use the value of quotes to determine how they should work. Thus, the quotation begins and ends with a double quotation mark.

With `quotes`, you can define quotation patterns to as many nesting levels as you like. In American English, for example, a common practice is to start out with a double quotation mark and then use single quotation marks for the quotation nested inside the first one. This can be re-created with *curly* quotation marks by using the following rules:

```
quotation: display: block;}
quote {quotes: '\201C' '\201D' '\2018' '\2019';}
quote::before, q::before{content: open-quote;}
quote::after, q::after {content: close-quote;}
```

When applied to the following XML, these rules will have the effect shown in Figure 16-26:

```
<quotation>
 <quote> In the beginning, there was nothing. And God said: <q>Let there
  be light!</q> And there was still nothing, but you could see it.</quote>
</quotation>
```

> “ In the beginning, there was nothing. And God said: ‘Let there be light!’ And there was still nothing, but you could see it.”

Figure 16-26. Nested curly quotes

If the nested level of quotation marks is greater than the number of defined pairs, the last pair is reused for the deeper levels. Thus, if we had applied the following rule to the markup shown in Figure 16-26, the inner quote would have had double quotation marks, the same as the outer quote:

```
quote {quotes: '\201C' '\201D';}
```



These particular rules use the hexadecimal Unicode positions for the curly quote symbols. If your CSS uses UTF-8 character encoding (and it really should), you can skip the escaped hexadecimal position approach and include just the curly quote characters directly, as in previous examples.

Generated quotes make possible one other common typographic effect. When quoted text spans several paragraphs, the `close-quote` of each paragraph is often omitted; only the opening quote marks are shown, with the exception of the last paragraph. This can be re-created using the `no-close-quote` value:

```
blockquote {quotes: '"' '"' "'" "'" '"' '"';}
blockquote p::before {content: open-quote;}
blockquote p::after {content: no-close-quote;}
blockquote p:last-of-type::after {content: close-quote;}
```

This will start each paragraph with a double quotation mark but no closing mark. This is true of the last paragraph as well, so the fourth line in the previous code block inserts a close quote at the end of the last paragraph.

This value is important because it decrements the quotation nesting level without generating a symbol. This is why each paragraph starts with a double quotation mark, instead of alternating between double and single marks, until the third paragraph is reached. The `no-close-quote` value closes the quotation nesting at the end of each paragraph, and thus every paragraph starts at the same nesting level.

This is significant because, as the CSS2.1 specification notes, "Quoting depth is independent of the nesting of the source document or the formatting structure." In other words, when you start a quotation level, it persists across elements until a `close-quote` is encountered, and the quote nesting level is decremented.

For the sake of completeness, there is a `no-open-quote` keyword, which has a symmetrical effect to `no-close-quote`. This keyword increments the quotation nesting level by one but does not generate a symbol.

Defining Counters

Counters are probably familiar to you, even if you don't realize it; for example, the markers of the list items in ordered lists are counters. Two properties and two `content` values make it possible to define almost any counting format, including subsection counters employing multiple styles, such as "VII.2.c."

Resetting and incrementing

We create counters by setting the starting point for a counter and then incrementing it by a specified amount. The former is handled by the property `counter-reset`.

counter-reset

Values	[<*identifier*> <*integer*>?]+ | none | inherit
Initial value	User agent-dependent
Applies to	All elements
Inherited	No
Computed value	As specified

A *counter identifier* is simply a label created by the author. For example, you might name your subsection counter `subsection`, `subsec`, `ss`, or `bob`. The simple act of resetting (or incrementing) an identifier is sufficient to call it into being. In the following rule, the counter `chapter` is defined as it is reset:

```
h1 {counter-reset: chapter;}
```

By default, a counter is reset to 0. If you want to reset to a different number, you can declare that number following the identifier:

```
h1#ch4 {counter-reset: chapter 4;}
```

You can also reset multiple identifiers all at once by listing space-separated identifier-integer pairs. If you leave out an integer, it defaults to 0:

```
h1 {counter-reset: chapter 4 section -1 subsec figure 1;}
   /* 'subsec' is reset to 0 */
```

As you can see from the previous example, negative values are permitted. It would be perfectly legal to set a counter to `-32768` and count up from there.



CSS does not define what user agents should do with negative counter values in nonnumeric counting styles. For example, there is no defined behavior for what to do if a counter's value is `-5` but its display style is `upper-alpha`.

To count up or down, you'll need a property to indicate that an element increments or decrements a counter. Otherwise, the counter would remain at whatever value it was given with a `counter-reset` declaration. The property in question is, not surprisingly, `counter-increment`.

counter-increment

Values	[<*identifier*> <*integer*>?]+ | `none` | `inherit`
Initial value	User-agent dependent
Applies to	All elements
Inherited	No
Computed value	As specified

Like `counter-reset`, `counter-increment` accepts identifier-integer pairs, and the integer portion of these pairs can be 0 or negative as well as positive. The difference is that if an integer is omitted from a pair in `counter-increment`, it defaults to 1, not 0.

As an example, here's how a user agent might define counters to re-create the traditional 1, 2, 3 counting of ordered lists:

```
ol {counter-reset: ordered;}  /* defaults to 0 */
ol li {counter-increment: ordered;}  /* defaults to 1 */
```

On the other hand, an author might want to count backward from 0 so that the list items use a rising negative system. This would require only a small edit:

```
ol {counter-reset: ordered;}  /* defaults to 0 */
ol li {counter-increment: ordered -1;}
```

The counting of lists would then be –1, –2, –3, and so on. If you replaced the integer `-1` with `-2`, lists would count –2, –4, –6, and so on.

Displaying counters

To display the counters, you need to use the `content` property in conjunction with one of the counter-related values. To see how this works, let's use an XML-based ordered list:

```
<list type="ordered">
 <item>First item</item>
 <item>Item two</item>
 <item>The third item</item>
</list>
```

By applying the following rules to XML employing this structure, you would get the result shown in Figure 16-27:

```
list[type="ordered"] {counter-reset: ordered;}  /* defaults to 0 */
list[type="ordered"] item {display: block;}
list[type="ordered"] item::before {counter-increment: ordered;
     content: counter(ordered) ". "; margin: 0.25em 0;}
```



Figure 16-27. Counting the items

The generated content is placed as inline content at the beginning of the associated element. Thus, the effect is similar to an HTML list with `list-style-position: inside;` declared.

The `<item>` elements are ordinary elements generating block-level boxes, which means that counters are not restricted only to elements with a `display` of `list-item`. In fact, any element can use a counter. Consider the following rules:

```
h1 {counter-reset: section subsec;
    counter-increment: chapter;}
h1::before {content: counter(chapter) ". ";}
h2 {counter-reset: subsec;
    counter-increment: section;}
h2::before {content: counter(chapter )"." counter(section) ". ";}
h3 {counter-increment: subsec;}
h3::before {content: counter(chapter) "." counter(section) "."
        counter(subsec) ". ";}
```

These rules would have the effect shown in Figure 16-28.

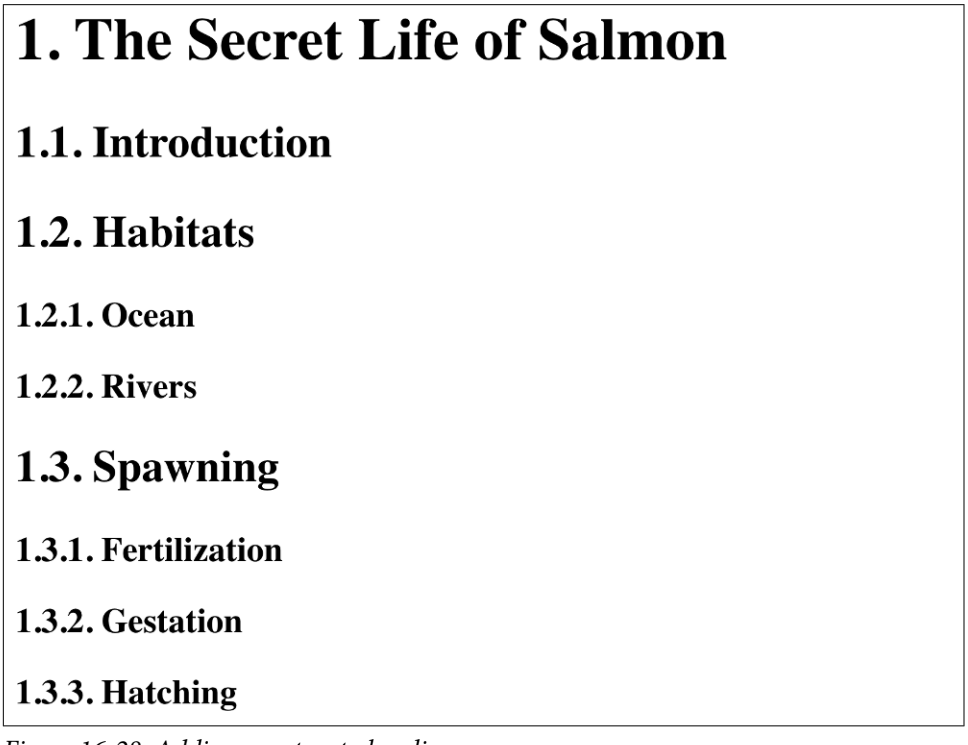


Figure 16-28. Adding counters to headings

Figure 16-28 illustrates some important points about counter resetting and incrementing. For instance, notice that the counters are reset on the elements, whereas the actual generated-content counters are inserted via the `::before` pseudo-elements. Attempting to reset counters in the pseudo-elements won't work: you'll get a lot of zeros.

Also notice that the `<h1>` element uses the counter `chapter`, which defaults to 0 and has a "1." before the element's text. When a counter is incremented and used by the same element, the incrementation happens *before* the counter is displayed. Similarly, if a counter is reset and shown in the same element, the reset happens before the counter is displayed. Consider the following:

```
h1::before, h2::before, h3::before {
  content: counter(chapter) "." counter(section) "." counter(subsec) ". ";}
h1 {counter-reset: section subsec;
  counter-increment: chapter;}
```

The first `<h1>` element in the document would be preceded by the text "1.0.0." because the counters `section` and `subsec` were reset but not incremented. Thus, if you want the first displayed instance of an incremented counter to be 0, you need to reset that counter to `-1`, as follows:

```
body {counter-reset: chapter -1;}
h1::before {counter-increment: chapter; content: counter(chapter) ". ";}
```

You can do some interesting things with counters. Consider the following XML:

```
<code type="BASIC">
  <line>PRINT "Hello world!"</line>
  <line>REM This is what the kids are calling a "comment"</line>
  <line>GOTO 10</line>
</code>
```

You can re-create the traditional format of a BASIC program listing with the following rules:

```
code[type="BASIC"] {counter-reset: linenum; font-family: monospace;}
code[type="BASIC"] line {display: block;}
code[type="BASIC"] line::before {counter-increment: linenum 10;
  content: counter(linenum) ": ";}
```

It's also possible to define a list style for each counter as part of the `counter()` format. You can do this by adding a comma-separated `list-style-type` keyword after the counter's identifier. The following modification of the heading-counter example is illustrated in Figure 16-29:

```
h1 {counter-reset: section subsec;
    counter-increment: chapter;}
h1::before {content: counter(chapter,upper-alpha) ". ";}
h2 {counter-reset: subsec;
    counter-increment: section;}
h2::before {content: counter(chapter,upper-alpha)"." counter(section) ". ";}
h3 {counter-increment: subsec;}
h3::before {content: counter(chapter,upper-alpha) "." counter(section) "."
        counter(subsec,lower-roman) ". ";}
```

Notice that the counter `section` is not given a style keyword, so it defaults to the decimal counting style. You can even set counters to use the styles `disc`, `circle`, `square`, and `none` if you so desire, though every instance of those counters will be just a single copy of the symbol you specified.

One interesting point to note is that elements with a `display` of `none` do not increment counters, even if the rule seems to indicate otherwise. In contrast, elements with a `visibility` of `hidden` *do* increment counters:

```
.suppress {counter-increment: cntr; display: none;}
  /* 'cntr' is NOT incremented */
.invisible {counter-increment: cntr; visibility: hidden;}
  /* 'cntr' IS incremented */
```

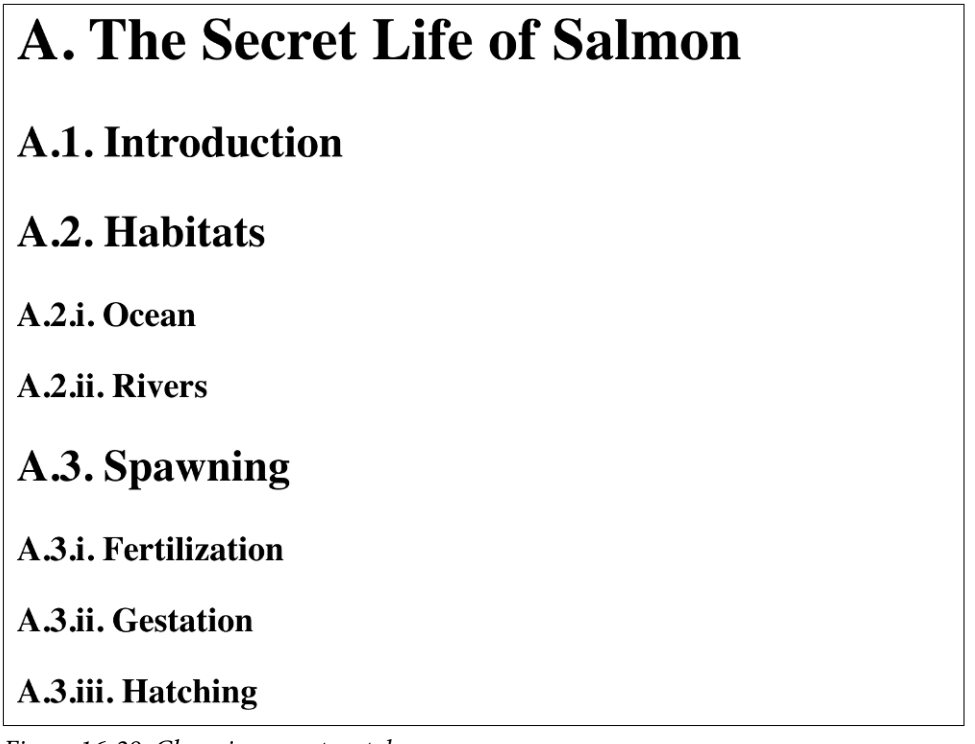


Figure 16-29. Changing counter styles

Counters and scope

So far, you've seen how to string multiple counters together to create section-and-subsection counting. Often, this is something authors desire for nested ordered lists as well, but trying to create enough counters to cover deep nesting levels would quickly become clumsy. Just to get counters to work for five-level-deep nested lists would require a bunch of rules like this:

```
ol ol ol ol ol li::before {
    counter-increment: ord1 ord2 ord3 ord4 ord5;
    content: counter(ord1) "." counter(ord2) "." counter(ord3) "."
        counter(ord4) "." counter(ord5) ".";}
```

Imagine writing enough rules to cover nesting up to 50 levels! (We're not saying you should nest ordered lists 50 deep. Just follow along for the moment.)

Fortunately, CSS 2.1 described the concept of *scope* when it comes to counters. Stated simply, every level of nesting creates a new scope for any given counter. Scope is what makes it possible for the following rules to cover nested-list counting in the usual HTML way:

```
ol {counter-reset: ordered;}
ol li::before {counter-increment: ordered; content: counter(ordered) ". ";}
```

These rules will all make ordered lists, even those nested inside others, start counting from 1 and increment each item by one—exactly the way it's been done in HTML from the beginning.

This works because a new instance of the counter `ordered` is created at each level of nesting. So, for the first ordered list, an instance of `ordered` is created. Then, for every list nested inside the first one, another new instance is created, and the counting starts anew with each list.

However, suppose you want ordered lists to count so that each level of nesting creates a new counter appended to the old: 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3, 2, 2.1, and so on. This can't be done with `counter()`, but it *can* be done with `counters()`. What a difference an *s* makes.

To create the nested-counter style shown in Figure 16-30, you need these rules:

```
ol {counter-reset: ordered; list-style: none;}
ol li:before {content: counters(ordered,".") ": "; counter-increment: ordered;}
```

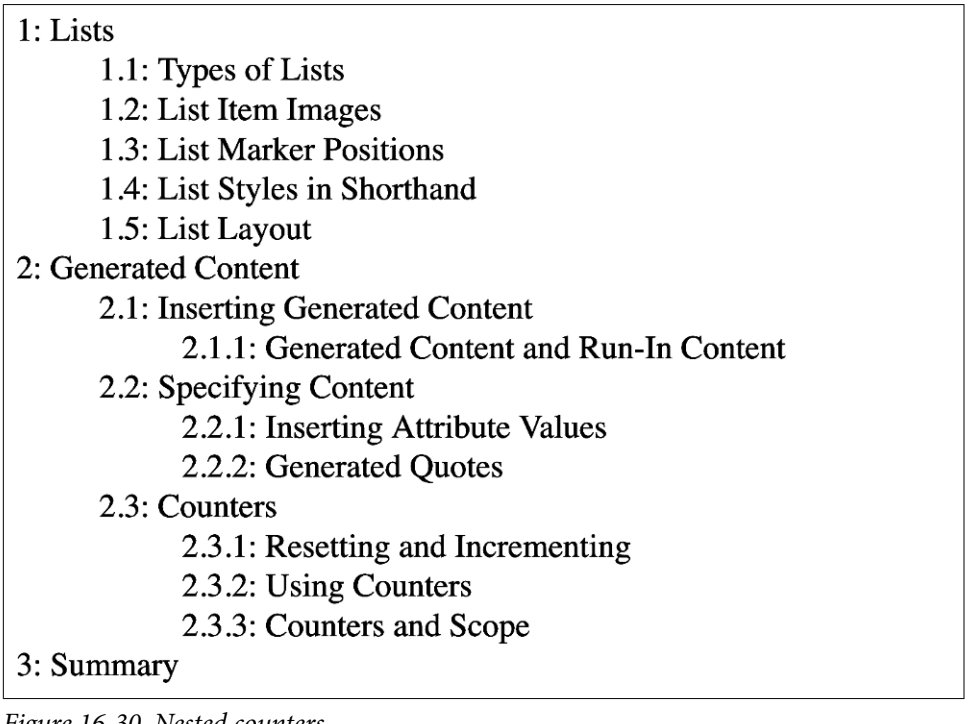


Figure 16-30. Nested counters

Basically, the keyword `counters(ordered,".")` displays the `ordered` counter from each scope with a period appended, and strings together all of the scoped counters for a given element. Thus, an item in a third-level nested list would be prefaced with the `ordered` value for the outermost list's scope, the scope of the list between the outer and current list, and the current list's scope, with each of those followed by a period. The rest of the `content` value causes a space, colon, and space to be added after all of those counters.

As with `counter()`, you can define a list style for nested counters, but the same style applies to all of the counters. Thus, if you changed your previous CSS to read as follows, the list items in Figure 16-30 would all use lowercase letters for the counters instead of numbers:

```
ol li::before {counter-increment: ordered;
    content: counters(ordered,".",lower-alpha) ": ";}
```

You may have noticed that `list-style: none` was applied to the `<ol>` elements in the previous examples. That's because the counters being inserted were generated content, not replacement list markers. In other words, had the `list-style: none` been omitted, each list item would have had its user agent-supplied list counter, *plus* the generated-content counters we defined.

That ability can be very useful, but sometimes you really just want to redefine the markers themselves. That's where counting patterns come in.

Defining Counting Patterns

If you want to get beyond simple nested counting, perhaps into defining base-60 counting or using patterns of symbols, CSS provides a way to define almost any counting pattern you can imagine. You can use `@counter-style` blocks, with dedicated descriptors to manage the outcome. The general pattern is as follows:

```
@counter-style <name> {
    …declarations…
}
```

Here, *<name>* is an author-supplied name for the pattern in question. For example, to create a series of alternating triangle markers, the block might look something like this:

```
@counter-style triangles {
    system: cyclic;
    symbols: ▶ ▷;
}
ol {list-style: triangles;}
```

Figure 16-31 shows the result.

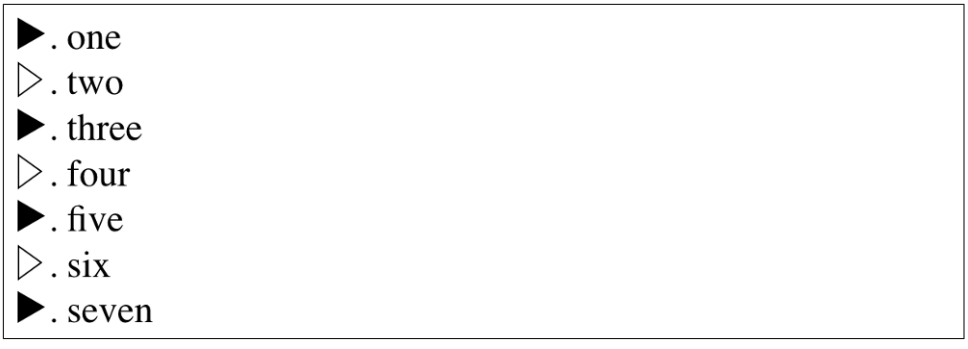


Figure 16-31. A simple counter pattern

Several descriptors are available, summarized here.

@counter-style descriptors

`system`	Defines the counter patterning system to be used.
`symbols`	Defines the counter symbols to be used in the counter pattern. This descriptor is required for all marker systems except `additive` and `extends`.
`additive-symbols`	Defines the counter symbols to be used in `additive` counter patterns.
`prefix`	Defines a string or symbol to be included just before each counter in the pattern.
`suffix`	Defines a string or symbol to be included just after each counter in the pattern.
`negative`	Defines strings or symbols to be included around any negative-value counter.
`range`	Defines the range of values in which the counter pattern should be applied. Any counter outside the defined range uses the fallback counter style.
`fallback`	Defines the counter pattern that should be used when the value can't be represented by the primary counter pattern, or the value is outside a defined `range` for the counters.
`pad`	Defines a minimum number of characters for all counters in the pattern, with any extra space filled in with a defined symbol or set of symbols.
`speak-as`	Defines a strategy for speaking the counter in text-to-speech systems.

We'll start with simple systems and work our way up in complexity, but first, let's see the precise definitions for the two most basic descriptors: `system` and `symbols`.

system descriptor

Values	`cyclic` | `numeric` | `alphabetic` | `symbolic` | `additive` | [`fixed` <*`integer`*>?] | [`extends` <*`counter-style-name`*>]
Initial value	`symbolic`

symbols descriptor

Values	<*`symbol`*>+
Initial value	n/a
Notes	A <*`symbol`*> can be any Unicode-compliant string, an image reference, or an identifier such as an escaped hexadecimal reference

For pretty much any `@counter-style` block, those are the minimum two descriptors. You can leave out `system` if you're defining a `symbolic` system, but it's generally better to include it so that you're clear about the kind of system you're setting up. Remember, the next person to work on the styles may not be as familiar with counter styling as you!

Fixed Counting Patterns

The simplest kind of counter pattern is a `fixed` system. Fixed systems are used when you want to define an exact sequence of counter markers that doesn't repeat after you've run out of markers. Consider this example, which has the result shown in Figure 16-32:

```
@counter-style emoji {
    system: fixed;
    symbols: 😁 😉 😂 🤔 🙃;
}
ol.emoji {list-style: emoji;}
```



Figure 16-32. A fixed counter pattern

Once the list gets past the fifth list item, the counter system runs out of emoji, and since no fallback was defined (we'll get to that shortly), the markers for subsequent list items fall back to the default for ordered lists.

Notice that the symbols in the `symbols` descriptor are separated by spaces. If they were all jammed together with no space separation, you'd get a result like that in Figure 16-33.

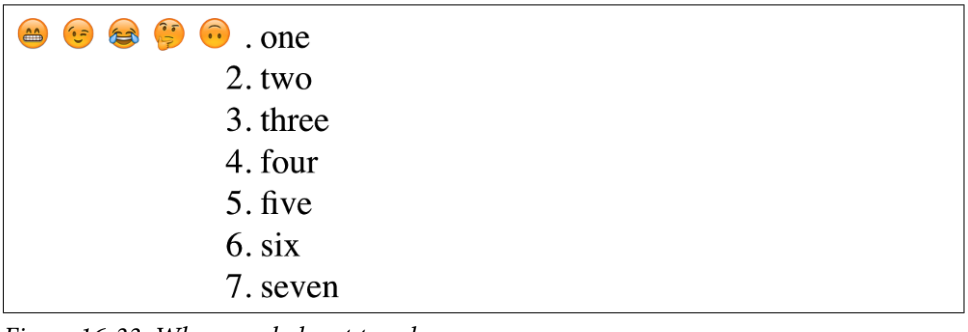


Figure 16-33. When symbols get too close

This does mean you can define a fixed sequence of markers in which each marker is composed of multiple symbols. (If you want to define a set of symbols that are combined in patterns to create a counting system, just wait: we're getting to that soon.)

If you want to use ASCII symbols in your markers, it's generally advisable to quote them. This avoids problems like angle brackets being mistaken for pieces of HTML by the parser. Thus you might do something like this:

```
@counter-style emoji {
    system: fixed;
    symbols: # $ % ">";
}
```

It's acceptable to quote all symbols, and it might be a good idea to get into that habit. That means more typing—the preceding value would become `"#" "$" "%" ">"`—but it's less error-prone.

In fixed counter systems, you can define a starting value in the `system` descriptor itself. If you want to start the counting at 5, for example, you'd write this:

```
@counter-style emoji {
    system: fixed 5;
    symbols: 😁 😉 😂 🤔 🙃;
}
ul.emoji {list-style: emoji;}
```

In this case, the first five symbols represent counters 5 through 9.



This ability to set a starting number is available in only fixed counter systems.

Cyclic Counting Patterns

The next step beyond fixed patterns is `cyclic` patterns, which are fixed patterns that repeat. Let's take the fixed emoji pattern from the previous section and convert it to cyclic. This will have the result shown in Figure 16-34:

```
@counter-style emojiverse {
    system: cyclic;
    symbols: 😁 😉 😂 🤔 🙃;
}

ul.emoji {list-style: emojiverse;}
```



Figure 16-34. A cyclic counter pattern

The defined symbols are used in order, over and over, until no more items remain left to count.

It's possible to use `cyclic` to supply a single marker that's used for the entire pattern, much like supplying a string for `list-style-type`. In this case, it would look something like this:

```
@counter-style thinker {
    system: cyclic;
    symbols: 🤔;
    /* equivalent to list-style-type: 🤔; */
}

ul.hmmm {list-style: thinker;}
```

One thing you may have noticed is that so far, all our counters have been followed by a full stop (or a period, if you prefer). This is due to the default value of the `suffix` descriptor, which has a cousin descriptor, `prefix`.

prefix and suffix descriptors

Value	*<symbol>*
Initial value	`""` (empty string) for `prefix`; `\2E` (the full stop, or period, `"."`) for `suffix`
Notes	A *<symbol>* can be any Unicode-compliant string, an image reference, or an identifier such as an escaped hexadecimal reference

With these descriptors, you can define symbols that are inserted before and after every marker in the pattern. Thus, we might give our thinker ASCII wings like so, as illustrated in Figure 16-35:

```
@counter-style wingthinker {
    system: cyclic;
    symbols: 🤔;
    prefix: "~";
    suffix: " ~";
}

ul.hmmm {list-style: wingthinker;}
```

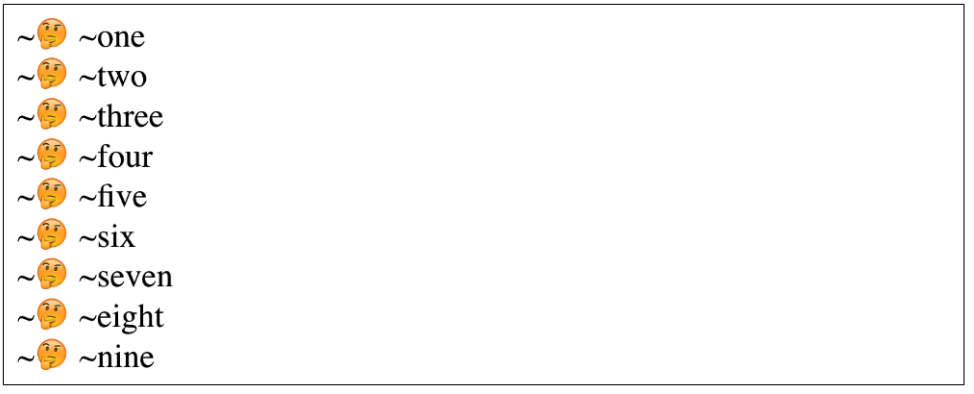


Figure 16-35. Putting "wings" on the thinker

The `suffix` descriptor is particularly useful if you want to remove the default suffix from your markers. Here's one example of how to do so:

```
@counter-style thisisfine {
    system: cyclic;
    symbols: 🔥 🐶 ☕;
    suffix: "";
}
```

You can also extend the markers in creative ways by using `prefix` and `suffix`, as shown in Figure 16-36:

```
@counter-style thisisfine {
    system: cyclic;
    symbols: 🔥 🐶 ☕;
    prefix: "🔥";
    suffix: "🔥";
}
```

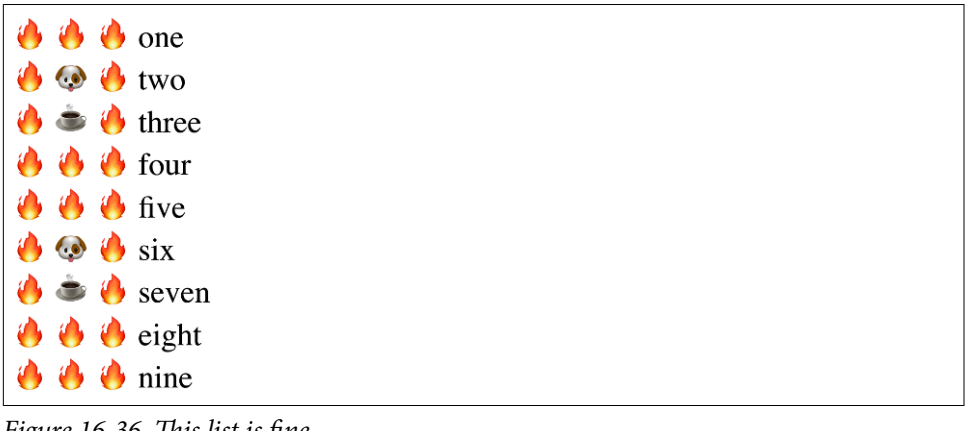


Figure 16-36. This list is fine

You might wonder why the `prefix` value is quoted in this example, while the `suffix` value is not. There is no reason other than to demonstrate that both approaches work. As stated before, quoting symbols is safer but is rarely required.

You may also see some differences between the Unicode glyphs in the CSS examples here and those shown in the figures. This is an unavoidable aspect of using emoji and other such characters—what appears on one person's user agent may be different on someone else's. Consider the differences in emoji rendering between macOS, iOS, Android, Samsung, Windows desktop, Windows mobile, Linux, and so on.

You can use images for your counters, at least in theory. As an example, suppose you want to use a series of Klingon glyphs, which have no Unicode equivalents. (It's a long-standing industry myth that Klingon is in Unicode. It was proposed in 1997 and rejected in 2001. A new proposal was made in 2016 and was once again rejected.) We won't represent the entire set of symbols here, but it would start something like this:

```
@counter-style klingon-letters {
    system: cyclic;
    symbols: url(i/klingon-a.svg) url(i/klingon-b.svg)
        url(i/klingon-ch.svg) url(i/klingon-d.svg)
        url(i/klingon-e.svg) url(i/klingon-gh.svg);
    suffix: url(i/klingon-full-stop.svg);
}
```

This would cycle from *A* through *GH* and then repeat, but still, you'd get some Klingon symbology, which might be enough. We'll see ways to build up alphabetic and numeric systems later in the chapter.



As of late 2022, browser support for any type of `<image>` as counting symbols is essentially nonexistent.

Symbolic Counting Patterns

A `symbolic` counting system is similar to a cyclic system, except in symbolic systems, for each restart of the symbol sequence, the number of symbols increases by one. Each marker is made up of a single symbol that is repeated the number of times the symbol sequence has repeated. This may be familiar to you from footnote symbols, or some varieties of alphabetic systems. Examples of each are shown here, with the result in Figure 16-37:

```
@counter-style footnotes {
    system: symbolic;
    symbols: "*" "†" "§";
    suffix: ' ';
}
@counter-style letters {
    system: symbolic;
```

```
    symbols: A B C D E;
}
```

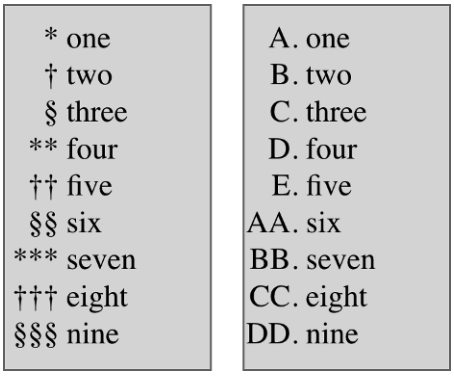


Figure 16-37. Two patterns of symbolic counting

One thing to watch out for is that if you have only a few symbols applied to a very long list, the markers will quickly get quite long. Consider the letter counters shown in the previous example. Figure 16-38 shows what the 135th through 150th entries in a list using that system would look like.

EEEEEEEEEEEEEEEEEEEEEEEEE. 135
AAAAAAAAAAAAAAAAAAAAAAAAAA. 136
BBBBBBBBBBBBBBBBBBBBBBBBB. 137
CCCCCCCCCCCCCCCCCCCCCCCCC. 138
DDDDDDDDDDDDDDDDDDDDDDDDD. 139
EEEEEEEEEEEEEEEEEEEEEEEEE. 140
AAAAAAAAAAAAAAAAAAAAAAAAAA. 141
BBBBBBBBBBBBBBBBBBBBBBBBB. 142
CCCCCCCCCCCCCCCCCCCCCCCCC. 143
DDDDDDDDDDDDDDDDDDDDDDDDD. 144
EEEEEEEEEEEEEEEEEEEEEEEEE. 145
AAAAAAAAAAAAAAAAAAAAAAAAAA. 146
BBBBBBBBBBBBBBBBBBBBBBBBB. 147
CCCCCCCCCCCCCCCCCCCCCCCCC. 148
DDDDDDDDDDDDDDDDDDDDDDDDD. 149
EEEEEEEEEEEEEEEEEEEEEEEEE. 150

Figure 16-38. Very long symbolic markers

This sort of consideration will become more of an issue from here on out, because the counter styles are all additive in one sense or another. To limit your exposure to these kinds of problems, you can use the `range` descriptor.

range descriptor

Values	[[<*integer*> | `infinite`]{2}]# | `auto`
Initial value	`auto`

With `range`, you can supply one or more space-separated pairs of values, with each pair separated from the others by commas. Let's suppose we want to stop the letter-doubling after three iterations. We have five symbols, so we can restrict their use to the first 15 list items like so, with the result shown in Figure 16-39 (which has been arranged in two columns to keep the figure size reasonable):

```
@counter-style letters {
    system: symbolic;
    symbols: A B C D E;
    range: 1 15;
}
```

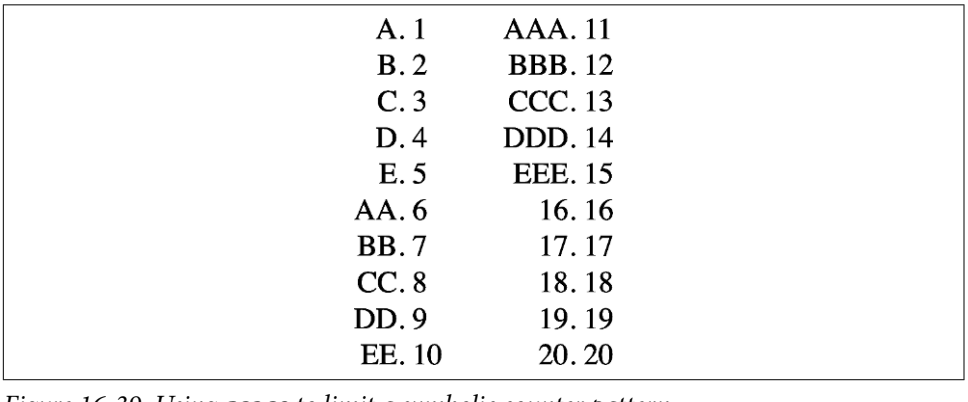


Figure 16-39. Using `range` to limit a symbolic counter pattern

If we needed, for whatever reason, to supply a second range of counter usage, it would look like this:

```
@counter-style letters {
    system: symbolic;
    symbols: A B C D E;
    range: 1 15, 101 115;
}
```

The symbolic letter system defined by `letters` would be applied in the range 1–15 as well as 101–115 (which would be "AAAAAAAAAAAAAAAAAAAAA" through "EEEEEEEEEEEEEEEEEEEEE," rather appropriately).

So what happens to the counters that fall outside of the range(s) defined by `range`? They fall back to a default marker style. You can leave that up to the user agent to handle, or you can provide some direction by means of the `fallback` descriptor.

fallback descriptor

Value	*<counter-style-name>*
Initial value	`decimal`
Note	*<counter-style-name>* can be any of the values allowed for `list-style-type`

As an example, you might decide to handle any beyond-the-range counters with Hebrew counting:

```
@counter-style letters {
    system: symbolic;
    symbols: A B C D E;
    range: 1 15, 101 115;
    fallback: hebrew;
}
```

You could just as easily use `lower-greek`, `upper-latin`, or even a noncounting style like `square`.

Alphabetic Counting Patterns

An `alphabetic` counting system is similar to a `symbolic` system, except the manner of repeating changes. Remember, with symbolic counting, the number of symbols goes up with each iteration through the cycle. In alphabetic systems, each symbol is treated as a digit in a numbering system. If you've spent any time in spreadsheets, this counting method may be familiar to you from the column labels.

To illustrate this, let's reuse the letter symbols from the previous section, and change from a symbolic to an alphabetic system. The result is shown in Figure 16-40 (once again formatted as two columns to fit):

```
@counter-style letters {
    system: alphabetic;
    symbols: A B C D E;
    /* once more cut off at 'E' to show the pattern's effects more quickly */
}
```

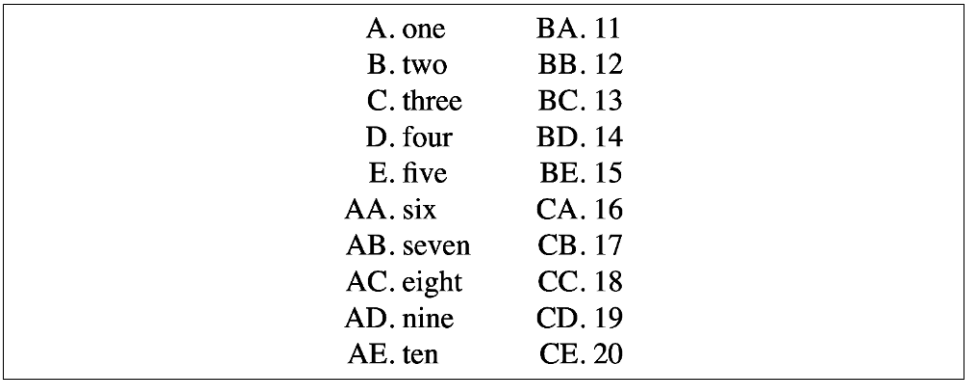


Figure 16-40. Alphabetic counting

Notice the second iteration of the pattern, which runs from "AA" to "AE" before switching over to "BA" through "BE," then on to "CA" and so on. In the symbolic version of this, we'd already be up to "EEEEEE" by the time "EE" was reached in the alphabetic system.

Note that to be valid, an alphabetic system must have a minimum of *two* symbols supplied in the `symbols` descriptor. If only one symbol is supplied, the entire `@counter-style` block is rendered invalid. Any two symbols are valid; they can be letters, numbers, or really anything in Unicode, as well as images (again, in theory).

Numeric Counting Patterns

When you define a `numeric` system, you're technically using the symbols you supply to define a *positional numbering* system—that is, the symbols are used as digits in a place-number counting system. Defining ordinary decimal counting, for example, would be done like this:

```
@counter-style decimal {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
}
```

This base is extensible to create hexadecimal counting, like so:

```
@counter-style hexadecimal {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D' 'E' 'F';
}
```

That counter style will count from 1 through F, roll over to 10 and count up to 1F, then 20 to 2F, 30 to 3F, etc. Much more simply, it's a breeze to set up binary counting:

```
@counter-style binary {
    system: numeric;
    symbols: '0' '1';
}
```

Examples of these three counting patterns are shown in Figure 16-41.

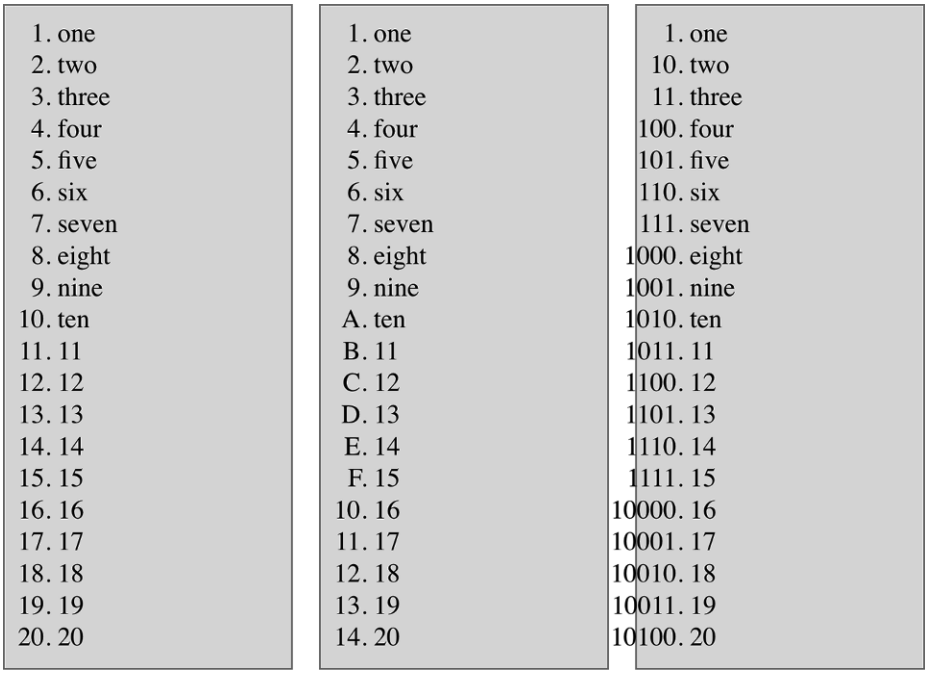


Figure 16-41. Three numeric counting patterns

An interesting question to consider is: what happens if a counter value is negative? In decimal counting, we generally expect negative numbers to be preceded by a minus sign (-), but what about in other systems, like symbolic? What if we define a letter-based numeric counting system? Or if we want to use accounting-style formatting, which puts negative values into parentheses? This is where the `negative` descriptor comes into play.

negative descriptor

Values	<*symbol*> <*symbol*>?
Initial value	`\2D` (the hyphen-minus symbol)
Notes	`negative` is applicable only in counting systems that allow negative values: `alphabetic`, `numeric`, `symbolic`, and `additive`.

The `negative` descriptor is like its own little self-contained combination of `prefix` and `suffix` that is applied only when the counter has a negative value. Its symbols are placed to the inside (that is, closer to the counter) of any prefix and suffix symbols.

So let's say we want to use accounting-style formatting, and to add prefix and suffix symbols to all counters. That would be done as follows, with the result shown in Figure 16-42:

```
@counter-style accounting {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    negative: "(" ")";
    prefix: "$";
    suffix: " - ";
}
ol.kaching {list-style: accounting;}

<ol start="-3">
…
</ol>
```

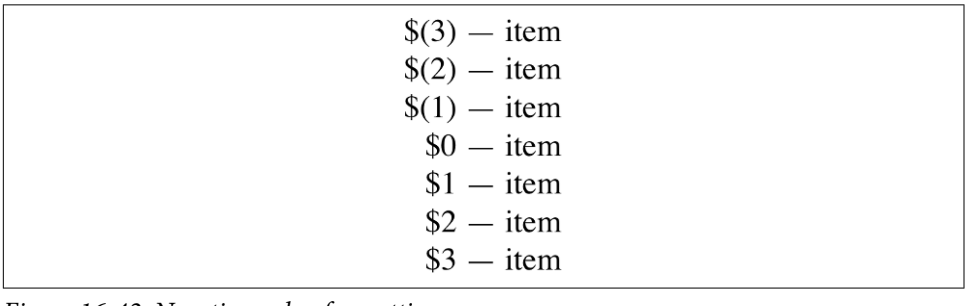


Figure 16-42. Negative-value formatting

Another common feature of numeric counting systems is the desire to pad out low values so that their length matches that of higher values. For example, rather than 1 and 100, a counting pattern might use leading zeros to create 001 and 100. This can be accomplished with the pad descriptor.

pad descriptor

Value	*<integer>* && *<symbol>*
Initial value	`0 ""`

The pattern of this descriptor is interesting. The first part is an integer that defines the number of digits that every counter should have. The second part is a string that's used to fill out any value containing fewer than the defined number of digits. Consider this example:

```
@counter-style padded {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '.';
```

```
    pad: 4 "0";
}

ol {list-style: decimal;}
ol.padded {list-style: padded;}
```

Given these styles, ordered lists will all used decimal counting by default: 1, 2, 3, 4, 5… Those with a `class` of `padded` will use padded decimal counting: 0001, 0002, 0003, 0004, 0005… Figure 16-43 shows an example.

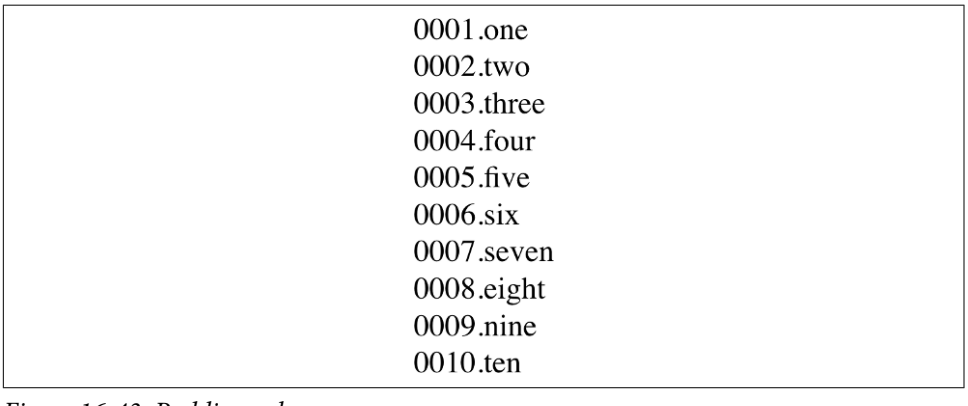


Figure 16-43. Padding values

Note that the padded counters use the `0` symbol to fill in any missing leading digits, in order to make every counter be at least four digits long. The "at least" part of that sentence is important: if a counter gets up to five digits, it won't be padded. More importantly, if a counter reaches five digits, none of the other shorter counters will get additional zeros. They'll stay four digits long, because of the `4` in `4 "0"`.

Any symbol can be used to pad values, not just `0`. You could use underlines, periods, emoji, arrow symbols, empty spaces, or anything else you like. In fact, you can have multiple characters in the <*`symbol`*> part of the value. The following is perfectly acceptable, if not necessarily desirable:

```
@counter-style crazy {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '.';
    pad: 7 "🤔 😉";
}

ol {list-style: decimal;}
ol.padded {list-style: padded;}
```

Given a counter value of `1`, the result of that crazy counting system would be "🤔😉🤔😉🤔😉1."

Note that negative symbols count toward symbol length and thus eat into padding. Also note that the negative sign will come *outside* any padding. Given the following styles, we'd get the result shown in Figure 16-44:

```
@counter-style negativezeropad {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '. ';
    negative: '-';
    pad: 4 "0";
}
@counter-style negativespacepad {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '. ';
    negative: '-';
    pad: 4 " ";
}
```

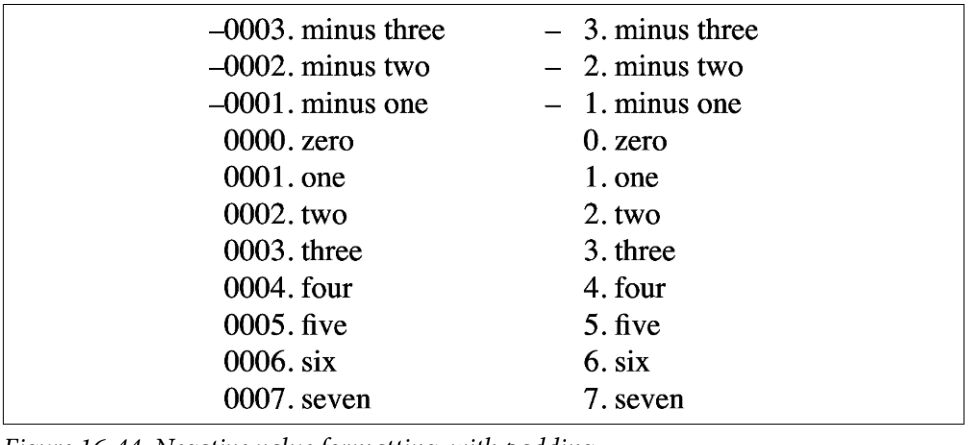


Figure 16-44. Negative value formatting, with padding

Additive Counting Patterns

We have one more system type to explore, which is `additive-symbol` counting. In additive counting systems, different symbols are used to represent values. Putting multiple symbols together properly and then adding up the numbers that each represents yields the counter value.

additive-symbols descriptor

Value	[<*integer*> && <*symbol*>]#
Initial value	n/a
Note	<*integer*> values must be nonnegative, and additive counters are not applied when a counter's value is negative.

It's much easier to show this than explain it. Here's an example adapted from Kseso:

```
@counter-style roman {
    system: additive;
    additive-symbols:
        1000 M, 900 CM, 500 D, 400 CD,
        100 C, 90 XC, 50 L, 40 XL,
        10 X, 9 IX, 5 V, 4 IV, 1 I;
}
```

This will count in classical Roman style. Another good example can be found in the specification for counting styles, which defines a dice-counting system:

```
@counter-style dice {
    system: additive;
    additive-symbols: 6 🁅, 5 🁄, 4 🁃, 3 🁂, 2 🁁, 1 🁀, 0 "__";
    suffix: " ";
}
```

The results of both counting systems are shown in Figure 16-45; this time, each list has been formatted as three columns.

-3. minus three	VI. six	XV. 15	-3 minus three	⚅ six	⚅⚄⚂ 15
-2. minus two	VII. seven	XVI. 16	-2 minus two	⚅⚀ seven	⚅⚄⚃ 16
-1. minus one	VIII. eight	XVII. 17	-1 minus one	⚅⚁ eight	⚅⚄⚄ 17
0. zero	IX. nine	XVIII. 18	__ zero	⚅⚂ nine	⚅⚅⚅ 18
I. one	X. ten	XIX. 19	⚀ one	⚅⚃ ten	⚅⚅⚅⚀ 19
II. two	XI. 11	XX. 20	⚁ two	⚅⚄ 11	⚅⚅⚅⚁ 20
III. three	XII. 12	XXI. 21	⚂ three	⚅⚅ 12	⚅⚅⚅⚂ 21
IV. four	XIII. 13	XXII. 22	⚃ four	⚅⚅⚀ 13	⚅⚅⚅⚃ 22
V. five	XIV. 14	XXIII. 23	⚄ five	⚅⚅⚁ 14	⚅⚅⚅⚄ 23

Figure 16-45. Additive values

Symbols can be quoted for clarity; e.g., `6 "🁅", 5 "🁄", 4 "🁃"`, and so on.

The most important thing to keep in mind is that the order of the symbols and their equivalent values matters. Notice that both the Roman and dice-counting systems supply values from largest to smallest, not the other way around? That's because if you put the values in any order other than descending, the entire block is rendered invalid.

Also notice the use of the `additive-symbols` descriptor instead of `symbols`. This is important to keep in mind, since defining an additive system and then trying to use the `symbols` descriptor will render the entire `counter-styles` block invalid. (Similarly, attempting to use the `additive-symbols` description in non-`additive` systems will render *those* blocks invalid.)

One last thing to note about additive systems is that, because of the way the additive-counter algorithm is defined, it's possible to create additive systems in which some values can't be represented even though it seems like they should be. Consider this definition:

```
@counter-style problem {
    system: additive;
    additive-symbols: 3 "Y", 2 "X";
    fallback: decimal;
}
```

This would yield the following counters for the first five numbers: 1, X, Y, 4, YX. You might think 4 should be XX, and that may make intuitive sense, but the algorithm for additive symbols doesn't permit it. To quote the specification: "While unfortunate, this is required to maintain the property that the algorithm runs in linear time relative to the size of the counter value."



So how does Roman counting manage to get III for 3? Again, the answer is in the algorithm. It's a little too complicated to get into here, so if you're truly curious, we recommend you read the CSS Counter Styles Level 3 specification, which defines the additive counting algorithm. If that doesn't interest you, just remember: make sure you have a symbol whose value equates to `1`, and you'll avoid this problem.

Extending Counting Patterns

There may come a time when you just want to vary an existing counting system a bit. For example, suppose you want to change regular decimal counting to use close-parentheses symbols as suffixes, and pad up to two leading zeros. You could write it all out longhand, like so:

```
@counter-style mydecimals {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: ") ";
    pad: 2 "0";
}
```

That works, but it's clumsy. Well, worry not: `extends` is here to help.

The `extends` option is sort of a system type, but only in the sense that it builds on an existing system type. The previous example would be rewritten with `extends` as follows:

```
@counter-style mydecimals {
    system: extends decimal;
    suffix: ") ";
    pad: 2 "0";
}
```

That takes the existing `decimal` system familiar from `list-style-type` and reformats it a bit. Thus, there's no need to retype the whole symbol chain. You just adjust the options, as it were.

In fact, you can *only* adjust the options: if you try to use either `symbols` or `additive-symbols` in an `extends` system, the entire `@counter-style` block will be invalid and ignored. In other words, symbols cannot be extended. As an example, you can't define hexadecimal counting by extending decimal counting.

However, you can vary the hexadecimal counting for different contexts. As an example, you could set up basic hex counting and then define variant display patterns, as shown in the following code and illustrated in Figure 16-46.



Each list jumps from 19 to 253, thanks to a `value="253"` on one of the list items.

```
@counter-style hexadecimal {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D' 'E' 'F';
}
@counter-style hexpad {
    system: extends hexadecimal;
    pad: 2 "0";
}
@counter-style hexcolon {
    system: extends hexadecimal;
    suffix: ": ";
}
@counter-style hexcolonlimited {
    system: extends hexcolon;
    range: 1 255; /* stops at FF */
}
```

-3. minus three	A. ten B. 11	-3. minus three	0A. ten 0B. 11	-3: minus three	A: ten B: 11	-3. minus three	A: ten B: 11
-2. minus two	C. 12 D. 13	-2. minus two	0C. 12 0D. 13	-2: minus two	C: 12 D: 13	-2. minus two	C: 12 D: 13
-1. minus one	E. 14 F. 15	-1. minus one	0E. 14 0F. 15	-1: minus one	E: 14 F: 15	-1. minus one	E: 14 F: 15
0. zero	10. 16	00. zero	10. 16	0: zero	10: 16	0. zero	10: 16
1. one	11. 17	01. one	11. 17	1: one	11: 17	1: one	11: 17
2. two	12. 18	02. two	12. 18	2: two	12: 18	2: two	12: 18
3. three	13. 19	03. three	13. 19	3: three	13: 19	3: three	13: 19
4. four	FD. 253	04. four	FD. 253	4: four	FD: 253	4: four	FD: 253
5. five	FE. 254	05. five	FE. 254	5: five	FE: 254	5: five	FE: 254
6. six	FF. 255	06. six	FF. 255	6: six	FF: 255	6: six	FF: 255
7. seven	100. 256	07. seven	100. 256	7: seven	100: 256	7: seven	256. 256
8. eight	101. 257	08. eight	101. 257	8: eight	101: 257	8: eight	257. 257
9. nine		09. nine		9: nine		9: nine	

Figure 16-46. Various hexadecimal counting patterns

Notice that the last of the four counter styles, `hexcolonlimited`, extends the third, `hexcolon`, which itself extends the first, `hexadecimal`. In `hexcolonlimited`, the hexadecimal counting stops at `FF` (255), thanks to the `range: 1 255;` declaration.

Speaking Counting Patterns

While it's fun to build counters out of symbols, the result can be a real mess for spoken technologies such as Apple's VoiceOver or the JAWS screen reader. Imagine, for example, a screen reader trying to read dice counters or phases of the moon. To help, the `speak-as` descriptor allows you to define an audible fallback.

speak-as descriptor

Values	`auto` | `bullets` | `numbers` | `words` | `spell-out` | <*`counter-style-name`*>
Initial value	`auto`



As of late 2022, `speak-as` is supported only by Mozilla-based browsers.

Let's take the values backward. With a <*`counter-style-name`*>, you're able to define an alternate counting style that the user agent likely already recognizes. For example, you likely want to provide an audio fallback for dice counting to be `decimal`, one of the well-supported `list-style-type` values, when spoken:

```
@counter-style dice {
    system: additive;
    speak-as: decimal;
    additive-symbols: 6 🎲, 5 ⚄, 4 ⚃, 3 ⚂, 2 ⚁, 1 ⚀;
    suffix: " ";
}
```

Given those styles, the counter ⚀⚀⚂ would be spoken as "fifteen." Alternatively, if the `speak-as` value is changed to `lower-latin`, that counter will be spoken as "oh" (capital letter *O*).

The `spell-out` value might seem fairly straightforward but it's a little more complicated than it first appears. What is spelled out by the user agent is a "counter representation," which is then spelled out letter by letter. It's hard to predict what that will mean, since the method of generating a counter representation isn't precisely defined: the specification says, "Counter representations are constructed by concatenating counter symbols together." And that's all.

The `words` value is similar to `spell-out`, except the counter representation is spoken as words instead of spelling out each letter. Again, the exact process is not defined.

With the value `numbers`, the counters are spoken as numbers in the document language. This is similar to the previous code sample, where ⚀⚀⚂ is spoken as "fifteen," at least in English documents. If it's another language, that language is used for counting: "quince" in Spanish, "fünfzehn" in German, "shíwǔ" in Chinese, and so on.

Given `bullets`, the user agent says whatever it says when reading a bullet (marker) in an unordered list. This may mean saying nothing at all, or producing an audio cue such as a chime or click.

Finally, consider the default value of `auto`. We saved this for last because its effect depends on the counting system in use. If it's an `alphabetic` system, `speak-as: auto` has the same effect as `speak-as: spell-out`. In `cyclic` systems, `auto` is the same as `bullets`. Otherwise, the effect is the same as `speak-as: numbers`.

The exception to this rule arises if the system is an `extends` system, in which case `auto`'s effects are determined based on the system being extended. Therefore, given the following styles, the counters in an `emojibrackets` list will be spoken as if `speak-as` were set to `bullets`:

```
@counter-style emojilist {
    emojiverse {
    system: cyclic;
    symbols: 😁;
@counter-style emojibrackets {
    system: extends emojilist;
    suffix: "]] ";
    speak-as: auto;
}
```

Summary

Even though list styling isn't as sophisticated as we might like, the ability to style lists is still highly useful. One relatively common use is to take a list of links, remove the markers and indentation, and thus create a navigation sidebar. The combination of simple markup and flexible layout is difficult to resist.

Remember, if a markup language doesn't have intrinsic list elements, generated content can be an enormous help—say, for inserting content such as icons to point to certain types of links (PDF files, Word documents, or even just links to another website). Generated content also makes it easy to print out link URLs, and its ability to insert and format quotation marks leads to true typographic joy. It's safe to say that the usefulness of generated content is limited only by your imagination. Even better, thanks to counters, you can now associate ordering information to elements that are not typically lists, such as headings or code blocks. If you want to support such features with design that mimics the appearance of the user's operating system, read on. The next chapter discusses ways to change the placement, shape, and even perspective of your design.

CHAPTER 17

Transforms

Ever since the inception of CSS, elements have been rectangular and firmly oriented on the horizontal and vertical axes. Several tricks arose to make elements look like they were tilted and so on, but underneath it all was a rigid grid.

With CSS *transforms*, you have the ability to break that visual grid and shake up the way your elements are presented. Whether it's as simple as rotating some photographs a bit to make them appear more natural, or creating interfaces where information can be revealed by flipping over elements, or doing interesting perspective tricks with sidebars, CSS transforms can—if you'll pardon the obvious expression—transform the way you design.

Coordinate Systems

Before embarking on this journey, let's take a moment to orient ourselves. Specifically, let's review the *coordinate systems* used to define positions or movement in space as a sequence of measurements. Two types of coordinate systems are used in transforms, and it's a good idea to be familiar with both.

The first is the *Cartesian coordinate system*, often called the *x/y/z coordinate system*. This system describes the position of a point in space by using two numbers (for two-dimensional placement) or three numbers (for three-dimensional placement). In CSS, the system uses three axes: the x-axis (horizontal); the y-axis (vertical); and the z-axis (depth). This is illustrated in Figure 17-1.

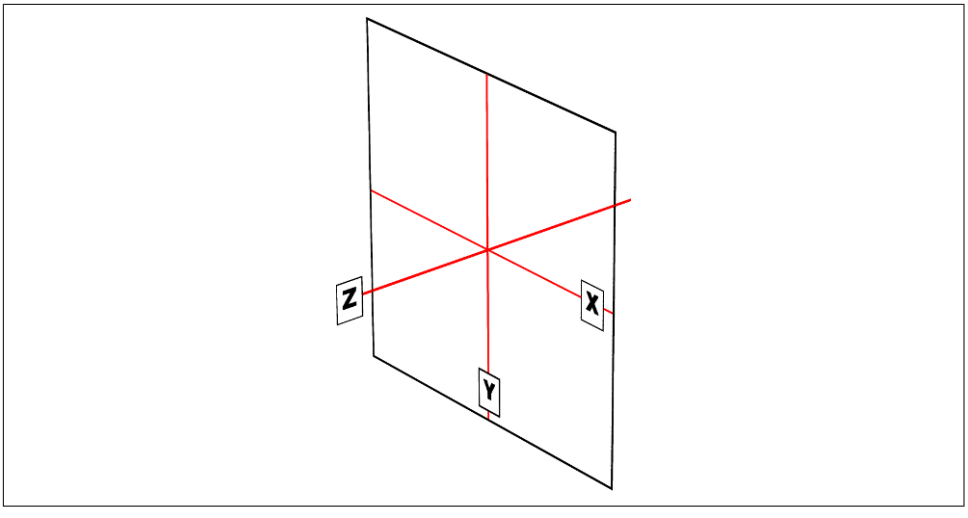


Figure 17-1. The three Cartesian axes used in CSS transforms

For any two-dimensional (2D) transform, you need to worry about only the x- and y-axes. By convention, positive x values go to the right, and negative values go to the left. Similarly, positive y values go downward along the y-axis, while negative values go upward along the y-axis.

That might seem a little weird, since we tend to think that higher numbers should place something higher up, not lower down, as many of us learned in pre-algebra. (This is why the "y" label is at the bottom of the y-axis in Figure 17-1: the labels are placed in the positive direction on all three axes.) If you are experienced with absolute positioning in CSS, think of the `top` property values for absolutely positioned elements: they get moved downward for positive `top` values, and upward when `top` has a negative length.

Given this, in order to move an element leftward and down, you would give it a negative x and a positive y value. Here is one way to do this:

```
translateX(-5em) translateY(33px)
```

That is, in fact, a valid transform value, as you'll see in just a bit. Its effect is to translate (move) the element 5 ems to the left and 33 pixels down, in that order.

If you want to transform something in three-dimensional (3D) space, you add a z-axis value. This axis is the one that "sticks out" of the display and runs straight through your head—in a theoretical sense, that is. Positive z values are closer to you, and negative z values are farther away from you. In this regard, it's very much like the `z-index` property.

So let's say that we want to take the element we moved before and add a z-axis value:

```
translateX(-5em) translateY(33px) translateZ(200px)
```

Now the element will appear 200 pixels closer to us than it would be without the z value.

Well, you might wonder exactly how an element can be moved 200 pixels closer to you, given that holographic displays are regrettably rare and expensive. How many molecules of air between you and your monitor are equivalent to 200 pixels? What does an element moving closer to you even look like, and what happens if it gets *too* close? These are excellent questions that we'll get to later. For now, just accept that moving an element along the z-axis appears to move it closer or farther away.

The really important thing to remember is that every element carries its own frame of reference and so considers its axes with respect to itself. If you rotate an element, the axes rotate along with it, as illustrated in Figure 17-2. Any further transforms are calculated with respect to those rotated axes, not the axes of the display.

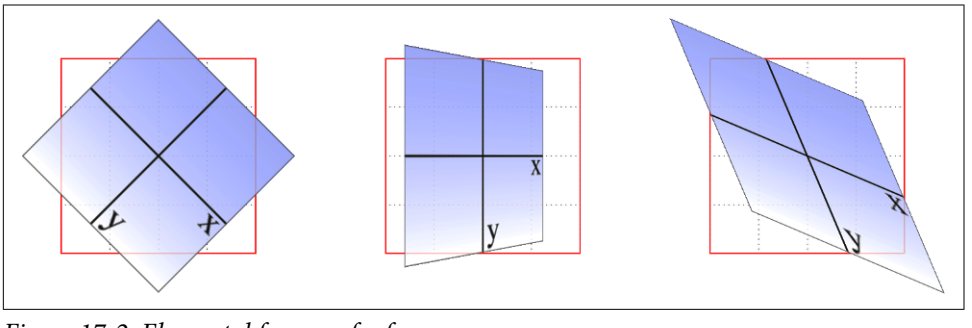


Figure 17-2. Elemental frames of reference

Now, suppose you want to rotate an element 45 degrees clockwise in the plane of the display (i.e., around the z-axis). Here's the transform value you're most likely to use:

```
rotate(45deg)
```

Change that to `-45deg`, and the element will rotate counterclockwise (anticlockwise for our international friends) around the z-axis. In other words, it will rotate in the *xy* plane, as illustrated in Figure 17-3.

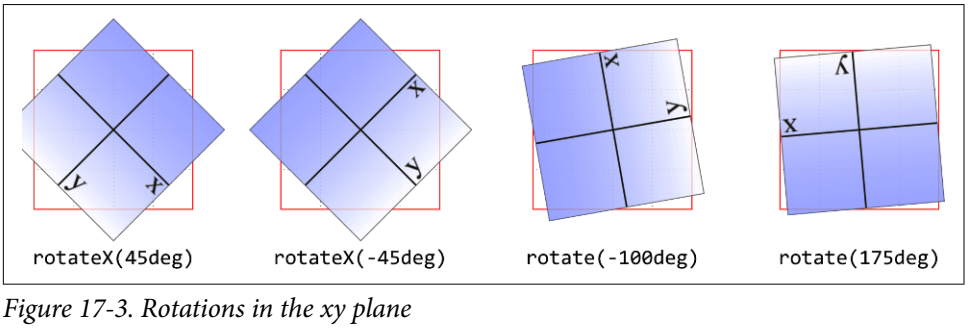


Figure 17-3. Rotations in the xy plane

Speaking of rotations, the other coordinate system used in CSS transforms is a *spherical system*, which describes angles in 3D space. It's illustrated in Figure 17-4.

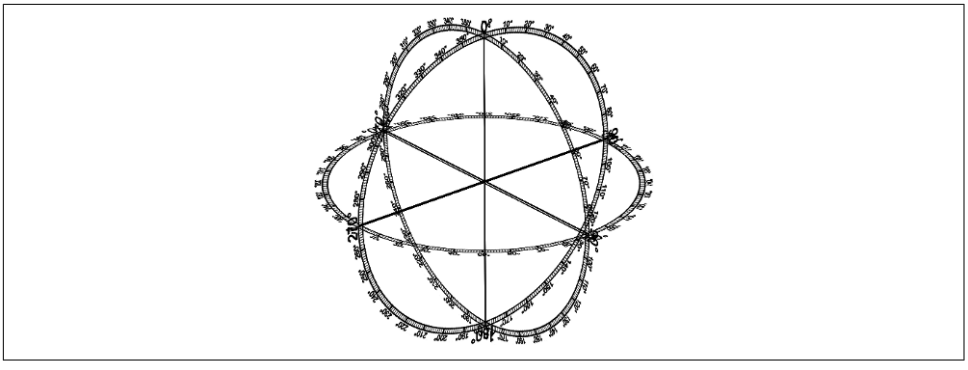


Figure 17-4. The spherical coordinate system used in CSS transforms

For the purposes of 2D transforms, you have to worry about only a single 360-degree polar system: the one that sits on the plane described by the x- and y-axes. When it comes to rotations, a 2D rotation actually describes a rotation around the z-axis. Similarly, rotations around the x-axis tilt the element toward or away from you, and rotations around the y-axis turn the element from side to side. These are illustrated in Figure 17-5.

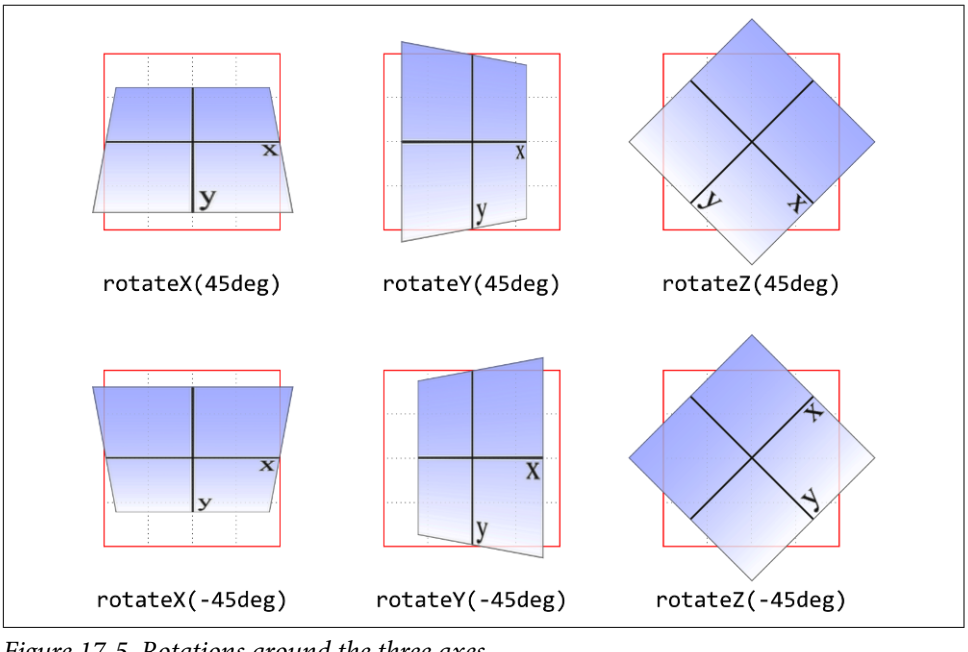


Figure 17-5. Rotations around the three axes

All right, now that we have our bearings, let's get started with using CSS transforms!

Transforming

One property applies all transforms as a single operation, and a few ancillary properties affect exactly how the transforms are applied or allow transforms in a single manner. We'll start with the big cheese.

transform

Values	<*transform-list*> | none
Initial value	none
Applies to	All elements except atomic inline-level boxes (see explanation)
Percentages	Refer to the size of the bounding box (see explanation)
Computed value	As specified, except for relative length values, which are converted to an absolute length
Inherited	No
Animatable	As a transform

A <*transform-list*> is a space-separated list of functions defining different transformations, like the examples used in the preceding section. We'll dig into the specific functions you can use in a moment.

First off, let's clear up the matter of the bounding box. For any element being affected by CSS, the *bounding box* is the border box—the outermost edge of the element's border. Any outlines and margins are ignored for the purposes of calculating the bounding box.



If a table-display element is being transformed, its bounding box is the table wrapper box, which encloses the table box and any associated caption box.

If you're transforming an SVG element with CSS, its bounding box is its SVG-defined *object bounding box*.

Note that all transformed elements (e.g., elements with `transform` set to a value other than `none`) have their own stacking context. (See "Placement on the Z-Axis" on page 444 for an explanation.)

While a scaled element may be much smaller or larger than it was before the transform was applied, the actual space on the page that the element occupies remains the same as before the transform was applied. This is true for all the transform functions: when you translate or rotate an element, its siblings don't automatically move out of the way.

Now, the value entry <*transform-list*> requires some explanation. It refers to a list of one or more transform functions, one after the other, in space-separated format. It looks something like this, with the result shown in Figure 17-6:

```
#example {transform: rotate(30deg) skewX(-25deg) scaleY(2);}
```



Figure 17-6. A transformed element

The functions are processed one at a time, starting with the first (leftmost) and proceeding to the last (rightmost). This first-to-last processing order is important, because changing the order can lead to drastically different results. Consider the following two rules, which have the results shown in Figure 17-7:

```
img#one {transform: translateX(200px) rotate(45deg);}
img#two {transform: rotate(45deg) translateX(200px);}
```

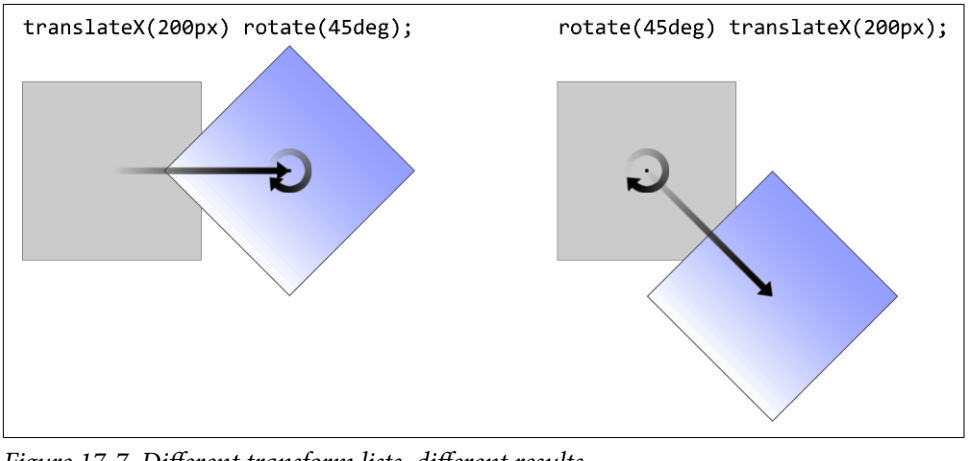


Figure 17-7. Different transform lists, different results

In the first instance, an image is translated (moved) 200 pixels along its x-axis and then rotated 45 degrees. In the second instance, an image is rotated 45 degrees and then moved 200 pixels along its x-axis—that's the x-axis of the transformed element, *not* of the parent element, page, or viewport. In other words, when an element is rotated, its x-axis (along with its other axes) rotates along with it. All element transforms are conducted with respect to the element's own frame of reference.

Note that when you have a series of transform functions, all of them must be properly formatted; that is, they must be valid. If even one function is invalid, it renders the entire value invalid. Consider the following:

```
img#one {transform: translateX(100px) scale(1.2) rotate(22);}
```

Because the value for `rotate()` is invalid—rotational values must be an *<angle>*—the entire value is dropped. The image in question will just sit there in its initial untransformed state, neither translated nor scaled, let alone rotated.

In addition, transforms are not usually cumulative. If you apply a transform to an element and then later want to add a transformation, you need to restate the original transform. Consider the following scenarios, illustrated in Figure 17-8:

```
#ex01 {transform: rotate(30deg) skewX(-25deg);}
#ex01 {transform: scaleY(2);}
#ex02 {transform: rotate(30deg) skewX(-25deg);}
#ex02 {transform: rotate(30deg) skewX(-25deg) scaleY(2);}
```

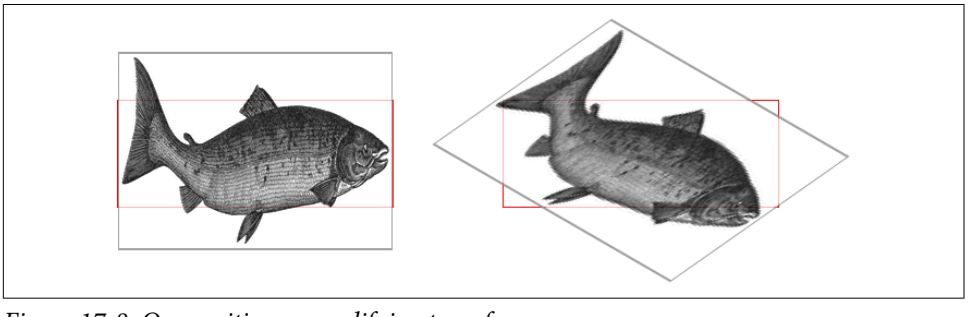


Figure 17-8. Overwriting or modifying transforms

In the first case, the second rule completely replaces the first, meaning that the element is scaled only along the y-axis. This makes some sense; it's the same as if you declare a font size and then elsewhere declare a different font size for the same element. You don't get a cumulative font size that way. You just get one size or the other. In the second example, the entirety of the first set of transforms is included in the second set, so they all are applied along with the `scaleY()` function.



If you're wishing for properties that apply to just a single type of transformation, such as a property that only rotates or a property that only scales elements, you'll see some later in the chapter, so hang in there.

There's one important caveat: as of this writing, transforms are not applied to *atomic inline-level* boxes. These are inline boxes like spans, hyperlinks, and so on. Those elements can be transformed if their block-level parent is transformed, in which case they go along for the ride. But you can't just rotate a `<span>` unless you've changed its display role via `display: block`, `display: inline-block`, or something along those lines. The reason for this limitation boils down to an uncertainty. Suppose you have a `<span>` (or any inline-level box) that breaks across multiple lines. If you rotate it, what happens? Does each line box rotate with respect to itself, or should all the line boxes be rotated as a single

group? There's no clear answer, and the debate continues, so for now you can't directly transform inline-level boxes.

The Transform Functions

CSS has 21 transform functions, as of early 2023, employing various value patterns to get their jobs done. The following is a list of all the available transform functions, minus their value patterns:

```
translate()   scale()   rotate()   skew()   matrix()
translate3d() scale3d() rotate3d() skewX()  matrix3d()
translateX()  scaleX()  rotateX()  skewY()  perspective()
translateY()  scaleY()  rotateY()
translateZ()  scaleZ()  rotateZ()
```

We'll tackle the most common types of transforms first, along with their associated properties if they exist, and then deal with the more obscure or difficult ones.

Translation

A *translation transform* is just a move along one or more axes. For example, `translateX()` moves an element along its own x-axis, `translateY()` moves it along its y-axis, and `translateZ()` moves it along its z-axis.

translateX(), translateY() functions

Values <*length*> | <*percentage*>

These are usually referred to as the 2D *translation functions*, since they can slide an element up and down, or side to side, but not forward or backward along the z-axis. Each of these functions accepts a single distance value, expressed as either a length or a percentage.

If the value is a length, the effect is about what you'd expect. Translate an element 200 pixels along the x-axis with `translateX(200px)`, and it will move 200 pixels to its right. Change that to `translateX(-200px)`, and the element will move 200 pixels to its left. For `translateY()`, positive values move the element downward, while negative values move it upward.

Keep in mind that translations are always declared with respect to the element itself. Thus, for example, if you flip the element upside down by rotation, positive `translateY()` values will move the element downward on the page, because that's a move upward from the upside-down element's point of view.

If the value is a percentage, the distance is calculated as a percentage of the element's own size. Thus, if an element is 300 pixels wide and 200 pixels tall, `translateX(50%)` will move it 150 pixels to its right, and `translateY(-10%)` will move that same element upward (with respect to itself) by 20 pixels.

translate() function

Values [*<length>* | *<percentage>*] [, *<length>* | *<percentage>*]?

If you want to translate an element along both the x- and y-axes at the same time, `translate()` makes it easy. Just supply the x value first and the y value second, separated by a comma, which is the same as if you included both a `translateX()` and a `translateY()`. If you omit the y value, it's assumed to be 0. Thus, `translate(2em)` is treated as if it were `translate(2em,0)`, which is also the same as `translateX(2em)`. See Figure 17-9 for some examples of 2D translation.

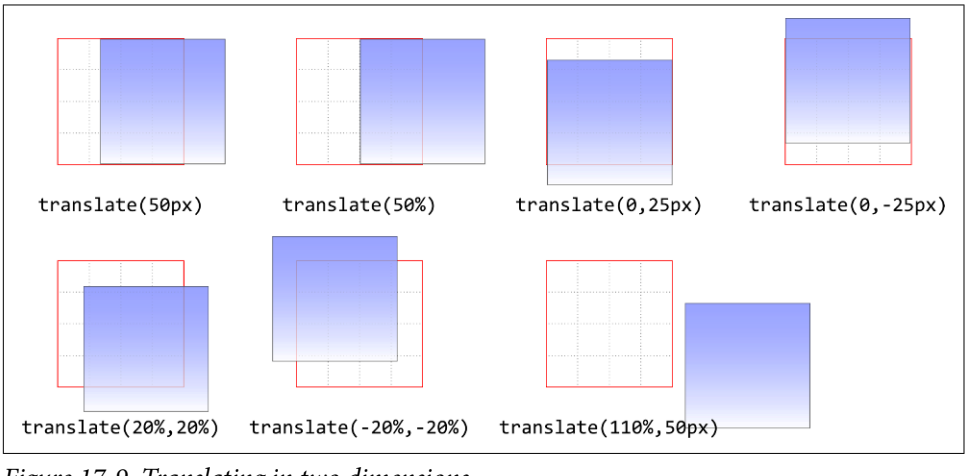


Figure 17-9. Translating in two dimensions

translateZ() function

Value *<length>*

The `translateZ()` function translates elements along the z-axis, thus moving them into the third dimension. Unlike the 2D translation functions, `translateZ()` accepts only

length values. Percentage values are *not* permitted for `translateZ()`, or indeed for any z-axis value.

translate3d() function

Values [<*length*> | <*percentage*>], [<*length*> | <*percentage*>], [<*length*>]

Much like `translate()` does for x and y translations, `translate3d()` is a shorthand function that incorporates the x, y, and z translation values into a single function. This is handy if you want to move an element over, up, and forward in one fell swoop.

See Figure 17-10 for an illustration of how 3D translation works. Each arrow represents the translation along that axis, arriving at a point in 3D space. The dashed lines show the distance and direction from the origin point (the intersection of the three axes) and the distance above the x*y* plane.

Unlike `translate()`, there is no fallback if `translate3d()` does not contain three values. Thus, `translate3d(1em,-50px)` should be treated as invalid by browsers, with no actual translation taking place as a result.

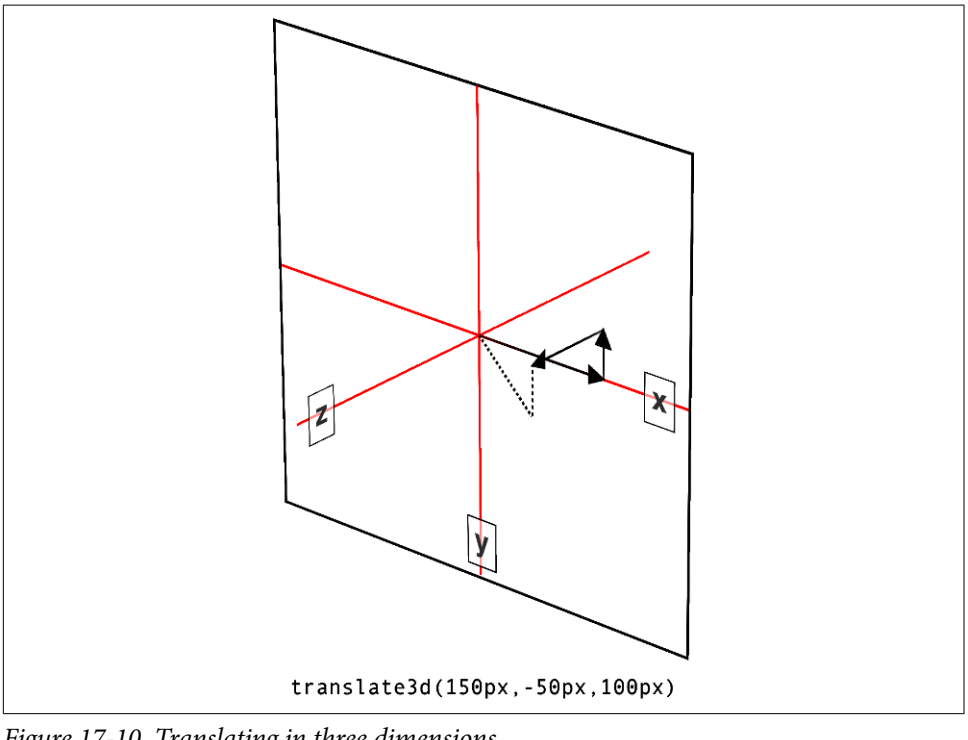


Figure 17-10. Translating in three dimensions

The translate property

When you want to translate an element without having to go through the `transform` property, you can use the `translate` property instead.

translate

Values	| none | [<*length*> | <*percentage*>]{1,2} <*length*>?
Initial value	none
Applies to	Any transformable element
Percentages	Refer to the corresponding size of the bounding box
Computed value	As specified, except for relative length values, which are converted to an absolute length
Inherited	No
Animatable	As a transform

Very much like the `translate()` function, the `translate` property accepts from one to three length values, or two percentages and a length value, or more reduced patterns such as a single length. Unlike the `translate()` function, the `transform` property does not use commas to separate its values.

If only one value is given, it is used as an x-axis translation. With two values, the first is the x-axis translation, and the second is the y-axis translation. With three values, they are taken in the order `x y z`. Any missing values default to `0px`.

If you refer back to Figure 17-9, the following would yield the same results as are shown there:

```
translate: 25px;    /* equivalent to 25px 0px 0px */
translate: 25%;
translate: 0 25px;  /* equivalent to 0 25px 0px */
translate: 0 -25px;
translate: 20% 20%;
translate: -20% -20%;
translate: 110% 25px;
```

Similarly, the following would have the same effect diagrammed in Figure 17-10:

```
translate: 150px -50px 100px;
```

The default value, `none`, means that no translation is applied.

Scaling

A *scale transform* makes an element larger or smaller, depending on the value you supply. These values are unitless real numbers, either positive or negative. On the 2D plane, you can scale along the x- and y-axes individually or scale them together.

scaleX(), scaleY(), scaleZ() functions

Values <*number*> | <*percentage*>

A number value supplied to a scale function is a multiplier; thus, `scaleX(2)` will make an element twice as wide as it was before the transformation, whereas `scaleY(0.5)` will make it half as tall. Percentage values are equivalent to number values at a ratio of 100:1; that is, `50%` will have the same effect as `0.5`, and `200%` will have the same effect as `2`, and so on.

scale() function

Values [<*number*> | <*percentage*>] [, <*number*> | <*percentage*>]?

If you want to scale along both axes simultaneously, use `scale()`. The x value is always first, and the y always second, so `scale(2,0.5)` will make the element twice as wide and half as tall as it was before being transformed. If you supply only one number, it is used as the scaling value for both axes; thus, `scale(2)` will make the element twice as wide *and* twice as tall. This is in contrast to `translate()`, where an omitted second value is always set to 0. Using `scale(1)` will scale an element to be exactly the same size it was before you scaled it, as will `scale(1,1)`—just in case you were dying to do that.

Figure 17-11 shows a few examples of element scaling, using the single-axis scaling functions as well as the combined `scale()`.

If you can scale in two dimensions, you can also scale in three. CSS offers `scaleZ()` for scaling just along the z-axis, and `scale3d()` for scaling along all three axes at once. These have an effect only if the element has any depth, which elements don't by default. If you do make a change that conveys depth—say, rotating an element around the x- or y-axis—then there is a depth that can be scaled, and either `scaleZ()` or `scale3d()` can do so.

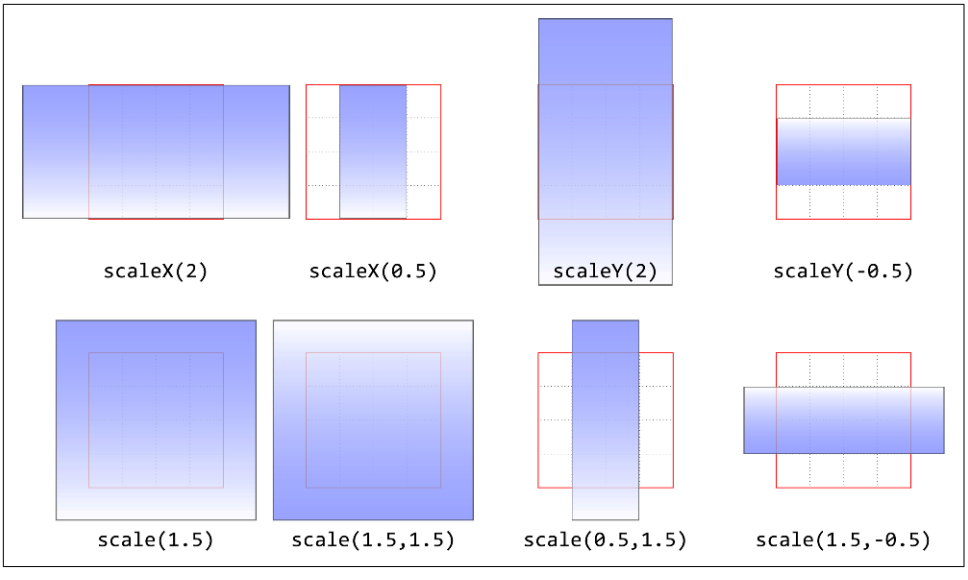


Figure 17-11. Scaled elements

scale3d() function

Values [*<number>* | *<percentage>*] , [*<number>* | *<percentage>*] , [*<number>* | *<percentage>*]

Similar to `translate3d()`, the `scale3d()` function requires all three numbers to be valid. If you fail to do this, the malformed `scale3d()` will invalidate the entire transform value to which it belongs.

Also note that scaling an element will change the effective distance of any translations. For example, the following will cause the element to be translated 50 pixels to its right:

```
transform: scale(0.5) translateX(100px);
```

This is because the element is shrunk by 50%, and then moved to the right by 100 pixels *within its own frame of reference*, which is half-size. Switch the order of the functions, and the element will be translated 100 pixels to its right and then shrunk 50% from that spot.

The scale property

Also similarly to translation, the `scale` property allows you to scale elements up or down without having to use the `transform` property.

scale

Values	none | [<*percentage*> | <*number*>]{1,3}
Initial value	none
Applies to	Any transformable element
Percentages	Refer to the corresponding size of the bounding box
Computed value	As specified
Inherited	No
Animatable	As a transform

The way `scale` handles its values differs little from the `translate` property. If you give only one value, such as `scale(2)`, that value is used to scale in both the x and y directions. With two values, the first is used to scale in the x-axis direction, and the second in the y-axis direction. With three values, the third is used to scale in the z-axis direction.

The following would have the same results as shown in Figure 17-11.

```
scale: 2 1;   /* equivalent to 200% 100% */
scale: 0.5 1; /* equivalent to 50% 100% */
scale: 1 2;
scale: 1 0.5;
scale: 1.5;
scale: 1.5;
scale: 0.5 1.5;
scale: 1 5 0.5;
```

The default value, none, means that no scaling is applied.

Element Rotation

A *rotation function* causes an element to be rotated around an axis, or around an arbitrary vector in 3D space. CSS has four simple rotation functions, and one less-simple function meant specifically for 3D.

rotate(), rotateX(), rotateY(), rotateZ() functions

Value	<*angle*>

All four basic rotation functions accept just one value: an angle. This can be expressed using a number, either positive or negative, and then any of the valid angle units (`deg`, `grad`, `rad`, and `turn`). (See "Angles" on page 166 for more details.) If a value's number runs

outside the usual range for the given unit, it will look as if it were given a value in the allowed range. In other words, a value of `437deg` will be tilted the same as if it were `77deg`, or, for that matter, `-283deg`.

Note, however, that these are visually equivalent only if you don't animate the rotation in some fashion. That is to say, animating a rotation of `1100deg` will spin the element around several times before coming to rest at a tilt of –20 degrees (or 340 degrees, if you like). By contrast, animating a rotation of `-20deg` will tilt the element a bit to the left, with no spinning; and animating a rotation of `340deg` will animate an almost full spin to the right. All three animations come to the same end state, but the process of getting there is very different in each case.

The `rotate()` function is a straight 2D rotation, and the one you're most likely to use. It is visually equivalent to `rotateZ()` because it rotates the element around the z-axis. In a similar manner, `rotateX()` causes rotation around the x-axis, thus causing the element to tilt toward or away from you; and `rotateY()` rotates the element around its y-axis, as though it were a door. These are all illustrated in Figure 17-12.

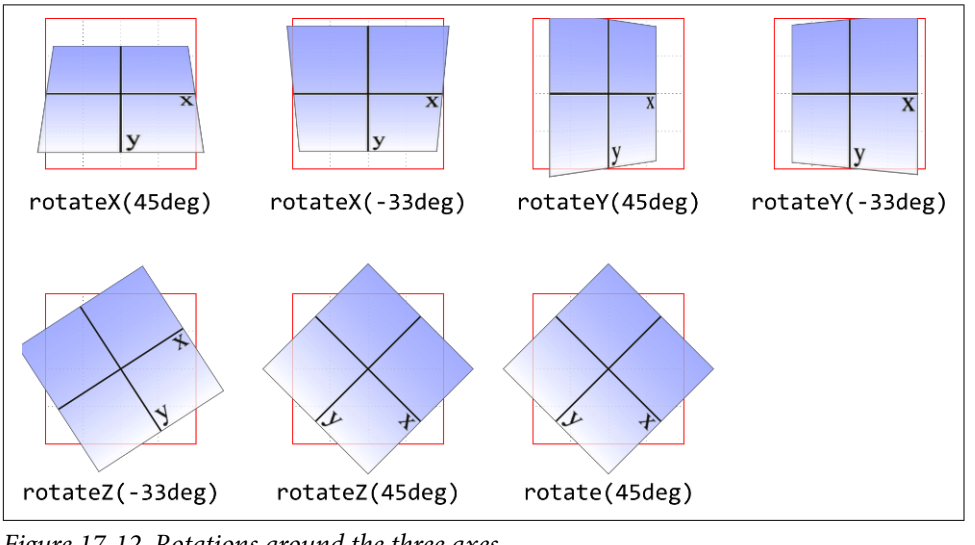


Figure 17-12. Rotations around the three axes



Several of the examples in Figure 17-12 present a fully 3D appearance. This is possible with certain values of the `transform-style` and `perspective` properties, described in "Choosing a 3D Style" on page 870 and "Changing Perspective" on page 873 and omitted here for clarity. This will be true throughout this text anytime 3D-transformed elements appear to be fully three-dimensional. This is important to keep in mind because if you just try to apply the transform functions shown, you won't get the same visual results as in the figures.

rotate3d() function

Values *<number>*, *<number>*, *<number>*, *<angle>*

If you're comfortable with vectors and want to rotate an element through 3D space, `rotate3d()` is for you. The first three numbers specify the x, y, and z components of a vector in 3D space, and the degree value (angle) determines the amount of rotation around the declared 3D vector.

To start with a basic example, the 3D equivalent of `rotateZ(45deg)` is `rotate3d(0,0,1,45deg)`. This specifies a vector of zero magnitude on the x- and y-axes, and a magnitude of 1 along the z-axis; in other words, the z-axis. The element is thus rotated 45 degrees around that vector, as shown in Figure 17-13. This figure also shows the appropriate `rotate3d()` values to rotate an element by 45 degrees around the x- and y-axes.

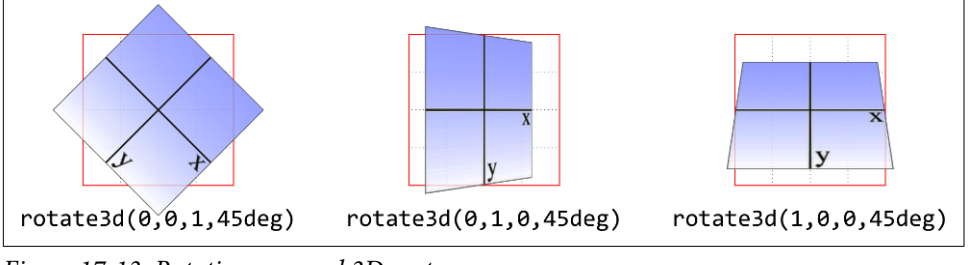


Figure 17-13. Rotations around 3D vectors

A little more complicated is something like `rotate3d(-0.95,0.5,1,45deg)`, where the described vector points off into 3D space between the axes. To understand how this works, let's start with a basic example: `rotateZ(45deg)` (illustrated in Figure 17-13). The equivalent is `rotate3d(0,0,1,45deg)`. The first three numbers describe the components of a vector that has no x or y magnitude, and a z magnitude of 1. Thus, it points along the z-axis in a positive direction—that is, toward the viewer. The element is then rotated clockwise as you look toward the origin of the vector.

Similarly, the 3D equivalent of `rotateX(45deg)` is `rotate3d(1,0,0,45deg)`. The vector points along the x-axis in the positive direction (to the right). If you stand at the end of that vector and look toward its origin, you rotate the element 45 degrees clockwise around the vector. Thus, from the usual viewer placement, the top of the element rotates away from, and the bottom rotates toward, the viewer.

Now let's make the example slightly more complex: suppose you have `rotate3d(1,1,0,45deg)`. When viewed on your monitor, that describes a vector running from the top-left to bottom-right corner, going right through the center of the element

(by default, anyway; we'll see how to change that later). So the element's rectangle has a line running through it at a 45-degree angle, effectively spearing it. Then the vector rotates 45 degrees, taking the element with it. The rotation is clockwise as you look back toward the vector's origin, so again, the top of the element rotates away from the viewer, while the bottom rotates toward the viewer. If we were to change the rotation to `rotate3d(1,1,0,90deg)`, the element would be edge-on to the viewer, tilted at a 45-degree angle and facing off toward the upper right. Try it with a piece of paper: draw a line from the top left to bottom right, and then rotate the paper around that line.

OK, so given all of that, now try visualizing how the vector is determined for `rotate3d(-0.95,0.5,1,45deg)`. If we assume a cube 200 pixels on a side, the vector's components are 190 pixels to the *left* along the x-axis, 100 pixels down along the y-axis, and 200 pixels toward the views along the z-axis. The vector goes from the origin point (0, 0, 0) to the point (–190 px, 100 px, 200 px). Figure 17-14 depicts that vector, as well as the final result presented to the viewer.

So the vector is like a metal rod speared through the element being rotated. As we look back along the line of the vector, the rotation is 45 degrees clockwise. But since the vector points left, down, and forward, that means the top-left corner of the element rotates toward the viewer, and the bottom right rotates away, as shown in Figure 17-14.

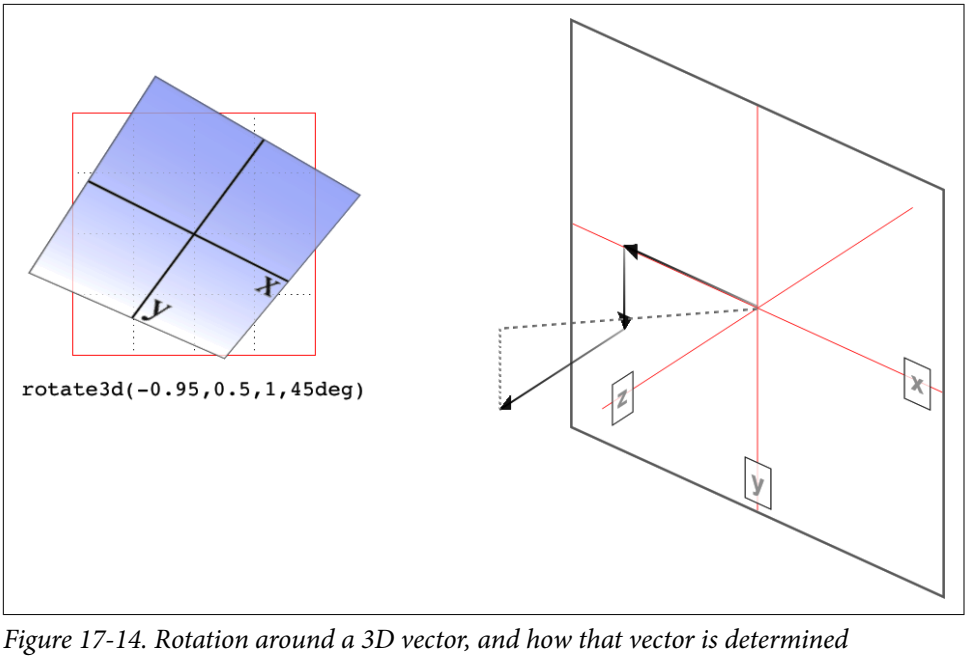


Figure 17-14. Rotation around a 3D vector, and how that vector is determined

Just to be crystal clear, `rotate3d(1,1,0,45deg)` is *not* equivalent to `rotateX(45deg) rotateY(45deg) rotateZ(0deg)`! It's an easy mistake to make, and many people—including your humble correspondent—have made it. It seems like it should be equivalent, but it really isn't. If we place that vector inside the imaginary 200 × 200 × 200 cube previously

mentioned, the axis of rotation would go from the origin point to a point 200 pixels right and 200 pixels down (200, 200, 0).

Having done that, the axis of rotation is shooting through the element from the top left to the bottom right, at a 45-degree angle. The element then rotates 45 degrees clockwise around that diagonal, as you look back toward its origin (the top left), which rotates the top-right corner of the element away and a bit to the left, while the bottom-left corner rotates closer and a bit to the right. This is distinctly different from the result of `rotateX(45deg) rotateY(45deg) rotateZ(0deg)`, as you can see in Figure 17-15.

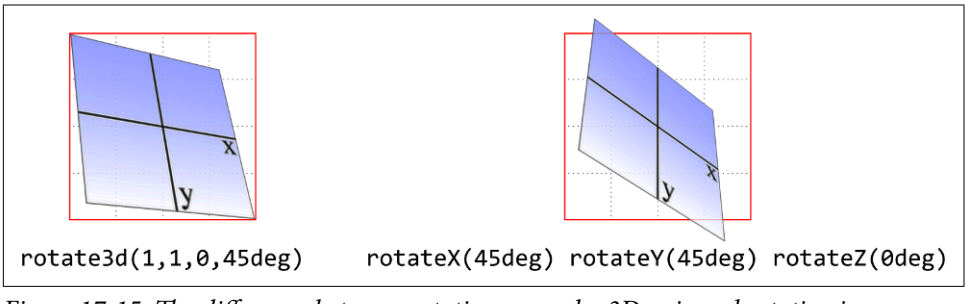


Figure 17-15. The difference between rotating around a 3D axis and rotating in sequence around three different axes

The rotate property

As with translations and scaling, CSS has a `rotate` property that allows you to rotate elements around various axes without having to use the `transform` property to do so. The value syntax to make that possible is a bit different, however.

rotate

Values	none | <*angle*> | [x | y | z | <*number*>{3}] && <*angle*>
Initial value	none
Applies to	Any transformable element
Percentages	Refer to the corresponding size of the bounding box
Computed value	As specified
Inherited	No
Animatable	As a transform

The valid values are divided into three mutually exclusive syntax options. The simplest is that the default value of `none` means no rotation is applied.

If you want to rotate around a single axis, it's easiest to give the axis identifier along with the angle you want to rotate. In the following code, each line contains two equivalent ways of rotating an element around a given axis:

```
transform: rotateX(45deg);    rotate: x 45deg;
transform: rotateY(33deg);    rotate: y 33deg;
transform: rotateZ(-45deg);   rotate: z -45deg;
transform: rotate(90deg);     rotate: 90deg;
```

The last line is similar to the handling of the `rotate()` function discussed earlier: a rotation with a single degree value is a 2D rotation on the *xy* plane. (See Figure 17-12 for a refresher.)

If you want to define a 3D vector as the axis of rotation, the value of `rotate` looks a little different. For example, suppose we want to rotate an element 45 degrees around the vector –0.95, 0.5, 1, as illustrated in Figure 17-14. Either of the following two declarations will have this effect:

```
transform: rotate3d(-0.95, 0.5, 1, 45deg);
rotate: -0.95 0.5 1 45deg;
```

If you want, you can use this pattern to rotate around the cardinal axes; that is, `rotate: z 23deg` and `rotate: 0 0 1 23deg` will have that same effect (as will `rotate: 23deg`). This can be useful when changing the vector of rotation via JavaScript, but is rarely useful in other cases.

Note that `transform` has a power that `rotate` cannot duplicate: the ability to chain rotations in sequence. For example, `transform: rotateZ(20deg) rotateY(30deg)` will first rotate the element 20 degrees around the z-axis, and then the result of that rotation is rotated around the y-axis. The `rotate` property can do only one or the other of these on its own. The only way to get the same result is to figure out the vector and angle that will leave the element in the same state as the `transform` operation did. The math to do that certainly exists, but is outside the scope of this book (although see "Matrix Functions" on page 861).

Individual Transform Property Order

When using the individual transform properties, the effects are always applied in the order `translate`, then `rotate`, then `scale`. The following two rules are functionally equivalent:

```
#mover {
        rotate: 30deg;
        scale: 1.5 1;
        translate: 10rem;}

#mover {
        transform: translate(10rem) rotate(30deg) scale(1.5, 1);
}
```

This matters because, for example, translating and then rotating is very different from rotating and then translating. If you need to have an element's transforms happen in an order other than transform-rotate-scale, use `transform` instead of the individual properties.

Skewing

When you *skew* an element, you slant it along one or both of the x- and y-axes. There is no z-axis or 3D skewing.

skewX(), skewY() functions

Value *<angle>*

In both cases, you supply an angle value, and the element is skewed to match that angle. It's much easier to show skewing rather than try to explain it in words, so Figure 17-16 shows skew examples along the x- and y-axes.

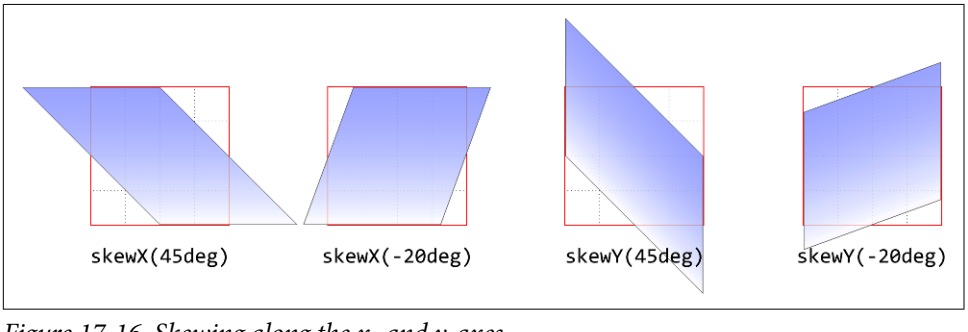


Figure 17-16. Skewing along the x- and y-axes

skew() function

Values *<angle>* [, *<angle>*]?

Using `skew(a,b)` is different from including `skewX(a)` with `skewY(b)`. The former specifies a 2D skew using the matrix operation `[ax,ay]`. Figure 17-17 shows examples of this matrix skewing and how they differ from double-skew transforms that look the same at first but aren't.



For a variety of reasons, including the way `skew(a,b)` is different from `skewX(a) skewY(b)`, the CSS specification explicitly discourages the use of `skew()`. You should avoid using it if at all possible; we document it here in case you find yourself coming across it in legacy code.

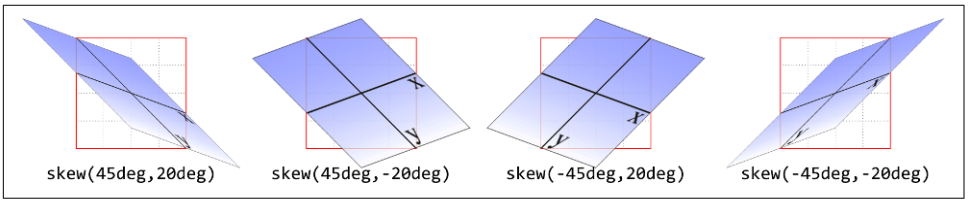


Figure 17-17. Skewed elements

If you supply two values, the x skew angle is always first, and the y skew angle comes second. If you leave out a y skew angle, it's treated as 0.



Unlike for translation, rotation, and scaling, CSS has no `skew` property as of late 2022, so any skewing has to be managed via the `transform` property.

Matrix Functions

If you're a particular fan of advanced math, or stale jokes derived from the Wachowski siblings' movies, the matrix functions will be your favorites. CSS has no `matrix` properties, to be clear.

matrix() function

Values *<number>* [, *<number>*]{5,5}

In the CSS transforms specification, we find the trenchant description of `matrix()` as a function that "specifies a 2D transformation in the form of a transformation matrix of the six values *a–f*."

First things first: a valid `matrix()` value is a list of six comma-separated numbers. No more, no less. The values can be positive or negative. Second, the value describes the final transformed state of the element, combining all of the other transform types (rotation, skewing, and so on) into a compact syntax. Third, very few people use this syntax to write code themselves, though it is often generated by drawing or animation software.

We're not going to go through the complicated process of doing the matrix math. For most readers, it would be an eye-watering wall of apparent gibberish; for the rest, it would be time wasted on familiar territory. You can certainly research the intricacies of matrix calculations online, and we encourage anyone with an interest to do so. We'll just look at the basics of syntax and usage in CSS.

Here's a brief rundown of how it works. Say you have this function applied to an element:

```
matrix(0.838671, 0.544639, -0.692519, 0.742636, 6.51212, 34.0381)
```

That's the CSS syntax used to describe this transformation matrix:

```
0.838671    -0.692519   0   6.51212
0.544639     0.742636   0   34.0381
0            0          1   0
0            0          0   1
```

Right. So what does that do? It has the result shown in Figure 17-18, which is exactly the same result as writing this:

```
rotate(33deg) translate(24px,25px) skewX(-10deg)
```

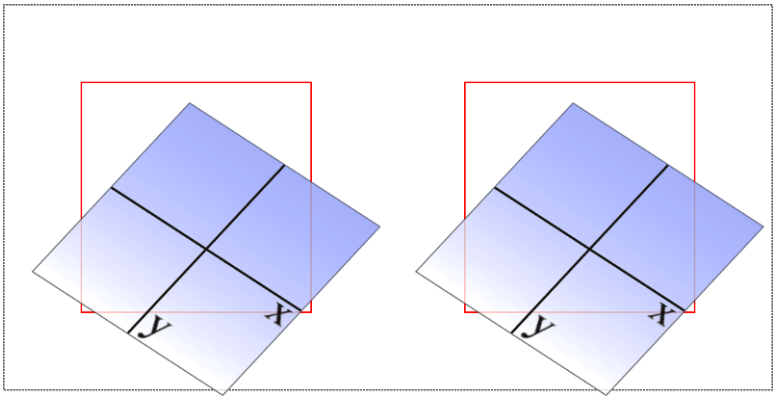


Figure 17-18. A matrix-transformed element and its functional equivalent

What this comes down to is that if you're familiar with or need to use matrix calculations, you can and should. Otherwise, you can chain much more human-readable transform functions together and get the element to the same end state.

Now, that was for plain old 2D transforms. What if you want to use a matrix to transform through three dimensions?

matrix3d() function

Values `<number>[,<number>]{15,15}`

Again, just for kicks, we'll savor the definition of `matrix3d()` from the CSS Transforms specification: "specifies a 3D transformation as a 4 × 4 homogeneous matrix of 16 values in column-major order." This means the parameter of `matrix3d()` *must* be a list of 16 comma-separated numbers, no more or less. Those numbers are arranged in a 4 × 4 grid in column order, so the first column of the matrix is formed by the first set of four numbers in the value, the second column by the second set of four numbers, the third column by the third set, and so on. Thus, you can take the following function,

```
matrix3d(
    0.838671, 0, -0.544639, 0.00108928,
    -0.14788, 1, 0.0960346, -0.000192069,
    0.544639, 0, 0.838671, -0.00167734,
    20.1281, 25, -13.0713, 1.02614)
```

and write it out as this matrix:

```
 0.838671   -0.14788        0.544639     20.1281
 0           1              0            25
-0.544639    0.0960346      0.838671    -13.0713
 0.00108928 -0.000192069   -0.00167734   1.02614
```

Both have an end state equivalent to the following, which is depicted in Figure 17-19.

```
perspective(500px) rotateY(33deg) translate(24px,25px) skewX(-10deg)
```

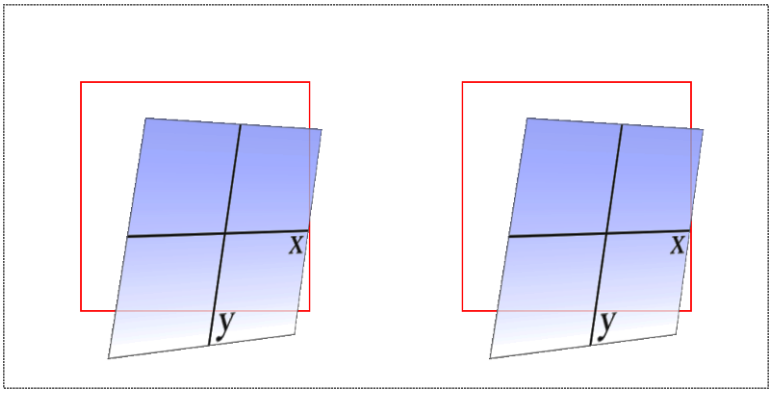


Figure 17-19. A `matrix3d()`-transformed element and its functional equivalent

A note on end-state equivalence

It's important to keep in mind that only the end states of a `matrix()` function, and of an equivalent chain of transform functions, can be considered identical. This is for the same reason discussed in "Element Rotation" on page 854: because a rotation angle of `393deg` will end with the same visible rotation as an angle of `33deg`. This matters if you are animating the transformation, since the former will cause the element to do a barrel roll in the animation, whereas the latter will not. The `matrix()` version of this end state won't

include the barrel roll, either. Instead, it will always use the shortest possible rotation to reach the end state.

To illustrate what this means, consider the following, a transform chain and its `matrix()` equivalent:

```
rotate(200deg) translate(24px,25px) skewX(-10deg)
matrix(-0.939693, -0.34202, 0.507713, -0.879385, -14.0021, -31.7008)
```

Note the rotation of 200 degrees. We naturally interpret this to mean a clockwise rotation of 200 degrees, which it is. If these two transforms are animated, however, they will act differently: the chained-functions version will indeed rotate 200 degrees clockwise, whereas the `matrix()` version will rotate 160 degrees counterclockwise. Both will end up in the same place but will get there in different ways.

Other differences can arise even when you might think they wouldn't. Once again, this is because a `matrix()` transformation will always take the shortest possible route to the end state, whereas a transform chain might not. (In fact, it probably doesn't.) Consider these apparently equivalent transforms:

```
rotate(160deg) translate(24px,25px) rotate(-30deg) translate(-100px)
matrix(-0.642788, 0.766044, -0.766044, -0.642788, 33.1756, -91.8883)
```

As ever, they end up in the same place. When animated, though, the elements will take different paths to reach that end state. They might not be obviously different at first glance, but the difference is still there.

None of this matters if you aren't animating the transformation, but it's an important distinction to make nevertheless, because you never know when you'll decide to start animating things. (Hopefully after reading Chapters 18 and 19!)

Setting Element Perspective

If you're transforming an element in 3D space, you most likely want it to have some perspective. *Perspective* gives the appearance of front-to-back depth, and you can vary the degree of perspective applied to an element.

perspective() function

Value *<length>*

It might seem a bit weird to specify perspective as a distance. After all, `perspective(200px)` seems odd when you can't really measure pixels along the z-axis. And yet, here we are. You supply a length, and the illusion of depth is constructed around that value.

Smaller numbers create a more extreme perspective, as though you are up close to the element. Higher numbers create a gentler perspective, as though viewing the element through a zoom lens from far away. *Really* high perspective values create an isometric effect, which looks the same as no perspective at all.

This makes a certain amount of sense. You can visualize perspective as a pyramid, with its apex point at the perspective origin (by default, the center of the untransformed element's position) and its base as the browser window that you're looking through. A shorter distance between apex and base will create a shallower pyramid, and thus a more extreme distortion. This is illustrated in Figure 17-20, with hypothetical pyramids representing 200-pixel, 800-pixel, and 2,000-pixel perspective distances.

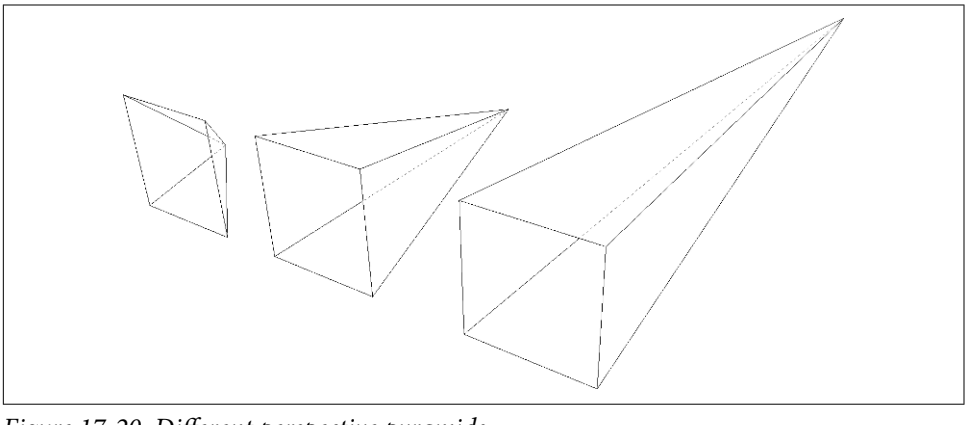


Figure 17-20. Different perspective pyramids

In documentation for Safari, Apple writes that perspective values below `300px` tend to be extremely distorted, values above `2000px` create "very mild" distortion, and values between `500px` and `1000px` create "moderate perspective." To illustrate this, Figure 17-21 shows a series of elements with the exact same rotation as displayed with varying perspective values.

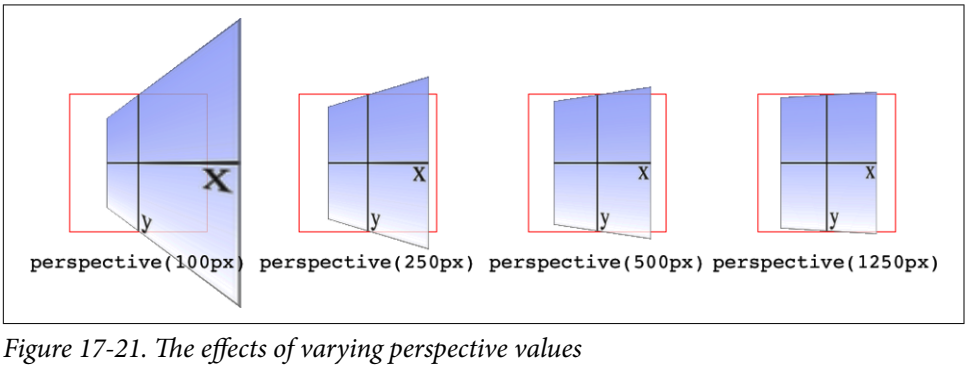


Figure 17-21. The effects of varying perspective values

Perspective values must always be positive, nonzero lengths. Any other value will cause the `perspective()` function to be ignored. Also note that its placement in the list of functions is important. If you look at the code for Figure 17-21, the `perspective()` function comes before the `rotateY()` function:

```
#ex1 {transform: perspective(100px) rotateY(-45deg);}
#ex2 {transform: perspective(250px) rotateY(-45deg);}
#ex3 {transform: perspective(500px) rotateY(-45deg);}
#ex4 {transform: perspective(1250px) rotateY(-45deg);}
```

If you were to reverse the order, the rotation would happen before the perspective is applied, so all four examples in Figure 17-21 would look exactly the same. So if you plan to apply a perspective value via the list of transform functions, make sure it comes first, or at the very least before any transforms that depend on it. This serves as a stark reminder that the order in which you write `transform` functions can be very important.

More Transform Properties

In addition to the base `transform` property and the standalone transform properties like `rotate`, a few related properties help to define how the elements transform the origin point of a transform, the perspective used for a "scene," and more.

Moving the Transform's Origin

So far, all of our transforms have shared one thing in common: we've used the precise center of the element as the *transform origin*. For example, when rotating the element, it rotated around its center, instead of, say, a corner. This is the default behavior, but with the property `transform-origin`, you can change it.

transform-origin

Values	[left | center | right | top | bottom | <*percentage*> | <*length*>] | [left | center | right | <*percentage*> | <*length*>] && [top | center | bottom | <*percentage*> | <*length*>]] <*length*>?
Initial value	50% 50% (0 0 in SVG)
Applies to	Any transformable element
Percentages	Refer to the size of the bounding box (see explanation)
Computed value	A percentage, except for length values, which are converted to an absolute length
Inherited	No
Animatable	<*length*>, <*percentage*>

The syntax definition looks really abstruse and confusing, but it's fairly simple in practice. With `transform-origin`, you supply two or three lengths or keywords to define the point around which transforms should be made: first the horizontal, then the vertical, and optionally a length along the z-axis. For the horizontal and vertical axes, you can use plain-English keywords like `top` and `right`, percentages, lengths, or a combination of keywords and percentage or length values. For the z-axis, you can't use plain-English keywords or percentages, but can use any length value. Pixels are by far the most common.

Length values are taken as a distance starting from the top-left corner of the element. Thus, `transform-origin: 5em 22px` will place the transform origin 5 ems in from the left side of the element, and 22 pixels down from the top of the element. Similarly, `transform-origin: 5em 22px -200px` will place it 5 ems over, 22 pixels down, and 200 pixels away (that is, 200 pixels behind the untransformed position of the element).

Percentages are calculated with respect to the corresponding axis and size of the element's bounding box, as offsets from the element's top-left corner. For example, `transform-origin: 67% 40%` will place the transform origin 67 percent of the width to the right of the element's left side, and 40 percent of the element's height down from the element's top side. Figure 17-22 illustrates a few origin calculations.

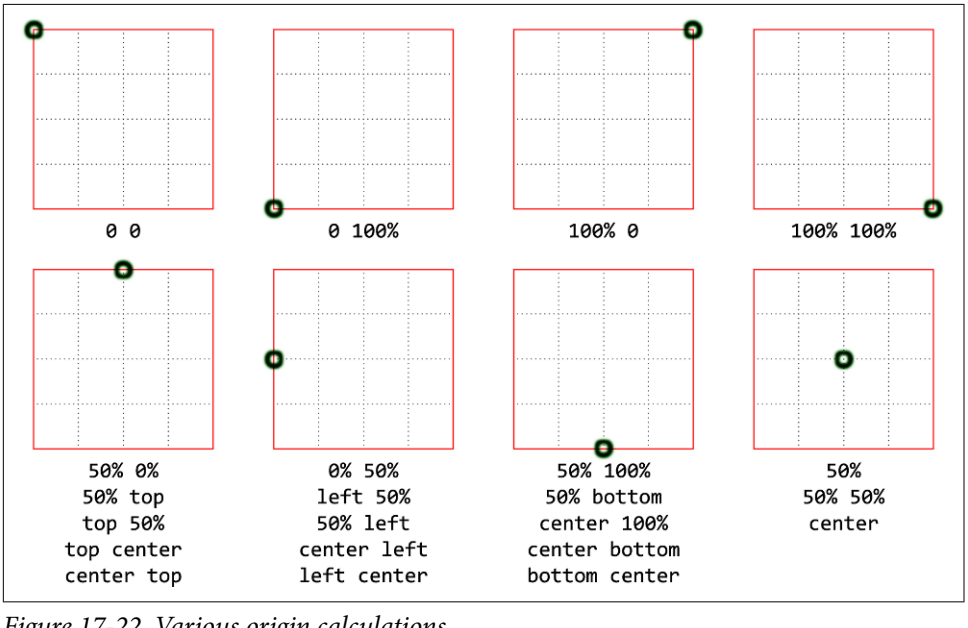


Figure 17-22. Various origin calculations

All right, so if you change the origin, what happens? The easiest way to visualize this is with 2D rotations. Suppose you rotate an element 45 degrees to the right. Its final placement will depend on its origin. Figure 17-23 illustrates the effects of several transform origins; in each case, the transform origin is marked with a circle.

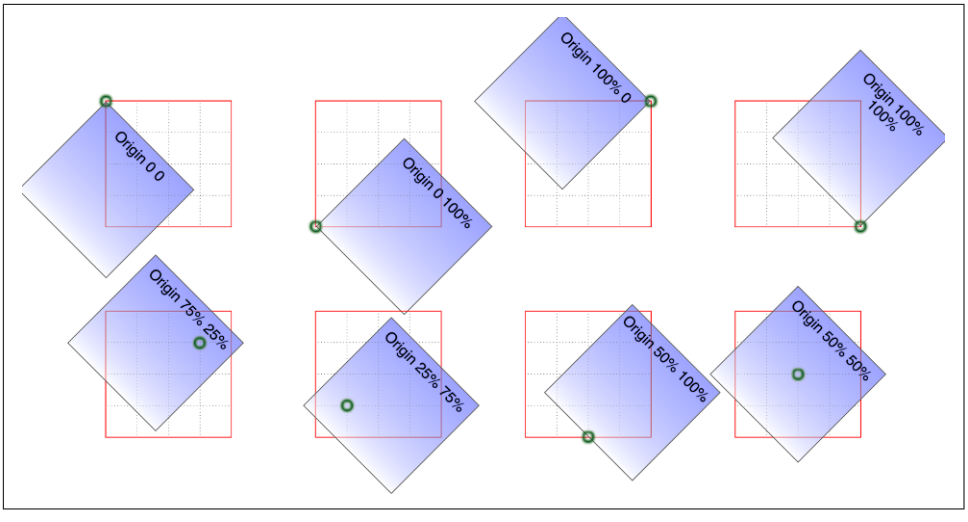


Figure 17-23. Rotational effects using various transform origins

The origin matters for other transform types, such as skews and scales. Scaling down an element with its origin in the center will pull in all sides equally, whereas scaling down an element with a bottom-right origin will cause it to shrink toward that corner. Similarly, skewing an element with respect to its center will result in the same shape as if it's skewed with respect to the top-right corner, but the placement of the shape will be different. Some examples are shown in Figure 17-24; again, each transform origin is marked with a circle.

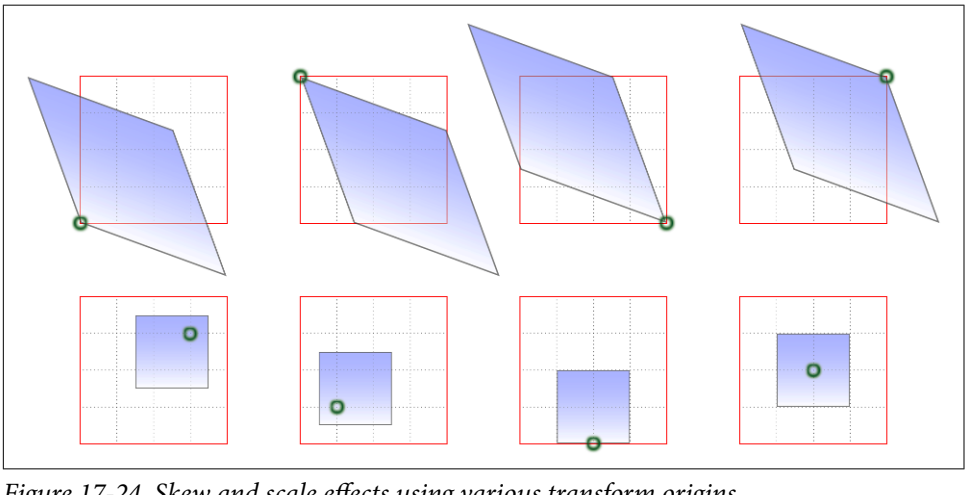


Figure 17-24. Skew and scale effects using various transform origins

The one transform type that isn't really affected by changing the transform origin is translation. If you move an element around with `translate()` or its cousins like `translateX()` and `translateY()`, or the `translate` property, the element is going to end up in the same

place regardless of where the transform's origin is located. If that's all the transforming you plan to do, setting the transform origin is irrelevant. If you ever do anything besides translating, though, the origin will matter. Use it wisely.

Choosing the Transform's Box

We wrote the previous section as though the transform origin is always calculated with respect to the outer border edge, and that is indeed the default in HTML, but not always in SVG. You can change this, at least in theory, with the property `transform-box`.

transform-box

Values	`border-box` | `content-box` | `fill-box` | `stroke-box` | `view-box`
Initial value	`view-box`
Applies to	Any transformable element
Computed value	As specified
Inherited	No
Animatable	No

Two of the values are directly related to CSS when styling HTML:

`border-box`
: Use the element's border box (defined by the outer border edge) as the reference box for transforms.

`content-box`
: Use the element's content box as the reference box for transforms.

The remaining three are designed for SVG purposes, though they can also apply in HTML contexts:

`fill-box`
: Use the element's object bounding box as the reference box.

`stroke-box`
: Use the element's stroke bounding box as the reference box.

`view-box`
: Use the element's nearest SVG viewport as the reference box.

Using `fill-box` in an SVG context causes transforms to be performed on the element in question, as we would expect from HTML. The default `view-box`, on the other hand, causes all transforms to be calculated with respect to the origin of the coordinate system

established by the SVG `viewBox` attribute. The difference is illustrated in Figure 17-25, which is the result of the following SVG file and the CSS it contains:

```
<svg xmlns="http://www.w3.org/2000/svg"
     width="500" height="200"
     fill="none" stroke="#000">
  <defs>
    <style>
      g rect {transform-origin: 0 0; transform: rotate(20deg);}
      g rect:nth-child(1) {transform-box: view-box;}
      g rect:nth-child(2) {transform-box: fill-box;}
    </style>
  </defs>
  <rect width="100%" height="100%" stroke-dasharray="4 3" />
  <rect x="100" y="50" width="100" height="100" />
  <rect x="300" y="50" width="100" height="100" />
  <g stroke-width="3" fill="#FFF8">
    <rect x="100" y="50" width="100" height="100" />
    <rect x="300" y="50" width="100" height="100" />
  </g>
</svg>
```

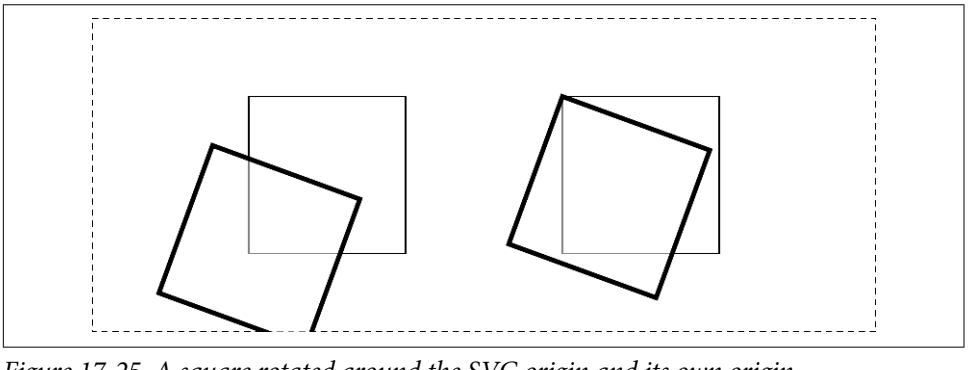


Figure 17-25. A square rotated around the SVG origin and its own origin

The first square, on the left, is rotated 20 degrees from its starting point, with the center of rotation as the top left of the entire SVG file (the top-left corner of the dashed-line box). This is because the value of `transform-box` for this square is `view-box`. The second square has a `transform-box` of `fill-box`, so it uses the top left of its own fill box—what in HTML we would call the background area—as the center of rotation.

Choosing a 3D Style

If you're setting elements to be transformed through three dimensions—using, say, `translate3d()` or `rotateY()`—you probably expect that the elements will be presented as though they're in a 3D space. The `transform-style` property helps bring that to life.

transform-style

Values	`flat` | `preserve-3d`
Initial value	`flat`
Applies to	Any transformable element
Computed value	As specified
Inherited	No
Animatable	No

Suppose you want to move an element "closer" to your eye, and then tilt it away a bit, with a moderate amount of perspective. You might use something like this rule:

```
div#inner {transform: perspective(750px) translateZ(60px) rotateX(45deg);}

<div id="outer">
outer
<div id="inner">inner</div>
</div>
```

So you do that and get the result shown in Figure 17-26—more or less what you might expect.

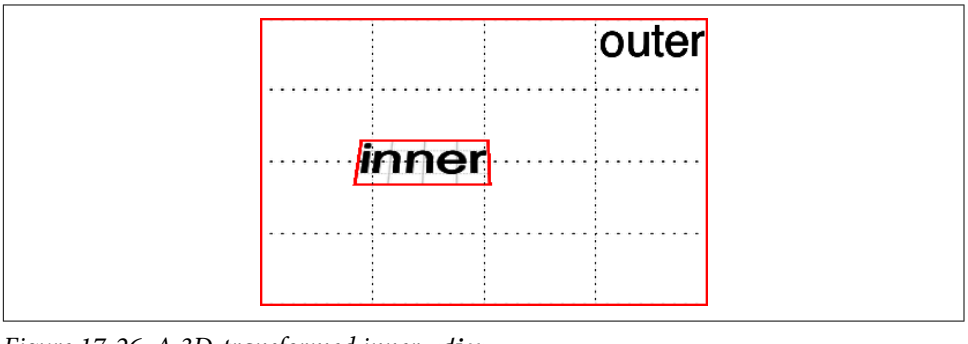


Figure 17-26. A 3D-transformed inner `<div>`

But then you decide to rotate the outer `<div>` to one side, and suddenly nothing makes sense anymore. The inner `<div>` isn't where you envisioned it. In fact, it just looks like a picture pasted to the front of the outer `<div>`.

Well, that's exactly what it is, because the default value of `transform-style` is `flat`. The inner `div` got drawn in its moved-forward, tilted-back state, and that was applied to the front of the outer `<div>` as if it were an image. So when you rotated the outer `<div>`, as shown in Figure 17-27, the flat picture rotated right along with it:

```
div#outer {transform: perspective(750px) rotateY(60deg) rotateX(-20deg);}
div#inner {transform: perspective(750px) translateZ(60px) rotateX(45deg);}
```

Change the value to `preserve-3d`, however, and the result is very different. The inner `div` will be drawn as a full 3D object with respect to its parent outer `<div>`, floating in space nearby, and *not* as a picture pasted on the front of the outer `<div>`. You can see the results of this change in Figure 17-27:

```
div#outer {transform: perspective(750px) rotateY(60deg) rotateX(-20deg);
    transform-style: preserve-3d;}
div#inner {transform: perspective(750px) translateZ(60px) rotateX(45deg);}
```

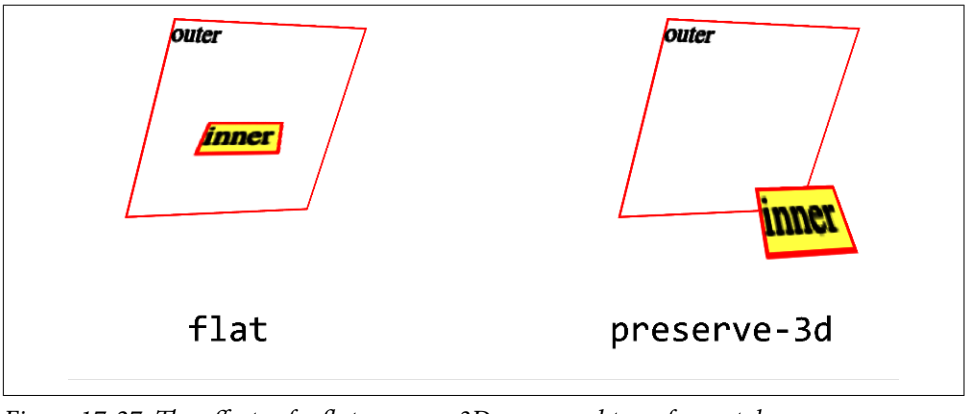


Figure 17-27. The effects of a flat versus a 3D-preserved transform style

One important aspect of `transform-style` is that it can be overridden by other properties. The reason is that some values of these other properties require a flattened presentation of an element and its children in order to work. In such cases, the value of `transform-style` is forced to be `flat`, regardless of what you may have declared.

So, to avoid this overriding behavior, make sure the following properties are set to the listed values on any 3D-transformed container elements that also have 3D-transformed children:

- `overflow: visible`
- `filter: none`
- `clip: auto`
- `clip-path: none`
- `mask-image: none`
- `mask-border-source: none`
- `mix-blend-mode: normal`
- `isolation: auto`

Those are all the default values for those properties, so as long as you don't try to change any of them for your preserved 3D elements, you're fine! But if you find that editing some CSS suddenly flattens out your lovely 3D transforms, one of these properties might be the culprit.

Changing Perspective

Two properties are used to define the way perspective is handled: one to define the perspective distance, as with the `perspective()` function discussed in an earlier section; and another to define the perspective's origin point.

Defining a group perspective

First, let's consider the property `perspective`, which accepts a length that defines the depth of the perspective pyramid. At first glance, it looks just like the `perspective()` function discussed earlier, but some critical differences exist.

perspective

Values	none | <*length*>
Initial value	none
Applies to	Any transformable element
Computed value	The absolute length, or else none
Inherited	No
Animatable	Yes

As a quick example, if you want to create a very deep perspective, one mimicking the results you'd get from a zoom lens, you might declare something like `perspective: 2500px`. For a shallow depth, one that mimics a close-up fish-eye lens effect, you might declare `perspective: 200px`.

So how does this differ from the `perspective()` function? When you use `perspective()`, you're defining the perspective effect for the element that is given that function. So if you write `transform: perspective(800px) rotateY(-50grad);`, you're applying that perspective to each element that has the rule applied.

With the `perspective` property, on the other hand, you're creating a shared perspective for all the child elements of the element that received the property. Here's an illustration of the difference, as shown in Figure 17-28:

```
div {transform-style: preserve-3d; border: 1px solid gray; width: 660px;}
img {margin: 10px;}
#func {perspective: none;}
#func img {transform: perspective(800px) rotateX(-50grad);}
```

```
#prop {perspective: 800px;}
#prop img {transform: rotateX(-50grad);}
```

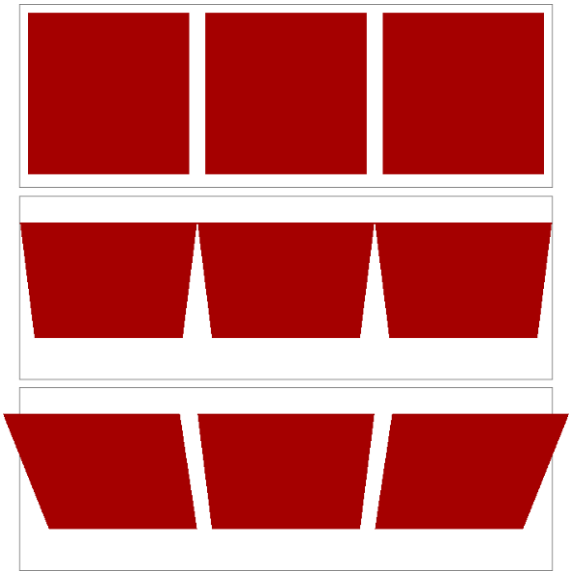


Figure 17-28. No perspective, individual `perspective()`, and shared `perspective`, respectively

In Figure 17-28, we first see a line of images that haven't been transformed. In the second line, each image has been rotated 50 gradians (equivalent to 45 degrees) toward us, but each one within its own individual perspective.

In the third line of images, none has an individual perspective. Instead, all are drawn within the perspective defined by `perspective: 800px;` that's been set on the `<div>` that contains them. Since they all operate within a shared perspective, they look "correct"—that is, as we would expect if we had three physical pictures mounted on a clear sheet of glass and rotated that toward us around its center horizontal axis.

This is the critical difference between `perspective`, the property, and `perspective()`, the function. The former creates a 3D space shared by all its children. The latter affects only the element to which it's applied. Another difference is that the effect of the `perspective()` function is different depending on when it is called in the chain of transforms. The `perspective` property is always applied before all other transforms, which is what you normally want to create a 3D effect.

In most cases, you're going to use the `perspective` property instead of the `perspective()` function. In fact, container `<div>`s (or other elements) are a common feature of 3D transforms—the way they used to be for page layout—largely to establish a shared perspective. In the previous example, the `<div id="two">` is there solely to serve

as a perspective container, so to speak. On the other hand, we couldn't have done what we did without it.

Moving the perspective's origin

When transforming elements in three dimensions, a perspective will be used. (See `transform-style` and `perspective` in previous sections.) That perspective will have an origin, which is also known as the *vanishing point*, and you can change its location with the `perspective-origin` property.

perspective-origin

Values	[`left` | `center` | `right` | `top` | `bottom` | <*percentage*> | <*length*>] | [`left` | `center` | `right` | <*percentage*> | <*length*>] && [`top` | `center` | `bottom` | <*percentage*> | <*length*>]]
Initial value	50% 50%
Applies to	Any transformable element
Percentages	Refer to the size of the bounding box (see explanation)
Computed value	A percentage, except for length values, which are converted to an absolute length
Inherited	No
Animatable	<*length*>, <*percentage*>

With `perspective-origin`, you define the point on which sight lines converge, and as with `perspective`, that point is defined relative to a parent container.

As with most 3D transform properties, this is more easily demonstrated than described. Consider the following CSS and markup, illustrated in Figure 17-29:

```
#container {perspective: 850px; perspective-origin: 50% 0%;}
#ruler {height: 50px; background: #DED url(tick.gif) repeat-x;
    rotate: x 60deg;
    transform-origin: 50% 100%;}

<div id="container">
    <div id="ruler"></div>
</div>
```

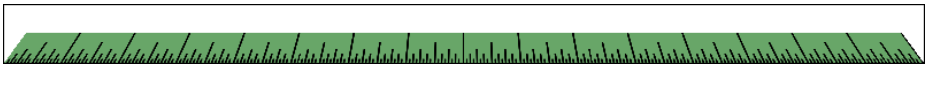


Figure 17-29. A basic "ruler"

We have a repeated background image of tick marks on a ruler, with the `<div>` that contains them tilted away from us by 60 degrees. All the lines point at a common vanishing

point, the top center of the container `<div>` (because of the `50% 0%` value for `perspective-origin`).

Now consider that same setup with various perspective origins (Figure 17-30).

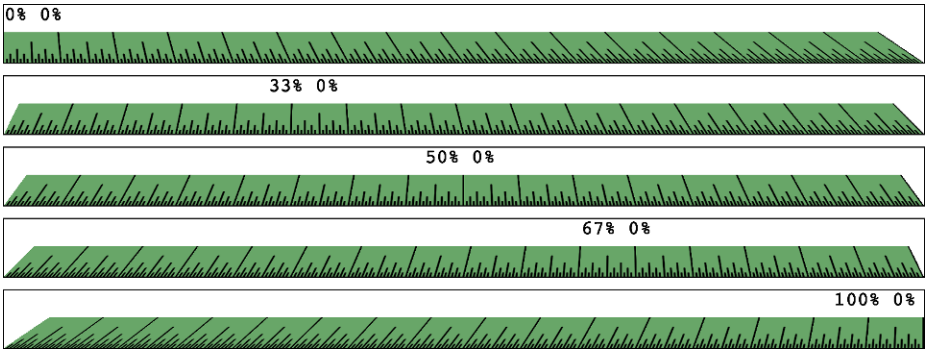


Figure 17-30. A basic "ruler" with different perspective origins

As you can see, moving the perspective origin changes the rendering of the 3D-transformed element. Note that these have an effect only because we supplied a value for `perspective`. If the value of `perspective` is ever the default `none`, any value given for `perspective-origin` will be ignored. That makes sense, since you can't have a perspective origin when there's no perspective at all!

Dealing with Backfaces

Over all the years you've been laying out elements, you've probably never thought, "What would it look like if we could see the back side of the element?" With 3D transforms, if there comes a day when you *do* see the back side of an element, CSS has you covered. What happens is determined by the property `backface-visibility`.

backface-visibility

Values	`visible` | `hidden`
Initial value	`visible`
Applies to	Any transformable element
Computed value	As specified
Inherited	No
Animatable	No

Unlike many of the other properties and functions we've already talked about, this one is pretty uncomplicated. All it does is determine whether the back side of an element is rendered when it's facing toward the viewer, or not. That's it.

So let's say you flip over two elements, one with `backface-visibility` set to the default value of `visible` and the other set to `hidden`. You get the result shown in Figure 17-31:

```
span {border: 1px solid red; display: inline-block;}
img {vertical-align: bottom;}
img.flip {rotate: x 180deg; display: inline-block;}
img#show {backface-visibility: visible;}
img#hide {backface-visibility: hidden;}

<span><img src="salmon.gif" alt="salmon"></span>
<span><img src="salmon.gif" class="flip" id="show" alt="salmon"></span>
<span><img src="salmon.gif" class="flip" id="hide" alt="salmon"></span>
```

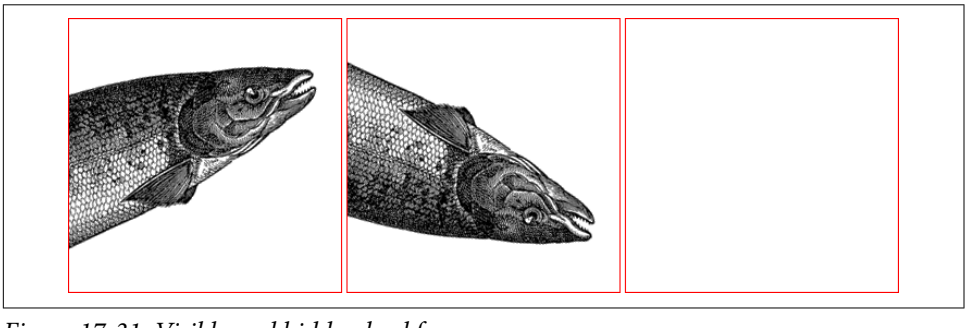


Figure 17-31. Visible and hidden backfaces

As you can see, the first image is unchanged. The second is flipped over its x-axis, so we see it from the back. The third has also been flipped, but we can't see it at all because its backface has been hidden.

This property can come in handy in several situations. In the simplest, you have two elements that represent the two sides of a UI element that flips over—say, a search area with preference settings on its back, or a photo with some information on the back. Let's take the latter case. The CSS and markup might look something like this:

```
section {position: relative;}
img, div {position: absolute; top: 0; left: 0; backface-visibility: hidden;}
div {rotate: y 180deg;}
section:hover {rotate: y 180deg; transform-style: preserve-3d;}

<section>
    <img src="photo.jpg" alt="">
    <div class="info">(…info goes here…)</div>
</section>
```

(This would be a lot more interesting with an animated rotation, causing the card to flip over in 3D space.)

A variant of this example uses the same markup, but slightly different CSS to show the image's backface when it's flipped over. This is probably more what was intended, since it makes information look like it's literally written on the back of the image. It leads to the end result shown in Figure 17-32:

```
section {position: relative;}
img, div {position: absolute; top: 0; left: 0;}
div {rotate: y 180deg; backface-visibility: hidden;
    background: rgba(255,255,255,0.85);}
section:hover {rotate: y 180deg; transform-style: preserve-3d;}
```

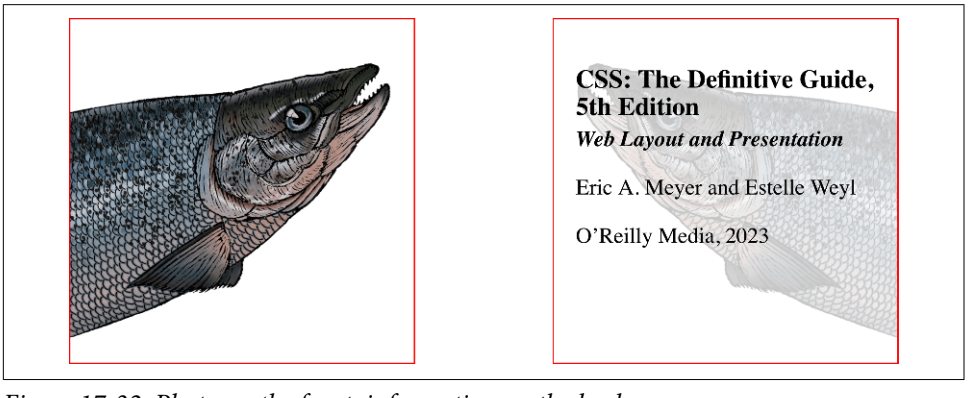


Figure 17-32. Photo on the front, information on the back

All we had to do to make that happen was shift `backface-visibility: hidden` to the `<div>` instead of applying it to both the `<img>` and the `<div>`. Thus, the `<div>`'s backface is hidden when it's flipped over, but that of the image is not. (Well, that and use a semitransparent background so we could see both the text and the flipped image underneath it.)

Summary

With the ability to transform elements in two- and three-dimensional space, CSS transforms provide a great deal of power to designers. From creating interesting combinations of 2D transforms, to creating a fully 3D-acting interface, transforms open up a great deal of new territory in the design space. Some dependencies exist between properties, which is something that not every CSS author will find natural at first, but they become second nature with practice.

One of the things authors often do with transforms is animate them, so that a card flips over, an element scales and rotates smoothly, and so on. In the next two chapters, we'll get into the details of how those transitions and animations are defined.

CHAPTER 18
Transitions

CSS transitions allow us to animate CSS properties from an original value to a new value over time. These changes *transition* an element from one state to another, in response to a change. This usually involves a user interaction but can also be due to a scripted change of class, ID, or other state.

Normally, when a CSS property value changes—when a *style change event* occurs—the change is instantaneous. The new property value replaces the old property in the milliseconds it takes to repaint the page (or to reflow and repaint, when necessary). Most value changes seem instantaneous, taking fewer than 16 milliseconds to render. Even if the changes take longer than that (like when a large image is replaced with one that isn't prefetched—which isn't a transition, just poor performance), it is still a single step from one value to the next. For example, when changing a background color on mouse hover, the background immediately changes from one color to the other, with no gradual transition.

CSS Transitions

CSS transitions provide a way to control how a property changes from one value to the next over a period of time. Thus, we can make the property values change gradually, creating (hopefully) pleasant and unobtrusive effects. For example:

```
button {color: magenta;
    transition: color 200ms ease-in 50ms;
}
button:hover {color: rebeccapurple;
    transition: color 200ms ease-out 50ms;
}
```

In this example, instead of instantaneously changing a button's `color` value on hover, that `transition` property means the button's `color` will gradually fade from `magenta` to `rebeccapurple` over 200 milliseconds, even adding a 50-millisecond delay before starting the transition.

In the unlikely event that a browser doesn't support CSS transition properties, the change is immediate instead of gradual, which is completely fine. If a given property or some property values aren't animatable, again, the change will be immediate instead of gradual.



When we say *animatable*, we mean any properties that can be animated, whether through transitions or animations (the subject of the next chapter, Chapter 19). The property definition boxes throughout the book indicate whether a given property is animatable.

Often you will want instantaneous value changes. For example, link colors usually change instantly on hover or focus, informing sighted users that an interaction is occurring and that the focused content is a link. Similarly, options in an autocomplete listbox shouldn't fade in: you want the options to appear instantly, rather than fade in more slowly than the user types. Instantaneous value changes are often the best user experience.

At other times, you'll want a property's value to change more gradually, bringing attention to what is occurring. For example, you may want to make a card game more realistic by taking 200 milliseconds to animate the flipping of a card, as the user may not realize what happened if there is no animation. ▶



Look for the Play symbol ▶ to know when an online example is available. All of the examples in this chapter can be found at *https://meyerweb.github.io/csstdg5figs/18-transitions*.

As another example, you may want some drop-down menus to expand or become visible over 200 milliseconds (instead of instantly, which may be jarring). With transitions, you can make a drop-down menu appear slowly. In Figure 18-1 ▶, we transition the submenu's height by making a scale transform. This is a common use for CSS transitions, which we will also explore later in this chapter.



Especially rapid transitions, particularly those that move over large distances or take up major parts of a page, *can potentially lead to seizures in some users.* To reduce or eliminate this risk, use the `prefers-reduced-motion` media query (see Chapter 21). Always keep these concerns in mind, and ensure the accessibility of your designs to people with epilepsy and other seizure disorders.

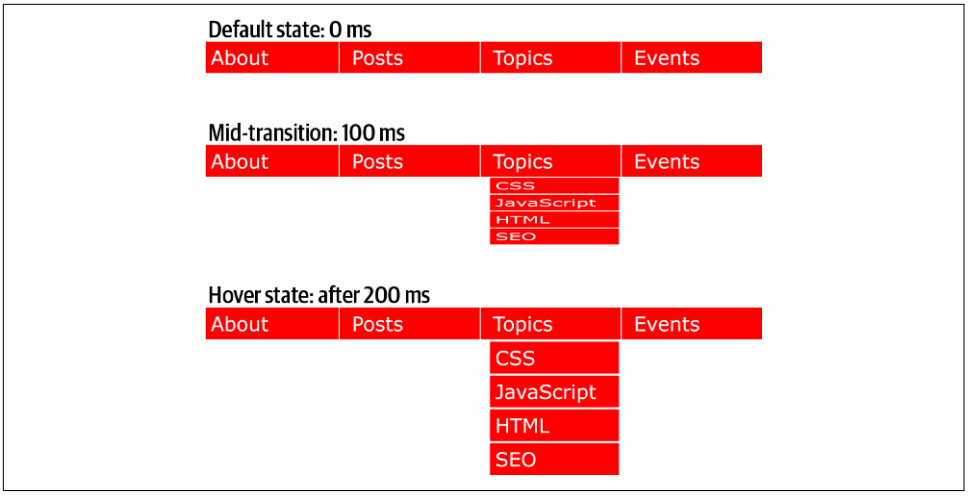


Figure 18-1. Initial transition, midtransition, and final state

Transition Properties

In CSS, transitions are written using four transition properties: `transition-property`, `transition-duration`, `transition-timing-function`, and `transition-delay`, along with the `transition` property as a shorthand for those four.

To create the drop-down navigation in Figure 18-1, we used all four CSS transition properties, as well as some transform properties defining the beginning and end states of the transition. The following code defines the transition for that example:

```
nav li ul {
    transition-property: transform;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
    transition-delay: 50ms;
    transform: scale(1, 0);
    transform-origin: top center;
}
nav li:is(:hover, :focus) ul {
    transform: scale(1, 1);
}
```

Although we are using the `:hover` and `:focus` states for the style change event in this example, you can transition properties in other scenarios too. For example, you might add or remove a class, or otherwise change the state—say, by changing an input from `:invalid` to `:valid` or from `:checked` to `:not(:checked)`. Or you might append a table row at the end of a zebra-striped table or a list item at the end of a list with styles based on `:nth-last-of-type` selectors.

In Figure 18-1, the initial state of the nested lists is `transform: scale(1, 0)` with a `transform-origin: top center`. The final state is `transform: scale(1, 1)`, while the `transform-origin` remains the same. (For more information on transform properties, see Chapter 17.)

In this example, the transition properties define a transition on the `transform` property: when the new `transform` value is set on `hover`, the nested unordered list scales to its original, default size, changing smoothly between the old value of `transform: scale(1, 0)` and the new value of `transform: scale(1, 1)`, all over a period of 200 milliseconds. This transition starts after a 50-millisecond delay, and *eases in*, which means it proceeds slowly at first, then picks up speed as it progresses.

Whenever an animatable target property changes, if a transition is set on that property, the browser will apply a transition to make the change gradual.

Note that all the transition properties were set for the default unhovered/unfocused state of the `<ul>` elements. These states were used to change only the transform, not the transition. There's a very good reason for this: it means that the menus not only will slide open when the state change happens, but also will slide closed when the hover or focus state ends.

Imagine that the transition properties were applied to the interaction states instead, like this:

```
nav li ul {
    transform: scale(1, 0);
    transform-origin: top center;
}
nav li:is(:hover, :focus) ul {
    transition-property: transform;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
    transition-delay: 50ms;
    transform: scale(1, 1);
}
```

That would mean that when *not* hovered or focused, the element would have default transition values—which is to say, no transitions or instantaneous transitions. The menus in our previous example would slide open, but instantly disappear when the interaction state ends—because no longer being in an interactive state, the transition properties would no longer apply!

Maybe you want exactly this effect: slide smoothly open but instantly disappear. If so, then apply the transitions as shown in the previous example. Otherwise, apply them to the element in the default state directly so that the transitions will apply as the interaction state is both entered and exited. When the state change is exited, the transition timing is reversed. You can override this default reverse transition by declaring different transitions in both the initial and changed states.

By *initial state*, we mean a state that matches the element at page load time. It could mean a content-editable element that could get `:focus`, as in the following:

```
/* selector that matches elements all the time */
p[contenteditable] {
    background-color: background-color: rgb(0 0 0 / 0);
}
/* selector that matches elements some of the time */
p[contenteditable]:focus {
    /* overriding declaration */
    background-color: background-color: rgb(0 0 0 / 0.1);
}
```

In this example, the fully transparent background is always the initial state, changing only when the user gives the element focus. This is what we mean when we say *initial* or *default* value throughout this chapter. The transition properties included in the selector that matches the element all the time will impact that element whenever the state changes, including from the initial state to the changed state (being focused, in the preceding example).

An initial state could also be a temporary state that may change, such as a `:checked` checkbox or a `:valid` form control, or even a class that gets toggled on and off:

```
/* selector that matches elements some of the time */
input:valid {
    border-color: green;
}
/* selector that matches elements some of the time,
   when the prior selector does NOT match. */
input:invalid {
    border-color: red;
}
/* selector that matches elements some of the time,
   whether the input is valid or invalid */
input:focus {
    /* alternative declaration */
    border-color: yellow;
}
```

In this example, either the `:valid` or `:invalid` selector can match any given element, but never both. The `:focus` selector, as shown in Figure 18-2, matches whenever an input has focus, regardless of whether the input is matching the `:valid` or `:invalid` selector simultaneously.

In this case, when we refer to the initial state, we are referring to the original value, which could be either `:valid` or `:invalid`. The changed state for a given element is the opposite of the initial `:valid` or `:invalid` state.

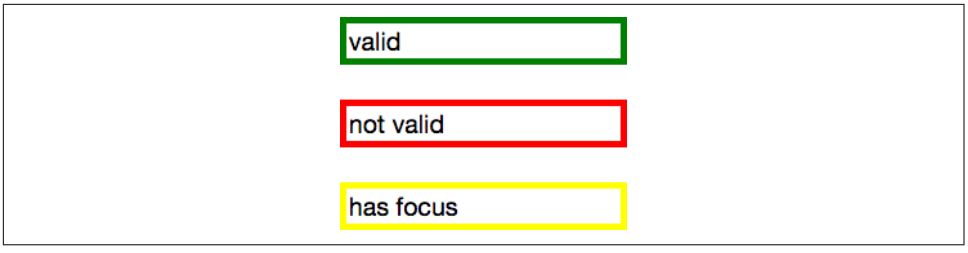


Figure 18-2. The input's appearance in the valid, invalid, and focused states

Remember, you can apply different transition values to the initial and changed states, but you always want to apply the value used when you *enter* a given state. Take the following code as an example, where the transitions are set up to have menus slide open over 2 seconds but close in just 200 milliseconds:

```
nav li ul {
    transition-property: transform;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
    transition-delay: 50ms;
    transform: scale(1, 0);
    transform-origin: top center;
}
nav li:is(:hover, :focus) ul {
    transition-property: transform;
    transition-duration: 2s;
    transition-timing-function: linear;
    transition-delay: 1s;
    transform: scale(1, 1);
}
```

This provides a horrible user experience, but it illustrates the point. ▶ When hovered or focused, the opening of the navigation takes a full 2 seconds. When closing, it quickly closes over 0.2 seconds. The transition properties in the changed state are in force when a list item is hovered or focused. Thus, the `transition-duration: 2s` defined for these states takes effect. When a menu is no longer hovered or focused, it returns to the default scaled-down state, and the transition properties of the initial state—the `nav li ul` condition—are used, causing the menu to take 200 milliseconds to close.

Look more closely at the example, specifically the default transition styles. When the user stops hovering over or focusing on the parent navigational element or the child drop-down menu, the drop-down menu delays 50 milliseconds before starting the `200ms` transition to close. This is actually a decent user experience style, because it give users a chance (however brief) to get the mouse pointer or focused ring back on a menu before it starts closing.

While the four transition properties can be declared separately, you will probably always use the shorthand. We'll take a look at the four properties individually first so you have a good understanding of what each one does.

Limiting Transition Effects by Property

The `transition-property` property specifies the names of the CSS properties you want to transition. This allows you to limit the transition to only certain properties, while having other properties change instantaneously. And, yes, it's weird to say "the `transition-property` property."

transition-property

Values	none | [all | <*property-name*>]#
Initial value	all
Applies to	All elements and :before and :after pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

The value of `transition-property` is a comma-separated list of properties; the keyword `none` if you want no properties transitioned; or the default `all`, which means "transition all the animatable properties." You can also include the keyword `all` within a comma-separated list of properties.

If you include `all` as the only keyword—or default to `all`—all animatable properties will transition in unison. Let's say you want to change a box's appearance on hover:

```
div {
    color: #ff0000;
    border: 1px solid #00ff00;
    border-radius: 0;
    transform: scale(1) rotate(0deg);
    opacity: 1;
    box-shadow: 3px 3px rgb(0 0 0 / 0.1);
    width: 50px;
    padding: 100px;
}
div:hover {
    color: #000000;
    border: 5px dashed #000000;
    border-radius: 50%;
    transform: scale(2) rotate(-10deg);
    opacity: 0.5;
    box-shadow: -3px -3px rgb(255 0 0 / 0.5);
    width: 100px;
    padding: 20px;
}
```

When the mouse pointer hovers over the `<div>`, every property that has a different value in the initial state versus the hovered (changed) state will change to the hover-state values. The `transition-property` property is used to define which of those properties are animated over time (versus those that change instantly, without animating). All the properties change from the default value to the hovered value on `hover`, but only the animatable properties included in the `transition-property` will change over the transition's duration. Nonanimatable properties like `border-style` change from one value to the next instantly.

If `all` is the only value or the last value in the comma-separated value for `transition-property`, all the animatable properties will transition in unison. Otherwise, provide a comma-separated list of properties to be affected by the transition properties.

Thus, if we want to transition all the properties, the following statements are almost equivalent:

```
div {
    color: #ff0000;
    border: 1px solid #00ff00;
    border-radius: 0;
    opacity: 1;
    width: 50px;
    padding: 100px;
    transition-property: color, border, border-radius, opacity,
        width, padding;
    transition-duration: 1s;
}
div {
    color: #ff0000;
    border: 1px solid #00ff00;
    border-radius: 0;
    opacity: 1;
    width: 50px;
    padding: 100px;
    transition-property: all;
    transition-duration: 1s;
}
```

Both `transition-property` property declarations will transition all the properties listed —but the former will transition only the six properties that may change.

The `transition-property: all` in the latter rule ensures that *all* animatable property values that would change based on any style change event—no matter which CSS rule block includes the changed property value—transitions over 1 second. The transition applies to all animatable properties applied to all the elements matched by the selector, not just the properties declared in the same style block as the `all`.

In this case, the first version limits the transition to only the six properties listed, but enables us to provide more control over how each property will transition. Declaring the properties individually lets us provide different speeds, delays, and/or durations to each property's transition:

```
div {
    color: #ff0000;
    border: 1px solid #0f0;
    border-radius: 0;
    opacity: 1;
    width: 50px;
    padding: 100px;
}
.foo {
    color: #00ff00;
    transition-property: color, border, border-radius, opacity,
        width, padding;
    transition-duration: 1s;
}
```

```
<div class="foo">Hello</div>
```

If you want to define the transitions for each property separately, write them all out, separating each of the properties with a comma. If you want to animate almost all the properties with the same duration, delay, and pace, with a few exceptions, you can use a combination of `all` and the individual properties you want to transition at different times, speeds, or paces. Just make sure to use `all` as the first value, because any properties listed before the `all` will be included in the `all`, overriding any other transition property values you intended to apply to those now overridden properties:

```
div {
    color: #f00;
    border: 1px solid #00ff00;
    border-radius: 0;
    opacity: 1;
    width: 50px;
    padding: 100px;
    transition-property: all, border-radius, opacity;
    transition-duration: 1s, 2s, 3s;
}
```

The `all` part of the comma-separated value includes all the properties listed in the example, as well as all the inherited CSS properties, and all the properties defined in any other CSS rule block matching or inherited by the element.

In the preceding example, all the properties getting new values will transition at the same duration, delay, and timing function, with the exception of `border-radius` and `opacity`, which we've explicitly included separately. Because we included them as part of a comma-separated list after the `all`, we can transition them at the same time, delay, and timing function as all the other properties, or we can provide different times, delays, and timing functions for these two properties. In this case, we transition all the properties over 1 second, except for `border-radius` and `opacity`, which we transition over 2 seconds and 3 seconds, respectively. (The `transition-duration` property is covered in an upcoming section.)

Suppressing transitions via property limits

While transitioning over time doesn't happen by default, if you do include a CSS transition and want to override that transition in a particular scenario, you can set `transition-property: none` to override the entire transition and ensure that no properties are transitioned.

The `none` keyword can be used as only a unique value of the property—you can't include it as part of a comma-separated list of properties. If you want to override the transition of a limited set of properties, you will have to list all of the properties you still want to transition. You can't use the `transition-property` property to exclude properties; rather, you can use that property only to include them.



Another method is to set the delay and duration of the property to `0s`. That way, it will appear instantaneously, as if no CSS transition is being applied to it.

Transition events

The TransitionEvent Interface provides for four transition-related events: `transition start`, `transitionrun`, `transitionend`, and `transitioncancel`. We'll concentrate on `transitionend`, as it's the one that can be triggered multiple times by a single piece of CSS.

A `transitionend` event is fired at the end of every transition, in either direction, for every property that is transitioned over any amount of time *or* after any delay. This happens whether the property is declared individually or is part of the `all` declaration. Some seemingly simple property declarations will use several `transitionend` events, as every animatable property within a shorthand property gets its own `transitionend` event. Consider the following:

```
div {
    color: #f00;
    border: 1px solid #00ff00;
    border-radius: 0;
    opacity: 1;
    width: 50px;
    padding: 100px;
    transition-property: all, border-radius, opacity;
    transition-duration: 1s, 2s, 3s;
}
```

When the transitions conclude, well over six `transitionend` events will have occurred. For example, the `border-radius` transition alone produces four `transitionend` events, one each for the following:

- `border-bottom-left-radius`
- `border-bottom-right-radius`
- `border-top-right-radius`
- `border-top-left-radius`

The `padding` property is also shorthand for four longhand properties:

- `padding-top`
- `padding-right`
- `padding-bottom`
- `padding-left`

The `border` shorthand property produces eight `transitionend` events: four values for the four properties represented by the `border-width` shorthand, and four for the properties represented by `border-color`:

- `border-left-width`
- `border-right-width`
- `border-top-width`
- `border-bottom-width`
- `border-top-color`
- `border-left-color`
- `border-right-color`
- `border-bottom-color`

The `border-style` properties have no `transitionend` events, however, as `border-style` is not an animatable property.

There will be 19 `transitionend` events in the scenario where six specific properties—`color`, `border`, `border-radius`, `opacity`, `width`, and `padding`—are listed, as those six include several shorthand properties. In the case of `all`, there will be at least 19 `transitionend` events: one for each of the longhand values making up the six properties we know are included in the pre- and post-transition states, and possibly from others that are inherited or declared in other style blocks impacting the element. ▶

You can listen for `transitionend` events like this:

```
document.querySelector("div").addEventListener("transitionend",
    , (e) => {
      console.log(e.propertyName);
});
```

The `transitionend` event includes three event-specific attributes:

`propertyName`
: The name of the CSS property that just finished transitioning.

`pseudoElement`
: The pseudo-element upon which the transition occurred, preceded by two semicolons, or an empty string if the transition was on a regular DOM node.

`elapsedTime`
: The amount of time the transition took to run, in seconds; usually this is the time listed in the `transition-duration` property.

A `transitionend` event will occur for each property that successfully transitions to a new value. It will not fire if the transition is interrupted, such as by removing the state change that initiated the transition or by another change to the same property on the same element. That said, a `transitionend` event *will* occur when it reverts back to its initial value, or when it finishes transitioning to the value made by that other property value change on the element.

When the properties return to their initial value, another `transitionend` event occurs. This event occurs as long as the transition started, even if it didn't finish its initial transition in the original direction.

Setting Transition Duration

The `transition-duration` property takes as its value a comma-separated list of lengths of time, in seconds (`s`) or milliseconds (`ms`). These time values describe the time it will take to transition from one state to another.

transition-duration

Values	*<time>#*
Initial value	`0s`
Applies to	All elements and `:before` and `:after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

When moving between two states, if a duration is declared for only one of those states, the transition duration will be used for only the transition *to* that state. Consider the following:

```
input {
    transition: background-color;
}
input:invalid {
    transition-duration: 1s;
    background-color: red;
}
input:valid {
    transition-duration: 0.2s;
    background-color: green;
}
```

Thus, it will take 1 second for the input to change to a red background when it becomes invalid, and only 200 milliseconds to transition to a green background when it becomes valid. ▶

The value of the `transition-duration` property is positive in either seconds (`s`) or milliseconds (`ms`). The time unit of `ms` or `s` is required by the specification, even if the duration is set to `0s`. By default, properties change from one value to the next instantly, showing no visible animation, which is why the default value for the duration of a transition is `0s`.

Unless a positive value for `transition-delay` is set on a property, if `transition-duration` is omitted, it is as if no `transition-property` declaration had been applied, and no `transitionend` event will occur. As long as the total duration time for a transition is greater than 0 seconds—as long as the `transition-duration` is greater than the `transition-delay`, including greater than the default `0s` delay—the transition will still be applied, and a `transitionend` event will occur when the transition finishes.

Negative values for `transition-duration` are invalid, and, if included, will invalidate the entire `transition-duration` declaration.

Using the same lengthy `transition-property` declaration from before, we can declare a single duration for all the properties, individual durations for each property, or we can make alternate properties animate for the same length of time. We can declare a single duration that applies to all properties during the transition by including a single `transition-duration` value:

```
div {
    color: #ff0000;
    …
    transition-property: color, border, border-radius, opacity,
        width, padding;
    transition-duration: 200ms;
}
```

We can also declare the same number of comma-separated time values for the `transition-duration` property value as the CSS properties listed in the `transition-property` property value. If we want each property to transition over a different length of time, we have to include a different comma-separated value for each property name declared:

```
div {
    color: #ff0000;
    …
    transition-property: color, border, border-radius, opacity,
        width, padding;
    transition-duration: 200ms, 180ms, 160ms, 120ms, 1s, 2s;
}
```

If the number of properties declared does not match the number of durations declared, the browser has specific rules on how to handle the mismatch. If we have more durations than properties, the extra durations are ignored. If we have more properties than durations, the durations are repeated. In the following example, `color`, `border-radius`, and `width` have a duration of 100 milliseconds; `border`, `opacity`, and `padding` will be set to 200 milliseconds:

```
div {
    …
    transition-property: color, border, border-radius, opacity,
        width, padding;
    transition-duration: 100ms, 200ms;
}
```

If we declare exactly two comma-separated durations, every odd property will transition over the first time declared, and every even property will transition over the second time value declared.



Always remember that user experience is important. If a transition is too slow, the website will appear slow or unresponsive, drawing unwanted focus to what should be a subtle effect. If a transition is too fast, it may be too subtle to be noticed. Visual effects should last long enough to be seen, but not so long as to make themselves the center of attention. Generally, the best duration for a visible, yet not distracting, transition is 100 to 300 milliseconds.

Altering the Internal Timing of Transitions

Do you want your transition to start off slow and get faster, start off fast and end slower, advance at an even keel, jump through various steps, or even bounce? The `transition-timing-function` provides a way to control the pace of the transition.

transition-timing-function

Values	*<timing-function>#*
Initial value	`ease`
Applies to	All elements and `:before` and `:after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

The `transition-timing-function` values include `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, `step-start`, `step-end`, `steps(n, start)`—where *n* is the number of steps —`steps(n, end)`, and `cubic-bezier(x1, y1, x2, y2)`. (These values are also the valid values for the `animation-timing-function`, and they are described in great detail in Chapter 19.)

Cubic Bézier timing

The nonstep keywords are easing timing functions that serve as aliases for cubic Bézier mathematical functions that provide smooth curves. The specification provides for five predefined easing functions, as shown in Table 18-1.

Table 18-1. Supported keywords for cubic Bézier timing functions

Timing function	Description	Cubic Bézier value
`cubic-bezier()`	Specifies a cubic Bézier curve	`cubic-bezier(x1, y1, x2, y2)`
`ease`	Starts slow, then speeds up, then slows down, then ends very slowly	`cubic-bezier(0.25, 0.1, 0.25, 1)`
`linear`	Proceeds at the same speed throughout transition	`cubic-bezier(0, 0, 1, 1)`
`ease-in`	Starts slow, then speeds up	`cubic-bezier(0.42, 0, 1, 1)`
`ease-out`	Starts fast, then slows down	`cubic-bezier(0, 0, 0.58, 1)`
`ease-in-out`	Similar to `ease`; faster in the middle, with a slow start but not as slow at the end	`cubic-bezier(0.42, 0, 0.58, 1)`

Cubic Bézier curves, including the underlying curves defining the five named easing functions in Table 18-1 and displayed in Figure 18-3, take four numeric parameters. For example, `linear` is the same as `cubic-bezier(0, 0, 1, 1)`. The first and third cubic Bézier function parameter values need to be between 0 and 1.

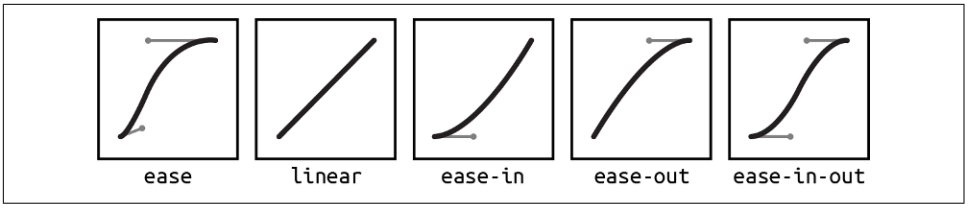


Figure 18-3. Curve representations of named cubic Bézier functions

The four numbers in a `cubic-bezier()` function define the *x* and *y* coordinates of two *handles* within a box. These handles are the endpoints of lines that stretch from the bottom-left and top-right corners of the box. The curve is constructed using the two corners, and the two handles' coordinates, via a Bézier function.

To get an idea of how this works, look at the curves and their corresponding values shown in Figure 18-4.

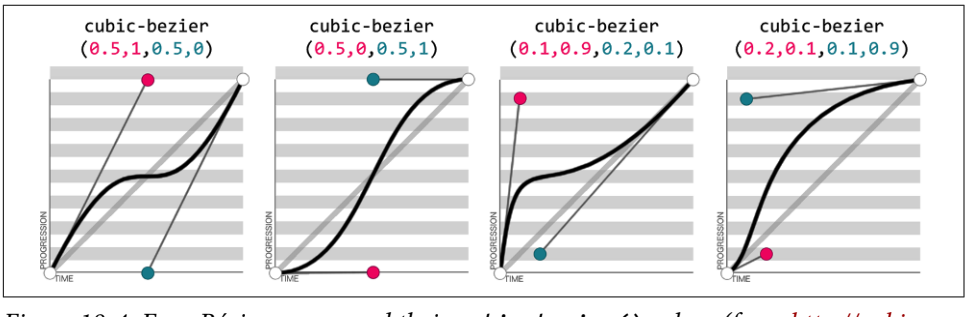


Figure 18-4. Four Bézier curves and their `cubic-bezier()` values (from http://cubic-bezier.com)

Consider the first example. The first two values, corresponding to *x1* and *y1*, are `0.5` and `1`. If you go halfway across the box (*x1* = 0.5) and all the way to the top of the box (*y1* = 1), you land at the spot where the first handle is placed. Similarly, the coordinates `0.5,0` for *x2,y2* describe the point at the center bottom of the box, which is where the second handle is placed. The curve shown there results from those handle placements.

In the second example, the handle positions are switched, with the resulting change in the curve. Ditto for the third and fourth examples, which are inversions of each other. Notice how the resulting curve differs when switching the handle positions.

The predefined key terms are fairly limited. To better follow the principles of animation, you may want to use a cubic Bézier function with four float values instead of the predefined key words. If you're a whiz at calculus or have a lot of experience with programs like Illustrator, you might be able to invent cubic Bézier functions in your head; otherwise, online tools let you play with different values, such as *http://cubic-bezier.com*, which lets you compare the common keywords against each other or against your own cubic Bézier function.

As shown in Figure 18-5, the website *http://easings.net* provides many additional cubic Bézier function values you can use to provide for a more realistic, delightful animation.

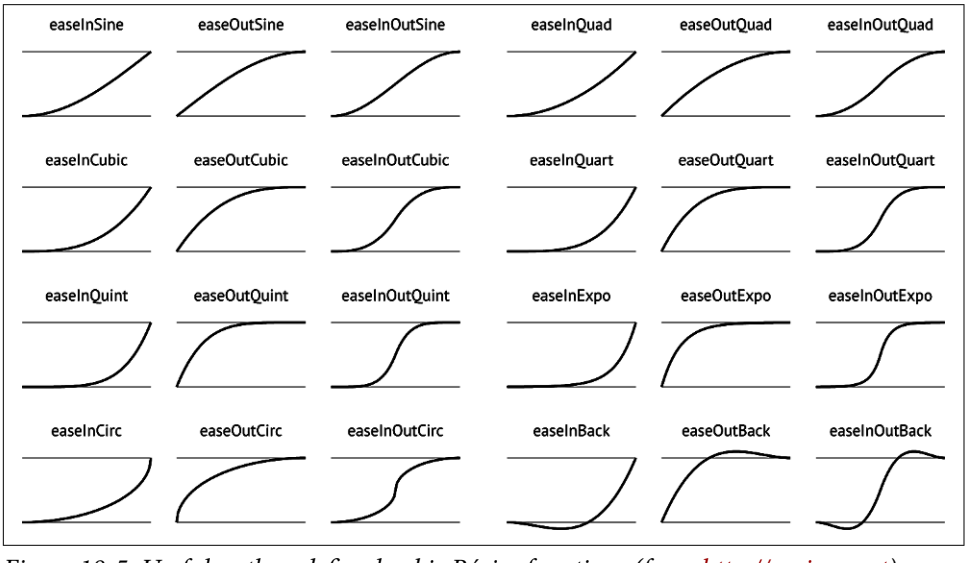


Figure 18-5. Useful author-defined cubic Bézier functions (from http://easings.net)

While the authors of the site named their animations, the preceding names are not part of the CSS specifications, and must be written as shown in Table 18-2.

Table 18-2. Cubic Bézier timings

Unofficial name	Cubic Bézier function value
`easeInSine`	`cubic-bezier(0.47, 0, 0.745, 0.715)`
`easeOutSine`	`cubic-bezier(0.39, 0.575, 0.565, 1)`
`easeInOutSine`	`cubic-bezier(0.445, 0.05, 0.55, 0.95)`
`easeInQuad`	`cubic-bezier(0.55, 0.085, 0.68, 0.53)`
`easeOutQuad`	`cubic-bezier(0.25, 0.46, 0.45, 0.94)`
`easeInOutQuad`	`cubic-bezier(0.455, 0.03, 0.515, 0.955)`
`easeInCubic`	`cubic-bezier(0.55, 0.055, 0.675, 0.19)`
`easeOutCubic`	`cubic-bezier(0.215, 0.61, 0.355, 1)`
`easeInOutCubic`	`cubic-bezier(0.645, 0.045, 0.355, 1)`
`easeInQuart`	`cubic-bezier(0.895, 0.03, 0.685, 0.22)`
`easeOutQuart`	`cubic-bezier(0.165, 0.84, 0.44, 1)`
`easeInOutQuart`	`cubic-bezier(0.77, 0, 0.175, 1)`
`easeInQuint`	`cubic-bezier(0.755, 0.05, 0.855, 0.06)`
`easeOutQuint`	`cubic-bezier(0.23, 1, 0.32, 1)`

Unofficial name	Cubic Bézier function value
`easeInOutQuint`	`cubic-bezier(0.86, 0, 0.07, 1)`
`easeInExpo`	`cubic-bezier(0.95, 0.05, 0.795, 0.035)`
`easeOutExpo`	`cubic-bezier(0.19, 1, 0.22, 1)`
`easeInOutExpo`	`cubic-bezier(1, 0, 0, 1)`
`easeInCirc`	`cubic-bezier(0.6, 0.04, 0.98, 0.335)`
`easeOutCirc`	`cubic-bezier(0.075, 0.82, 0.165, 1)`
`easeInOutCirc`	`cubic-bezier(0.785, 0.135, 0.15, 0.86)`
`easeInBack`	`cubic-bezier(0.6, -0.28, 0.735, 0.045)`
`easeOutBack`	`cubic-bezier(0.175, 0.885, 0.32, 1.275)`
`easeInOutBack`	`cubic-bezier(0.68, -0.55, 0.265, 1.55)`

Step timing

Step timing functions also are available, as well as four predefined step values; see Table 18-3.

Table 18-3. Step timing functions

Timing function	Definition
`steps(<integer>, jump-start)`	Displays <*integer*> keyframes, showing the last keyframe for the last *n*/100% of the transition duration; the first jump happens at the very beginning of the transition. `start` can be used in place of `jump-start`
`steps(<integer>, jump-end)`	Displays <*integer*> keyframes, staying on the initial state for the first *n*/100% of the transition duration; the last jump happens at the very end of the transition. `end` can be used in place of `jump-end`
`steps(<integer>, jump-both)`	Displays <*integer*> keyframes, starting with an immediate jump and taking the final jump at the very end of the transition duration; this effectively adds one step to the transition
`steps(<integer>, jump-none)`	Displays <*integer*> keyframes, but there is no jump at either the beginning or end of the transition duration, instead staying on the initial values for the first *n*/100% of the time *and* showing the final values for the last *n*/100% of the time; this effectively removes one step from the transition
`step-start`	Stays on the final keyframe throughout transition duration; equal to `steps(1, jump-start)`
`step-end`	Stays on the initial keyframe throughout transition duration; equal to `steps(1, jump-end)`

As Figure 18-6 shows, the step timing functions show the progression of the transition from the initial value to the final value in steps, rather than as a smooth curve.

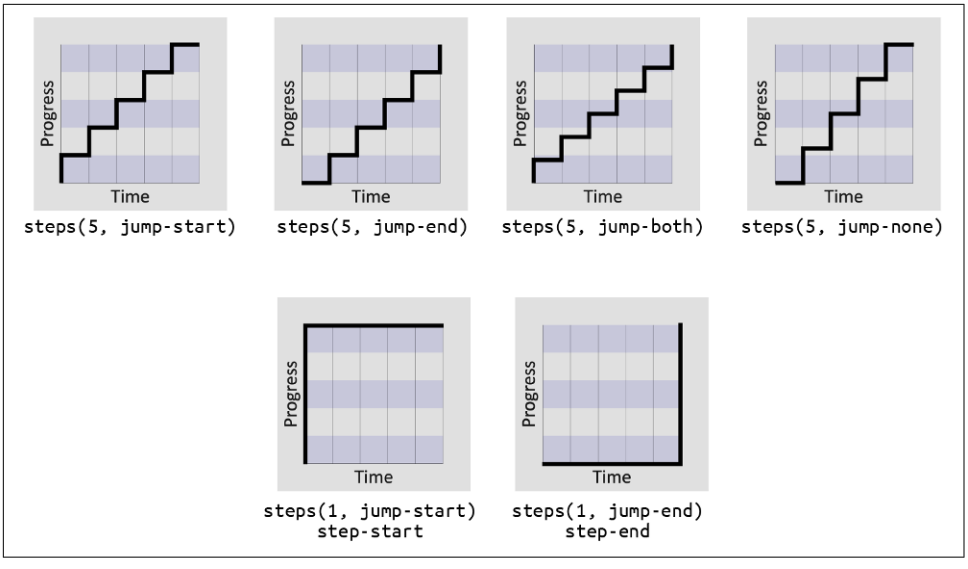


Figure 18-6. Step timing functions

The step timing functions allow you to divide the transition over equidistant steps, by defining the number and direction of steps.

With `jump-start`, the first step happens at the animation or transition start. With `jump-end`, the last step happens at the animation or transition end. For example, `steps(5, jump-end)` would jump through the equidistant steps at 0%, 20%, 40%, 60%, and 80%; and `steps(5, jump-start)` would jump through the equidistant steps at 20%, 40%, 60%, 80%, and 100%.

The `step-start` function is the same as `steps(1, jump-start)`. When used, transitioned property values stay on their *final* values from the beginning until the end of the transition. The `step-end` function, which is the same as `steps(1, jump-end)`, sets transitioned values to their *initial* values, staying there throughout the transition's duration.



Step timing, especially the precise meaning of `jump-start` and `jump-end`, is discussed in depth in Chapter 19.

Continuing on with the same lengthy `transition-property` declaration we've used before, we can declare a single timing function for all the properties, or define individual timing functions for each property, and so on. Here, we've set all the transitioned properties to a single duration and timing function:

```
div {
    transition-property: color, border-width, border-color, border-radius,
```

```
        opacity, width, padding;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
}
```

Always remember that the `transition-timing-function` does not change the time it takes to transition properties: that is set with the `transition-duration` property. It just changes how the transition progresses during that set time. Consider the following:

```
div {
    …
    transition-property: color, border-width, border-color, border-radius,
        opacity, width, padding;
    transition-duration: 200ms;
    transition-timing-function: ease-in, ease-out, ease-in-out,
        step-end, step-start, steps(5, jump-start), steps(3, jump-end);
}
```

If we include these seven timing functions for the seven properties, as long as they have the same transition duration and delay, all the properties start and finish transitioning at the same time. (The preceding transition would be a terrible user experience, by the way. Please don't do that.)

The best way to familiarize yourself with the various timing functions is to play with them and see which one works best for the effect you're looking for. While testing, set a relatively long `transition-duration` to better visualize the difference between the various functions. At higher speeds, you may not be able to tell the difference between different easing functions. Just don't forget to set the transition back to a faster speed before publishing the result!

Delaying Transitions

The `transition-delay` property enables you to introduce a delay between the time that the change initiating the transition is applied to an element and the time the transition actually begins.

transition-delay

Values	*<time>#*
Initial value	`0s`
Applies to	All elements, `:before` and `:after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

A `transition-delay` of `0s` (the default) means the transition will begin immediately; it will start executing as soon as the state of the element is altered. This is familiar from the instant-change effect of `a:hover`, for example.

With a value other than `0s`, the <*time*> value of `transition-delay` defines the time offset from the moment the property values would ordinarily have changed until the property values declared in the `transition` or `transition-property` values begin animating to their final values.

Interestingly, negative values of time are valid. The effects you can create with negative `transition-delay`s are described in "Negative delay values" on page 900.

Continuing with the 6- (or 19-) property `transition-property` declaration we've been using, we can make all the properties start transitioning right away by omitting the `transition-delay` property, or by including it with a value of `0s`. Another possibility is to start half the transitions right away, and the rest 200 milliseconds later, as in the following:

```
div {
    transition-property: color, border, border-radius, opacity,
        width, padding;
    transition-duration: 200ms;
    transition-timing-function: linear;
    transition-delay: 0s, 200ms;
}
```

By including `transition-delay: 0s, 200ms` on a series of properties, each taking 200 milliseconds to transition, we make `color`, `border-radius`, and `width` begin their transitions immediately. All the rest begin their transitions as soon as the other transitions have completed, because their `transition-delay` is equal to the `transition-duration` applied to all the properties.

As with `transition-duration` and `transition-timing-function`, when the comma-separated `transition-delay` values outnumber the comma-separated `transition-property` values, the extra delay values are ignored. When the comma-separated `transition-property` values outnumber the comma-separated `transition-delay` values, the delay values are repeated.

We can even declare seven `transition-delay` values so that each property begins transitioning after the previous property has transitioned, as follows:

```
div {
    …
    transition-property: color, border-width, border-color, border-radius,
        opacity, width, padding;
    transition-duration: 200ms;
    transition-timing-function: linear;
    transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s;
}
```

In this example, we declare each transition to last 200 milliseconds with the `transition-duration` property. We then declare a `transition-delay` that provides comma-separated

delay values for each property that increments by 200 milliseconds, or 0.2 seconds—the same time as the duration of each property's transition. The end result is that each property starts transitioning at the point the previous property has finished.

We can use math to give every transitioning property different durations and delays, ensuring that they all complete transitioning at the same time:

```
div {
    …
    transition-property: color, border-width, border-color, border-radius,
        opacity, width, padding;
    transition-duration: 1.4s, 1.2s, 1s, 0.8s, 0.6s, 0.4s, 0.2s;
    transition-timing-function: linear;
    transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s;
}
```

In this example, each property completes transitioning at the 1.4-second mark, but each with a different duration and delay. For each property, the `transition-duration` value plus the `transition-delay` value will add up to 1.4 seconds.

Generally, you'll want all the transitions to begin at the same time. You can make that happen by including a single `transition-delay` value, which gets applied to all the properties. In our drop-down menu in Figure 18-1, we included a delay of 50 milliseconds. This delay is not long enough for the user to notice and will not cause the application to appear slow. Rather, a 50-millisecond delay can help prevent the navigation from shooting open unintentionally as the user accidentally hovers over the menu items while moving the cursor from one part of the page or app to another, or as they quickly move the focus ring through the document.

Negative delay values

A negative value for `transition-delay` that is smaller than the `transition-duration` will cause the transition to start immediately, partway through the transition. For example: ▶

```
div {
  transform: translateX(0);
  transition-property: transform;
  transition-duration: 200ms;
  transition-delay: -150ms;
  transition-timing-function: linear;
}
div:hover {
  transform: translateX(200px);
}
```

Given the `transition-delay` of `-150ms` on a `200ms` transition, the transition will start three-quarters of the way through the transition and will last 50 milliseconds. In that scenario, given the linear timing function, the `<div>` jumps to being translated `150px` along the x-axis immediately on hover and then animates the translation from 150 pixels to 200 pixels over 50 milliseconds.

If the absolute value of the negative `transition-delay` is greater than or equal to the `transition-duration`, the change of property values is immediate, as if no `transition` had been applied, *and* no `transitionend` event occurs.

When transitioning back from the hovered state to the original state, by default, the same value for the `transition-delay` is applied. In the preceding scenario, since the `transition-delay` is not overridden in the hover state, when the user stops hovering over the element, the `<div>` will jump to being translated 50 pixels along the x-axis and then take 50 milliseconds to return to its initial position of being translated 0 pixels along the x-axis.

Using the transition Shorthand

The `transition` property combines the four properties we've covered thus far—`transition-property`, `transition-duration`, `transition-timing-function`, and `transition-delay`—into a single shorthand property.

transition

Values	[[none | <*transition-property*>] || <*time*> || <*transition-timing-function*> || <*time*>]#
Initial value	`all 0s ease 0s`
Applies to	All elements and `:before` and `:after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

The `transition` property accepts the value of `none`, or any number of comma-separated list of *single transitions*. A single transition contains a single property to transition, or the keyword `all` to transition all the properties; the duration of the transition; the timing function; and the transition delay.

If a single transition within the `transition` shorthand omits the property to transition, that single transition will default to `all`. If the `transition-timing-function` value is omitted, it will default to `ease`. If only one time value is included, that will be the duration, and no delay will occur, as if `transition-delay` were set to `0s`.

Within each single transition, the order of the duration versus the delay is important: the first value that can be parsed as a time will be set as the duration. If an additional time value is found before the comma or the end of the statement, that will be set as the delay.

Here are three equivalent ways to write the same transition effects:

```
nav li ul {
    transition: transform 200ms ease-in 50ms,
                  opacity 200ms ease-in 50ms;
}
nav li ul {
    transition: all 200ms ease-in 50ms;
}
nav li ul {
    transition: 200ms ease-in 50ms;
}
```

In the first example, we see a shorthand way to express each of the two properties that are being transitioned. Because we are transitioning all the properties that will be changed (in other rules not shown in the code block), we could use the keyword `all`, as shown in the second example. And, since `all` is the default value, we could write the shorthand with just the duration, timing function, and delay. Had we used `ease` instead of `ease-in`, we could have omitted the timing function, since `ease` is the default. Had we not wanted a delay, we could have omitted the second time value, since `0s` is the default.

We did have to include the duration, or no transition would be visible. In other words, the only portion of the `transition` property value that can truly be considered required is `transition-duration`.

If we wanted to only delay the change from closed menu to open menu without a gradual transition, we would still need to include a duration of `0s`. Remember, the first value parsable as time will be set as the duration and the second one will be set as the delay:

```
nav li ul {
  transition: 0s 200ms;
}
```



This transition will wait 200 milliseconds, then show the drop-down fully open and opaque with no gradual transition. Creating delays with no transitions is a horrible user experience, so please do not do it.

If we have a comma-separated list of transitions (versus just a single declaration) and the word `none` is included, the entire transition declaration is invalid and will be ignored. You can declare comma-separated values for the four longhand transition properties, or you can include a comma-separated list of multiple shorthand transitions:

```
div {
    transition-property: color, border-width, border-color, border-radius,
        opacity, width, padding;
    transition-duration: 200ms, 180ms, 160ms, 140ms, 100ms, 2s, 3s;
    transition-timing-function: ease, ease-in, ease-out, ease-in-out,
        step-end, steps(5, start), steps(3, end);
    transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s;
}
```

```
div {
    transition:
        color 200ms ease,
        border-width 180ms ease-in 200ms,
        border-color 160ms ease-out 400ms,
        border-radius 140ms ease-in-out 600ms,
        opacity 100ms step-end 0.8s,
        width 2s steps(5, start) 1s,
        padding 3s steps(3, end) 1.2s;
}
```

The two preceding CSS rule blocks are functionally equivalent. Use care when stringing multiple shorthand transitions into a list of transitions: `transition: color, opacity 200ms ease-in 50ms` will ease in the opacity over 200 milliseconds after a 50-millisecond delay, but the `color` change will be instantaneous, with no `transitionend` event. It is still valid, but may not be the effect you were seeking.

Reversing Interrupted Transitions

When a transition is interrupted before it is able to finish (such as mousing off a drop-down menu before it finishes its opening transition), property values are reset to the values they had before the transition began, and the properties transition back to those values. Because repeating the duration and timing functions on a reverting partial transition can lead to an odd or even bad user experience, the CSS Transitions specification provides for making the reverting transition shorter.

Let's say we have a `transition-delay` of `50ms` set on the default state of a menu, and no transition properties declared on the hover state; thus, browsers will wait 50 milliseconds before beginning the reverse (or closing) transition.

When the forward animation finishes transitioning to the final values and the `transitionend` event is fired, all browsers will duplicate the `transition-delay` in the reverse states. Let's say the user moves off that menu 75 milliseconds after it started transitioning. This means the drop-down menu will animate closed without ever being fully opened and fully opaque. The browser should have a 50-millisecond delay before closing the menu, just as it waited 50 milliseconds before starting to open it. This is actually a good user experience, as it provides a few milliseconds of delay before closing, preventing jerky behavior if the user accidentally navigates off the menu.

In the case of a step timing function, if the transition is 10 seconds with 10 steps, and the properties revert after 3.25 seconds, ending a quarter of the way between the third and fourth steps (completing three steps, or 30% of the transition), it will take 3 seconds to revert to the previous values. In the following example, the width of our `<div>` will grow to 130 pixels wide before it begins reverting back to 100 pixels wide on mouseout:

```
div {
    width: 100px;
    transition: width 10s steps(10, jump-start);
}
```

```
div:hover {
    width: 200px;
}
```

While the reverse duration will be rounded down to the time it took to reach the most recently executed step, the reverse *direction* will be split by the originally declared number of steps, not the number of steps that completed. In our 3.25-second case, it will take 3 seconds to revert through 10 steps. These reverse transition steps will be shorter in duration at 300 milliseconds each, each step shrinking the width by 3 pixels, instead of 10 pixels.

If the timing function is linear, the duration will be the same in both directions. All other `cubic-bezier` functions will have a duration that is proportional to progress the initial transition made before being interrupted. Negative `transition-delay` values are also proportionally shortened. Positive delays remain unchanged in both directions.

No browser will have a `transitionend` for the hover state, as the transition did not end; but all browsers will have a `transitionend` event in the reverse state when the menu finishes collapsing. The `elapsedTime` for that reverse transition depends on whether the browser took the full 200 milliseconds to close the menu, or if the browser takes as long to close the menu as it did to partially open the menu.

To override these values, include transition properties in both the initial and final states (e.g., both the unhovered and hovered styles). While this does not impact the reverse shortening, it does provide more control.



Beware of having transitions on both ancestors and descendants. For example, transitioning inherited properties on an element soon after transitioning the same property on ancestor or descendant nodes can have unexpected outcomes. If the transition on the descendant completes before the transition on the ancestor, the descendant will then resume inheriting the (still transitioning) value from its parent. This effect may not be what you expect.

Animatable Properties and Values

Before implementing transitions and animations, it's important to understand that not all properties are animatable. You can transition (or animate) any animatable CSS properties; but which properties are animatable?

One key to developing a sense for which properties can be animated is to identify which have values that can be interpolated. *Interpolation* is the construction of data points between the values of known data points. The key guideline to determining if a property value is animatable is whether the *computed value* can be interpolated. If a property's computed values are keywords, they can't be interpolated; if its keywords compute to a number of some sort, they can be. The quick gut check is that if you can determine a midpoint between two property values, those property values are probably animatable.

For example, the `display` values like `block` and `inline-block` aren't numeric and therefore don't have a midpoint; they aren't animatable. The `transform` property values of `rotate(10deg)` and `rotate(20deg)` have a midpoint of `rotate(15deg)`; they are animatable.

The `border` property is shorthand for `border-style`, `border-width`, and `border-color` (which, in turn, are themselves shorthand properties for the four side values). While there is no midpoint between any of the `border-style` values, the `border-width` property length units are numeric, so they can be animated. The keyword values of `medium`, `thick`, and `thin` have numeric equivalents and can be interpolated: the computed value of the `border-width` property translates those keywords to lengths.

In the `border-color` value, colors are numeric—the named colors can all be represented using hexadecimal or other numeric color values—so colors are animatable as well. If you transition from `border: red solid 3px` to `border: blue dashed 10px`, the border width and border colors will transition at the defined speed, but `border-style` will jump from `solid` to `dashed` immediately.

In the same vein, CSS functions that take numeric values as parameters generally are animatable. An exception to this rule is properties with discrete animation types like `visibility`: while there is no midpoint between the values of `visible` and `hidden`, `visibility` values jump between the discrete values, jumping from visible to not visible. With the `visibility` property, when the initial value or the destination value is `visible`, the value will change at the end of the transition from `visible` to `hidden`. For a transition from `hidden` to `visible`, the value changes at the start of the transition.

The `auto` value should generally be considered nonanimatable and should be avoided for animations and transitions. According to the specification, it is not an animatable value, but some browsers interpolate the current numeric value of `auto` (such as `height: auto`) to be `0px` or possibly a `fit-content()` function. The `auto` value is nonanimatable for properties like `height`, `width`, `top`, `bottom`, `left`, `right`, and `margin`.

Often an alternative property or value may work. For example, instead of changing `height: 0` to `height: auto`, use `max-height: 0` to `max-height: 100vh`, which will generally create the expected effect. The `auto` value is animatable for `min-height` and `min-width`, since `min-height: auto` actually computes to 0.

How Property Values Are Interpolated

Numbers are interpolated as floating-point numbers. Integers are interpolated as whole numbers, and thus increment or decrement as whole numbers.

In CSS, length and percentage units are translated into real numbers. When transitioning or animating `calc()`, from one type of length to or from a percentage, the values will be converted into a `calc()` function and interpolated as real numbers.

Colors, whether they are HSLA, RGB, or named colors like `aliceblue`, are translated to their RGBA equivalent values for transitioning, and interpolated across the RGBA color space. If you want to interpolate across a different color space, such as HSL, ensure the pre- and post-transition colors are in the same color space (in this case, HSL).

When animating font weights, if you use keywords like `bold`, they'll be converted to numeric values and animated.

When including animatable property values that have more than one component, each component is interpolated appropriately for that component. For example, `text-shadow` has up to four components: the color, `x`, `y`, and `blur`. The color is interpolated as `color`, whereas the `x`, `y`, and `blur` components are interpolated as lengths.

Box shadows have two additional optional keywords: `inset` (or lack thereof) and `spread`. Because `spread` is a length, it is interpolated. The `inset` keyword cannot be converted to a numeric equivalent, so there is no way to gradually transition between inset and drop shadows.

Similar to values with more than one component, gradients can be transitioned only if you are transitioning gradients of the same type (linear, radial, or conic) with equal numbers of color stops. The colors of each color stop are then interpolated as colors, and the position of each color stop is interpolated as length and percentage units.

Interpolating Repeating Values

When you have simple lists of other types of properties, each item in the list is interpolated appropriately for that type—as long as the lists have the same number of items or repeatable items, and each pair of values can be interpolated. For example:

```
.img {
    background-image:
        url(1.gif), url(2.gif), url(3.gif), url(4.gif),
        url(5.gif), url(6.gif), url(7.gif), url(8.gif),
        url(9.gif), url(10.gif), url(11.gif), url(12.gif);
    transition: background-size 1s ease-in 0s;
    background-size: 10px 10px, 20px 20px, 30px 30px, 40px 40px;
}
.img:hover {
    background-size: 25px 25px, 50px 50px, 75px 75px, 100px 100px;
}
```

In transitioning four `background-sizes`, with all the sizes in both lists listed in pixels, the third `background-size` from the pretransitioned state can gradually transition to the third `background-size` of the transitioned list. In the preceding example, background images 1, 5, and 9 will transition from `10px` to `25px` in height and width when hovered. Similarly, images 3, 7, and 11 will transition from `30px` to `75px`, and so forth.

Thus, the `background-size` values are repeated three times, as if the CSS had been written as follows:

```
.img {
    …
    background-size: 10px 10px, 20px 20px, 30px 30px, 40px 40px,
                     10px 10px, 20px 20px, 30px 30px, 40px 40px,
                     10px 10px, 20px 20px, 30px 30px, 40px 40px;
    …
}
.img:hover {
    background-size: 25px 25px, 50px 50px, 75px 75px, 100px 100px,
                     25px 25px, 50px 50px, 75px 75px, 100px 100px,
                     25px 25px, 50px 50px, 75px 75px, 100px 100px;
}
```

If a property doesn't have enough comma-separated values to match the number of background images, the list of values is repeated until there are enough, even when the list in the animated state doesn't match the initial state:

```
.img:hover {
    background-size: 33px 33px, 66px 66px, 99px 99px;
}
```

If we transition from four `background-size` declarations in the initial state to three `background-size` declarations in the animated state, all in pixels and still with 12 background images, the animated and initial state values are repeated (three and four times, respectively) until we have the 12 necessary values, as if the following had been declared:

```
.img {
    …
    background-size: 10px 10px, 20px 20px, 30px 30px,
                     40px 40px, 10px 10px, 20px 20px,
                     30px 30px, 40px 40px, 10px 10px,
                     20px 20px, 30px 30px, 40px 40px;
    …
}
.img:hover {
    background-size: 33px 33px, 66px 66px, 99px 99px,
                     33px 33px, 66px 66px, 99px 99px,
                     33px 33px, 66px 66px, 99px 99px,
                     33px 33px, 66px 66px, 99px 99px;
}
```

If a pair of values cannot be interpolated—for example, if the `background-size` changes from `contain` in the default state to `cover` when hovered—then, according to the specification, the lists are not interpolatable. However, some browsers ignore that particular pair of values for the purposes of the transition, and still animate the interpolatable values.

Some property values can animate if the browser can infer implicit values. For example, for shadows, the browser will infer an implicit shadow `box-shadow: transparent 0 0 0` or `box-shadow: inset transparent 0 0 0`, replacing any values not explicitly included in the pre- or post-transition state. These examples are in the chapter files for this book.

Only animatable property value changes trigger `transitionend` events.

If you accidentally include a property that can't be transitioned, fear not. The entire declaration will not fail: the browser will simply not transition the property that is not animatable.

Note that a nonanimatable property or nonexistent CSS property is not exactly ignored. The browser passes over unrecognized or nonanimatable properties, keeping their place in the property list order to ensure that the other comma-separated transition properties described next are not applied to the wrong properties.[1]



Transitions can occur only on properties that are not currently being impacted by a CSS animation. If the element is being animated, properties may still transition, as long as they are not properties that are currently controlled by the animation. CSS animations are covered in Chapter 19.

Printing Transitions

When web pages or web applications are printed, the stylesheet for print media is used. If your style element's media attribute matches only `screen`, the CSS will not impact the printed page at all.

Often, no media attribute is included; it is as if `media="all"` were set, which is the default. Depending on the browser, when a transitioned element is printed, either the interpolating values are ignored or the property values in their current state are printed.

You can't see the element transitioning on a piece of paper, but in some browsers, like Chrome, if an element transitioned from one state to another, the current state at the time the `print` function is called will be the value on the printed page, if that property is printable. For example, if a background color changed, neither the pre-transition or the post-transition background color will be printed, as background colors are generally not printed. However, if the text color mutated from one value to another, the current value of `color` will get printed on a color printer or to a PDF.

In other browsers, like Firefox, whether the pre-transition or post-transition value is printed depends on how the transition was initiated. If it's initiated with a hover, for example, the nonhovered value will be printed, as you are no longer hovering over the element while you interact with the print dialog. If it transitioned with a class addition, the post-transition value will be printed, even if the transition hasn't completed. The printing acts as if the transition properties are ignored.

1 This might change. The CSS Working Group is considering making all property values animatable, switching from one value to the next at the midpoint of the timing function if there is no midpoint between the pre and post values.

Given that CSS has separate print stylesheets or `@media` rules for print, browsers compute style separately. In the print style, styles don't change, so there just aren't any transitions. The printing acts as if the property values changed instantly instead of transitioning over time.

Summary

Transitions are a useful and quite powerful way to add UI enhancements. Worrying about archaic browsers should not prevent you from including them, since if a browser doesn't support CSS transitions, the changes will still be applied. They'll just "transition" from the initial state to the end state instantaneously when the style recomputation occurs. A user may miss out on an interesting (or possibly annoying) effect, but will not miss out on any content.

The defining feature of transitions is that they are applied when an element transitions from one state to another, whether that happens because of user action or some kind of scripted change to the DOM. If you want elements to animate regardless of user action or DOM changes, the next chapter will show you the way.

CHAPTER 19
Animation

CSS transitions, covered in the previous chapter, enable simple animations that are triggered by changes in the DOM state and proceed from a beginning state to an end state. CSS *animations* are similar to transitions in that values of CSS properties change over time, but animations provide much more control over the way those changes happen. Specifically, CSS keyframe animations let us decide if and how an animation repeats, give us granular control over what happens throughout the animation, and more. While transitions trigger implicit property value changes, animations are explicitly executed when keyframe animations are applied.

With CSS animations, you can change property values that are not part of the set pre- or post-state of an element. The property values set on the animated element don't necessarily have to be part of the animation progression. For example, when using a transition, going from black to white will animate only through various shades of gray. With animation, that same element doesn't have to be black or white or even in-between shades of gray during the animation.

While you *can* transition through shades of gray, you could instead turn the element yellow, then animate from yellow to orange. Alternatively, you could animate through various colors, starting with black and ending with white, but progressing through the entire rainbow along the way.



Look for the Play symbol ▶ to know when an online example is available. All of the examples in this chapter can be found at *https://meyerweb.github.io/csstdg5figs/19-animation*.

Accommodating Seizure and Vestibular Disorders



While you can use animations to create ever-changing content, *repeated rapid changing of content can lead to seizures in some users.* Always keep this in mind, and ensure the accessibility of your website for people with epilepsy and other seizure disorders.

We don't usually start a chapter with a warning, but in this case, it's warranted. Visual change, especially rapid visual change, can trigger medical emergencies in users who are prone to seizures. They can also cause severe unease in users who are prone to vestibular disorder (motion sickness).

To reduce or eliminate this risk, use the `prefers-reduced-motion` media query (see Chapter 21). This allows you to apply styles when the user has a "Reduce motion" or similar preference set for their browser or operating system. An approach such as this may be considered:

```
@media (prefers-reduced-motion) {
  * {animation: none !important; transition: none !important;}
}
```

This disables all animations and transitions, assuming no other `!important` animations are specified (and they shouldn't be). This is not a nuanced or perfect solution, but it's a first step. You can invert this approach by segregating all of your animations and transitions in a media block for those who do *not* have motion reduction enabled, like this:

```
@media not (prefers-reduced-motion) {
  /* all animations and transitions */
}
```

Not all animations are dangerous or disorienting, and having at least some animations for all users may be necessary. Transitions and animations can be very helpful in informing users what has changed and guiding them to focus on specific content. In such cases, use `prefers-reduced-motion` to tone down animations that are essential to understanding the UI, and to switch off those that are not essential.

Defining Keyframes

To animate an element, you need to refer to the name of a keyframe animation; to do *that*, we need a named keyframe animation. The first step is to define this reusable CSS keyframe animation by using the `@keyframes` at-rule, thus giving our animation a name.

A `@keyframes` at-rule includes the *animation identifier*, or name, and one or more *keyframe blocks*. Each keyframe block includes one or more keyframe selectors with declaration blocks of property-value pairs. The entire `@keyframes` at-rule specifies the behavior of a single full iteration of the animation. The animation can iterate zero or more times,

depending mainly on the `animation-iteration-count` property value, which we'll discuss in "Declaring Animation Iterations" on page 924.

Each keyframe block includes one or more *keyframe selectors*. These are percentage-of-time positions along the duration of the animation; they are declared either as percentages or with the keywords `from` or `to`. Here's the generic structure of an animation:

```
@keyframes animation_identifier {
  keyframe_selector {
    property: value;
    property: value;
  }
  keyframe_selector {
    property: value;
    property: value;
  }
}
```

Here are a couple of basic examples:

```
@keyframes fadeout {
    from {
        opacity: 1;
    }
    to {
        opacity: 0;
    }
}

@keyframes color-pop {
    0% {
        color: black;
        background-color: white;
    }
    33% { /* one-third of the way through the animation */
        color: gray;
        background-color: yellow;
    }
    100% {
        color: white;
        background-color: orange;
    }
}
```

The first set of keyframes shown takes an element, sets its `opacity` to `1` (fully opaque), and animates it to `0` opacity (fully transparent). The second keyframe set animates an element's foreground to black and its background to white, then animates the foreground from black to gray and then white, and the background from white to yellow and then orange.

Note that the keyframes don't say how long this animation should take—that's handled by a CSS property dedicated to the purpose. Instead they say, "Go from this state to that

state" or "Hit these various states at these percentage points of the total animation." That's why keyframe selectors are always percentages, or `from` and `to`. If you try to use time values (like `1.5s`) as your keyframe selectors, you'll render them invalid.

Setting Up Keyframe Animations

Within the opening and closing curly braces of a keyframe set, you include a series of keyframe selectors with blocks of CSS that declare the properties you want to animate. Once the keyframes are defined, you "attach" the animation to an element by using the `animation-name` property. We'll discuss that property shortly, in "Invoking a Named Animation" on page 920.

Start with the at-rule declaration, followed by the animation name and braces:

```
@keyframes nameOfAnimation {
...
}
```

The name, which you create, is an identifier or a string. Originally, the keyframe names had to be an identifier, but both the specification and the browsers also support quoted strings.

Identifiers are unquoted and have specific rules. You can use any characters `a-z`, `A-Z`, and `0-9`, the hyphen (`-`), underscore (_), and any ISO 10646 character U+00A0 and higher. ISO 10646 is the universal character set; this means you can use any character in the Unicode standard that matches the regular expression `[-_a-zA-Z0-9\u00A0-\u10FFFF]`.▶ The identifier can't start with a digit (0–9) and should not start with two hyphens (though some browsers allow this). One hyphen is fine, as long as it is not followed by a digit—unless you escape the digit or hyphen with a backslash.

If you include any escape characters within your animation name, make sure to escape them with a backslash (\). For example, `Q&A!` must be written as `Q\&A\!`. The name `âœž` can be left as âœž (no, that's not a typo), and ✎ is a valid name as well. But if you're going to use any keyboard characters that aren't letters or digits in an identifier, like `!`, `@`, `#`, `$`, and so on, escape them with a backslash.

Also, don't use any of the keywords covered in this chapter as the name of your animation. For example, possible values for the various animation properties we'll be covering later in the chapter include `none`, `paused`, `running`, `infinite`, `backwards`, and `forwards`. Using an animation property keyword, while not prohibited by the spec, will likely break your animation ▶ when using the `animation` shorthand property (discussed in "Bringing It All Together" on page 950). So, while you can legally name your animation `paused` (or another keyword,) we *strongly* recommend against it.

Defining Keyframe Selectors

Keyframe selectors define points during an animation where we set the values of the properties we want to animate. If you want a value at the start of the animation, you declare it at the `0%` mark. If you want a different value at the end of the animation, you declare the property value at the `100%` mark. If you want a value a third of the way through the animation, you declare it at the `33%` mark. These marks are defined with keyframe selectors.

Keyframe selectors consist of a comma-separated list of one or more percentage values or the keywords `from` or `to`. The keyword `from` is equal to `0%`. The keyword `to` equals `100%`. The keyframe selectors are used to specify the percentage along the duration of the animation the keyframe represents. The keyframe itself is specified by the block of property values declared on the selector. The `%` unit must be used on percentage values. In other words, `0` is invalid as a keyframe selector:

```
@keyframes W {
    from {  /* equivalent to 0% */
      left: 0;
      top: 0;
    }
    25%, 75% {
      top: 100%;
    }
    50% {
      top: 50%;
    }
    to {  /* equivalent to 100% */
      left: 100%;
      top: 0;
    }
}
```

This `@keyframes` animation, named `W`, when attached to a nonstatically positioned element, would move that element along a W-shaped path. `W` has five keyframes: one each at the `0%`, `25%`, `50%`, `75%`, and `100%` marks. The `from` is the `0%` mark, while the `to` is the `100%` mark. ▶

Because the property values we set for the `25%` and `75%` mark are the same, we can put the two keyframe selectors together as a comma-separated list. This is very similar to regular selectors, which you can group together with commas. Whether you keep those selectors on one line (as in the example) or put each selector on its own line is up to your personal preference.

Notice that keyframe selectors do not need to be listed in ascending order. In the preceding example, we have the `25%` and `75%` on the same line, with the `50%` mark coming after that declaration. For legibility, it is highly encouraged to progress from the `0%` to the `100%` mark. However, as demonstrated by the `75%` keyframe in this example, it is not required. You could define your keyframes with the last first and the first last, or scramble them up randomly, or whatever works for you.

Omitting from and to Values

If a `0%` or `from` keyframe is not specified, the user agent (browser) constructs a `0%` keyframe. The implicit `0%` keyframe uses the original values of the properties being animated, as if the `0%` keyframe were declared with the same property values that impact the element when no animation was applied—that is, unless another animation applied to that element is currently animating the same property (see "Invoking a Named Animation" on page 920 for details). Similarly, if the `100%` or `to` keyframe is not defined and no other animations are being applied, the browser creates a faux `100%` keyframe using the value the element would have had if no animation had been set on it.

Say we have a `background-color` change animation:

```
@keyframes change_bgcolor {
    45% { background-color: green; }
    55% { background-color: blue; }
}
```

If the element originally had `background-color: red` set on it, it would be as if the animation were as follows: ▶

```
@keyframes change_bgcolor {
    0%   { background-color: red; }
    45%  { background-color: green; }
    55%  { background-color: blue; }
    100% { background-color: red; }
}
```

Or, remembering that we can include multiple identical keyframes as a comma-separated list, this faux animation also could be written as shown here:

```
@keyframes change_bgcolor {
    0%, 100% { background-color: red; }
    45%  { background-color: green; }
    55%  { background-color: blue; }
}
```

Note the `background-color: red;` declarations are not part of the original keyframe animation; they've just been filled in here for clarity. We can include this `change_bgcolor` animation on many elements, and the perceived animation will differ based on the element's value for the `background-color` property in the nonanimated state. Thus, an element that has a yellow background will animate from yellow to green to blue and then back to yellow.

Although we've been using exclusively integer values for our percentages, noninteger percentage values, such as `33.33%`, are perfectly valid. Negative percentages, values greater than `100%`, and values that aren't otherwise percentages or the keywords `to` or `from` are invalid and will be ignored.

Repeating Keyframe Properties

Much like the rest of CSS, the values in keyframe declaration blocks with identical keyframe values cascade. Thus, the earlier `W` animation can be written with the `to`, or `100%`, declared twice, overriding the value of the `left` property:

```
@keyframes W {
  from, to {
    top: 0;
    left: 0;
  }
  25%, 75% {
    top: 100%;
  }
  50% {
    top: 50%;
  }
  to {
    left: 100%;
  }
}
```

Notice that `to` is declared along with `from` as keyframe selectors for the first code block? That sets both `top` and `left` for the `to` keyframe. Then, the `left` value is overridden for the `to` in the last keyframe block.

Animatable Properties

It's worth taking a moment to note that not all properties are *animatable*. If you list a property that can't be animated within an animation's keyframes, it's simply ignored. (For that matter, so are properties and values that the browser doesn't recognize at all, just like any other part of CSS.)



Exceptions to the midpoint rule include `animating-timing-function` and `visibility`, which are discussed in the next section.

As long as an animatable property is included in at least one block with a value that is different from the nonanimated attribute value, and there is a calculable midpoint between those two values, that property will animate.

If an animation is set between two property values that don't have a calculable midpoint, the property may not animate correctly—or at all. For example, you shouldn't declare an element's height to animate between `height: auto` and `height: 300px`, because there is no easily defined midpoint between `auto` and `300px`. The element will still animate, but browsers will jump from the preanimated state to the postanimated state halfway through the animation. Thus, for a 1-second animation, the element will jump from `auto` height to

300px height at the 500-millisecond point in the animation. ▶ Other properties may animate over the length of the same animation; e.g., if you change the background color, it will animate smoothly over the animation. Only those properties that can't be animated between will jump halfway through.

The behavior of your animation will be most predictable if you declare both a 0% and a 100% value for every property you animate. For example, if you declare `border-radius: 50%;` in your animation, you may want to declare `border-radius: 0%;` as well, because the default value of `border-radius` is `none`, not `0`, and there is no midpoint between `none` and other values. Consider the difference in the following two animations:

```
@keyframes round {
    100% {
        border-radius: 50%;
    }
}
@keyframes square_to_round {
    0% {
        border-radius: 0%;
    }
    100% {
        border-radius: 50%;
    }
}
```

The `round` animation will animate an element from the original `border-radius` value of that element to `border-radius: 50%` over the duration of the animation. The `square_to_round` animation will animate an element from `border-radius: 0%` to `border-radius: 50%` over the duration of the animation. If the element starts out with square corners, the two animations will have exactly the same effect. But if the element starts out with rounded corners, `square_to_round` will jump to rectangular corners before it starts animating.

Using Nonanimatable Properties That Aren't Ignored

Exceptions to the midpoint rule include `visibility` and `animation-timing-function`.

The `visibility` property is animatable, even though there is no midpoint between `visibility: hidden` and `visibility: visible`. When you animate from `hidden` to `visible`, the visibility value jumps from one value to the next at the keyframe where the change is declared. So you don't get a smooth fade from visible to hidden, or vice versa. The state changes in an instant.

While the `animation-timing-function` is not, in fact, an animatable property, when included in a keyframe block, the animation timing will switch to the newly declared value at that point in the animation for the properties within that keyframe selector block. The change in animation timing is not animated; it simply switches to the new value for those properties only, and only until the next keyframe. This allows you to vary the

timing function from one keyframe to another. (This is covered in "Changing the Internal Timing of Animations" on page 937.)

Scripting @keyframes Animations

The CSSKeyframesRule API enables finding, appending, and deleting keyframe rules. You can change the content of a keyframe block within a given `@keyframes` declaration with `appendRule(n)` or `deleteRule(n)`, where *n* is the full selector of that keyframe. You can return the contents of a keyframe with `findRule(n)`. Consider this:

```
@keyframes W {
  from, to { top: 0; left: 0; }
  25%, 75% { top: 100%; }
  50%      { top: 50%; }
  to       { left: 100%; }
}
```

The `appendRule()`, `deleteRule()`, and `findRule()` methods take the full keyframe selector as an argument, as shown in the following:

```
// Get the selector and content block for a keyframe
var aRule = myAnimation.findRule('25%, 75%').cssText;

// Delete the 50% keyframe
myAnimation.deleteRule('50%');

// Add a 53% keyframe to the end of the animation
myAnimation.appendRule('53% {top: 50%;}');
```

The statement `myAnimation.findRule('25%, 75%').cssText`, where `myAnimation` is pointing to a keyframe animation, returns the keyframe that matches `25%, 75%`. It would not match any block using either `25%` or `75%` only. If `myAnimation` refers to the `W` animation, `myAnimation.findRule('25%, 75%').cssText` returns `25%, 75% { top: 100%; }`.

Similarly, `myAnimation.deleteRule('50%')` will delete the *last* 50% keyframe—so if we have multiple `50%` keyframes, the last one listed will be the first to go. Conversely, `myAnimation.appendRule('53% {top: 50%;}')` appends a 53% keyframe after the last keyframe of the `@keyframes` block. ▶

CSS has four animation events: `animationstart`, `animationend`, `animationiteration`, and `animationcancel`. The first two occur at the start and end of an animation, and the last between the end of an iteration and the start of a subsequent iteration. Any animation for which a valid keyframe rule is defined will generate the start and end events, even animations with empty keyframe rules. The `animationiteration` event occurs only when an animation has more than one iteration, as the `animationiteration` event does not fire if the `animationend` event would fire at the same time. The `animationcancel` event is fired whenever a running animation is stopped before reaching its last keyframe.

Animating Elements

Once you have created a keyframe animation, you can apply that animation to elements and/or pseudo-elements. CSS provides numerous animation properties to attach a keyframe animation to an element and control its progression. At a minimum, you need to include the name of the animation for the element to animate, and a duration if you want the animation to be visible. (Without a duration, the animation will happen in zero time.)

You can attach animation properties to an element in two ways: include all the animation properties separately, or declare all the properties in one line by using the `animation` shorthand property (or a combination of shorthand and longhand properties). Let's start with the individual properties.

Invoking a Named Animation

The `animation-name` property takes as its value a comma-separated list of names of keyframe animations you want to apply to the selected elements. The names are the unquoted identifiers or quoted strings (or a mixture of both) you created in your `@keyframes` rules.

animation-name

Values	[*<single-animation-name>* | none]#
Initial value	none
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

The default value is `none`, which means no animation is applied to the selected elements. The `none` value can be used to override any animation applied elsewhere in the CSS cascade. (This is also the reason you don't want to name your animation `none`, unless you're a masochist.) ▶

Using the `change_bgcolor` keyframe animation defined in "Omitting from and to Values" on page 916, we have this:

```
div {
    animation-name: change_bgcolor;
}
```

This simple rule applies the `change_bgcolor` animation to all `<div>` elements, however many or few are on the page. To apply more than one animation, include more than one comma-separated animation name:

```
div {
   animation-name: change_bgcolor, round, W;
}
```

If one of the included keyframe identifiers does not exist, the series of animations will not fail; rather, the failed animation will be ignored, and the valid animations will be applied. While ignored initially, the failed animation will be applied if and when that keyframe animation comes into existence as a valid animation. Consider the following:

```
div {
   animation-name: change_bgcolor, spin, round, W;
}
```

In this example, assume that no `spin` keyframe animation is defined. The `spin` animation will not be applied, while the `change_bgcolor`, `round`, and `W` animations will occur. Should a `spin` keyframe animation come into existence through scripting, it will be applied at that time. ▶

If more than one animation is applied to an element and those animations have repeated properties, the later animations override the property values in the earlier animations. For example, if more than two background color changes are applied concurrently in two different keyframe animations, whichever animation was listed later will override the background property declarations of animations earlier in the list, but *only* if the properties (background colors, in this case) are being animated at the same time. ▶ For more on this, see "Animation, Specificity, and Precedence Order" on page 953.

For example, assume the following, and further assume that the animations happen over a period of 10 seconds:

```
div {animation-name: change_bgcolor, bg-shift;}

@keyframes bg-shift {
    0%, 100% {background-color: cyan;}
    35% {background-color: orange;}
    55% {background-color: red;}
    65% {background-color: purple;}
}
@keyframes change_bgcolor {
    0%, 100% {background-color: yellow;}
    45% {background-color: green;}
    55% {background-color: blue;}
}
```

The background will animate from cyan to orange to red to purple and then back to cyan, thanks to `bg-shift`. Because it comes last in the list of animations, its keyframes take precedence. Anytime multiple animations specify behavior for the same property at the same point in time, the animation listed last in the value of `animation-name` will be in effect.

What's interesting is what happens if the `from` (`0%`) or `to` (`100%`) keyframes are omitted from the animation in force. For example, let's remove the first keyframes defined in `bg-shift`:

```
div {animation-name: change_bgcolor, bg-shift;}

@keyframes bg-shift {
    35% {background-color: orange;}
    55% {background-color: red;}
    65% {background-color: purple;}
}
@keyframes change_bgcolor {
    0%, 100% {background-color: yellow;}
    45% {background-color: green;}
    55% {background-color: blue;}
}
```

Now no background colors are defined at the beginning and end of `bg-shift`. In a situation like this, when a `0%` or `100%` keyframe is not specified, the user agent constructs a `0%`/`100%` keyframe by using the computed values of the properties being animated.

These are concerns only when two different keyframe blocks are trying to change the same property's values. In this case, it is `background-color`. On the other hand, if one keyframe block animates `background-color` while another animates `padding`, the two animations will not collide, and both the background color and padding will be animated together.

Simply applying an animation to an element is not enough for the element to visibly animate. For that to happen, the animation must take place over some amount of time. For that, we have the `animation-duration` property.

Defining Animation Lengths

The `animation-duration` property defines how long a single animation iteration should take in seconds (`s`) or milliseconds (`ms`).

animation-duration

Values	*<time>#*
Initial value	`0s`
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

The `animation-duration` property defines the length of time, either in seconds (`s`) or milliseconds (`ms`), it should take to complete one cycle through all the keyframes of the animation. If you don't declare `animation-duration`, the animation will still be run with a duration of `0s`, with `animationstart` and `animationend` still being fired even though the animation, taking `0s`, is imperceptible. Negative time values are not permitted for `animation-duration`.

When specifying a duration, you must include the second (`s`) or millisecond (`ms`) unit. If you have more than one animation, you can include a different `animation-duration` for each by including more than one comma-separated time duration:

```
div {
   animation-name: change_bgcolor, round, W;
   animation-duration: 200ms, 100ms, 0.5s;
}
```

If you supply an invalid value within your comma-separated list of durations (e.g., `animation-duration: 200ms, 0, 0.5s`) the entire declaration will fail, and it will behave as if `animation-duration: 0s` had been declared; `0` is not a valid time value. ▶

Generally, you will want to include an `animation-duration` value for each `animation-name` provided. If you have only one duration, all the animations will last the same amount of time. Having fewer `animation-duration` values than `animation-name` values in your comma-separated property value list will not fail: rather, the values will be repeated as a group. Say we have the following:

```
div {
    animation-name: change_bgcolor, spin, round, W;
    animation-duration: 200ms, 5s;
       /* same effect as '200ms, 5s, 200ms, 5s' */
}
```

The `change_bgcolor` and `round` animations will be run over `200ms`, and the `spin` and `W` animations will run for `5s`.

If you have a greater number of `animation-duration` values than `animation-name` values, the extra values will be ignored. If one of the included animations does not exist, the series of animations and animation durations will not fail; the failed animation, along with its duration, is ignored:

```
div {
    animation-name: change_bgcolor, spinner, round, W;
    animation-duration: 200ms, 5s, 100ms, 0.5s;
}
```

In this example, the duration `5s` is associated with `spinner`. There is no `spinner` animation, though, so `spinner` doesn't exist, and the `5s` and `spinner` are both ignored. Should a `spinner` animation come into existence, it will be applied to `<div>` elements and last 5 seconds.

Declaring Animation Iterations

Simply including the required `animation-name` will lead to the animation playing once, and only once, resetting to the initial state at the end of the animation. If you want to iterate through the animation more or fewer times than the default one time, use the `animation-iteration-count` property.

animation-iteration-count

Values	[<*number*> | `infinite`]#
Initial value	`1`
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

By default, the animation will occur once (because the default value is `1`). If another value is given for `animation-iteration-count`, and there isn't a negative value for the `animation-delay` property, the animation will repeat the number of times specified by the value of the property, which can be any number or the keyword `infinite`. The following declarations will cause their animations to be repeated 2, 5, and 13 times, respectively:

```
animation-iteration-count: 2;
animation-iteration-count: 5;
animation-iteration-count: 13;
```

If the value of `animation-iteration-count` is not an integer, the animation will still run, but will cut off mid-iteration on the final iteration. For example, `animation-iteration-count: 1.25` will iterate through the animation one and a quarter times, cutting off 25% of the way through the second iteration. If the value is `0.25` on an 8-second animation, the animation will play about 25% of the way through, ending after 2 seconds.

Negative numbers are not permitted. If an invalid value is given, the default value of `1` will lead to a default single iteration. ▶

Interestingly, `0` is a valid value for the `animation-iteration-count` property. When set to `0`, the animation still occurs, but zero times. This is similar to setting `animation-duration: 0s`: it will throw both an `animationstart` and an `animationend` event.

If you are attaching more than one animation to an element or pseudo-element, include a comma-separated list of values for `animation-name`, `animation-duration`, and `animation-iteration-count`:

```
.flag {
    animation-name: red, white, blue;
    animation-duration: 2s, 4s, 6s;
    animation-iteration-count: 3, 5;
}
```

The `iteration-count` values (and all other animation property values) will be assigned in the order of the comma-separated `animation-name` property value. Extra values are ignored. Missing values cause the existing values to be repeated, as with `animation-iteration-count` in the preceding scenario.

The preceding example has more name values than count values, so the count values will repeat: `red` and `blue` will iterate three times, and `white` will iterate five times. We have the same number of name values as duration values; therefore, the duration values will not repeat. The `red` animation lasts 2 seconds, iterating three times, and therefore will run for a total of 6 seconds. The `white` animation lasts 4 seconds, iterating five times, for a total of 20 seconds. The `blue` animation is 6 seconds per iteration with the repeated three iterations value, animating for a total of 18 seconds.

Invalid values will invalidate the entire declaration, leading to the animations being played once each.

If we want all three animations to end at the same time, even though their durations differ, we can control that with `animation-iteration-count`:

```
.flag {
    animation-name: red, white, blue;
    animation-duration: 2s, 4s, 6s;
    animation-iteration-count: 6, 3, 2;
}
```

In this example, the `red`, `white`, and `blue` animations will last for a total of 12 seconds each, because the product of the durations and iteration counts in each case totals 12 seconds.

You can also include the keyword `infinite` instead of a number, for a duration. This will make the animation iterate forever, or until something makes it stop, such as removing the animation name, removing the element from the DOM, or pausing the play state.

Setting an Animation Direction

With the `animation-direction` property, you can control whether the animation progresses from the 0% keyframe to the 100% keyframe, or from the 100% keyframe to the 0% keyframe. You can also define whether all the iterations progress in the same direction, or set every other animation cycle to progress in the opposite direction.

animation-direction

Values	[`normal` | `reverse` | `alternate` | `alternate-reverse`]#
Initial value	`normal`
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

The `animation-direction` property defines the direction of the animation's progression through the keyframes. It has four possible values:

`normal`
: Each iteration of the animation progresses from the 0% keyframe to the 100% keyframe; this value is the default.

`reverse`
: Sets each iteration to play in reverse keyframe order, always progressing from the 100% keyframe to the 0% keyframe. Reversing the animation direction also reverses the `animation-timing-function` (which is described in "Changing the Internal Timing of Animations" on page 937).

`alternate`
: The first iteration (and each subsequent odd-numbered iteration) proceeds from 0% to 100%, and the second iteration (and each subsequent even-numbered cycle) reverses direction, proceeding from 100% to 0%. This has an effect only if you have more than one iteration.

`alternate-reverse`
: Similar to the `alternate` value, except it's the reverse. The first iteration (and each subsequent odd-numbered iteration) will proceed from 100% to 0%, and the second iteration (and each subsequent even-numbered cycle) reverses direction, proceeding from 100% to 0%:

```
.ball {
    animation-name: bouncing;
    animation-duration: 400ms;
    animation-iteration-count: infinite;
    animation-direction: alternate-reverse;
}
@keyframes bouncing {
    from {
        transform: translateY(500px);
    }
```

```
        to {
            transform: translateY(0);
        }
    }
```

In this example, we are bouncing a ball, but we want to start by dropping it, not by throwing it up in the air: we want it to alternate between going down and up, rather than up and down, so `animation-direction: alternate-reverse` is the most appropriate value for our needs. ▶

This is a rudimentary way of making a ball bounce. When balls are bouncing, they are moving slowest when they reach their apex and fastest when they reach their nadir. We include this example here to illustrate the `alternate-reverse` animation directions. We'll revisit the bouncing animation to make it more realistic with the addition of timing (in "Changing the Internal Timing of Animations" on page 937). We'll also discuss how, when the animation is iterating in the reverse direction, the `animation-timing-function` is reversed.

Delaying Animations

The `animation-delay` property defines how long the browser waits after the animation is attached to the element before beginning the first animation iteration.

animation-delay

Values	*<time>#*
Initial value	`0s`
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

By default, an animation begins iterating as soon as it is applied to the element, with a 0-second delay. A positive value for `animation-delay` delays the start of the animation until the time listed as the value of the property has elapsed.

Negative values for `animation-delay` are allowed and create interesting effects. A negative delay will execute the animation immediately but will begin animating the element partway through the attached animation. For example, if `animation-delay: -4s` and `animation-duration: 10s` are set on an element, the animation will begin immediately but will start approximately 40% of the way through the first animation, and will end 6 seconds later.

We say *approximately* because the animation will not necessarily start at precisely the 40% keyframe block: when the 40% mark of an animation occurs depends on the value of the `animation-timing-function`. If `animation-timing-function: linear` is set, the animation state will start 40% of the way through the animation:

```
div {
  animation-name: move;
  animation-duration: 10s;
  animation-delay: -4s;
  animation-timing-function: linear;
}

@keyframes move {
  from {
    transform: translateX(0);
  }
  to {
    transform: translateX(1000px);
  }
}
```

In this `linear` animation example, we have a 10-second animation with a delay of –4 seconds. In this case, the animation will start immediately 40% of the way through the animation, with the `<div>` translated 400 pixels to the right of its original position, and last only 6 seconds. ▶

If an animation is set to occur 10 times, with a delay of –600 milliseconds and an animation duration of 200 milliseconds, the element will start animating right away, at the beginning of the fourth iteration:

```
.ball {
  animation-name: bounce;
  animation-duration: 200ms;
  animation-delay: -600ms;
  animation-iteration-count: 10;
  animation-timing-function: ease-in;
  animation-direction: alternate;
}
@keyframes bounce {
  from {
    transform: translateY(0);
  }
  to {
    transform: translateY(500px);
  }
}
```

Instead of animating for 2,000 milliseconds (200 ms × 10 = 2,000 ms, or 2 seconds), starting in the normal direction, the ball will animate for 1,400 milliseconds (or 1.4 seconds) with the animation starting immediately—but at the start of the fourth iteration, *and* in the reverse direction.

The animation starts out in reverse because `animation-direction` is set to `alternate`, meaning every even-numbered iteration proceeds from the 100% keyframe to the 0% keyframe. The fourth iteration, which is an even-numbered iteration, is the first visible iteration. ▶

In this case, the animation will throw the `animationstart` event immediately. The `animationend` event will occur at the 1,400-millisecond mark. The ball will be tossed up, rather than bounced, throwing six `animationiteration` events, after 200, 400, 600, 800, 1,000, and 1,200 milliseconds. While the iteration count is set to 10, we get only six `animationiteration` events because we are getting only seven iterations; three iterations don't occur because of the negative `animation-delay`, and the last iteration concludes at the same time as the `animationend` event. Remember, when an `animationiteration` event would occur at the same time as an `animationend` event, the `animationiteration` event does not occur.

Let's take a deeper look at animation events before continuing.

Exploring Animation Events

The three types of animation events are `animationstart`, `animationiteration`, and `animationend`. Each event has three read-only properties: `animationName`, `elapsedTime`, and `pseudoElement`.

The `animationstart` event fires at the start of the animation: after the `animation-delay` (if present) has expired, or immediately if no delay is set. If a negative `animation-delay` value is present, the `animationstart` will fire immediately, with an `elapsedTime` equal to the absolute value of the delay.

The `animationend` event fires when the animation finishes. If the `animation-iteration-count` is set to `infinite`, then as long as the `animation-duration` is set to a time greater than `0`, the `animationend` event will never fire. If the `animation-duration` is set or defaults to 0 seconds, even when the iteration count is infinite, `animationstart` and `animationend` will occur virtually simultaneously, and in that order. These are illustrated in the following code:

```
.noAnimationEnd {
    animation-name: myAnimation;
    animation-duration: 1s;
    animation-iteration-count: infinite;
}
.startAndEndSimultaneously {
    animation-name: myAnimation;
    animation-duration: 0s;
    animation-iteration-count: infinite;
}
```

The `animationiteration` event fires *between* iterations. The `animationend` event ▶ fires at the conclusion of iterations that do not occur at the same time as the conclusion of the animation itself; thus, the `animationiteration` and `animationend` events do *not* fire simultaneously:

```
.noAnimationIteration {
    animation-name: myAnimation;
    animation-duration: 1s;
    animation-iteration-count: 1;
}
```

In the `.noAnimationIteration` example, with the `animation-iteration-count` set to a single occurrence, the animation ends at the conclusion of the first and only iteration. Whenever the `animationiteration` event would occur at the same time as an `animationend` event, the `animationend` event occurs but the `animationiteration` event does not.

When the `animation-iteration-count` property is omitted, or when its value is `1` or less, no `animationiteration` event will be fired. As long as an iteration finishes (even if it's a partial iteration) and another iteration begins, if the duration of that subsequent iteration is greater than `0s`, an `animationiteration` event will be fired:

```
.noAnimationIteration {
    animation-name: myAnimation;
    animation-duration: 1s;
    animation-iteration-count: 4;
    animation-delay: -3s;
}
```

When an animation iterates through fewer cycles than listed in the `animation-iteration-count` because of a negative `animation-delay`, there are no `animationiteration` events for the cycles that didn't occur. The preceding example code has no `animationiteration` events, as the first three cycles do not occur (because of the `-3s` `animation-delay`), and the last cycle finishes at the same time the animation ends. ▶

In that example, the `elapsedTime` on the `animationstart` event is `3`, as it is equal to the absolute value of the delay.

Animation chaining

You can use `animation-delay` to chain animations together so the next animation starts immediately after the conclusion of the preceding animation:

```
.rainbow {
    animation-name: red, orange, yellow, blue, green;
    animation-duration: 1s, 3s, 5s, 7s, 11s;
    animation-delay: 3s, 4s, 7s, 12s, 19s;
}
```

In this example, the `red` animation starts after a 3-second delay and lasts 1 second, meaning the `animationend` event occurs at the 4-second mark. This example starts each subsequent animation at the conclusion of the previous animation. This is known as *CSS animation chaining*. ▶

By including a 4-second delay on the second animation, the `orange` animation will begin interpolating the `@keyframe` property values at the 4-second mark, starting the `orange` animation immediately at the conclusion of the `red` animation. The `orange` animation concludes at the 7-second mark—it lasts 3 seconds, starting after a 4-second delay—which is the delay set on the third, or `yellow`, animation, making the `yellow` animation begin immediately after the `orange` animation ends.

This is an example of chaining animations on a single element. You can also use the `animation-delay` property to chain the animations for different elements:

```
li:first-of-type {
    animation-name: red;
    animation-duration: 1s;
    animation-delay: 3s;
}
li:nth-of-type(2) {
    animation-name: orange;
    animation-duration: 3s;
    animation-delay: 4s;
}
li:nth-of-type(3)  {
    animation-name: yellow;
    animation-duration: 5s;
    animation-delay: 7s;
}
li:nth-of-type(4) {
    animation-name: green;
    animation-duration: 7s;
    animation-delay: 12s;
}
li:nth-of-type(5) {
    animation-name: blue;
    animation-duration: 11s;
    animation-delay: 19s;
}
```

If you want a group of list items to animate in order, ▶ appearing as if the animations were chained in sequence, the `animation-delay` of each list item should be the combined time of the `animation-duration` and `animation-delay` of the previous animation.

While you can use JavaScript and the `animationend` event from one animation to determine when to attach a subsequent animation, which we discuss shortly, the `animation-delay` property is an appropriate method of using CSS animation properties to chain animations. There is one caveat: animations are the lowest priority on the UI thread. Therefore, if you have a script running that is occupying the UI thread, depending on the

browser and which properties are being animated and what property values are set on the element, the browser may let the delays expire while waiting until the UI thread is available before starting more animations.

Animation Performance

Some, but not all, animations take place on the UI thread. In most browsers, when opacity or transforms are being animated, the animation takes place on the graphics processing unit (GPU) instead of the central processing unit (CPU), and doesn't rely on the UI thread's availability. If those properties are not part of the animation, the unavailability of the UI thread can lead to visual stutters (sometimes called *jank*):

```
/* Don't do this */
* {
    transform: translateZ(0);
}
```

Putting an element into 3D space by using 3D transforms (see Chapter 17) moves that element into its own layer, allowing for jank-free animations. For this reason, the `translateZ` hack—the thing we just told you not to do—became overused, and led to the creation of the `will-change` property (see "Using the will-change Property" on page 954 for more).

While putting a few elements onto their own layers with this hack is OK, some devices have limited video memory. Each independent layer you create uses video memory and takes time to move from the UI thread to the composited layer on the GPU. The more layers you create, the higher the performance cost.

For improved performance, whenever possible, include `transform` and `opacity` in your animations rather than `top`, `left`, `bottom`, `right`, and `visibility`. Not only does it improve performance by using the GPU over the CPU, but when you change box-model properties, the browser needs to reflow and repaint, which is bad for performance. Just don't put everything on the GPU, or you'll run into different performance issues.

If you are able to rely on JavaScript, another way of chaining animations is listening for `animationend` events to start subsequent animations: ▶

```
<script>
  document.querySelectorAll('li')[0].addEventListener( 'animationend',
    () => {
        document.querySelectorAll('li')[1].style.animationName = 'orange';
    },
    false );

  document.querySelectorAll('li')[1].addEventListener( 'animationend',
    () => {
        document.querySelectorAll('li')[2].style.animationName = 'yellow';
```

```
    },
    false );

  document.querySelectorAll('li')[2].addEventListener( 'animationend',
    () => {
        document.querySelectorAll('li')[3].style.animationName = 'green';
    },
    false );

  document.querySelectorAll('li')[3].addEventListener( 'animationend',
    () => {
        document.querySelectorAll('li')[4].style.animationName = 'blue';
    },
    false );
</script>

<style>
  li:first-of-type {
    animation-name: red;
    animation-duration: 1s;
  }
  li:nth-of-type(2) {
    animation-duration: 3s;
  }
  li:nth-of-type(3)  {
    animation-duration: 5s;
  }
  li:nth-of-type(4) {
    animation-duration: 7s;
  }
  li:nth-of-type(5)  {
    animation-duration: 11s;
  }
</style>
```

In this example, there is an event handler on each of the first four list items, listening for that list item's `animationend` event. When the `animationend` event occurs, the event listeners add an `animation-name` to the subsequent list item.

As you can see in the styles, this animation chaining method doesn't employ `animation-delay` at all. Instead, the JavaScript event listeners attach animations to each element by setting the `animation-name` property when the `animationend` event is thrown.

You'll also note that the `animation-name` is included for only the first list item. The other list items have only an `animation-duration` with no `animation-name`, and therefore no attached animations. Adding `animation-name` via JavaScript is what attaches and starts the animation, at least in this example. To start or restart an animation, the animation name must be removed and then added back—at which point all the animation properties take effect, including `animation-delay`.

Instead of writing the following:

```
<script>
  document.querySelectorAll('li')[2].addEventListener( 'animationend',
    () => {
        document.querySelectorAll('li')[3].style.animationName = 'green';
    },
    false );

  document.querySelectorAll('li')[3].addEventListener( 'animationend',
    () => {
        document.querySelectorAll('li')[4].style.animationName = 'blue';
    },
    false );
</script

<style>
  li:nth-of-type(4) {
    animation-duration: 7s;
  }
  li:nth-of-type(5)  {
    animation-duration: 11s;
  }
</style>
```

we could have written this:

```
<script>
  document.querySelectorAll('li')[2].addEventListener( 'animationend',
    () => {
        document.querySelectorAll('li')[3].style.animationName = 'green';
        document.querySelectorAll('li')[4].style.animationName = 'blue';
    },
  false );
</script>

<style>
  li:nth-of-type(4) {
    animation-duration: 7s;
  }
  li:nth-of-type(5)  {
    animation-delay: 7s;
    animation-duration: 11s;
  }
</style>
```

When the `blue` animation name is added to the fifth list item at the same time we added `green`, the delay on the fifth element takes effect at that point in time and starts expiring.

While changing the values of animation properties (other than name) on the element during an animation has no effect on the animation, removing or adding an `animation-name` does have an impact. You can't change the animation duration from `100ms` to `400ms` in the middle of an animation. You can't switch the delay from `-200ms` to `5s` once the

delay has already been applied. You can, however, stop and start the animation by removing it and reapplying it. In the preceding JavaScript example, we started the animations by applying them to the elements.

In addition, setting `display: none` on an element terminates any animation. Updating the `display` back to a visible value restarts the animation from the beginning. If `animation-delay` has a positive value, the delay will have to expire before the `animationstart` event happens and any animations occur. If the delay is negative, the animation will start midway through an iteration, exactly as it would have if the animation had been applied any other way.

Animation iteration delay

What is an animation iteration delay? Sometimes you want an animation to occur multiple times but want to wait a specific amount of time between each iteration.

While there is no such thing as an animation iteration delay property, you can employ the `animation-delay` property, incorporate delays within your keyframe declaration, or use JavaScript to fake it. The best method for faking it depends on the number of iterations, performance, and whether the delays are all equal in length.

Let's say you want your element to grow three times, but want to wait 4 seconds between each 1-second iteration. You can include the delay within your keyframe definition and iterate through it three times:

```
.animate3times {
    background-color: red;
    animation: color_and_scale_after_delay;
    animation-iteration-count: 3;
    animation-duration: 5s;
}

@keyframes color_and_scale_after_delay {
    80% {
        transform: scale(1);
        background-color: red;
    }
    80.1% {
        background-color: green;
        transform: scale(0.5);
    }
    100% {
        background-color: yellow;
        transform: scale(1.5);
    }
}
```

Note the first keyframe selector is at the 80% mark and matches the default state. ▶ This will animate your element three times: it stays in the default state for 80% of the 5-second animation (not changing for 4 seconds) and then moves from green to yellow and small

to big over the last 1 second of the animation before iterating again, stopping after three iterations.

This method works for any number of iterations of the animation. Unfortunately, it is a good solution only if the delay between each iteration is identical and you don't want to reuse the animation with any other timing, such as a delay of 6 seconds. ▶ If you want to change the delay between each iteration while not changing the duration of the change in size and color, you have to write a new `@keyframes` definition.

To enable multiple iteration delays between animations, we could create a single animation and bake the effect of three different delays into the animation keyframe definition:

```
.animate3times {
    background-color: red;
    animation: color_and_scale_3_times;
    animation-iteration-count: 1;
    animation-duration: 15s;
}

@keyframes color_and_scale_3_times {
  0%, 13.32%, 20.01%, 40%, 46.67%, 93.32% {
        transform: scale(1);
        background-color: red;
  }
    13.33%, 40.01%, 93.33% {
        background-color: green;
        transform: scale(0.5);
  }
    20%, 46.66%, 100% {
        background-color: yellow;
        transform: scale(1.5);
  }
}
```

This method may be more difficult to code and maintain, however. ▶ It works for only a single cycle of the animation. To change the number of animations or the iteration delay durations, another `@keyframes` declaration would be required. This example is even less robust than the previous one, but it does allow for different between-iteration delays.

A solution is specifically allowed in the animation specification: declare an animation multiple times, each with a different `animation-delay` value: ▶

```
.animate3times {
  animation: color_and_scale, color_and_scale, color_and_scale;
  animation-delay: 0, 4s, 10s;
  animation-duration: 1s;
}

@keyframes color_and_scale {
    0% {
        background-color: green;
        transform: scale(0.5);
```

```
        }
        100% {
            background-color: yellow;
            transform: scale(1.5);
        }
    }
```

Here, we've attached the animation three times, each with a different delay. In this case, each animation iteration concludes before the next one proceeds.

If animations overlap while they're concurrently animating, the values will be the values from the last declared animation. As is true whenever multiple animations are changing an element's property at the same time, the animation that occurs last in the sequence of animation names will override any animations occurring before it in the list of names. In declaring three `color_and_scale` animations but at different intervals, the value of the property of the last iteration of the `color_and_scale` animation will override the values of the previous ones that haven't yet concluded. ▶

The safest, most robust, and most cross-browser-friendly method of faking an animation iteration delay property is to use JavaScript's animation events. Detach the animation from the element on `animationend`, and then reattach it after the iteration delay. If all the iteration delays are the same, you can use `setInterval`; if they vary, use `setTimeout`:

```
let iteration = 0;
const el = document.getElementById('myElement');

el.addEventListener('animationend', () => {
  let time = ++iteration * 1000;

  el.classList.remove('animationClass');

  setTimeout( () => {
    el.classList.add('animationClass');
  }, time);

});
```

Changing the Internal Timing of Animations

All right! The scripting was fun, but let's get back to straight CSS and talk about timing functions. Similar to the `transition-timing-function` property, the `animation-timing-function` property describes how the animation will progress from one keyframe to the next.

animation-timing-function

Values	[ease | linear | ease-in | ease-out | ease-in-out | step-start | step-end | steps(<*integer*>, start) | steps(<*integer*>, end) | cubic-bezier(<*number*>, <*number*>, <*number*>, <*number*>)]#
Initial value	ease
Applies to	All elements, ::before and ::after pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

Other than the step timing functions, described in "Using step timing functions" on page 941, the timing functions are all Bézier curves. Just like the `transition-timing-function`, the CSS specification provides for five predefined Bézier curve keywords, which we described in the preceding chapter (see Table 18-1 and Figure 18-3).

A handy tool to visualize Bézier curves and to create your own is Lea Verou's cubic Bézier visualizer.

The default `ease` has a slow start, then speeds up, and ends slowly. This function is similar to `ease-in-out`, which has a greater acceleration at the beginning. The `linear` timing function, as the name describes, creates an animation that animates at a constant speed.

The `ease-in` timing function creates an animation that is slow to start, gains speed, and then stops abruptly. The opposite `ease-out` timing function starts at full speed, then slows progressively as it reaches the conclusion of the animation iteration.

If none of these suit your needs, you can create your own Bézier curve timing function by passing four values, such as the following:

```
animation-timing-function: cubic-bezier(0.2, 0.4, 0.6, 0.8);
```

While the *x* values must be between 0 and 1, by using values for *y* that are greater than 1 or less than 0, you can create a bouncing effect, making the animation bounce up and down between values, rather than going consistently in a single direction. Consider the following timing function, whose rather outlandish Bézier curve is (partly) illustrated in Figure 19-1:

```
.snake {
  animation-name: shrink;
  animation-duration: 10s;
  animation-timing-function: cubic-bezier(0, 4, 1, -4);
  animation-fill-mode: both;
}
```

```
@keyframes shrink {
  0% {
    width: 500px;
  }
  100% {
    width: 100px;
  }
}
```

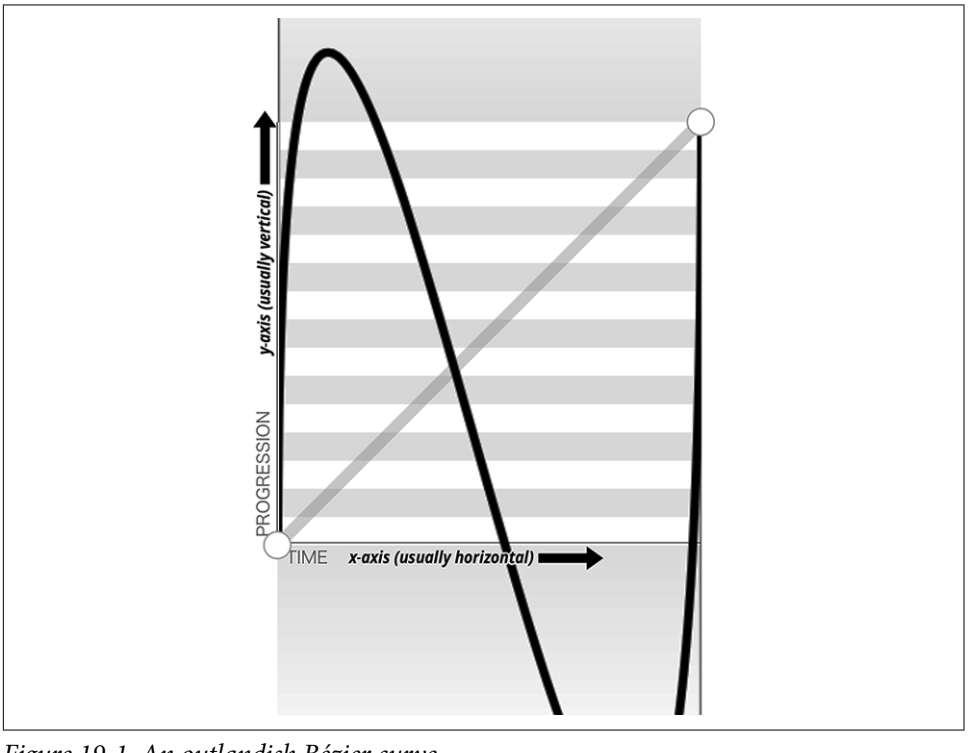


Figure 19-1. An outlandish Bézier curve

This `animation-timing-function` curve makes the animated property's values go outside the boundaries of the values set in the `0%` and `100%` keyframes. In this example, we are shrinking an element from `500px` to `100px`. However, because of the `cubic-bezier` values, the element we're shrinking will actually grow to be wider than the `500px` width defined in the `0%` keyframe and narrower than the `100px` width defined in the `100%` keyframe, as shown in Figure 19-2.

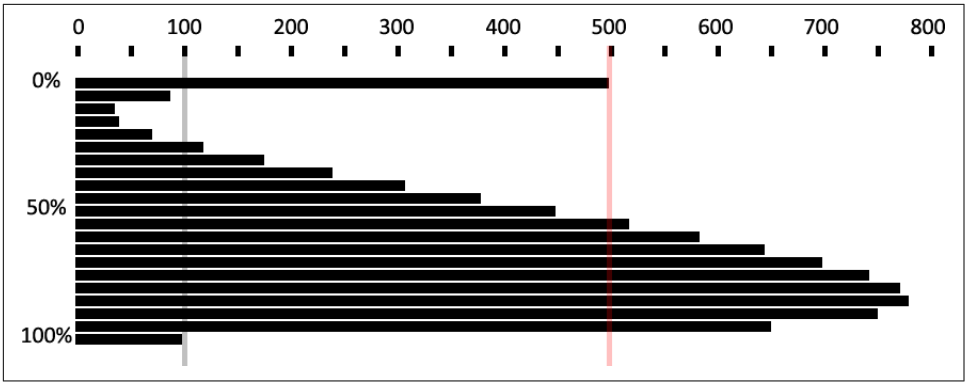


Figure 19-2. Effect of outlandish Bézier curve

In this scenario, the element starts with a width of `500px`, defined in the `0%` keyframe. It then quickly shrinks to a width of about `40px`, which is narrower than `width: 100px` defined in the `100%` keyframe. From there, it slowly expands to about `750px` wide, which is larger than the original width of `500px`. It then quickly shrinks back to `width: 100px`, ending the animation iteration. ▶

You may have realized that the curve created by our animation is the same curve as the Bézier curve. Just as the S-curve goes outside the normal bounding box, the width of the animated element goes narrower than the smaller width we set of `100px`, and wider than the larger width we set of `500px`.

The Bézier curve has the appearance of a snake because one *y* coordinate is positive, and the other negative. If both are positive values greater than 1 or both are negative values less than –1, the Bézier curve is arc-shaped, going above or below one of the values set, but not bouncing out of bounds on both ends like the S-curve.

Any timing function declared with `animation-timing-function` sets the timing for the normal animation direction, when the animation is progressing from the `0%` keyframe to the `100%` keyframe. When the animation is running in the reverse direction, from the `100%` keyframe to the `0%` keyframe, the animation timing function is reversed.

Remember the bouncing-ball example in "animation-direction" on page 926? The bouncing wasn't very realistic, because the original example defaulted to `ease` for its timing function. With `animation-timing-function`, we can apply `ease-in` to the animation so that when the ball is dropping, it gets faster as it nears its nadir at the `100%` keyframe. When it is bouncing upward, it animates in the reverse direction, from `100%` to `0%`, so the `animating-timing-function` is reversed as well—in this case, to `ease-out`—slowing down as it reaches the apex: ▶

```
.ball {
  animation-name: bounce;
  animation-duration: 1s;
  animation-iteration-count: infinite;
```

```
  animation-timing-function: ease-in;
  animation-direction: alternate;
}

@keyframes bounce {
  0% {
    transform: translateY(0);
  }
  100% {
    transform: translateY(500px);
  }
}
```

Using step timing functions

The step timing functions, `step-start`, `step-end`, and `steps()`, aren't Bézier curves. They're not curves at all. Rather, they're *tweening* definitions. The `steps()` function is most useful when it comes to character or sprite animation.

The `steps()` function divides the animation into a series of equal-length steps. The function takes two parameters: the number of steps and the change point (more on that in a moment).

The number-of-steps parameter value must be a positive integer. The animation length will be divided equally into the number of steps provided. For example, if the animation duration is 1 second and the number of steps is 5, the animation will be divided into five 200-millisecond steps, with the element being redrawn to the page five times, at 200-millisecond intervals, moving 20% through the animation at each interval.

To understand how this works, think of a flip book. Each page in a flip book contains a single drawing or picture that changes slightly from one page to the next, like one frame from a movie reel stamped onto each page. When the pages of a flip book are rapidly flipped through (hence the name), the pictures appear as an animated motion. You can create similar animations with CSS by using an image sprite, the `background-position` property, and the `steps()` timing function.

Figure 19-3 shows an image sprite containing several images that change just slightly, like the drawings on the individual pages of a flip book.



Figure 19-3. Sprite of dancing

We put all of our slightly differing images into a single image called a *sprite*. Each image in our sprite is a frame in the single animated image we're creating.

We then create a container element that is the size of a single image of our sprite, and attach the sprite as the container element's background image. We animate the

`background-position`, using the `steps()` timing function so we see only a single instance of the changing image of our sprite at a time. The number of steps in our `steps()` timing function is the number of occurrences of the image in our sprite. The number of steps defines how many stops our background image makes to complete a single animation.

The sprite in Figure 19-3 has 22 images, each 56 × 100 pixels. The total size of our sprite is 1,232 × 100 pixels. We set our container to the individual image size: 56 × 100 pixels. We set our sprite as our background image: the initial or default value of `background-position` is `top left`, which is the same as `0 0`. Our image will appear at `0 0`, which is a good default. Browsers that don't support CSS animation, like Opera Mini, will simply display the first image from our sprite:

```
.dancer {
  height: 100px;
  width: 56px;
  background-image: url(../images/dancer.png);
  ....
}
```

The trick is to use `steps()` to change the `background-position` value so that each frame is a view of a separate image within the sprite. Instead of sliding in the background image from the left, the `steps()` timing function will pop in the background image in the number of steps we declared.

So we create an animation that simply changes the left-right value of the `background-position`. The image is 1,232 pixels wide, so we move the background image from `0 0`, which is the left top, to `0 -1232px`, putting the sprite fully outside of our 56 × 100 pixel `<div>` viewport.

The values of `-1232px 0` will move the image completely to the left, outside of our containing block viewport. It will no longer show up as a background image in our 100 × 56 pixel `<div>` at the 100% mark unless `background-repeat` is set to repeat along the x-axis. We don't want that to happen!

This is what we want:

```
@keyframes dance_in_place {
  from {
      background-position: 0 0;
  }
  to {
      background-position: -1232px 0;
  }
}

.dancer {
  ....
  background-image: url(../images/dancer.png);
  animation-name: dance_in_place;
  animation-duration: 4s;
  animation-timing-function: steps(22, end);
```

```
    animation-iteration-count: infinite;
}
```

What may have seemed like a complex animation is very simple: just as in a flip book, we see one frame of the sprite at a time. Our keyframe animation simply moves the background. ▶

So that covers the first parameter, the number of steps. The second parameter takes one of a few values: `step-start`, `start`, `step-end`, `end`, `jump-none`, and `jump-both`. The given value specifies whether the change for the first step's interval takes place at the beginning or at the end of a given interval. (Chapter 18 describes these values in more detail.)

With the default value, `end`, or its equivalent `step-end`, the change take place at the end of the first step. In other words, given 200-ms step lengths, the first change in the animation will not occur until 200 ms into the animation's overall duration. With `start` or `step-start`, the first change will take place at the beginning of the first step's interval; that is to say, the instant the animation begins. Figure 19-4 provides a timeline diagram of how the two values work, based on the following styles:

```
@keyframes grayfade {
    from {background-color: #BBB;}
    to {background-color: #333;}
}

.slowfader  {animation: grayfade 1s steps(5,end);}
.quickfader {animation: grayfade 1s steps(5,start);}
```

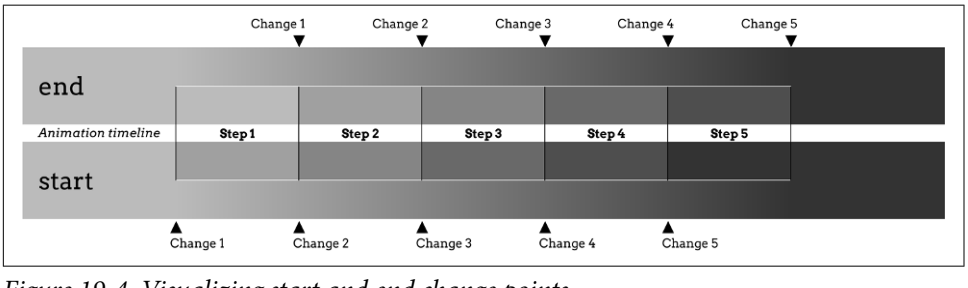


Figure 19-4. Visualizing start and end change points

The boxes embedded into each timeline represent the background color during that step interval. Notice that in the `end` timeline, the first interval is the same as the background before the animation started. This is because the animation waits until the end of the first frame to make the color change for the first step (the color between "Step 1" and "Step 2").

In the `start` timeline, on the other hand, the first interval makes that color change at the start of the interval, instantly switching from the starting background color to the color between "Step 1" and "Step 2." This is sort of like jumping ahead one interval, an impression reinforced by the fact that the background color in "Step 2" of the `end` timeline is the same as that in "Step 1" of the `start` timeline.

A similar effect can be seen at the end of each animation, where the background in the fifth step of the `start` timeline is the same as the ending background color. In the `end` timeline, it's the color at the point between "Step 4" and "Step 5," and doesn't switch to the ending background color until the end of "Step 5," when the animation is finished.

The change parameter can be hard to keep straight. If it helps, think of it this way: in a normal animation direction, the `start` value "skips" the 0% keyframe, because it makes the first change as soon as the animation starts, and the `end` value "skips" the `100%` keyframe.

The `step-start` value is equal to `steps(1, start)`, with only a single step displaying the `100%` keyframe. The `step-end` value is equal to `steps(1, end)`, which displays only the `0%` keyframe.

Animating the timing function

The `animation-timing-function` is not an animatable property, but it can be included in keyframes to alter the current timing of the animation.

Unlike animatable properties, the `animation-timing-function` values aren't interpolated over time. When included in a keyframe within the `@keyframes` definition, the timing function for the properties declared within that same keyframe will change to the new `animation-timing-function` value when that keyframe is reached, as shown in Figure 19-5:

```
.pencil {animation: W 3s infinite linear;}
@keyframes width {
  0% {
    width: 200px;
    animation-timing-function: linear;
  }
  50% {
    width: 350px;
    animation-timing-function: ease-in;
  }
  100% {
    width: 500px;
  }
}
```

In the preceding example, as shown in Figure 19-5, halfway through the animation, we switch from a linear animation progression for the `width` property to one that eases in. The `ease-in` timing starts from the keyframe in which the timing function changes.

Specifying the `animation-timing-function` within the `to` or `100%` keyframe will have no effect on the animation. When included in any other keyframe, the animation will follow the `animation-timing-function` specified in that keyframe definition until it reaches the next keyframe, overriding the element's default or declared `animation-timing-function`.

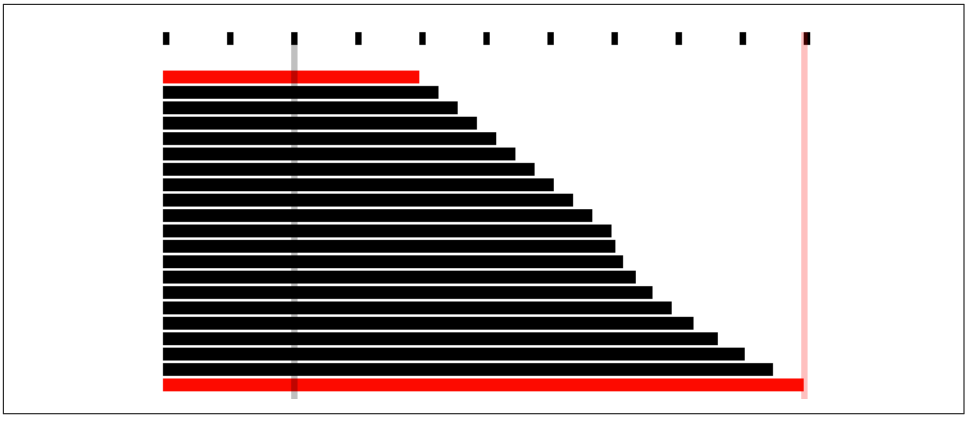


Figure 19-5. Changing the animation timing function mid-animation

If the `animation-timing-function` property is included in a keyframe, only the properties also included in that keyframe block will have their timing function impacted. The new timing function will be in play on that property until the next keyframe containing that property is reached, at which point it will change to the timing function declared within that block, or revert back to the original timing function assigned to that element. Take our `W` animation as an example:

```
@keyframes W {
    from     { left: 0; top: 0; }
    25%, 75% { top: 100%; }
    50%      { top: 50%; }
    to       { left: 100%; top: 0; }
}
```

This follows the idea that conceptually, when an animation is set on an element or pseudo-element, it is as if a set of keyframes is created for each property present in any of the keyframes, as if an animation is run independently for each property being animated. It's as if the `W` animation were made up of two animations that run simultaneously—`W_part1` and `W_part2`:

```
@keyframes W_part1 {
    from, to { top: 0; }
    25%, 75% { top: 100%; }
    50%      { top: 50%; }
}
@keyframes W_part2 {
    from { left: 0; }
    to   { left: 100%; }
}
```

The `animation-timing-function` that is set on any of the keyframes is added to the progression of only the properties defined at that keyframe:

```
@keyframes W {
    from     { left: 0; top: 0; }
    25%, 75% { top: 100%; }
    50%      { animation-timing-function: ease-in; top: 50%; }
    to       { left: 100%; top: 0; }
}
```

The preceding code will change the `animation-timing-function` from whatever was set on the CSS selector block to `ease-in` for the `top` property only, not the `left` property, impacting only the `W_part1` section of our `W` animation, and only from the middle of the animation to the 75% mark.

However, with the following animation, the `animation-timing-function` will have no effect, because it's been placed in a keyframe block that has no property-value declarations:

```
@keyframes W {
    from     { left: 0; top: 0; }
    25%, 75% { top: 100%; }
    50%      { animation-timing-function: ease-in; }
    50%      { top: 50%; }
    to       { left: 100%; top: 0; }
}
```

How is it useful to change the timing function mid-animation? In the bounce animation, we had a frictionless environment: the ball bounced forever, never losing momentum. The ball sped up as it dropped, and slowed as it rose, because the timing function was inverted from `ease-in` to `ease-out` by default as the animation proceeded from the `normal` to `reverse` direction every other iteration.

In reality, friction exists; momentum is lost. Balls will not continue to bounce indefinitely. If we want our bouncing ball to look natural, we have to make it bounce less high as it loses energy with each impact. To do this, we need a single animation that bounces multiple times, losing momentum on each bounce, while switching between `ease-in` and `ease-out` at each apex and nadir:

```
@keyframes bounce {
  0% {
    transform: translateY(0);
    animation-timing-function: ease-in;
  }
  30% {
    transform: translateY(100px);
    animation-timing-function: ease-in;
  }
  58% {
    transform: translateY(200px);
    animation-timing-function: ease-in;
  }
  80% {
    transform: translateY(300px);
    animation-timing-function: ease-in;
```

```
    }
    95% {
      transform: translateY(360px);
      animation-timing-function: ease-in;
    }
    15%, 45%, 71%, 89%, 100% {
      transform: translateY(380px);
      animation-timing-function: ease-out;
    }
  }
```

This animation loses height after a few bounces, eventually stopping. ▶

Since this new animation uses a single iteration, we can't rely on `animation-direction` to change our timing function. We need to ensure that while each bounce causes the ball to lose momentum, it still speeds up with gravity and slows down as it reaches its apex. Because we will have only a single iteration, we control the timing by including `animation-timing-function` within our keyframes. At every apex, we switch to `ease-in`, and at every nadir, or bounce, we switch to `ease-out`.

Setting the Animation Play State

If you need to pause and resume animations, the `animation-play-state` property defines whether the animation is running or paused.

animation-play-state

Values	[`running` | `paused`]#
Initial value	`running`
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

When set to the default value of `running`, the animation proceeds as normal. If set to `paused`, the animation will be, well, paused. When `paused`, the animation is still applied to the element, just frozen at the progress it had made before being paused. If stopped mid-iteration, the properties that were in the process of animating stay at their mid-iteration values. When set back to `running`, the animation restarts from where it left off, as if the "clock" that controls the animation had stopped and started again.

If the property is set to `paused` during the delay phase of the animation, the delay clock is also paused and resumes as soon as `animation-play-state` is set back to running. ▶

Animation Fill Modes

The `animation-fill-mode` property enables us to define whether an element's property values continue to be applied by the animation outside of the animation's duration time.

animation-fill-mode

Values	[`none` | `forwards` | `backwards` | `both`]#
Initial value	`none`
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

This property is useful because, by default, the changes in an animation apply only during the animation itself. Before the animation starts, the animation property values aren't applied. Once the animation is done, the values will all revert to their pre-animation values. Thus, if you take an element whose background is red, and then animate the background from green to blue, the background will (by default) stay red until the animation delay expires, and instantly revert to red after the animation finishes.

Similarly, an animation will not affect the property values of the element immediately if a positive `animation-delay` is applied. Rather, animation property values are applied when the `animation-delay` expires, at the moment the `animationstart` event is fired.

With `animation-fill-mode`, we can define how the animation impacts the element on which it is set before the `animationstart` and after the `animationend` events are fired. Property values set in the 0% keyframe can be applied to the element during the expiration of any animation delay, and property values can persist after the `animationend` event is fired.

The default value for `animation-fill-mode` is `none`, which means the animation has no effect when it is not executing. Property values from the animation's 0% keyframe (or the 100% keyframe in reverse animations) are not applied to the animated element until the `animation-delay` has expired, when the `animationstart` event is fired.

When the value is set to `backwards` and the `animation-direction` is either `normal` or `alternate`, the property values from the `0%` keyframe are applied immediately, without waiting for the `animation-delay` time to expire. If the `animation-direction` is either `reversed` or `reversed-alternate`, the property values from the `100%` keyframe are applied.

The value of `forwards` means that when the animation is done executing—that is, has concluded the last part of the last iteration as defined by the `animation-iteration-count`

value, and the `animationend` event has fired—it continues to apply the values of the properties as they were when the `animationend` event occurred. If the `iteration-count` has an integer value, this will be either the `100%` keyframe, or, if the last iteration was in the reverse direction, the `0%` keyframe.

The `both` value applies both the `backwards` effect of applying the property values as soon as the animation is attached to the element, *and* the `forwards` value of persisting the property values past the `animationend` event. ▶

If the `animation-iteration-count` is a float value, and not an integer, the last iteration will not end on the `0%` or `100%` keyframe; the animation will instead end its execution partway through an animation cycle. If the `animation-fill-mode` is set to `forwards` or `both`, the element maintains the property values it had when the `animationend` event occurred. For example, if the `animation-iteration-count` is `6.5`, and the `animation-timing-function` is linear, the `animationend` event fires and the values of the properties at the 50% mark (whether or not a 50% keyframe is explicitly declared) will stick, as if the `animation-play-state` had been set to `pause` at that point.

For example, consider the following code:

```
@keyframes move_me {
  0% {
    transform: translateX(0);
  }
  100% {
    transform: translateX(1000px);
  }
}

.moved {
  transform: translateX(0);
  animation-name: move_me;
  animation-duration: 10s;
  animation-timing-function: linear;
  animation-iteration-count: 0.6;
  animation-fill-mode: forwards;
}
```

The animation will go through only 0.6 iterations. Being a linear 10-second animation, it will stop at the 60% mark, 6 seconds into the animation, when the element is translated 600 pixels to the right. With `animation-fill-mode` set to `forwards` or `both`, the animation will stop animating when it is translated 600 pixels to the right, holding the moved element 600 pixels to the right of its original position. This will keep it translated indefinitely, or at least until the animation is detached from the element. Without the `animation-fill-mode: forwards`, the element with class `moved` will pop back to its original transform: `translateX(0)`, as defined in the moved selector code block.

Bringing It All Together

The `animation` shorthand property allows you to use one declaration, instead of eight, to define all the parameters for an element's animation. The `animation` property value is a list of space-separated values for the various longhand animation properties. If you are setting multiple animations on an element or pseudo-element, you can use a comma-separated list of animations.

animation

Values	[*<animation-name>* ‖ *<animation-duration>* ‖ *<animation-timing-function>* ‖ *<animation-delay>* ‖ *<animation-iteration-count>* ‖ *<animation-direction>* ‖ *<animation-fill-mode>* ‖ *<animation-play-state>*]#
Initial value	`0s ease 0s 1 normal none running none`
Applies to	All elements, `::before` and `::after` pseudo-elements
Computed value	As specified
Inherited	No
Animatable	No

The animation shorthand takes as its value all the other preceding animation properties, including `animation-duration`, `animation-timing-function`, `animation-delay`, `animation-iteration-count`, `animation-direction`, `animation-fill-mode`, `animation-play-state`, and `animation-name`. For example, the following two rules are precisely equivalent:

```
#animated {
  animation: 200ms ease-in 50ms 1 normal running forwards slidedown;
}
#animated {
  animation-name: slidedown;
  animation-duration: 200ms;
  animation-timing-function: ease-in;
  animation-delay: 50ms;
  animation-iteration-count: 1;
  animation-fill-mode: forwards;
  animation-direction: normal;
  animation-play-state: running;
}
```

We didn't have to declare all of the values in the animation shorthand; any values that aren't declared are set to the default or initial values. In the preceding example, three of the properties are set to their default values, so they are not strictly necessary, though

sometimes it's a good idea to write them in as a reminder to future you (or whoever takes over maintenance of your code).

The order of the shorthand is important in two specific ways. First, two time properties are permitted, for <*animation-duration*> and <*animation-delay*>. When two are listed, the first is *always* the duration. The second, if present, is interpreted as the delay.

Second, the placement of the `animation-name` is also important. If you use an animation property value as an animation name—which you shouldn't, but let's say you do—then the `animation-name` should be placed as the *last* property value in the `animation` shorthand. The first occurrence of a keyword that is a valid value for any of the other animation properties, such as `ease` or `running`, is assumed to be part of the shorthand of the animation property the keyword is associated with, rather than the `animation-name`. The following rules are equivalent:

```
#failedAnimation {
    animation: paused 2s;
}

#failedAnimation {
    animation-name: none;
    animation-duration: 2s;
    animation-delay: 0;
    animation-timing-function: ease;
    animation-iteration-count: 1;
    animation-fill-mode: none;
    animation-direction: normal;
    animation-play-state: paused;
}
```

This happens because `paused` is a valid animation name. While it may seem that the animation named `paused` with a duration of `2s` is being attached to the element, that is not what happens. Because words within the shorthand animation are first checked against possible valid values of all animation properties other than `animation-name`, `paused` is set as the value of the `animation-play-state` property. Because no recognizable animation names are found, the `animation-name` value remains at its default, `none`.

Here's another example of what you shouldn't do:

```
#anotherFailedAnimation {
    animation: running 2s ease-in-out forwards;
}

#anotherFailedAnimation {
    animation-name: none;
    animation-duration: 2s;
    animation-delay: 0s;
    animation-timing-function: ease-in-out;
    animation-iteration-count: 1;
    animation-fill-mode: forwards;
    animation-direction: normal;
```

```
    animation-play-state: running;
}
```

Here, the author probably has a keyframe animation called `running`. The browser, however, sees the term and assigns it to the `animation-play-state` property rather than the `animation-name` property. With no `animation-name` declared, no animation is attached to the element.

The way to get around this is shown here:

```
#aSuccessfulIfInadvisableAnimation {
    animation: running 2s ease-in-out forwards running;
}
```

This will apply the first `running` to `animation-play-state`, and the second `running` to `animation-name`. Again: this is *not* advised. The potential for confusion and error is too great.

In light of all this, `animation: 2s 3s 4s;` may seem valid, as if the following were being set:

```
#invalidName {
    animation-name: 4s;
    animation-duration: 2s;
    animation-delay: 3s;
}
```

But as mentioned in "Setting Up Keyframe Animations" on page 914, `4s` is *not* a valid identifier. Identifiers cannot start with a digit unless escaped. For this animation to be valid, it would have to be written as `animation: 2s 3s \4s;`.

To attach multiple animations to a single element or pseudo-element, comma-separate the animations:

```
.snowflake {
  animation: 3s ease-in 200ms 32 forwards falling,
             1.5s linear 200ms 64 spinning;
}
```

Each snowflake will fall while spinning for 96 seconds, spinning twice during each 3-second fall. ▶ At the end of the last animation cycle, the snowflake will stay fixed on the `100%` keyframe of the `falling` animation. We declared six of the eight animation properties for the `falling` animation and five for the spinning animation, separating the two animations with a comma.

While you'll most often see the animation name as the first value—it's easier to read that way, because of the issue with animation property keywords being valid keyframe identifiers—it is not a best practice. That is why we put the animation name at the end.

To sum up: using the `animation` shorthand is a fine idea. Just remember that the placements of the duration, delay, and name within that shorthand are important, and omitted values will be set to their default values.

Also note that although `none` is basically the only word that can't be a valid animation name, using any animation keywords as your identifier is never a good idea.

Animation, Specificity, and Precedence Order

In terms of specificity, the cascade, and which property values get applied to an element, animations supersede all other values in the cascade.

Specificity and !important

In general, the weight of a property attached with an ID selector `1-0-0` should take precedence over a property applied by an element selector `0-0-1`. However, if that property value is changed via a keyframe animation, the new value will be applied as if that property-value pair were added as an inline style and override the previous value.

The animation specification states, "Animations override all normal rules, but are overridden by `!important` rules." That being said, don't add `!important` to properties set inside your animation declaration block; this use is invalid, and the property-value combination to which `!important` is added will be ignored.

Animation Iteration and display: none;

If the `display` property is set to `none` on an element, any animation iterating on that element or its descendants will cease, as if the animation were detached from the element. Updating the `display` property back to a visible value will reattach all the animation properties, restarting the animation from scratch:

```
.snowflake {
  animation: spin 2s linear 5s 20;
}
```

In this case, the snowflake will spin 20 times; each spin takes 2 seconds, with the first spin starting after 5 seconds. If the snowflake element's `display` property gets set to `none` after 15 seconds, it would have completed 5 spins before disappearing (after getting through the 5-second delay, then executing 5 spins at 2 seconds each). If the snowflake `display` property changes back to anything other than `none`, the animation starts from scratch: a 5-second delay will elapse again before it starts spinning 20 times. It makes no difference how many animation cycles iterated before it disappeared from view the first time. ▶

Animation and the UI Thread

CSS animations have the *lowest* priority on the UI thread. If you attach multiple animations on page load with positive values for `animation-delay`, the delays expire as specified, but the animations may not begin until the UI thread is available to animate.

Assume the following:

- The animations all require the UI thread (that is, they aren't on the GPU as described in "Animation chaining" on page 930).
- You have 20 animations with the `animation-delay` property set to `1s`, `2s`, `3s`, `4s`, and so on in order to start each subsequent animation 1 second after the previous animation.
- The document or application takes a long time to load, with 11 seconds between the time the animated elements were drawn to the page and the time the JavaScript finished being downloaded, parsed, and executed.

Given all that, the delays of the first 11 animations will have expired once the UI thread is available, and those first 11 animations will all commence simultaneously. Each remaining animation will then begin animating at 1-second intervals.

Using the will-change Property

You could create animations so complex that they render badly, stuttering or displaying what's sometimes referred to as *jank*. In situations such as these, it may be helpful to tell the browser what needs to be animated ahead of time via the `will-change` property.

will-change

Values	`auto` | [`scroll-position` | `contents` | *`<custom-ident>`*]#
Initial value	`auto`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

The general idea here is to give the browser hints about the sorts of pre-optimizations it can make if expensive calculations may be required.



You should use `will-change` only when you have animation problems that you have been unable to resolve through other methods, such as simplifying the animation in subtle but significant ways, and that you believe pre-optimizations will address. If you try `will-change` and see no worthwhile improvement, you should remove `will-change` rather than leaving it in place.

The default value of `auto` leaves optimization work to the browser, as per usual. The `scroll-position` value says that animation of, or at least some change of, the document's scroll position is expected. By default, browsers generally take only the contents of the viewport and a little of the content to either side of it into account. The `scroll-position` value might cause the browser to bring more of the content to either side of the viewport into its layout calculations. Although this might produce smoother scroll animations, the expanded scope could easily slow the rendering of the content visible in the viewport.

With `contents`, the browser is told to expect animation of the element's contents. This is most likely to cause browsers to reduce or eliminate caching of the layout of the viewport's contents. This would require the browser to recompute the layout of the page from scratch every frame. Having to constantly recompute the page layout could slow the rendering of the page to slower than 60 frames per second, which is the benchmark that browser makers usually try to meet. On the other hand, if the contents will be changed and animated quite a lot, telling the browser to cache less can make sense. Again, try this only if you already know the animations are overtaxing the browser—never assume ahead of time.

It's also possible to tell the browser which properties to watch out for by using a *<custom-ident>*, which, in this case, is a fancy way of saying "properties." For example, if you have a complicated animation set that changes position, filter, and text shadow, and they're proving to be slow or stuttery, you could try this:

```
will-change: top, left, filter, text-shadow;
```

If this smooths out the animation, it's worth removing one property at a time to see if the smoothness remains. You might, for example, discover that removing the `top` and `left` properties doesn't affect the new smoothness, but removing either `filter` or `text-shadow` causes the stuttering to return. In that case, keep it at `will-change: filter, text-shadow`.

Also keep in mind that listing a shorthand property like `font` or `background` causes all of the longhand properties to be considered changeable. Thus, the following two rules are equivalent:

```
.textAn {will-change: font;}

.textAn {will-change: font-family, font-size, font-weight, font-style,
     font-variant, line-height;}
```

This is why, in nearly any case, a shorthand property should not be listed in `will-change`. Instead, identify the longhand properties being animated, and list those.

Printing Animations

When an animated element is printed, its end state should print. You can't see the element animating on a piece of paper; but if, for example, an animation causes an element to have a `border-radius` of `50%`, the printed element will have a `border-radius` of `50%`.

Summary

As we hope this chapter shows, animations can be powerful additions to a user interface, as well as to decorative parts of a design. Whether an animation is simple, complex, short, or lengthy, all these aspects and more are in your hands.

Always exercise caution, as animation can affect some users negatively, whether they have vestibular disorders or simply are sensitive to motion. Fortunately, `prefers-reduced-motion` is available to reduce or eliminate animations for those who do not want them.

CHAPTER 20

Filters, Blending, Clipping, and Masking

Several special properties allow authors to alter the appearance of elements with visual filters, specify different ways to visually blend elements with whatever is behind them, and alter the presentation of elements by showing parts and hiding other parts. While these may seem like disparate concepts, they all share one thing in common: they allow elements to be altered in ways that were previously difficult or impossible.

CSS Filters

CSS provides a way to apply built-in visual filter effects, as well as custom filters defined in the page or in external files, to elements by way of the `filter` property.

filter

Values	[`none` | `blur()` | `brightness()` | `contrast()` | `drop-shadow()` | `grayscale()` | `hue-rotate()` | `invert()` | `opacity()` | `sepia()` | `saturate()` | `url()`]#
Initial value	`none`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	Yes

The value syntax permits a space-separated list of filter functions, with each filter applied in sequence. Thus, given the declaration `filter: opacity(0.5) blur(1px);`, the opacity is applied to the element, and the semitransparent result is then blurred. If the order is

reversed, so too is the order of application: the fully opaque element is blurred, and the resulting blur made semitransparent.

The CSS specification talks of "input images" when discussing `filter`, but this doesn't mean `filter` is used only on images. Any HTML element can be filtered, and all graphic SVG elements can be filtered. The *input image* is a visual copy of the rendered element *before* it is filtered. Filters are applied to this input, and the final filtered result is then rendered to the display medium (e.g., the device display).

All the values permitted (save `url()`) are function values, with the permitted value types for each function being dependent on the function in question. We've grouped these functions into a few broad categories for ease of understanding.

Basic Filters

The following filters are basic in the sense that they cause the changes that their names directly describe—blurring, drop shadows, and opacity changes:

`blur( <length> )`
: Blurs the element's contents by using a Gaussian blur whose standard deviation is defined by the <*length*> value supplied, where a value of `0` leaves the element unchanged. Negative lengths are not permitted.

`opacity( [ <number> | <percentage> ] )`
: Applies a transparency filter to the element in a manner very similar to the `opacity` property, where the value `0` yields a completely transparent element and a value of `1` or `100%` leaves the element unchanged. Negative values are not permitted. Values greater than `1` and `100%` are permitted, but are clipped to be `1` or `100%` for the purposes of computing the final value.



The specification makes clear that `filter: opacity()` is *not* meant to be a replacement or shorthand for the `opacity` property, and in fact both can be applied to the same element, resulting in a sort of double-transparency.

`drop-shadow( <length>{2,3} <color>? )`
: Creates a drop shadow that matches the shape of the element's alpha channel, with a blur and using an optional color. The handling of the lengths and colors is the same as for the property `box-shadow`, which means that while the first two <*length*> values can be negative, the third (which defines the blur) cannot. Unlike box-shadow, though, the `inset` value is not permitted. To apply multiple drop shadows, provide multiple space-separated `drop-shadow()` functions; unlike `box-shadow`, comma-separated shadows don't work here. If no <*color*> value is supplied, the used color is the same as the computed value of the `color` property for the element.

Figure 20-1 shows some effects of these filter functions.

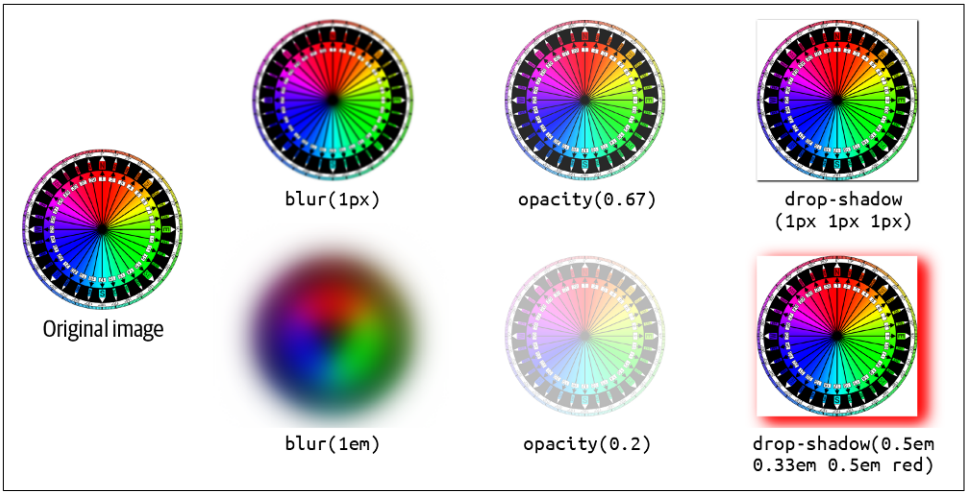


Figure 20-1. Basic filter effects

Before we go on, two things deserve further exploration. The first is how `drop-shadow()` really operates. Just by looking at Figure 20-1, it's easy to get the impression that drop shadows are bound to the element box, because of the boxlike nature of the drop shadows shown there. But that's just because the image used to illustrate filters is a PNG, which is to say a raster image, and more importantly one that doesn't have any alpha channel. The white parts of the image are opaque white, in other words.

If the image has transparent bits, `drop-shadow()` will use those in computing the shadow. To see what this means, consider Figure 20-2.

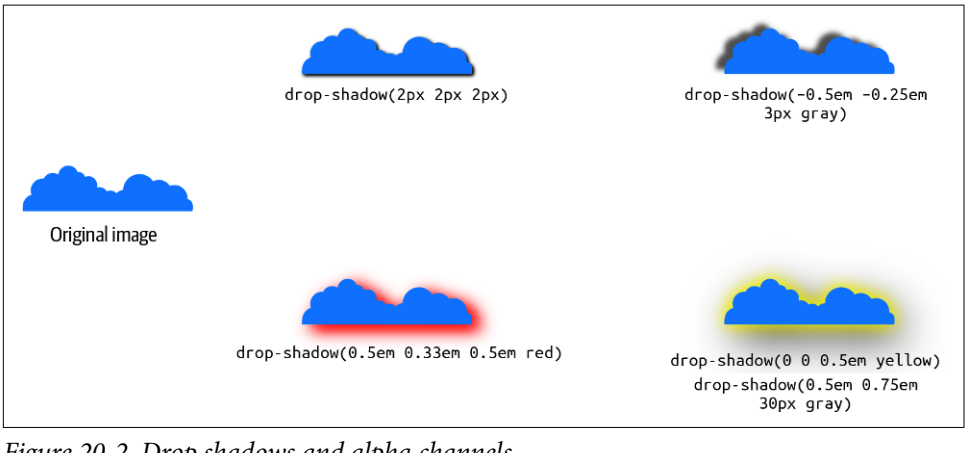


Figure 20-2. Drop shadows and alpha channels

The other thing to point out in Figure 20-2 is the last image has two drop shadows. This was accomplished as follows:

```
filter: drop-shadow(0 0 0.5em yellow) drop-shadow(0.5em 0.75em 30px gray);
```

Any number of filters can be chained together like this. To pick another example, you could write the following:

```
filter: blur(3px) drop-shadow(0.5em 0.75em 30px gray) opacity(0.5);
```

That would get you a blurry, drop-shadowed, half-opaque element. It might not be the most reader-friendly effect for text, but it's possible nonetheless. This function-chaining is possible with all `filter` functions, both those you've seen and those to come.

Color Filtering

This next set of `filter` functions alter the colors present in the element. This can be as simple as leaching out the colors, or as complex as shifting all the colors by way of an angle value.

Note that for the first three of the four following functions, all of which accept either a <*number*> or <*percentage*>, negative values are not permitted; the fourth permits positive and negative angle values:

`grayscale(` *[<number> | <percentage>]* `)`
: Alters the colors in the element to be shifted toward shades of gray. A value of `0` leaves the element unchanged, and a value of `1` or `100%` will result in black and white, as a fully grayscale element.

`sepia(` *[<number> | <percentage>]* `)`
: Alters the colors in the element to be shifted toward shades of sepia tones (sepia is the reddish-brown color used in antique photography, defined by Wikipedia to be equivalent to `#704214` or `rgba(112,66,20)` in the sRGB color space). A value of `0` leaves the element unchanged, and a value of `1` or `100%` will result in a fully sepia element.

`invert(` *[<number> | <percentage>]* `)`
: Inverts all colors in the element. Each of the R, G, and B values for a given color are inverted by subtracting them from 255 (in 0–255 notation) or from 100% (in 0%–100% notation). For example, a pixel with the color `rgb(255 128 55)` will be rendered as `rgb(0 127 200)`; a different pixel with the value `rgb(75% 57.2% 23%)` will become `rgb(25% 42.8% 77%)`. A value of `0` leaves the element unchanged, and a value of `1` or `100%` results in a fully inverted element. A value of `0.5` or `50%` stops the inversion of each color at the midpoint of the color space, leading to an element of uniform gray regardless of the input element's appearance.

`hue-rotate(` <*angle*> `)`
: Alters the colors of the image by shifting their hue angle around an HSL color wheel, leaving saturation and lightness unchanged. A value of `0deg` means no difference between the input and output images. A value of `360deg` (a full single rotation) will also present an apparently unchanged element, though the rotation-angle value is

maintained. Values above `360deg` are permitted. Negative values are also permitted, and cause a counterclockwise rotation as opposed to the clockwise rotation caused by positive values. (In other words, the rotation is "compass-style," with 0° at the top and increasing angle values in the clockwise direction.)

Examples of the preceding `filter` functions are shown in Figure 20-3, though fully appreciating them depends on a color rendering.

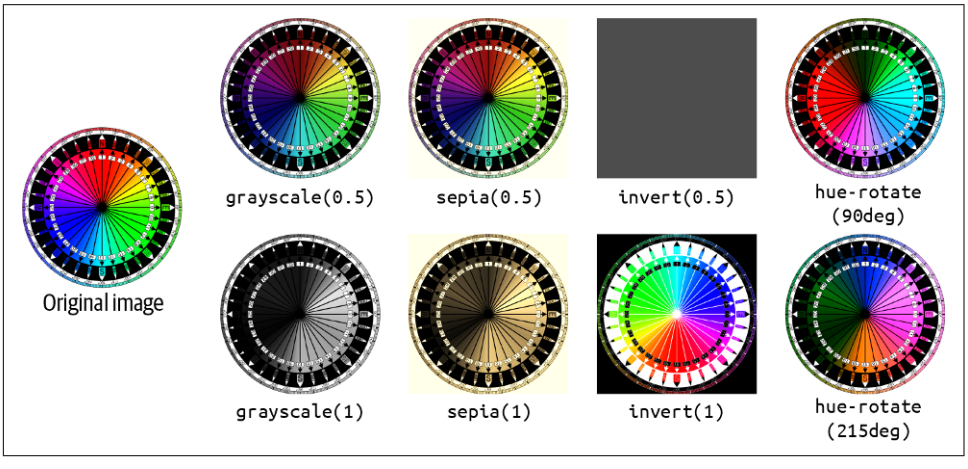


Figure 20-3. Color filter effects

Brightness, Contrast, and Saturation

While the following `filter` functions also manipulate color, they do so in closely related ways, and are a familiar grouping to anyone who's worked with images, particularly photographic images. For all these functions, values greater than `1` and `100%` are permitted, but are clipped to be `1` or `100%` for the purposes of computing the final value:

`brightness(` *[<number> | <percentage>]* `)`
: Alters the brightness of the element's colors. A value of `0` leaves the element a solid black, and a value of `1` or `100%` leaves it unchanged. Values above `1` and `100%` yield colors brighter than the input element, and can eventually reach a state of solid white.

`contrast(` *[<number> | <percentage>]* `)`
: Alters the contrast of the element's colors. The higher the contrast, the more colors are differentiated from each other; the lower the contrast, the more they converge on each other. A value of `0` leaves the element a solid gray, and a value of `1` or `100%` leaves it unchanged. Values above `1` and `100%` yield colors with greater contrast than is present in the input element.

`saturate(` *[<number> | <percentage>]* `)`
: Alters the saturation of the element's colors. The more saturated an element's colors, the more intense they become; the less saturated they are, the more muted they

appear. A value of `0` leaves the element completely unsaturated, making it effectively grayscale, whereas a value of `1` or `100%` leaves the element unchanged. Similar to `brightness()`, `saturate()` permits *and* acts upon values greater than `1` or `100%`; such values result in *supersaturation*.

Examples of the preceding `filter` functions are shown in Figure 20-4, though fully appreciating them depends on a color rendering. Also, the effects of greater-than-one values may be hard to make out in the figure, but they are present.

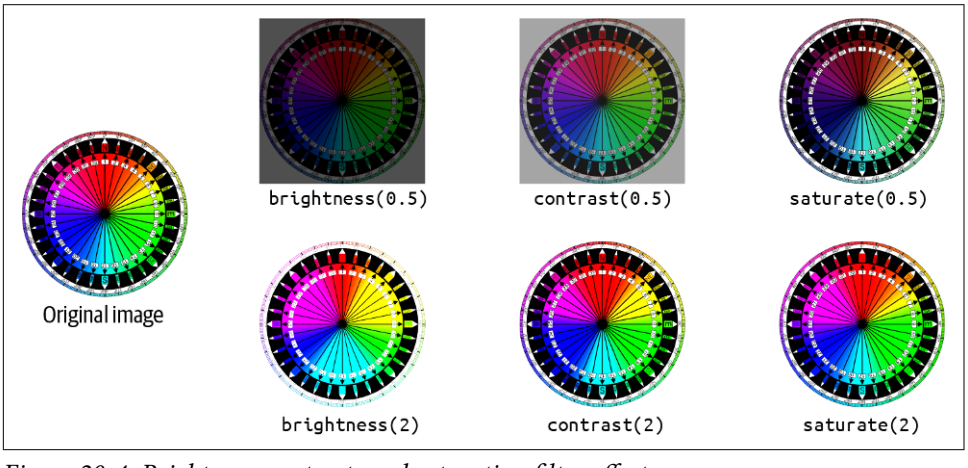


Figure 20-4. Brightness, contrast, and saturation filter effects

SVG Filters

The last `filter` value type is a function of a familiar kind: the `url()` value type. This allows you to point to a (potentially very complicated) filter defined in SVG, whether it's embedded in the document or stored in an external file.

This takes the form `url(<uri>)`, where the `<uri>` value points to a filter defined using SVG syntax, specifically the `<filter>` element. This can be a reference to a single SVG image that contains only a filter, such as `url(wavy.svg)`, or it can be a pointer to an identified filter embedded in an SVG image, such as `url(filters.svg#wavy)`. The advantage of the latter pattern is that a single SVG file can define multiple filters, thus consolidating all your filtering into one file for easy loading, caching, and referencing.

If a `url()` function points to a nonexistent file, or points to an SVG fragment that is not a `<filter>` element, the function is invalid and the *entire* function list is ignored (thus rendering the `filter` declaration invalid).

Examining the full range of filtering possibilities in SVG is well beyond the scope of this book, but let's just say that the power of the offered features is substantial. A few simple examples of SVG filtering are shown in Figure 20-5, with brief captions to indicate the kinds of operations the filters were built to create. (The actual CSS used to apply these filters looks like `filter: url(filters.svg#rough)`.)

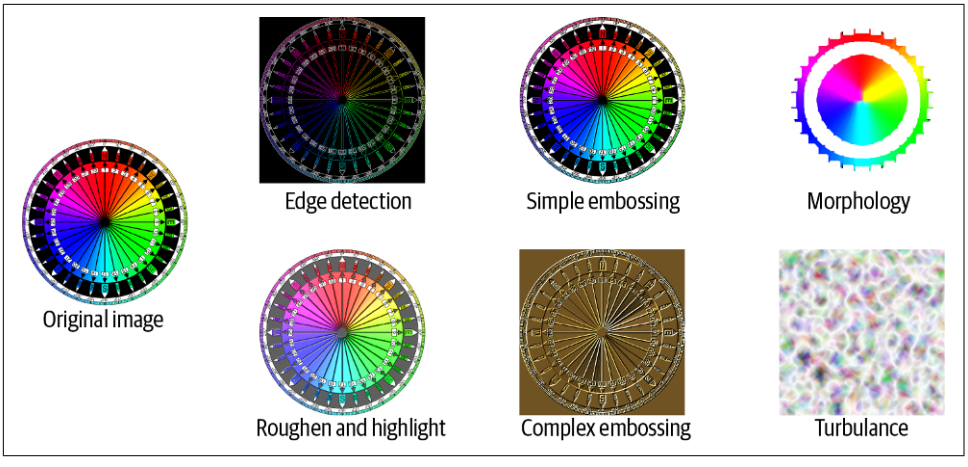


Figure 20-5. SVG filter effects

It's easily possible to put every last bit of filtering you do into SVG, including replacements for every other `filter` function you've seen. (In fact, all the other `filter` functions are defined by the specification as literal SVG filters, to give a precise rendering target for implementors.) Remember, however, that you can chain CSS functions together. Thus, you might define a specular-highlight filter in SVG, and modify it with blurring or grayscale functions as needed. For example:

```
img.logo {filter: url(/assets/filters.svg#spotlight);}
img.logo.print {filter: url(/assets/filters.svg#spotlight) grayscale(100%);}
img.logo.censored {filter: url(/assets/filters.svg#spotlight) blur(3px);}
```

Always keep in mind that the filter functions are applied in order. That's why the `grayscale()` and `blur()` functions come after the `url()`-imported spotlight filter. If they were reversed, the logos would be made grayscale or blurred first, and then have the spotlight filter applied afterward.

Compositing and Blending

In addition to filtering, CSS enables you to determine how elements are *composited* together. Take, for example, two elements that partially overlap because of positioning. By default, the element in front, if fully opaque, completely obscures the one behind, wherever they overlap. If the one in front is semitransparent, the element in back is partially visible.

This is sometimes called *simple alpha compositing*, in that you can see whatever is behind an element as long as some (or all) of it has alpha channel values less than `1`. Think of how you can see the background through an element with `opacity: 0.5`, or in the areas of a PNG or GIF that are set to be transparent. That's simple alpha compositing.

But if you're familiar with image-editing programs like Photoshop or GIMP, you know that overlapping image layers can be blended together in a variety of ways. CSS has the same ability. CSS has two blending strategies (at least as of late 2022): blending entire elements with whatever is behind them, and blending the background layers of a single element together. While similar to filter effects in many ways, blending mode values are predefined—they don't accept a parameter—and while both filter effects and blend modes support multiple values, the properties that support blend modes use a comma-separated list of values instead of a space-separated list. (This inconsistency in value syntaxes is rooted deep in the history of CSS, and is something we just have to live with for the time being.)

Blending Elements

If elements overlap, you can change the way they blend together by using the `mix-blend-mode` property.

mix-blend-mode

Values	`normal` | `multiply` | `screen` | `overlay` | `darken` | `lighten` | `color-dodge` | `color-burn` | `hard-light` | `soft-light` | `difference` | `exclusion` | `hue` | `saturation` | `color` | `luminosity`
Initial value	`normal`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

The CSS specification indicates that this property "defines the formula that must be used to mix the colors with the backdrop." The element is blended with whatever is behind it (the "backdrop"), whether that's pieces of another element, or just the background of an ancestor element such as the `<body>`.

The default value, `normal`, shows the element's pixels as is, without any mixing with the backdrop, except where the alpha channel is less than `1`. This is the simple alpha compositing mentioned previously. It's what we're all used to, which is why it's the default value. Figure 20-6 shows a few examples.

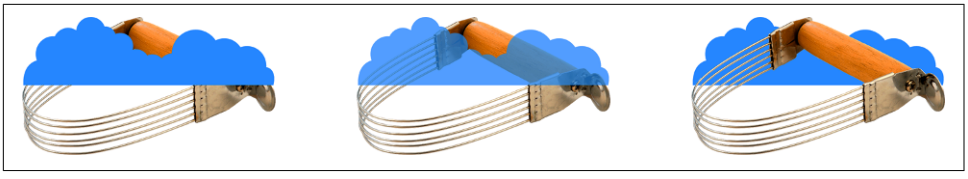


Figure 20-6. Simple alpha-channel blending

For the rest of the `mix-blend-mode` keywords, we've grouped them into a few categories. Let's also nail down a few definitions we'll be using in the blend mode descriptions:

Foreground
: The element that has `mix-blend-mode` applied to it.

Backdrop
: Whatever is behind an element. This can be other elements, the background of an ancestor element, and so on.

Pixel component
: The color component of a given pixel: R, G, and B.

If it helps, think of the foreground and backdrop as layers atop each other in an image-editing program. With `mix-blend-mode`, you can change the blend mode applied to the top element (the foreground).

Darken, lighten, difference, and exclusion

The following blend modes might be called *simple-math modes*—they achieve their effect by directly comparing values in some way, or using simple addition and subtraction to modify pixels:

`darken`
: Each pixel in the foreground is compared with the corresponding pixel in the backdrop, and for each of the R, G, and B values (the pixel components), the smaller of the two is kept. Thus, if the foreground pixel has a value corresponding to `rgb(91 164 22)` and the backdrop pixel is `rgb(102 104 255)`, the resulting pixel will be `rgb(91 104 22)`.

`lighten`
: This blend is the inverse of `darken`: when comparing the R, G, and B components of a foreground pixel and its corresponding backdrop pixel, the larger of the two values is kept. Thus, if the foreground pixel has a value corresponding to `rgb(91 164 22)` and the backdrop pixel is `rgb(102 104 255)`, the resulting pixel will be `rgb(102 164 255)`.

`difference`
: The R, G, and B components of each pixel in the foreground are compared to the corresponding pixel in the backdrop, and the absolute value of subtracting one from the other is the final result. Thus, if the foreground pixel has a value corresponding to `rgb(91 164 22)` and the backdrop pixel is `rgb(102 104 255)`, the resulting pixel will be `rgb(11 60 233)`. If one of the pixels is white, the resulting pixel will be the inverse of the nonwhite pixel. If one of the pixels is black, the result will be exactly the same as the nonblack pixel.

`exclusion`
: This blend is a milder version of `difference`. Rather than $|back - fore|$, the formula is $back + fore - (2 \times back \times fore)$, where *back* and *fore* are values in the range 0 to 1. For example, an exclusion calculation of an orange (`rgb(100% 50% 0%)`) and a medium gray (`rgb(50% 50% 50%)`) will yield `rgb(50% 50% 50%)`. For the green component, as an example, the math is $0.5 + 0.5 - (2 \times 0.5 \times 0.5)$, which reduces to 0.5, corresponding to `50%`. Compare this to `difference`, where the result would be `rgb(50% 0% 50%)`, since each component is the absolute value of subtracting one from the other.

This last definition highlights that, for all blend modes, the actual values being operated on are in the range 0–1. The previous examples showing values like `rgb(11 60 233)` are normalized from the 0–1 range. In other words, given the example of applying the `difference` blend mode to `rgb(91 164 22)` and `rgb(102 104 255)`, the actual operation is as follows:

1. `rgb(91 164 22)` is $R = 91 \div 255 = 0.357$; $G = 164 \div 255 = 0.643$; $B = 22 \div 255 = 0.086$. Similarly, `rgb(102 104 255)` corresponds to $R = 0.4$; $G = 0.408$; $B = 1$.
2. Each component is subtracted from the corresponding component, and the absolute value taken. Thus, $R = |0.357 - 0.4| = 0.043$; $G = |0.643 - 0.408| = 0.235$; $B = |1 - 0.086| = 0.914$. This could be expressed as `rgba(4.3% 23.5% 91.4%)`, or (by multiplying each component by 255) as `rgb(11 60 233)`.

From all this, you can perhaps understand why the full formulas are not written out for every blend mode we cover. If you're interested in the fine details, each blend mode's formula is provided in the "Compositing and Blending Level 2" specification.

Figure 20-7 depicts examples of the blend modes in this section.

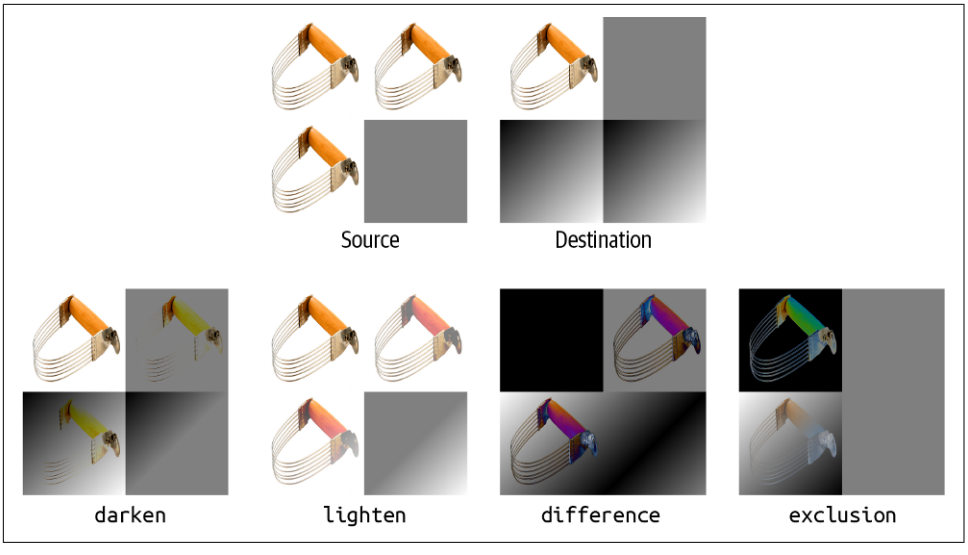


Figure 20-7. Darken, lighten, difference, and exclusion blending with `mix-blend-mode:` applied to the foreground image

Multiply, screen, and overlay

The following blend modes might be called the *multiplication modes*—they achieve their effect by multiplying values together:

`multiply`
: Each pixel component in the foreground is multiplied by the corresponding pixel component in the backdrop. This yields a darker version of the foreground, modified by what is underneath. This blend mode is *symmetric*, in that the result will be exactly the same even if you were to swap the foreground with the backdrop.

`screen`
: Each pixel component in the foreground is inverted (see `invert` in "Color Filtering" on page 960), multiplied by the inverse of the corresponding pixel component in the backdrop, and the result inverted again. This yields a lighter version of the foreground, modified by what is underneath. Like `multiply`, `screen` is symmetric.

`overlay`
: This blend is a combination of `multiply` and `screen`. For foreground pixel components darker than 0.5 (50%), the `multiply` operation is carried out; for foreground pixel components whose values are above 0.5, `screen` is used. This makes the dark areas darker, and the light areas lighter. This blend mode is *not* symmetric, because swapping the foreground for the backdrop would mean a different pattern of light and dark, and thus a different pattern of multiplying versus screening.

Figure 20-8 depicts examples of these blend modes.

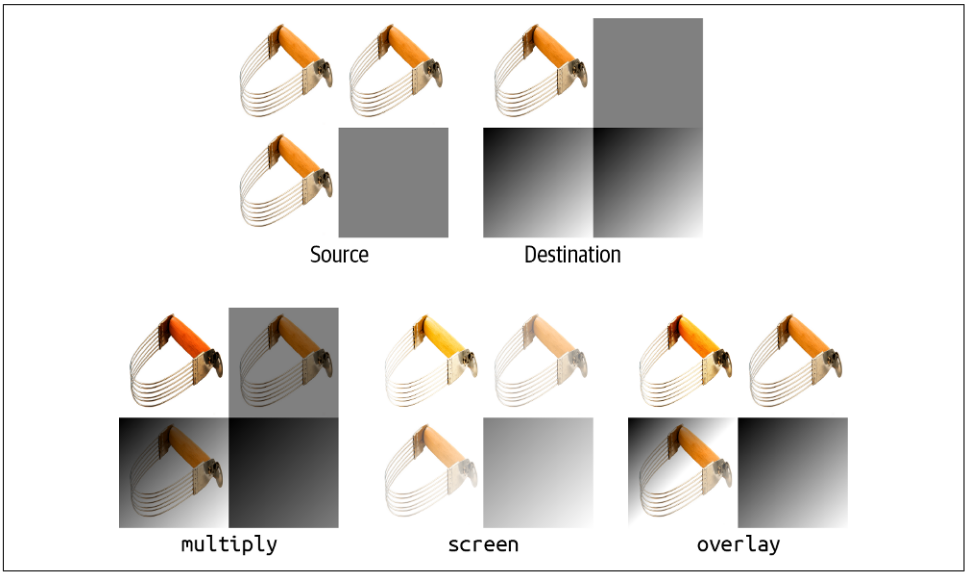


Figure 20-8. Images with `mix-blend-mode` property set showing multiply, screen, and overlay blending

Hard and soft light

The following blend modes are covered here because the first is closely related to a previous blend mode, and the other is just a muted version of the first:

`hard-light`
: This blend is the inverse of `overlay` blending. Like `overlay`, it's a combination of `multiply` and `screen`, but the determining layer is the backdrop. Thus, for backdrop pixel components darker than 0.5 (50%), the `multiply` operation is carried out; for backdrop pixel components lighter than 0.5, `screen` is used. This makes it appear somewhat as if the foreground is being projected onto the backdrop with a projector that employs a harsh light.

`soft-light`
: This blend is a softer version of `hard-light`. This mode uses the same operation but is muted in its effects. The intended appearance is as if the foreground is being projected onto the backdrop with a projector that employs a diffuse light.

Figure 20-9 depicts examples of these blend modes.

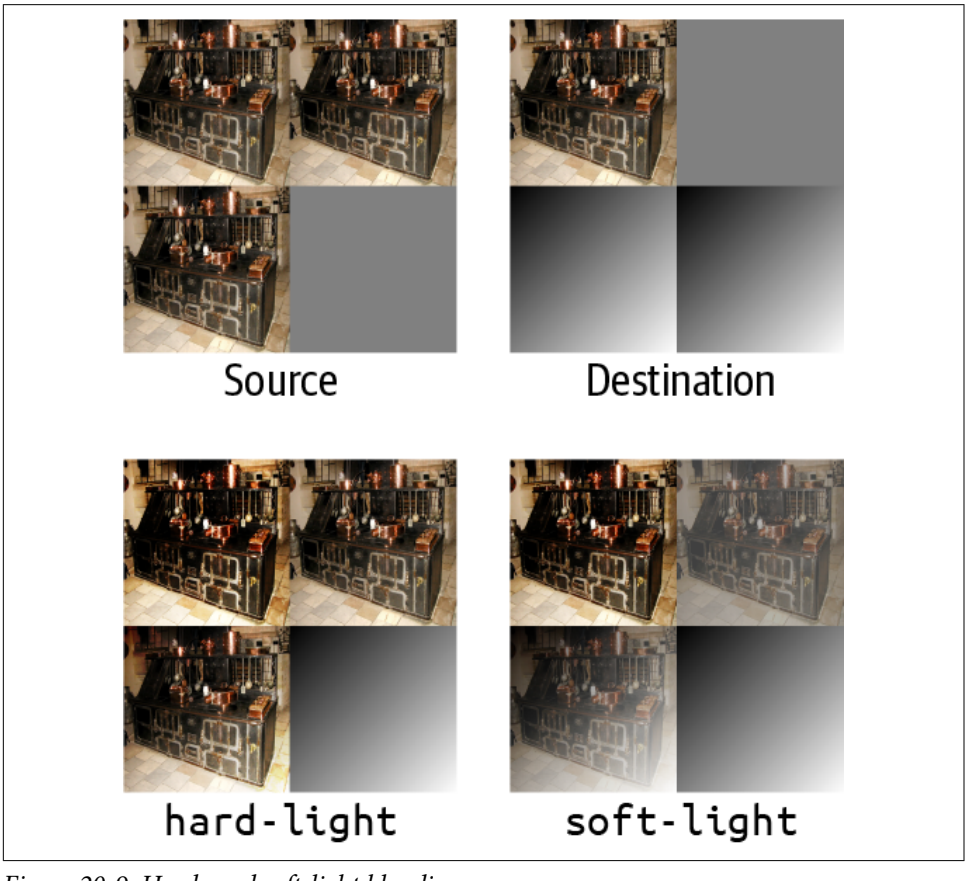


Figure 20-9. Hard- and soft-light blending

Color dodge and burn

Color dodging and burning—terms that come from old darkroom techniques performed on chemical film stock—are meant to lighten or darken a picture with a minimum of change to the colors themselves. These modes are as follows:

`color-dodge`
: Each pixel component in the foreground is inverted, and the component of the corresponding backdrop pixel component is divided by the inverted foreground value. This yields a brightened backdrop unless the foreground value is `0`, in which case the backdrop value is unchanged.

`color-burn`
: This blend is a reverse of `color-dodge`: each pixel component in the backdrop is inverted, the inverted backdrop value is divided by the unchanged value of the corresponding foreground pixel component, and the result is then inverted. This yields a result where the darker the backdrop pixel, the more its color will burn through the foreground pixel.

Figure 20-10 depicts examples of these blend modes.

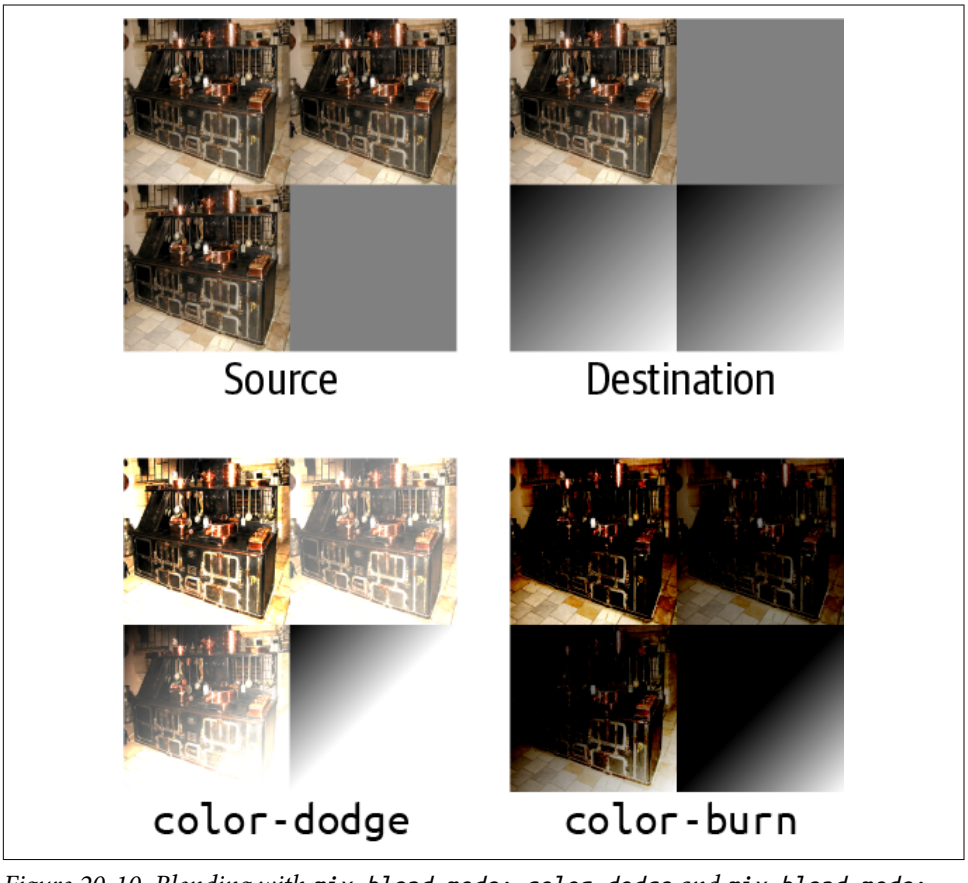


Figure 20-10. Blending with `mix-blend-mode: color-dodge` and `mix-blend-mode: color-burn`

Hue, saturation, luminosity, and color

The final four blend modes are different from those we've shown before, because they do *not* perform operations on the R/G/B pixel components. Instead, they perform operations to combine the hue, saturation, luminosity, and color of the foreground and backdrop in different ways. These modes are as follows:

`hue`
: For each pixel, combines the luminosity and saturation levels of the backdrop with the hue angle of the foreground.

`saturation`
: For each pixel, combines the hue angle and luminosity level of the backdrop with the saturation level of the foreground.

`color`
: For each pixel, combines the luminosity level of the backdrop with the hue angle and saturation level of the foreground.

`luminosity`
: For each pixel, combines the hue angle and saturation level of the backdrop with the luminosity level of the foreground.

Figure 20-11 depicts examples of these blend modes.

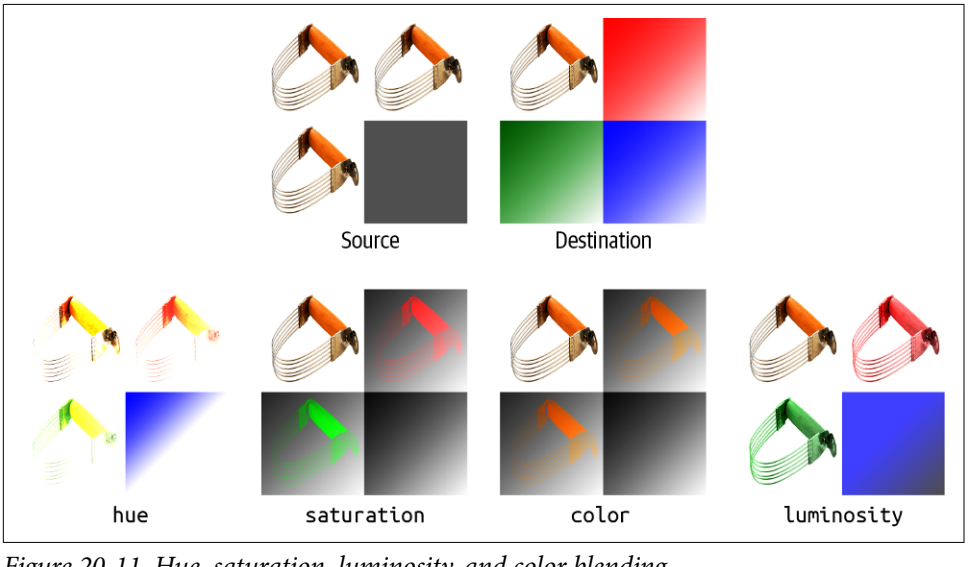


Figure 20-11. Hue, saturation, luminosity, and color blending

These blend modes can be a lot harder to grasp without busting out raw formulas, and even those can be confusing if you aren't familiar with how things like saturation and luminosity levels are determined. If you don't feel like you quite have a handle on how these modes work, the best solution is to practice with a bunch of images and simple color patterns.

Two points to note:

- Remember that an element always blends with its backdrop. If there are other elements behind an element, it will blend with them; if there's a patterned background on the parent element, the blending will be done against that pattern.
- Changing the opacity of a blended element will change the outcome, though not always in the way you might expect. For example, if an element with `mix-blend-mode: difference` is also given `opacity: 0.8`, the difference calculations will be scaled by 80%. More precisely, a scaling factor of 0.8 will be applied to the color-value calculations. This can cause some operations to trend toward flat middle gray and others to shift the color changes.

Blending Backgrounds

Blending an element with its backdrop is one thing, but what if an element has multiple background images that overlap and also need to be blended together? That's where `background-blend-mode` comes in.

background-blend-mode

Values	[`normal` | `multiply` | `screen` | `overlay` | `darken` | `lighten` | `color-dodge` | `color-burn` | `hard-light` | `soft-light` | `difference` | `exclusion` | `hue` | `saturation` | `color` | `luminosity`]#
Initial value	`normal`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

We won't go through an exhaustive list of all the blend modes and what they mean, because we did that in "Blending Elements" on page 964. What they meant there, they mean here.

The difference is that when it comes to blending multiple background images, they're blended with one other against an empty background—that is, a completely transparent, uncolored backdrop. They do *not* blend with the backdrop of the element, except as directed by `mix-blend-mode`. To see what that means, consider the following:

```
#example {background-image:
        url(star.svg),
        url(diamond.png),
        linear-gradient(135deg, #F00, #AEA);
    background-blend-mode: color-burn, luminosity, darken;}
```

Here we have three background images, each with its own blend mode. These are blended together into a single result, shown in Figure 20-12.

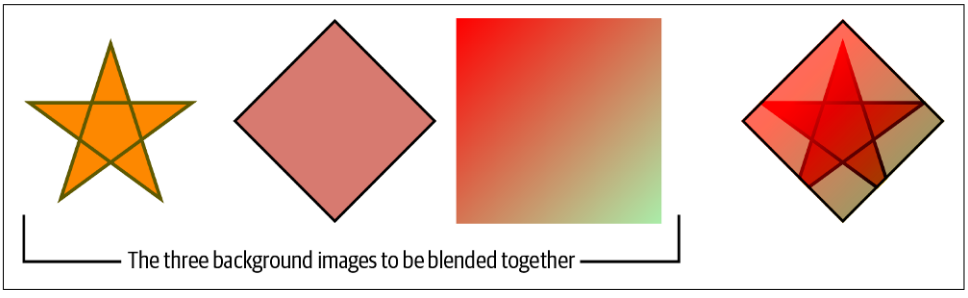


Figure 20-12. Three backgrounds blended together

So far, fine. Here's the kicker: the result will be the same regardless of what might appear behind the element. We can change the parent's background to white, gray, fuchsia, or a lovely pattern of repeating gradients, and in every case those three blended backgrounds will look exactly the same, pixel for pixel. They're blended in *isolation*, a term we'll return to shortly. We can see the previous example (Figure 20-12) sitting atop a variety of backgrounds in Figure 20-13.

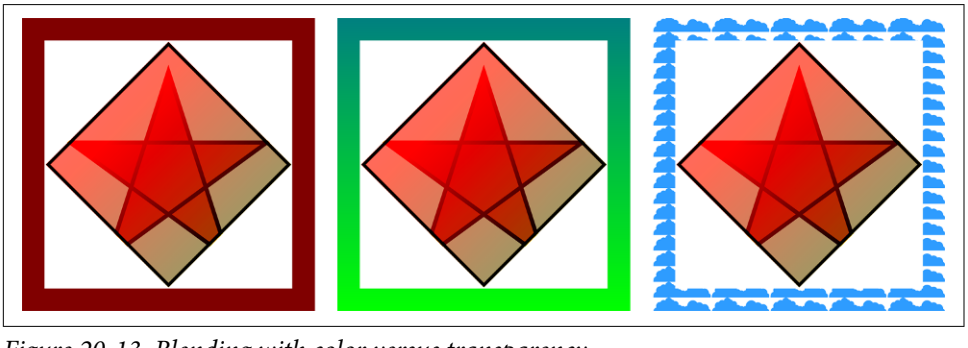


Figure 20-13. Blending with color versus transparency

Like multiple blended elements stacked atop one another, the blending of background layers works from the back to the front. Thus, if you have two background images over a solid background color, the background layer in the back is blended with the background color, and then the frontmost layer is blended with the result of the first blend. Consider the following:

```
.bbm {background-image:
        url(star.svg),
        url(diamond.png);
    background-color: goldenrod;
    background-blend-mode: color-burn, luminosity;}
```

Given these styles, *diamond.png* is blended with the background color `goldenrod` using the `luminosity` blend. Once that's done, *star.svg* is blended with the results of the diamond-goldenrod blend using a `color-burn` blend.

Although it's true that the background layers are blended in isolation, they're also part of an element that may have its own blending rules via `mix-blend-mode`. Thus, the final result of the isolated background blend may be blended with the element's backdrop after all. Given the following styles, the first example's background will sit atop the element's backdrop, but the rest will end up blended with it in some fashion, as illustrated in Figure 20-14:

```
.one {mix-blend-mode: normal;}
.two {mix-blend-mode: multiply;}
.three {mix-blend-mode: darken;}
.four {mix-blend-mode: luminosity;}
.five {mix-blend-mode: color-dodge;}
```

```
<div class="bbm one"></div>
<div class="bbm two"></div>
<div class="bbm three"></div>
<div class="bbm four"></div>
<div class="bbm five"></div>
```

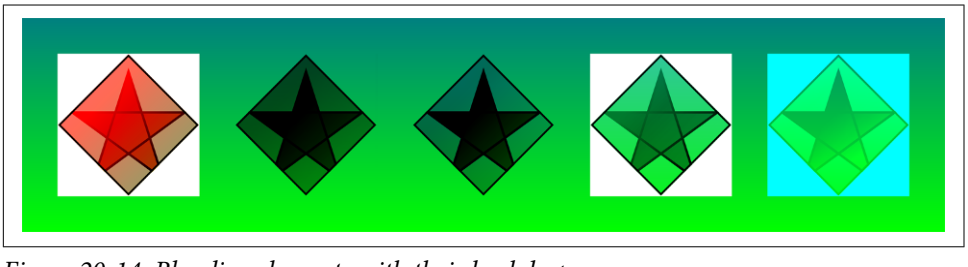


Figure 20-14. Blending elements with their backdrops

Throughout this section, we've touched on the concept of blending in isolation as a thing that backgrounds naturally do. Elements, on the other hand, do not naturally blend in isolation. As you'll see next, that behavior can be changed.

Blending in Isolation

Sometimes you might want to blend multiple elements together, but in a group of their own, in the same way background layers on an element are blended. This is, as you've seen, called blending in *isolation*. If that's what you're after, the `isolation` property is for you.

isolation

Values	`auto` | `isolate`
Initial value	`auto`
Applies to	All elements (in SVG, it applies to container elements, graphics elements, and graphics-referencing elements)
Computed value	As declared
Inherited	No
Animatable	No

This pretty much does exactly what it says: it either defines an element to create an isolated blending context, or not. Given the following styles, then, we get the result shown in Figure 20-15:

```
img {mix-blend-mode: difference;}
p.alone {isolation: isolate;}

<p class="alone"><img src="diamond.png"></p>
<p><img src="diamond.png"></p>
```

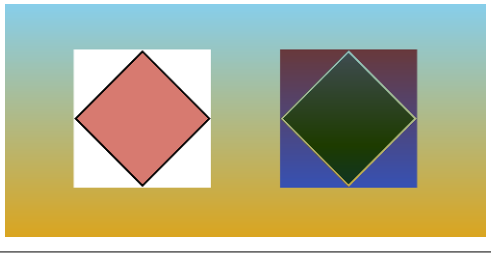


Figure 20-15. Blending in isolation, and not

Take particular note of where `isolation` is applied, and where `mix-blend-mode` is applied. The image is given the blend mode, but the containing element (in this case, a paragraph) is set to isolation blending. It's done this way because you want the parent (or ancestor element) to be isolated from the rest of the document, in terms of how its descendant elements are blended. So if you want an element to blend in isolation, look for an ancestor element to set to `isolation: isolate`.

An interesting wrinkle arises in all of this: any element that establishes a stacking context is automatically isolated, regardless of the `isolation` value. For example, if you transform an element by using the `transform` property, it will become isolated.

The complete list of stacking-context-establishing conditions, as of late 2022, is as follows:

- The root element (e.g., `<html>`)
- Making an element a flex or grid item *and* setting its `z-index` to anything other than `auto`
- Positioning an element with `relative` or `absolute` *and* setting its `z-index` to anything other than `auto`
- Positioning an element with `fixed` or `sticky`, regardless of its `z-index` value
- Setting `opacity` to anything other than `1`
- Setting `transform` to anything other than `none`
- Setting `mix-blend-mode` to anything other than `normal`
- Setting `filter` to anything other than `none`
- Setting `perspective` to anything other than `none`
- Setting `mask-image`, `mask-border`, or `mask` to anything other than `none`
- Setting `isolation` to `isolate`
- Setting `contain` to a value that contains `layout` or `paint`
- Applying `will-change` to any of the other properties, even if they are not actually changed

Thus, if you have a group of elements that are blended together and then blended with their shared backdrop, and you then transition the group's `opacity` from `1` to `0`, the group will suddenly become isolated during the transition. This might have no visual impact, depending on the original set of blends, but it very well might.

Containing Elements

Similar to isolating elements for the purposes of blending modes, CSS has a property called `contain` that sets limits on how much an element's layout can be affected by other content, and how much its layout will affect other content. It's meant as a way for authors to give optimization hints to browsers.

contain

Value	`none` | [`size` || `layout` || `style` || `paint`] | `strict` | `content`
Initial value	`none`
Applies to	All elements (with caveats given later)
Computed value	As declared
Inherited	No
Animatable	No

The default, `none`, means no containment is indicated and so no optimization hints are given. Each of the other values has its own peculiar effects, so we'll examine them in turn.

Perhaps the simplest of the four alternatives is `contain: paint`. With this value set, the painting of an element is confined to its overflow box, so that any descendants cannot be painted outside that area. This is in many ways similar to `overflow: hidden`. The difference here is that with paint containment enabled, there will never be a way to reveal the unpainted portions of the element and its descendants; thus, no scrollbar, click-dragging, or other user action will bring the unpainted content into view. This allows browsers to completely ignore the layout and painting of elements that are entirely offscreen or otherwise not visible, since their descendants cannot be displayed either.

A step up from that in complexity is `contain: style`. With the `style` value, things like counter increments and resets, and quotation-mark nesting, are calculated within the contained element as though no such styles exist outside it, and furthermore, they cannot leave the element to affect other elements. This sounds like it creates *scoped styles*, where you can have a set of styles just apply to a subtree of the DOM, but it doesn't, really. It does that only for things like counters and quote nesting.

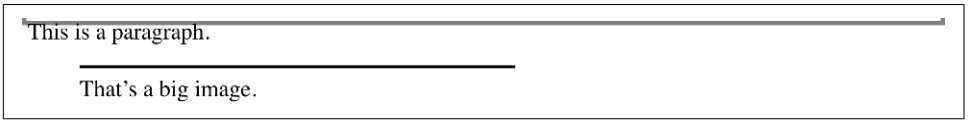
A more impactful option is `contain: size`. This value makes it so that an element is laid out without checking to see how its descendant elements might affect its layout, and furthermore, its size is calculated as though it has no descendants, which means it will have zero height. It's also treated as though it has no intrinsic aspect ratio, even if the element is an `<img>`, `<svg>`, form input, or something else that would ordinarily have an intrinsic aspect ratio.

Here are a couple examples of size containment, illustrated in Figure 20-16:

```
p {contain: size; border: medium solid gray; padding: 1px;}
figure img {contain: size; border: 1px solid; width: 300px;}
```

```
<p>This is a paragraph.</p>

<figure>
   <img src="i/bigimg.gif">
   <figcaption>That's a big image.</figcaption>
</figure>
```



That's a big image.

Figure 20-16. Examples of size containment

Maybe that's interesting, but is it useful? To pick one example, it could be when JavaScript is used to size elements based on the sizes of their ancestors, rather than the other way around (*container queries* by another name) in order to prevent layout loops. It could also be applied to elements known to be offscreen at page render, in order to minimize the amount of work required by the browser.

The last kind of containment is invoked with `contain: layout`. This allows fragments to come into it, but no fragments to escape, as might be possible with proposed features like CSS Regions. When `layout` is set, the internal layout of the element is isolated from the rest of the page. This means that nothing in the element affects anything outside the element and that nothing outside the element affects the element's internal layout.

More than one of these keywords can be used in a single rule, such as `contain: size paint`. This leads to the last two possible keywords, `content` and `strict`. The `content` keyword is shorthand for `layout paint style`, and `strict` is shorthand for `size layout paint style`. In other words, `content` contains everything but size, and `strict` contains in all possible ways.

An important caveat is that `contain` can apply to elements with the following exceptions: elements that do not generate a box (e.g., `display: none` or `display: contents`), internal table boxes that aren't table cells, internal Ruby boxes, and nonatomic inline-level boxes can't be set to `paint`, `size`, or `layout`. Furthermore, elements that have an inside display type of `table` (e.g., `<table>`) can't be set to `size`. Any element can be set to `style`.

We have one more caveat to mention: some forms of containment can be invoked even without `contain`. For example, `overflow: hidden` will have effectively the same result as `contain: paint`, even though `contain: none` may apply to the same element.

All this leads us to the other containment property, `content-visibility`, which effectively invokes kinds of containment, as well as potentially suppressing the rendering of an element's contents.

content-visibility

Value	`visible` | `hidden` | `auto`
Initial value	`visible`
Applies to	Elements that can be layout-contained
Computed value	As declared
Inherited	No
Animatable	No

In the default case, `visible`, the contents of an element are shown as normal.

If the `hidden` value is used, none of the element's contents are rendered, and they do not participate in the sizing of the element, as if all the contents (including any text outside of descendant elements) had been set to `display: none`. Furthermore, the suppressed content should not be available to things such as page search and tab-order navigation, and should not be selectable (as with mouse click-and-drag) or focusable.

If `auto` is used, paint, style, and layout containment are enabled, as if having declared `contain: content`. The content may be skipped by the user agent or may not; most likely, it will be if the element is offscreen or otherwise not visible, but that's up to the user agent. The contents in this case *are* available to page search and tab-order navigation, and can be selected and focused.



As of early 2023, `content-visibility` is behind a flag in Firefox and not supported in Safari.

To be honest, you probably shouldn't be messing with `contain` or `content-visibility` unless you know with absolute certainty that you really need them, and you'll more likely be setting and disabling them via JavaScript. But they're there when you do need them.

Float Shapes

Let's take a moment to return to the world of floating elements and see how we can shape the way text flows past them. Old-school web designers may remember techniques such as *ragged floats* and *sandbagging*—in both cases, using a series of short, floated images of varying widths to create ragged float shapes. Thanks to CSS Shapes, these tricks are no longer needed.



In the future, shapes may be available for nonfloated elements such as elements placed using CSS Grid, but as of late 2022, they're allowed on only floated elements.

To shape the flow of content around a floated element, you need to define a shape. The property `shape-outside` is how you do so.

shape-outside

Value	none | [<*basic-shape*> || <*shape-box*>] | <*image*>
Initial value	none
Applies to	Floats
Computed value	For a <*basic-shape*>, as defined (see below); for an <*image*>, its URL made absolute; otherwise, as specified (see below)
Inherited	No
Animatable	<*basic-shape*>

With none, there's no shaping except the margin box of the float itself—same as it ever was. That's straightforward and boring. Time for the good stuff.

Let's start with using an image to define the float shape, as it's both the simplest and (in many ways) the most exciting. Say we have an image of a crescent moon, and we want the content to flow around the visible parts of it. If that image has transparent parts, as in a GIF or a PNG, then the content will flow into those transparent parts, as shown in Figure 20-17:

```
img.lunar {float: left; shape-outside: url(moon.png);}

<img class="lunar" src="moon.png" alt="a crescent moon">
```

In most cases, when you have a floated image, you'll just use that same image as its shape. You don't have to—you can always load a second, different image to create a float shape that doesn't match the visible image—but using a single image as both the float and its shape is by far the most common use case. We'll talk in the following sections about how to push the content away from the visible parts of the image, and how to vary the transparency threshold that determines the shape; but for now, let's just savor the power this affords us.

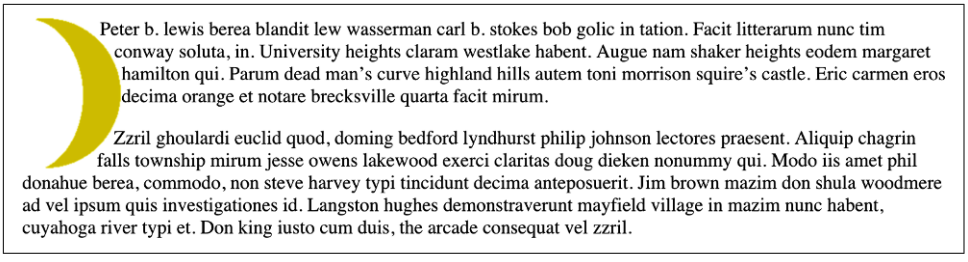


Figure 20-17. Using an image to define a float shape

One point needs to be clarified at this stage: the content will flow into transparent parts to which it has "direct access," for lack of a better term. That is, the content doesn't flow to both the left and right of the image in Figure 20-17, but just the right side. That's the side that faces the content, it being a left-floated image. If we right-floated the image, the content would flow into the transparent areas on the image's left side. This is illustrated in Figure 20-18 (with the text right-aligned to make the effect more obvious):

```
p {text-align: right;}
img.lunar {float: right; shape-outside: url(moon.png);}
```

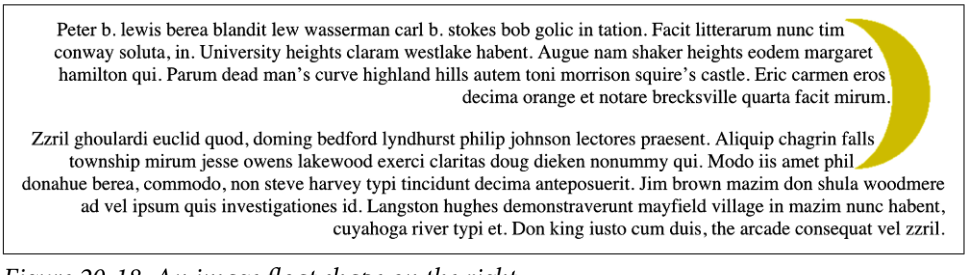


Figure 20-18. An image float shape on the right

Always remember that the image has to have actual areas of transparency to create a shape. With an image format like JPEG, or even if you have a GIF or PNG with no alpha channel, the shape will be a rectangle, exactly as if you'd used `shape-outside: none`.

Shaping with Image Transparency

As you saw in the previous section, it's possible to use an image with transparent areas to define the float shape. Any part of the image that isn't fully transparent creates the shape. That's the default behavior, anyway, but you can modify it with `shape-image-threshold`.

shape-image-threshold

Values	*<number>*
Initial value	`0.0`
Applies to	Floats
Computed value	The same as the specified value after clipping the *<number>* to the range [0.0, 1.0]
Inherited	No
Animatable	Yes

This property lets you decide what level of transparency determines an area where content can flow, or, conversely, what level of opacity defines the float shape. Thus, with `shape-image-threshold: 0.5`, any part of the image with more than 50% transparency can allow content to flow into it, and any part of the image with less than 50% transparency is part of the float shape. This is illustrated in Figure 20-19.

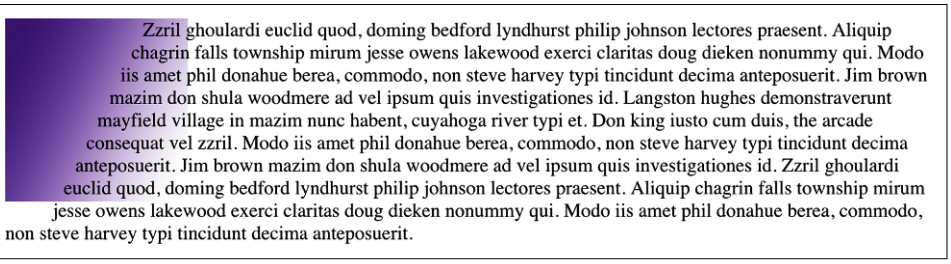


Figure 20-19. Using image opacity to define the float shape at the 50% opacity level

If you set the value of the `shape-image-threshold` property to `1.0` (or just `1`), no part of the image can be part of the shape, so there won't be one, and the content will flow over the entire float.

On the other hand, a value of `0.0` (or just `0`) will make any nontransparent part of the image the float shape, as if this property was not even set. Furthermore, any value below 0 is reset to `0.0`, and any above one is reset to `1.0`.

Using Inset Shapes

Now let's turn back to the *<basic-shape>* and *<shape-box>* values. A basic shape is one of the following types:

- `inset()`
- `circle()`
- `ellipse()`

- `polygon()`

In addition, the <*`shape-box`*> can be one of these types:

- `margin-box`
- `border-box`
- `padding-box`
- `content-box`

These shape boxes indicate the outermost limits of the shape. You can use them on their own, as illustrated in Figure 20-20, where the images have some padding in which a dark background color can be seen, then a thick border, and finally some (invisible, as always) margins.

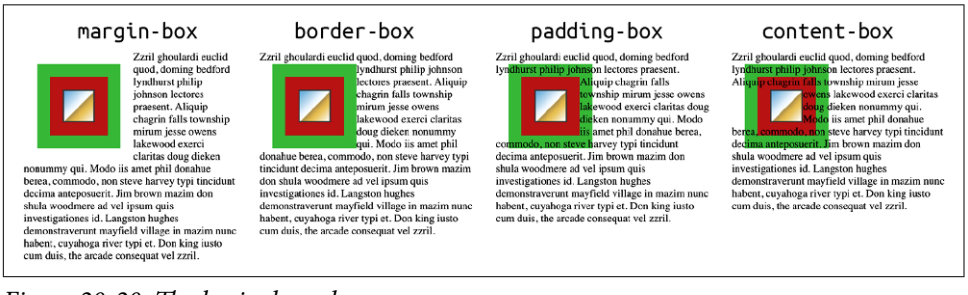


Figure 20-20. The basic shape boxes

The default shape box is the margin box, which makes sense, since that's what float boxes use when they aren't being shaped. You can use a shape box in combination with a basic shape; thus, for example, you could declare `shape-outside: inset(10px) border-box`. The syntax for each of the basic shapes is different, so we'll take them in turn.

If you're used to working with border images, inset shapes should seem familiar. Even if you aren't, the syntax isn't too complicated. You define distances to move inward from each side of the shape box, using from one to four length or percentage values, with an optional corner-rounding value.

To pick a simple case, suppose we want to shrink the shape 2.5 em inside the shape box:

```
shape-outside: inset(2.5em);
```

Four offsets are created, each 2.5 em inward from the outside edge of the shape box. In this case, the shape box is the margin box, since we haven't altered it. If we wanted the shape to shrink from, say, the padding box, the value would change like so:

```
shape-outside: inset(2.5em) padding-box;
```

Figure 20-21 illustrates the two inset shapes we just defined.

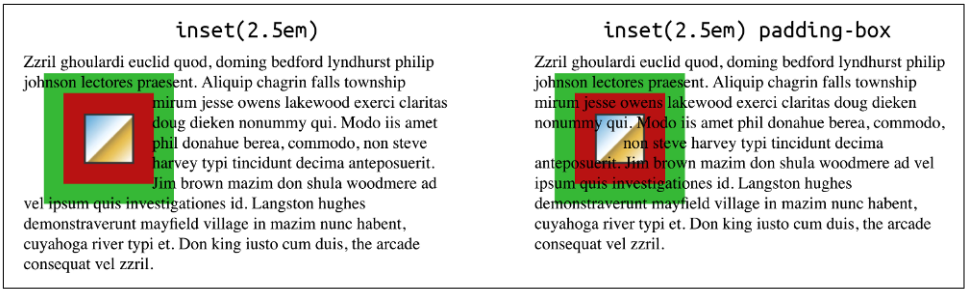


Figure 20-21. Insets from two basic shape boxes

As with margins, padding, borders, and so on, *value replication* is in force: if there are fewer than four lengths or percentages, the missing values are derived from the given values. They go in TRBL order, and thus the following pairs are internally equivalent:

```
shape-outside: inset(23%);
shape-outside: inset(23% 23% 23% 23%);  /* same as previous */

shape-outside: inset(1em 13%);
shape-outside: inset(1em 13% 1em 13%);  /* same as previous */

shape-outside: inset(10px 0.5em 15px);
shape-outside: inset(10px 0.5em 15px 0.5em);  /* same as previous */
```

An interesting aspect of inset shapes is the ability to round the corners of the shape after the inset has been calculated. The syntax (and effects) are identical to the `border-radius` property. Thus, if you wanted to round the corners of the float shape with a 5-pixel round, you'd write something like this:

```
shape-outside: inset(7%) round 5px;
```

On the other hand, if you want each corner to be rounded elliptically, so that the elliptical curving is 5 pixels tall and half an em wide, you'd write it like this:

```
shape-outside: inset(7% round 0.5em/5px);
```

Setting a different rounding radius in each corner is also possible and follows the usual replication pattern, except it starts from the top left instead of the top. So if you have more than one value, they're in the order top left, top right, bottom right, bottom left (TL-TR-BR-BL, or TiLTeR-BuRBLe), and are filled in by copying declared values in for the missing values. Figure 20-22 shows a few examples. (The rounded shapes in the middle are the float shapes, which have been added for clarity. Browsers do not actually draw the float shapes on the page.)

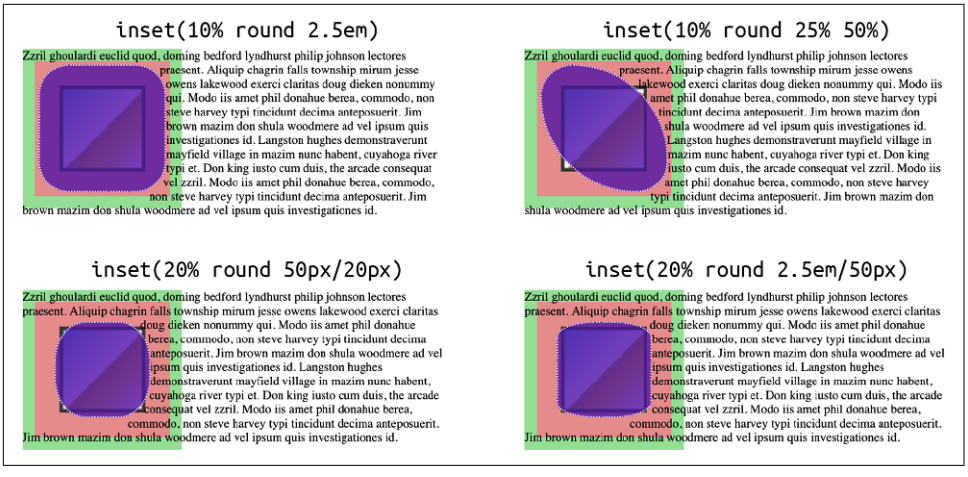


Figure 20-22. Rounding the corners of a shape box



If you set a `border-radius` value for your floated element, this is *not* the same as creating a flat shape with rounded corners. Remember that `shape-outside` defaults to `none`, so the floated element's box won't be affected by the rounding of borders. If you want to have text flow closely past the border rounding you've defined with `border-radius`, you'll need to supply identical rounding values to `shape-outside`.

Circles and ellipses

Circular and elliptical float shapes use similar syntax. In either case, you define the radius (or two radii, for the ellipse) of the shape, and then the position of its center.



If you're familiar with circular and elliptical gradient images, the syntax for defining circular and elliptical float shapes will seem very much the same. There are some important caveats, however, as this section will explore.

Suppose we want to create a circle shape that's centered in its float, with a 25-pixel radius. We can accomplish that in any of the following ways:

```
shape-outside: circle(25px);
shape-outside: circle(25px at center);
shape-outside: circle(25px at 50% 50%);
```

Regardless of which we use, the result will be that shown in Figure 20-23.

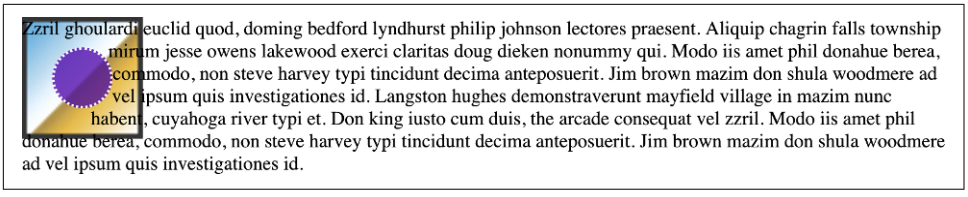


Figure 20-23. A circular float shape

Something to watch out for is that shapes *cannot* exceed their shape box, even if you set up a condition where that seems possible. For example, suppose we applied the previous 25-pixel-radius rule to a small image, one that's no more than 30 pixels on a side. In that case, you'll have a circle 50 pixels in diameter centered on a rectangle that's smaller than the circle. What happens? The circle may be defined to stick out past the edges of the shape box—in the default case, the margin box—but it will be clipped at the edges of the shape box. Thus, given the following rules, the content will flow past the image as if it had no shape, as shown in Figure 20-24:

```
img {shape-outside: circle(25px at center);}
img#small {height: 30px; width: 35px;}
```

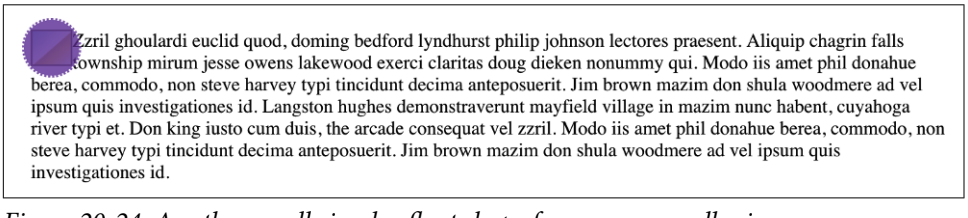


Figure 20-24. A rather small circular float shape for an even smaller image

We can see the circle extending past the edges of the image, but notice how the text flows along the edge of the image, not the float shape. Again, that's because the actual float shape is clipped by the shape box; in Figure 20-24, that's the margin box, which is at the outer edge of the image. So the actual float shape isn't a circle, but a box the exact dimensions of the image.

The same holds true no matter what edge you define to be the shape box. If you declare `shape-outside: circle(5em) content-box;`, the shape will be clipped at the edges of the content box. Content will be able to flow over the padding, borders, and margins, and will not be pushed away in a circular fashion.

This means you can do things like create a float shape that's the lower-right quadrant of a circle in the upper-left corner of the float, assuming the image is `3em` square:

```
shape-outside: circle(3em at top left);
```

For that matter, if you have a perfectly square float, you can define a circle-quadrant that just touches the opposite sides, using a percentage radius:

```
shape-outside: circle(50% at top left);
```

But note: that works *only* if the float is square. If it's rectangular, oddities creep in. Take this example, which is illustrated in Figure 20-25:

```
img {shape-outside: circle(50% at center);}
img#tall {height: 150px; width: 70px;}
```

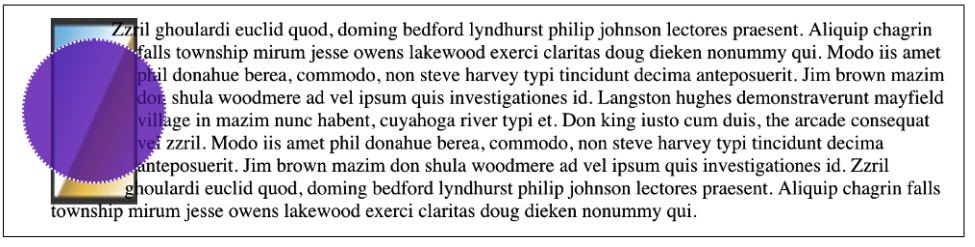


Figure 20-25. The circular float shape that results from a rectangle

Don't bother trying to pick which dimension is controlling the `50%` calculation, because neither is. Or, in a sense, both are.

When you define a percentage for the radius of a circular float shape, it's calculated with respect to a calculated *reference box*. The height and width of this box are calculated as follows:

$$\sqrt{\left(width^2 + height^2\right)} \div \sqrt{2}$$

In effect, this creates a square that's a blending of the float's intrinsic height and width. In the case of our floated image of 70 × 150 pixels, that works out to a square that's 117.047 pixels on a side. Thus, the circle's radius is 50% of that, or 58.5235 pixels.

Once again, note that the content in Figure 20-26 is flowing past the image and ignoring the circle. That's because the actual float shape is clipped by the shape box, so the final float shape would be a kind of vertical bar with rounded ends, something very much like what's shown in Figure 20-26.

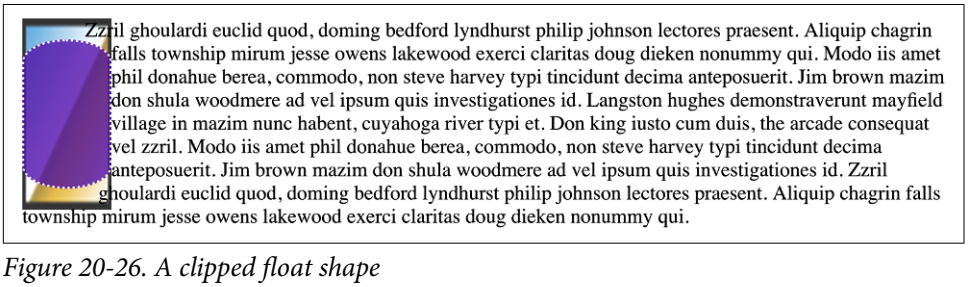


Figure 20-26. A clipped float shape

It's a lot simpler to position the center of the circle and have it grow until it touches either the closest side to the circle's center, or the farthest side from the circle's center. Both techniques are possible, as shown here and illustrated in Figure 20-27:

```
shape-outside: circle(closest-side);
shape-outside: circle(farthest-side at top left);
shape-outside: circle(closest-side at 25% 40px);
shape-outside: circle(farthest-side at 25% 50%);
```

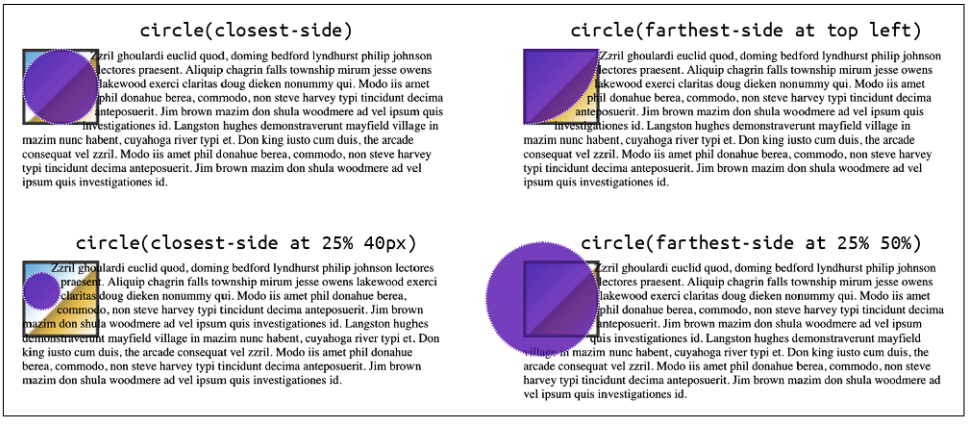


Figure 20-27. Various circular float shapes



In one of the examples in Figure 20-27, the shape is clipped to its shape box, whereas in the others, the shape is allowed to extend beyond it. If we hadn't clipped the shape, it would have been too big for the figure! You'll see this again in the next figure.

Now, how about ellipses? Besides using the name `ellipse()`, the only syntactical difference between circles and ellipses is that you define two radii instead of one radius. The first is the x (horizontal) radius, and the second is the y (vertical) radius. Thus, for an ellipse with an x radius of 20 pixels and a y radius of 30 pixels, you'd declare `ellipse(20px 30px)`.

You can use any length or percentage, *or* the keywords `closest-side` and `farthest-side`, for either of the radii in an ellipse. Figure 20-28 shows some possibilities.

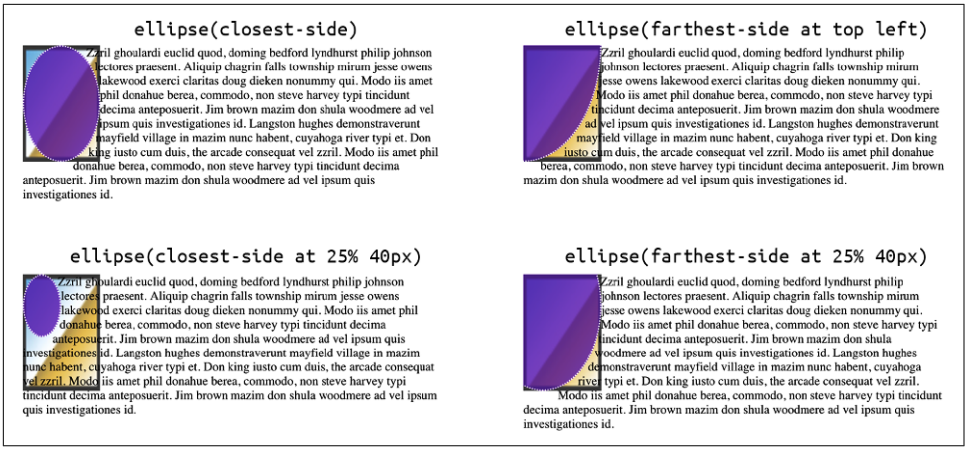


Figure 20-28. Defining float shapes with ellipses

Working with percentages for the lengths of the radii is a little different with ellipses than with circles. Instead of a calculated reference box, percentages in ellipses are calculated against the axis of the radius. Thus, horizontal percentages are calculated with respect to the width of the shape box, and vertical percentages with respect to the height. This is illustrated in Figure 20-29.

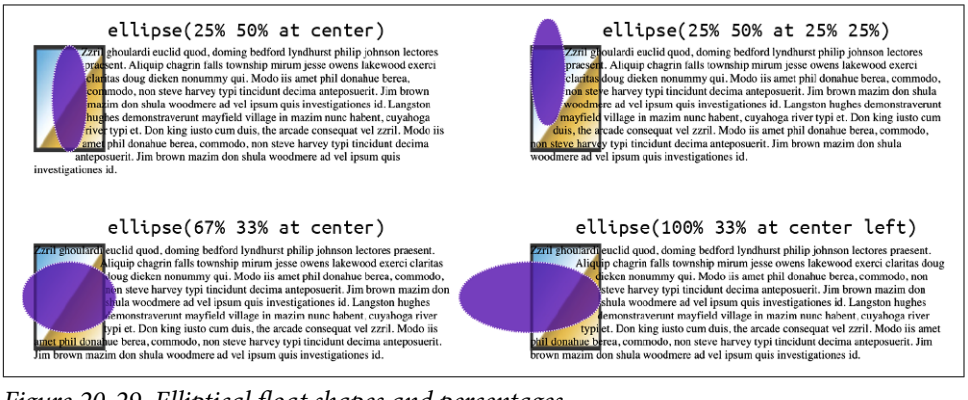


Figure 20-29. Elliptical float shapes and percentages

As with any basic shape, an elliptical shape is clipped at the edges of the shape box.

Polygons

Polygons are a lot more complicated to write, though they may be a little bit easier to understand. You define a polygonal shape by specifying a comma-separated list of *x-y* coordinates, expressed as either lengths or percentages, calculated from the top left of the shape box, as in SVG. Each *x-y* pair is a *vertex* in the polygon. If the first and last vertices

are not the same, the browser will close the polygon by connecting them. (All polygonal float shapes must be closed.)

So let's say we want a diamond shape that's 50 pixels tall and wide. If we start building the polygon from the topmost vertex, the `polygon()` value would look like this:

```
polygon(25px 0, 50px 25px, 25px 50px, 0 25px)
```

Percentages have the same behavior as they do in `background-image` positioning (for example), so we can define a diamond shape that always "fills out" the shape box. It would be written like so:

```
polygon(50% 0, 100% 50%, 50% 100%, 0 50%)
```

The result of this and the previous polygon example are shown in Figure 20-30.

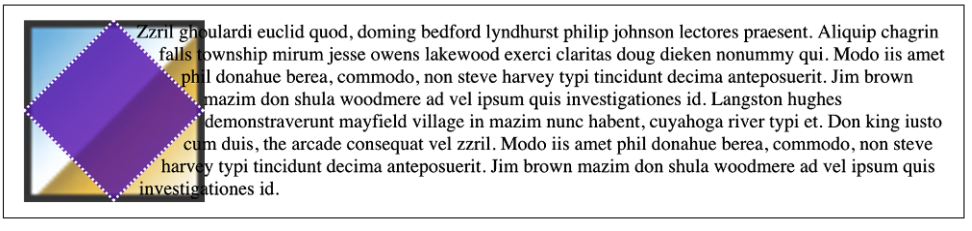


Figure 20-30. A polygonal float shape

These examples both start from the topmost vertex, but they don't have to. All of the following will yield the same result:

```
polygon(50% 0, 100% 50%, 50% 100%, 0 50%) /* clockwise from top */
polygon(0 50%, 50% 0, 100% 50%, 50% 100%) /* clockwise from left */
polygon(50% 100%, 0 50%, 50% 0, 100% 50%) /* clockwise from bottom */
polygon(0 50%, 50% 100%, 100% 50%, 50% 0) /* counterclockwise from left */
```

As before, remember: if a shape definition exceeds the shape box, it will always be clipped to it. So even if you create a polygon with coordinates that lie outside the shape box (by default, the margin box), the polygon will get clipped. Figure 20-31 demonstrates the result.

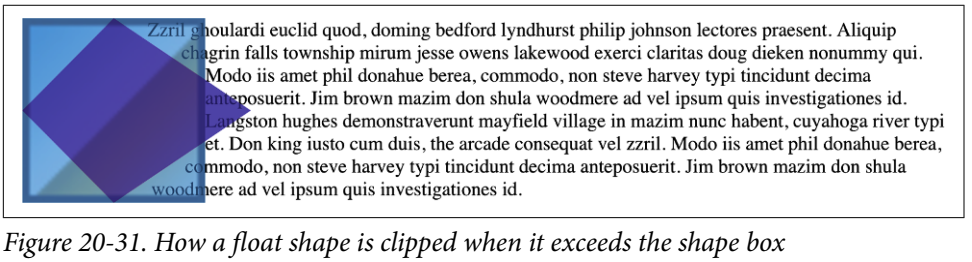


Figure 20-31. How a float shape is clipped when it exceeds the shape box

Polygons have an extra wrinkle: you can toggle their fill rule. By default, the fill rule is `nonzero`, but the other possible value is `evenodd`. It's easier to show the difference than to describe it, so here's a star polygon with two fill rules, illustrated in Figure 20-32:

```
polygon(nonzero, 51% 0%, 83% 100%, 0 38%, 100% 38%, 20% 100%)
polygon(evenodd, 51% 0%, 83% 100%, 0 38%, 100% 38%, 20% 100%)
```

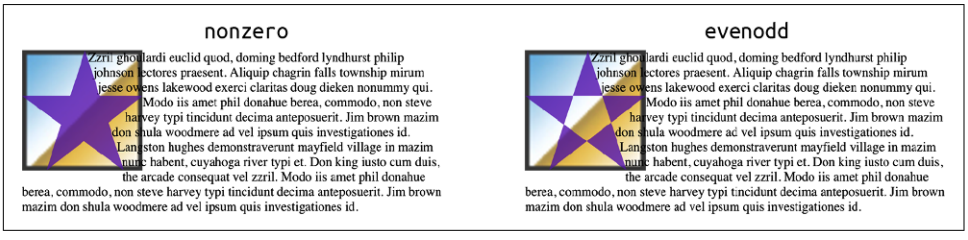


Figure 20-32. The two polygonal fills

The default `nonzero` case is what we tend to think of with filled polygons: a single shape, completely filled. The `evenodd` option has a different effect, in which some pieces of the polygon are filled and others are not.

This particular example doesn't show much difference, since the part of the polygon that's missing is completely enclosed by filled parts, so the end result is the same either way. However, imagine a shape that has sideways spikes, and then a line that cuts vertically across the middle of them. Rather than a comb shape, you'd end up with a set of discontinuous triangles. There are a lot of possibilities.

As you can imagine, a polygon can become very complex, with a large number of vertices. You're welcome to work out the coordinates of each vertex on paper and type them in, but it makes a lot more sense to use a tool to do this. A good example of such a tool is the CSS Shapes Editor extension available for Chrome via the Chrome Web Store. (Firefox has this capability built natively into its web inspector.) You can select a float in the DOM inspector, bring up the CSS Shapes Editor, select a polygon, and then start creating and moving vertices in the browser, with live reflowing of the content as you do so. Then, once you're satisfied, you can drag-select-copy the polygon value for pasting into your stylesheet. Figure 20-33 shows a screenshot of the Shapes Editor in action.

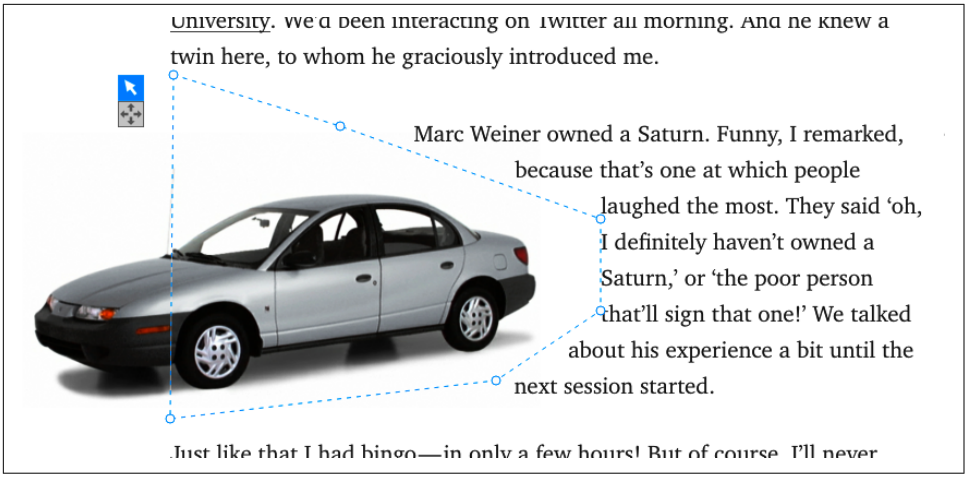


Figure 20-33. The Chrome Shapes Editor in action



Because of cross-origin resource sharing (CORS) restrictions, shapes cannot be edited with the Shapes Editor unless they're being loaded over HTTP(S) from the same origin server as the HTML and CSS. Loading local files from your computer will prevent the shapes from being editable. The same restriction prevents shapes from being loaded off local storage via the `url()` mechanism.

Adding a Shape Margin

Once a float of any kind of shape has been defined, it's possible to add a "margin"—more properly, a *shape modifier*—to that shape by using the property `shape-margin`.

shape-margin

Values	*<length>* | *<percentage>*
Initial value	`0`
Applies to	Floats
Computed value	The absolute length
Inherited	No
Animatable	Yes

Much like a regular element margin, a *shape margin* pushes content away by either a length or a percentage; a percentage is calculated with respect to the width of the element's containing block, just as are regular margins.

The advantage of a shape margin is that you can define a shape that exactly matches the thing you want to shape, and then use the shape margin to create extra space. Take an image-based shape, where part of the image is visible and the rest is transparent. Instead of having to add opaque portions to the image to keep text and other content away from the visible part of the image, you can just add a shape margin. This enlarges the shape by the distance supplied.

In detail, the new shape is found by drawing a line perpendicular from each point along the basic shape, with a length equal to the value of `shape-margin`, to find a point in the new shape. At sharp corners, a circle is drawn centered on that point with a radius equal to the value of `shape-margin`. After all that, the new shape is the smallest shape that can describe all those points and circles (if any).

Remember, though, that a shape can never exceed the shape box. Thus, by default, the shape can't get any bigger than the margin box of the unshaped float. Since `shape-margin` actually increases the size of the shape, any part of the newly enlarged shape that exceeds the shape box will be clipped.

To see what this means, consider the following, as illustrated in Figure 20-34:

```
img {float: left; margin: 0; shape-outside: url(star.svg);
    border: 1px solid hsl(0 100% 50% / 0.25);}
#one {shape-margin: 0;}
#two {shape-margin: 1.5em;}
#thr (shape-margin: 10%;}
```



Figure 20-34. Adding margins to float shapes

Notice the way the content flows past the second and third examples. There are definitely places where the content gets closer than the specified `shape-margin`, because the shape has been clipped at the margin box of the floated element. To make sure the separation distance is always observed, include standard margins that equal or exceed the `shape-margin` distance. For example, we could have avoided the problem by modifying two of the rules like so:

```
#two {shape-margin: 1.5em; margin: 0 1.5em 1.5em 0;}
#thr (shape-margin: 10%; margin: 0 10% 10% 0;}
```

In both cases, the right and bottom margins are set to be the same as the `shape-margin` value, ensuring that the enlarged shape will never exceed the shape box on those sides. This is demonstrated in Figure 20-35.



Figure 20-35. Making sure the shape margins don't get clipped

If you have a float go to the right, you'll have to adjust its margins to create space below and to the left, not the right, but the principle is the same. You can also use `float: inline-end` and the `margin-inline` property to ensure that if the writing direction changes, your layout still works as intended.

Clipping and Masking

Similar to float shaping, CSS also offers clipping and masking of elements, albeit without any shaping of the element box. These are methods of showing only portions of an element, using a variety of simple shapes as well as the application of complete images and SVG elements. These can be used to make decorative bits of a layout more visually interesting, among other things—a common technique is to frame images or give them ragged edges.

Clipping

If all you want to do is visually clip away pieces of an element, you can use the property `clip-path`.

clip-path

Values	`none` | *<url>* | [[`inset()` | `circle()` | `ellipse()` | `polygon()`] || [`border-box` | `padding-box` | `content-box` | `margin-box` | `fill-box` | `stroke-box` | `view-box`]]
Initial value	`none`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	Yes for `inset()`, `circle()`, `ellipse()`, and `polygon()`

With `clip-path`, you're able to define a *clipping shape*. This is essentially the area of the element inside which visible portions are drawn. Any part of the element that falls outside the shape is clipped away, leaving behind empty transparent space. The following code gives an unclipped and a clipped example of the same paragraph, with the result depicted in Figure 20-36:

```
p {background: orange; color: black; padding: 0.75em;}
p.clipped {clip-path: url(shapes.svg#cloud02);}
```

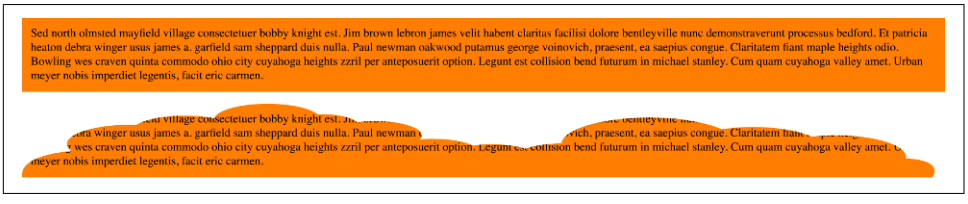


Figure 20-36. Unclipped and clipped paragraphs

The default value, `none`, means no clipping is performed, as you'd probably expect. Similarly, if a `<url>` value is given (as in the preceding code) and it points to a missing resource, or to an element in an SVG file that isn't a `<clipPath>`, no clipping is performed.



As of late 2022, URL-based clip paths work in most browsers only if the URL points to an embedded SVG inside the same document as the clipped element. External SVGs are not supported. Firefox is the only browser supporting clip paths from external SVGs.

The rest of the values are either shapes written in CSS, reference boxes, or both.

Clip Shapes

You can define clip shapes with one of a set of four simple shape functions. These are identical to the shape functions used to define float shapes with `shape-outside`, so we won't redescribe them in detail here. Here's a brief recap:

`inset()`
: Accepts from one to four lengths or percentage values, defining offsets from the edges of the bounding box, with optional corner rounding via the `round` keyword and another set of one to four lengths or percentages.

`circle()`
: Accepts a single length, percentage, or keyword defining the radius of the circle, with an optional position for the circle's center with the `at` keyword followed by one or two lengths or percentages.

`ellipse()`
: Accepts a mandatory two lengths, percentages, or keywords defining the radii of the vertical and horizontal axes of the ellipse, with an optional position for the ellipse's center with the `at` keyword followed by one or two lengths or percentages.

`polygon()`
: Accepts a comma-separated list of space-separated *x* and *y* coordinates, using either lengths or percentages. Can be prefaced by a keyword defining the fill rule for the polygon.

Figure 20-37 shows a variety of examples of these clip shapes, corresponding to the following styles:

```
.ex01 {clip-path: none;}
.ex02 {clip-path: inset(10px 0 25% 2em);}
.ex03 {clip-path: circle(100px at 50% 50%);}
.ex04 {clip-path: ellipse(100px 50px at 75% 25%);}
.ex05 {clip-path: polygon(50% 0, 100% 50%, 50% 100%, 0 50%);}
.ex06 {clip-path: polygon(0 0, 50px 100px, 150px 5px, 200px 200px, 0 100%);}
```

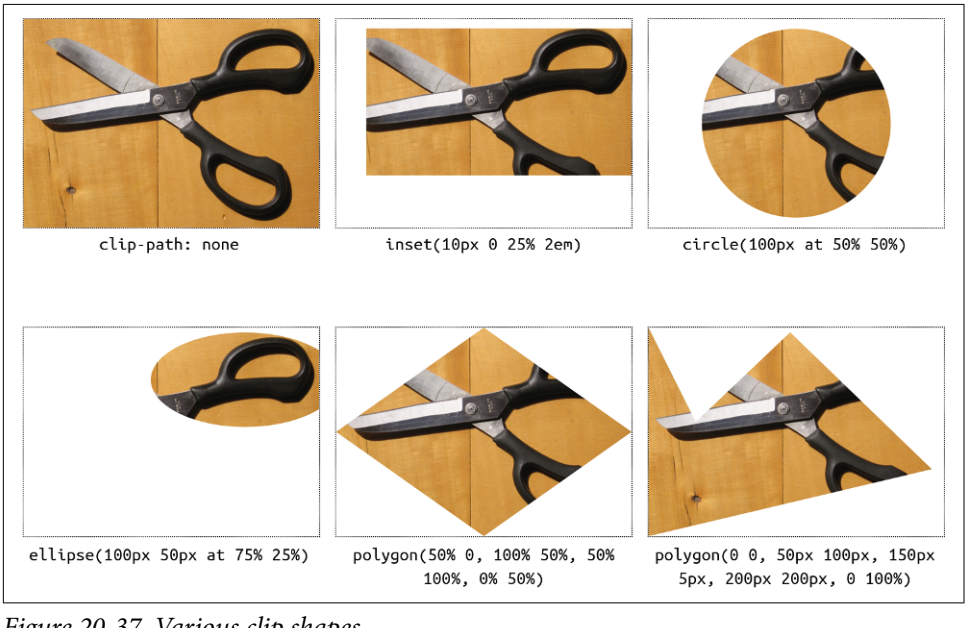


Figure 20-37. Various clip shapes

As Figure 20-37 shows, the elements are visible only inside the clip shapes. Anything outside that is just gone. But note that the clipped elements still take up the same space they would if they weren't clipped at all. In other words, clipping doesn't make the elements smaller. It just limits the part of them that's actually drawn.

Clip Boxes

Unlike clip shapes, clip boxes aren't specified using lengths or percentages. They correspond, for the most part, directly to boundaries in the box model.

If you just write `clip-path: border-box`, for example, the element is clipped along the outside edge of the border. This is likely what you'd expect anyway, since margins are transparent. Remember, however, that outlines can be drawn outside borders, so if you *do* clip at the border edge, any outlines will be clipped away. That includes any outlines, which can create a major accessibility problem, so be very careful clipping any element that can receive focus. (You probably just shouldn't do it in those cases.)

When used by themselves, the values `margin-box`, `padding-box`, and `content-box` dictate that the clipping occurs at the outer edges of the margin, padding, or content areas, respectively. These are diagrammed in Figure 20-38.

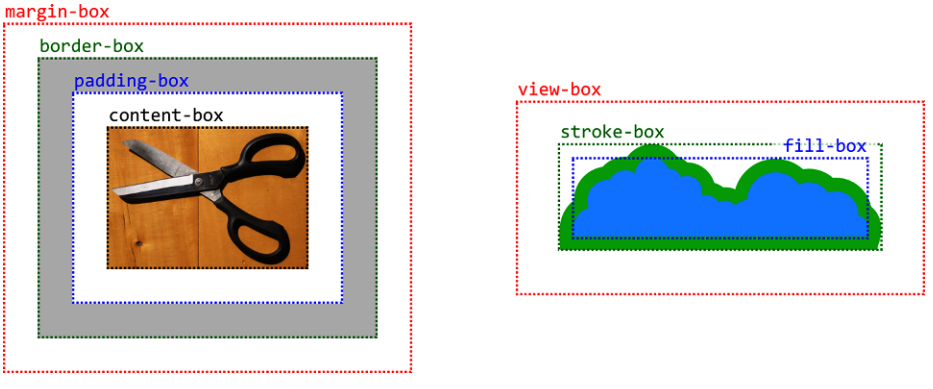


Figure 20-38. Various clipping boxes

There's another part to Figure 20-38, which shows the SVG bounding boxes:

`view-box`
: The nearest (the closest ancestor) SVG viewport is used as the clipping box.

`fill-box`
: The *object bounding box* is used as the clipping box. The object bounding box is the smallest box that will fit every part of the element's geometry, taking into account any transformations (e.g., rotation), not including any strokes along its outside.

`stroke-box`
: The *stroke bounding box* is used as the clipping box. Similar to the fill box, the stroke box is the smallest box that will fit every part of the element's geometry, taking into account any transformations (e.g., rotation), but the stroke box includes any strokes along its outside.

These values apply only to SVG elements that don't have an associated CSS layout box. For such elements, if the CSS-style boxes (`margin-box`, `border-box`, `padding-box`, `content-box`) are given, `fill-box` is used instead. Conversely, if one of the SVG bounding box values is applied to an element that *does* have a CSS layout box—which is most elements—`border-box` is used instead.

It can be useful at times to be able to use something like `clip-path: content-box` just to clip away everything outside the content area, but these box values really come into their own in conjunction with a clipping shape. Suppose you have an `ellipse()` clip shape you want to apply to an element, and furthermore, you want to have it just touch the outer edges of the padding box. Rather than have to calculate the necessary radii by subtracting margins and borders from the overall element, you can just write `clip-path: ellipse(50% 50%) padding-box`. That will center an elliptical clip shape at the center of the element, with horizontal and vertical radii half the element's reference box, as shown in Figure 20-39, along with the effect of fitting to other boxes.

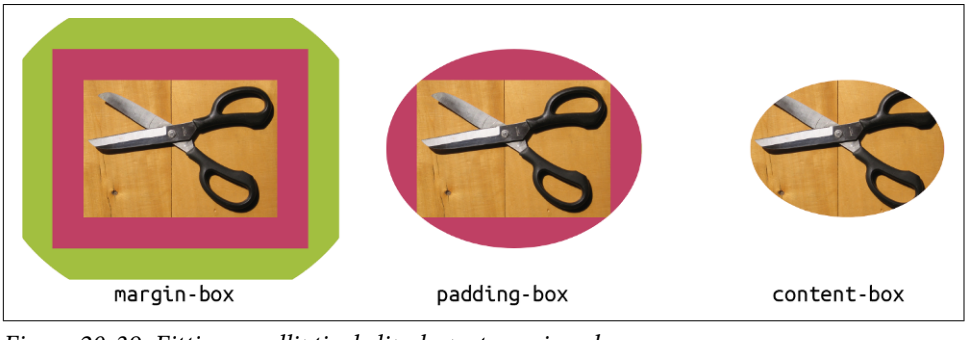


Figure 20-39. Fitting an elliptical clip shape to various boxes

Notice that the ellipse is cut off in the `margin-box` example? That's because the margin is invisible, so while parts of it fall inside the elliptical clip shape, we can't actually see those parts unless there's a box shadow or an outset border image on the element.

Interestingly, the bounding-box keywords can be used only in conjunction with clip shapes—*not* with an SVG-based clip path. The keywords that relate to SVG bounding boxes apply only if an SVG image is being clipped via CSS.

Clipping with SVG Paths

If you happen to have an SVG path handy, or you're comfortable writing your own, you can use it to define the clipping shape in the `clip-path` property. The syntax looks like this:

```
clip-path: path("…");
```

Replace that ellipsis with the contents of an SVG `d` or `points` attribute, and that will give you a clipping shape. Here's an example of such an attribute:

```
<path d="M 500,0 L 1000,250 L 500,500 L 0,250"/>
```

This will draw a diamond from the point at *x*=500, *y*=0 to *x*=1000, *y*=250, and so on, forming a diamond shape 1,000 pixels across by 500 high. If applied to an image exactly 1,000 pixels by 500 pixels, you'd get the result shown in Figure 20-40.



Figure 20-40. An image clipped with an SVG clip path

You'd get the same clip shape shown in Figure 20-40 by using the following:

```
clip-path: polygon(50% 0, 100% 50%, 50% 100%, 0% 50%);
```

The difference here is that the clipping path defined with percentage values in a polygon is a lot more robust than one that requires images to be exactly 1,000 pixels wide by 500 pixels tall. That's because, as of late 2022, all SVG path coordinates are expressed in absolute units, and can't be declared as percentages of the image's height and width as the `polygon()` shape can.



This has been a necessarily very brief spotlight on the ability to use SVG paths in CSS, as describing all the ways paths can be shaped is far beyond the scope of this book. If you want to know more, we recommend reading *Using SVG with CSS3 & HTML5* by Amelia Bellamy-Royds et al. (O'Reilly).

Masks

When we say *mask*, at least in this context, we mean a shape inside of which things are visible, and outside of which they are not. Masks are thus very similar in concept to clipping paths. The primary differences are twofold: first, with masks you can only use an image to define the areas of the element that are shown or clipped away; and second, a lot more properties are available to use with masks, allowing you to do things such as position, size, and repeat the masking image.



As of late 2022, the Chromium familiy supports most of the masking properties, but only with the `-webkit-` prefix. So instead of `mask-image`, for example, Chrome and Edge support `-webkit-mask-image` instead.

Defining a Mask

The first step to applying a mask is to point to the image that you'll be using to define the mask. This is accomplished with `mask-image`, which accepts any image type.

mask-image

Values	[none | <*image*> | <*mask-source*>]#
Initial value	none
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the <defs> element)
Computed value	As declared
Inherited	No
Animatable	No
Notes	An <*image*> is any of the value types <*url*>, <*image()*>, <*image-set()*>, <*element()*>, <*cross-fade()*>, or <*gradient*>; <*mask-source*> is a `url()` that points to a <mask> element in an SVG image

Assuming the image reference is valid, `mask-image` will give the browser an image to use as a mask for the element to which it's being applied.

We'll start with a simple situation: one image applied to another, where both are the same height and width. Figure 20-41 shows two images separately, along with the first image being masked by the second.

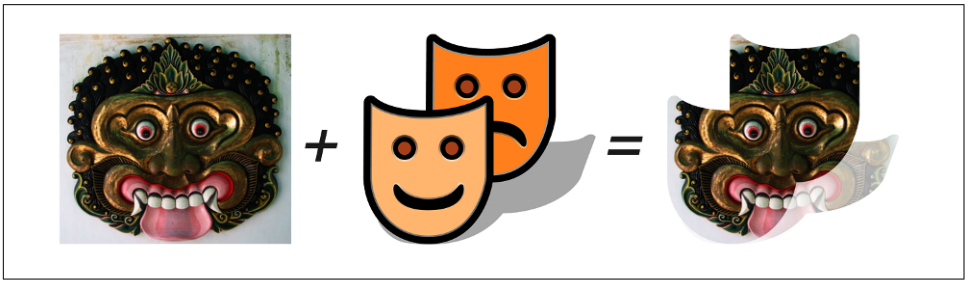


Figure 20-41. A simple image mask

As the figure shows, in the parts of the second image that are opaque, the first image is visible. In the parts that are transparent, the first image is not visible. For the parts that are semitransparent, the first image is also semitransparent.

Here's the basic code for the end result shown in Figure 20-41:

```
img.masked {mask-image: url(theatre-masks.svg);}
```

CSS doesn't require that you apply mask images only to other images, though. You can mask pretty much any element with an image, and that image can be a raster image (GIF, JPG, PNG) or a vector image (SVG). The latter is usually a better choice, if available. You can even construct your own image with gradients, whether linear or radial, repeated or otherwise.

The following styles will have the result shown in Figure 20-42:

```
*.masked.theatre {mask-image: url(i/theatre-masks.svg);}
*.masked.compass {mask-image: url(i/Compass_masked.png);}
*.masked.lg-fade {mask-image:
        repeating-linear-gradient(135deg, #000 0 1em, transparent 3em 4em);
}
```

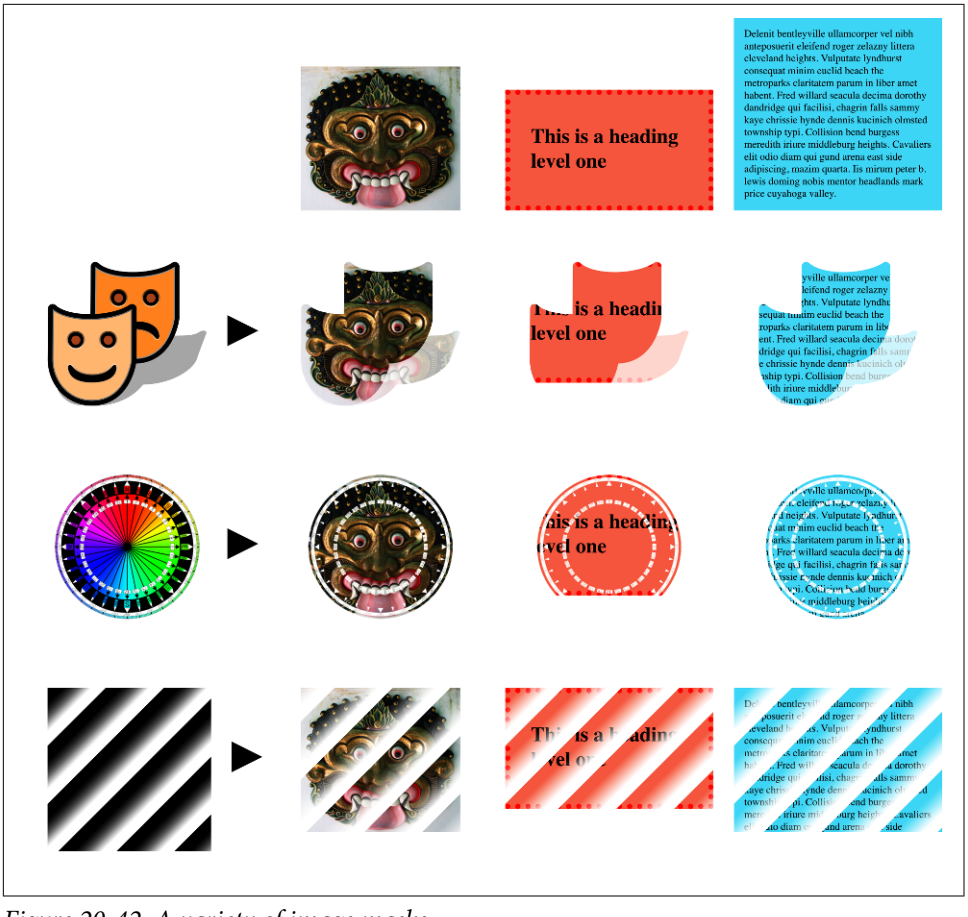


Figure 20-42. A variety of image masks

An important point to keep in mind is that when a mask clips away pieces of an element, it clips away *all* pieces. The best example of this occurs when you apply an image that clips away the outer edges of elements, and the markers on list items can very easily become invisible. Figure 20-43 shows an example, which is the result of the following:

```
*.masked {mask-image: url(i/Compass_masked.png);}

<ol class="masked">
    <li>One</li>
    <li>Two</li>
    <li>Three</li>
    <li>Four</li>
    <li>Five</li>
</ol>
```

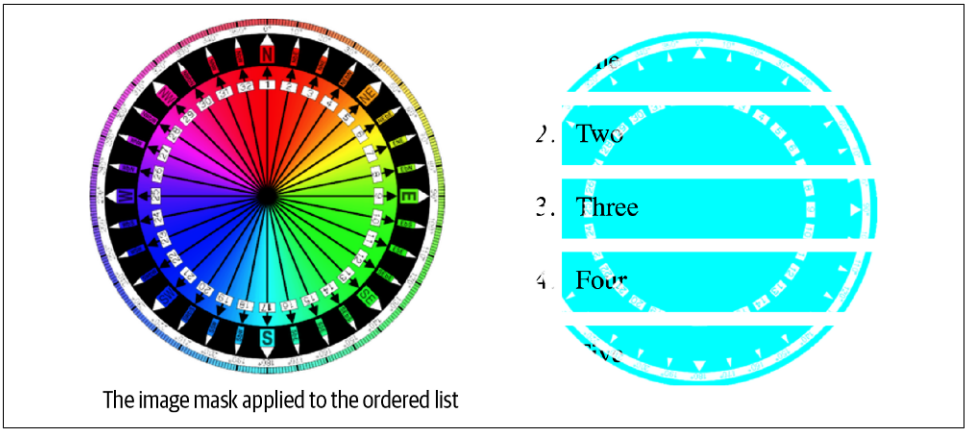


Figure 20-43. A PNG with areas of transparency masking an unordered list

One other value option enables you to point directly at a `<mask>` element in SVG to use the mask it defines. This is analogous to pointing to a `<clipPath>` or other SVG element from the property `clip-path`. Here's an example of how a mask might be defined:

```
<svg>
        <mask id="hexlike">
           <path fill="#FFFFFF"
             d="M 50,0 L 100,25 L 100,75 L 50,100 L 0,75 L 0,25" />
        </mask>
</svg>
```

With that SVG embedded in the HTML file directly, the mask can be referenced like this:

```
.masked {mask-image: url(#hexlike);}
```

If the SVG is in an external file, this is how to reference it from CSS:

```
.masked {mask-image: url(masks.svg#hexlike);}
```

The difference between using an image as a mask versus an SVG `<mask>` is that SVG masking is based on luminance, rather than alpha transparency. This difference can be inverted with the `mask-mode` property.

Changing the Mask's Mode

You've just seen the two ways to use an image as a mask. Masking is accomplished by applying an image with an alpha channel to another element. Masking can also be done by using the brightness of each part of the masking image to define the mask. Switching between these two options is accomplished with the `mask-mode` property.

mask-mode

Values	[`alpha` | `luminance` | `match-source`]#
Initial value	`match-source`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	No

Two of the three values are straightforward: `alpha` means the alpha channel of the image should be used to compute the mask, and `luminance` means the brightness levels should be used. The difference is illustrated in Figure 20-44, which is the result of the following code:

```
img.theatre {mask-image: url(i/theatre-masks.svg);}
img.compass {mask-image: url(i/Compass_masked.png);}
img.lum {mask-mode: luminance;}
```

```
<img src="i/theatre-masks.svg" role="img" alt="theater mask">
<img class="theatre" src="i/mask.jpg" alt="mask">
<img class="theatre lum" src="i/mask.jpg" alt="mask">
<img src="i/Compass_masked.png" alt="mask">
<img class="compass" src="i/mask.jpg" alt="mask">
<img class="compass lum" src="i/mask.jpg" alt="mask">
```

When `luminance` is used to calculate the mask, brightness is treated the same way alpha values are in alpha masking. Consider how alpha masking works: any part of the image with opacity of 0 hides that part of the masked element. A part of the image with opacity of 1 (fully opaque) reveals that part of the masked element.

The same is true with luminance-based masking. A part of the mask with luminosity of 1 reveals that part of the masked element. A part of the mask with luminosity of 0 (fully black) hides that part of the masked element. But note that any fully transparent part of the mask is *also* treated as having a luminance of 0. This is why the shadow portion of the theater-mask image doesn't show any part of the masked image: its alpha value is greater than 0.

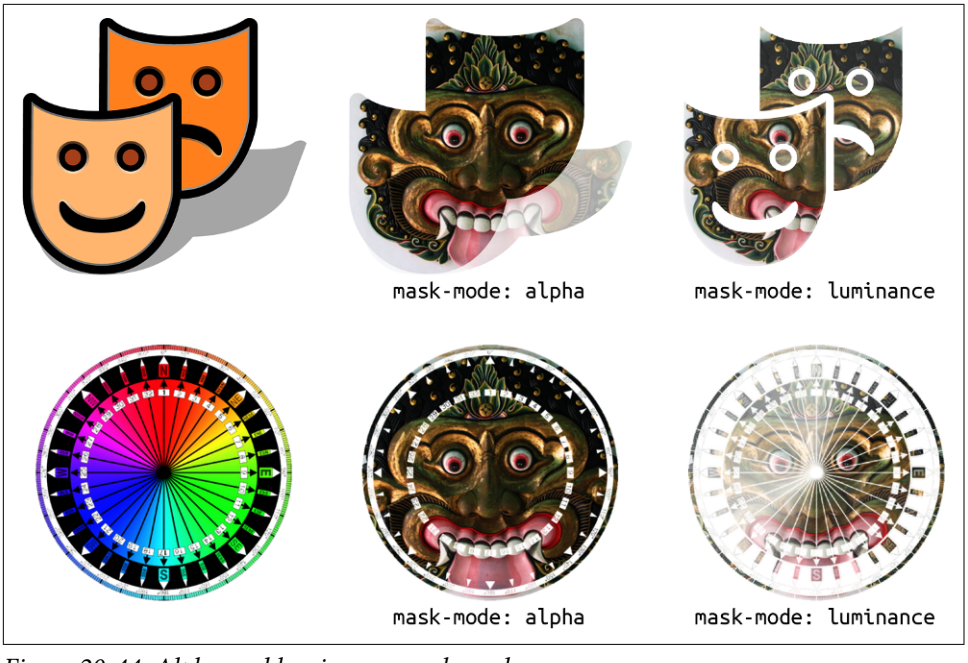


Figure 20-44. Alpha and luminance mask modes

The third (and default) value, `match-source`, is a combination of `alpha` and `luminance`, choosing between them based on the actual source image for the mask as follows:

- If the source is a type of <*image*>, use `alpha`. The <*image*> can be an image such as a PNG or visible SVG, a CSS gradient, or a piece of the page referred to by the `element()` function.
- If the source is an SVG `<mask>` element, use `luminance`.

Sizing and Repeating Masks

Thus far, nearly all the examples have been carefully crafted to make each mask's size match the size of the element it's masking. (This is why we keeping applying masks to images.) In many cases, mask images may be a different size than the masked element. CSS has a couple of ways to deal with this, starting with `mask-size`.

mask-size

Values	[[<*length*> | <*percentage*> | `auto`]{1,2} | `cover` | `contain`]#
Initial value	`auto`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	<*length*>, <*percentage*>

If you've ever sized background images, you know exactly how to size masks, because the value syntax is *exactly* the same, as are the behaviors. As an example, consider the following styles, which have the result shown in Figure 20-45:

```
p {mask-image: url(i/hexlike.svg);}
p:nth-child(1) {mask-size: 100% 100%;}
p:nth-child(2) {mask-size: 50% 100%;}
p:nth-child(3) {mask-size: 2em 3em;}
p:nth-child(4) {mask-size: cover;}
p:nth-child(5) {mask-size: contain;}
p:nth-child(6) {mask-size: 200% 50%;}
```

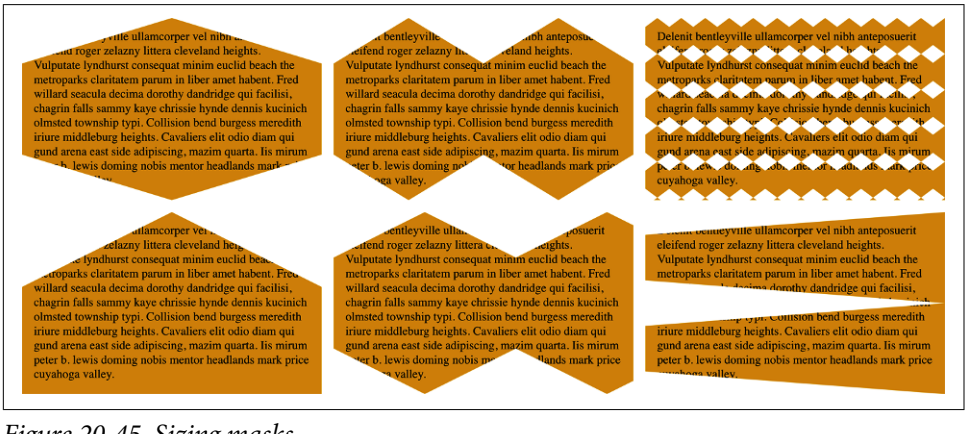


Figure 20-45. Sizing masks

Again, these should be immediately familiar to you if you've ever sized backgrounds. If not, see "Sizing Background Images" on page 343 for a more detailed exploration of the possibilities.

In a like vein, just as the pattern of backgrounds repeating throughout the background area of the element can be changed or suppressed, mask images can be affected with `mask-repeat`.

mask-repeat

Values	[`repeat-x` | `repeat-y` | [`repeat` | `space` | `round` | `no-repeat`]{1,2}]#
Initial value	`repeat`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	Yes
Note	The keywords for `mask-repeat` are reproduced from `background-repeat` and have the same behaviors

The values available here are the same as those for `background-repeat`. Figure 20-46 shows some examples, based on the following styles:

```
p {mask-image: url(i/theatre-masks.svg);}
p:nth-child(1) {mask-repeat: no-repeat; mask-size: 10% auto;}
p:nth-child(2) {mask-repeat: repeat-x; mask-size: 10% auto;}
p:nth-child(3) {mask-repeat: repeat-y; mask-size: 10% auto;}
p:nth-child(4) {mask-repeat: repeat; mask-size: 30% auto;}
p:nth-child(5) {mask-repeat: repeat round; mask-size: 30% auto;}
p:nth-child(6) {mask-repeat: space no-repeat; mask-size: 21% auto;}
```

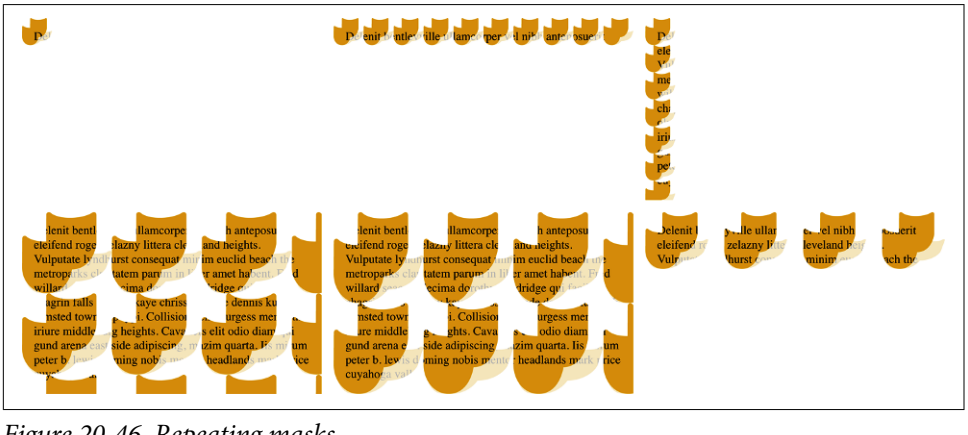


Figure 20-46. Repeating masks

Positioning Masks

Given that sizing and repetition of mask images mirrors the sizing and repetition of background images, you might think that the same is true for positioning the origin mask image, similar to `background-position`, as well as the origin box, similar to `background-origin`. And you'd be exactly right.

mask-position

Values	*<position>#*
Initial value	`0% 0%`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	*<length>*, *<percentage>*
Notes	*<position>* is exactly the same as the values permitted for `background-position`, and has the same behaviors

Once again, if you've ever positioned a background image, you know how to position mask images. Following are a few examples, illustrated in Figure 20-47:

```
p {mask-image: url(i/Compass_masked.png);
       mask-repeat: no-repeat; mask-size: 67% auto;}
p:nth-child(1) {mask-position: center;}
p:nth-child(2) {mask-position: top right;}
p:nth-child(3) {mask-position: 33% 80%;}
p:nth-child(4) {mask-position: 5em 120%;}
```

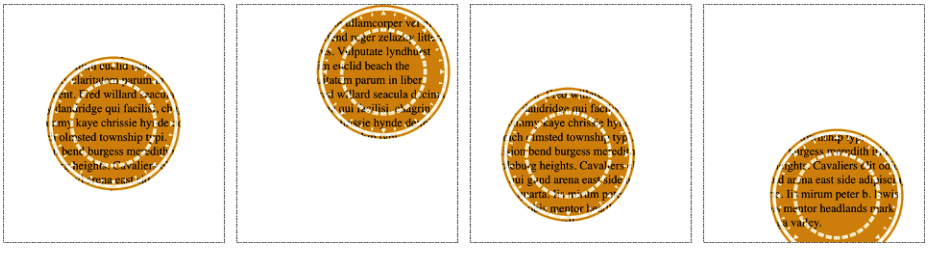


Figure 20-47. Positioning masks

By default, the origin box for mask images is the outer border edge. If you want to move it further inward, or define a specific origin box in an SVG context, then `mask-origin` does for masks what `background-origin` does for backgrounds.

mask-origin

Values	[`content-box` | `padding-box` | `border-box` | `margin-box` | `fill-box` | `stroke-box` | `view-box`]#
Initial value	`border-box`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	No

For the full story, see "Changing the positioning box" on page 328, but for a quick example, see Figure 20-48.



Figure 20-48. Changing the origin box

Clipping and Compositing Masks

One more property echoes backgrounds, and that's `mask-clip`, the mask equivalent of `background-clip`.

mask-clip

Values	[`content-box` | `padding-box` | `border-box` | `margin-box` | `fill-box` | `stroke-box` | `view-box` | `no-clip`]#
Initial value	`border-box`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No

Animatable	No

All this does is clip the overall mask to a specific area of the masked element. In other words, it restricts the area in which the visible parts of the element are in fact visible. Figure 20-49 shows the result of the following styles:

```
p {padding: 2em; border: 2em solid purple; margin: 2em;
        mask-image: url(i/Compass_masked.png);
        mask-repeat: no-repeat; mask-size: 125%;
        mask-position: center;}
p:nth-child(1) {mask-clip: border-box;}
p:nth-child(2) {mask-clip: padding-box;}
p:nth-child(3) {mask-clip: content-box;}
```

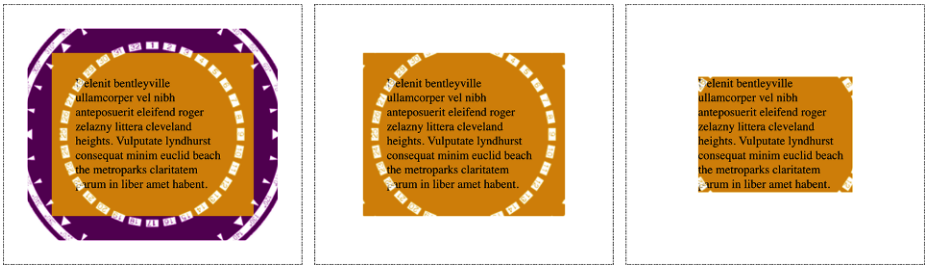


Figure 20-49. Clipping the mask

The last longhand masking property, `mask-composite`, is quite interesting because it can radically change the way multiple masks interact.



As of early 2023, `mask-composite` is supported only by Firefox, but all browsers (even Firefox) support the prefixed form `-webkit-mask-composite`.

mask-composite

Values	[`add` | `subtract` | `intersect` | `exclude`]#
Initial value	`add`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No

Animatable	No

If you are not familiar with compositing operations, a diagram is in order. See Figure 20-50.

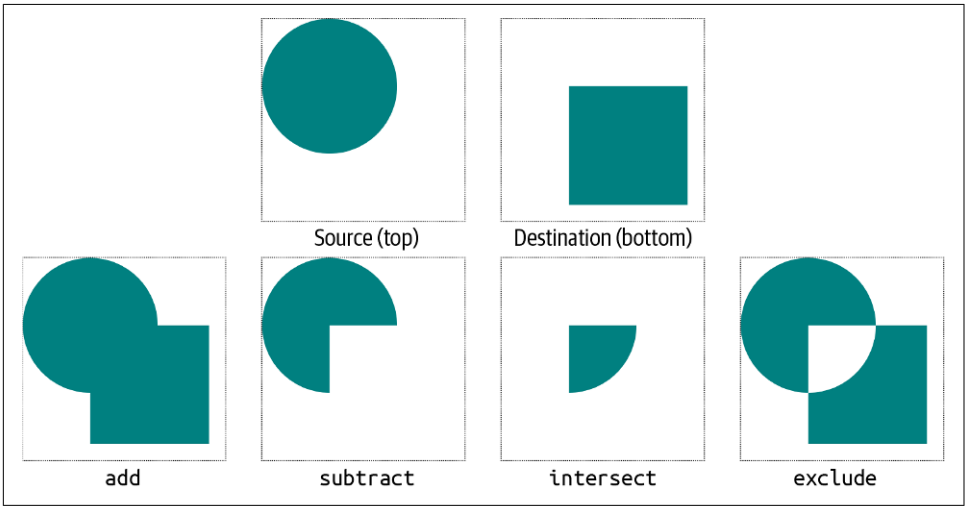


Figure 20-50. Compositing operations

The image on top in the operation is called the *source*, and the image beneath it is called the *destination*.

This doesn't particularly matter for three of the four operations: `add`, `intersect`, and `exclude`, all of which have the same result regardless of which image is the source and which the destination. But for `subtract`, the question is: which image is being subtracted from which? The answer: the destination is subtracted from the source.

The distinction between source and destination also becomes important when compositing multiple masks together. In these cases, the compositing order is from back to front, with each succeeding layer being the source and the already-composited layers beneath it being the destination.

To see why, consider Figure 20-51, which shows the various ways three overlapping masks are composited together, and how results change with changes to their order and compositing operations.

The figure is constructed to show the bottommost mask at the bottom, the topmost above the other two, and the resulting mask at the very top. Thus, in the first column, the triangle and circle are composited with an exclusion operation. The resulting shape is then composited with the square using an additive operation. That results in the mask shown at the top of the first column.

Just remember that when doing a subtraction composite, the bottom shape is subtracted from the shape above it. Thus, in the third column, the addition of the triangle and circle is subtracted from the square above them. This is accomplished with `mask-composite: add, subtract`.

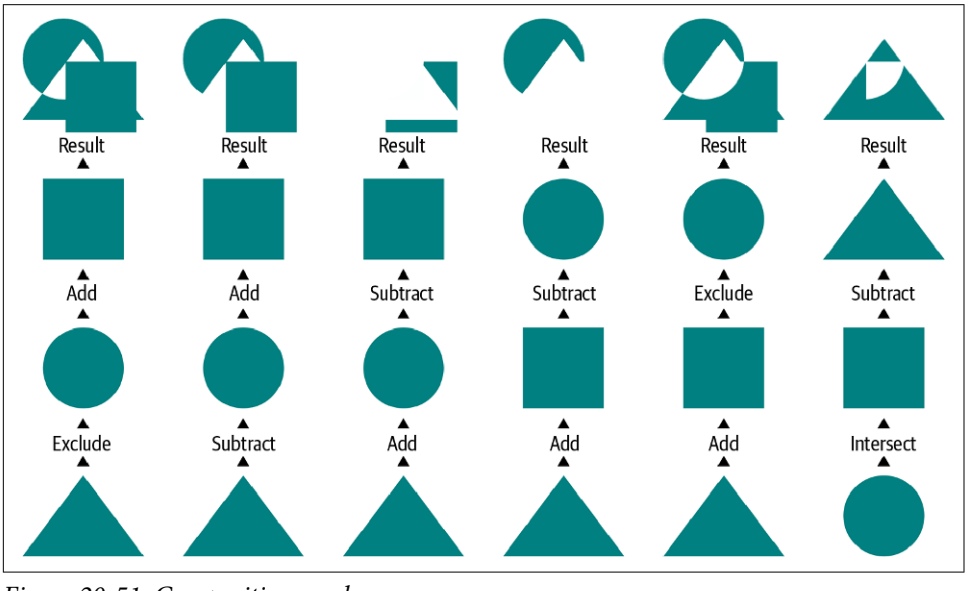


Figure 20-51. Compositing masks

Bringing It All Together

All of the preceding mask properties are brought together in the shorthand property `mask`.

mask

Values	[<*mask-image*> || <*mask-position*> [/ <*mask-size*>]? || <*mask-repeat*> || <*mask-clip*> || <*mask-origin*> || <*mask-composite*> || <*mask-mode*>]#
Initial value	See individual properties
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element)
Computed value	As declared
Inherited	No
Animatable	Refer to individual properties

Like all the other masking properties, `mask` accepts a comma-separated list of masks. The order of the values in each mask can be anything except for the mask size, which always follows the position and is separated from it by a forward slash (`/`).

Thus, the following rules are equivalent:

```
#example {
    mask-image: url(circle.svg), url(square.png), url(triangle.gif);
    mask-repeat: repeat-y, no-repeat;
    mask-position: top right, center, 25% 67%;
    mask-composite: subtract, add;
    mask-size: auto, 50% 33%, contain;
}
#example {
    mask:
      url(circle.svg) repeat-y top right / auto subtract,
      url(square.png) no-repeat center / 50% 33% add,
      url(triangle.gif) repeat-y 25% 67% / contain;
}
```

The triangle and square are added together, and then the result of that additive composite is subtracted from the circle. The result is shown in Figure 20-52 as applied to a square element (the teal shape on the left) and a shape wider than it is tall (the goldenrod shape on the right).



Figure 20-52. Two masks

Setting Mask Types

When you're using CSS to style SVG elements, and you want to set the SVG `<mask>` type, then `mask-type` is for you.

mask-type

Values	`luminance` | `alpha`
Initial value	`luminance`
Applies to	SVG `<mask>` elements
Computed value	As declared

Inherited	No
Animatable	No

This property is similar to `mask-mode`, except there is no `match-source` equivalent. You can choose only `luminance` or `alpha`.

The interesting thing is that if `mask-type` is set for a `<mask>` element that's used to mask an element, and `mask-mode` is declared for that masked element, `mask-mode` wins. As an example, consider the following rules:

```
svg #mask {mask-type: alpha;}
img.masked {mask: url(#mask) no-repeat center/cover luminance;}
```

Given these rules, the masked images will have a mask with luminance compositing, not alpha compositing. If the `mask-mode` value were left at its default value, `match-source`, then `mask-type`'s value would be used instead.

Border-Image Masking

The same specification that defines clipping paths and element masking, CSS Masking, also defines properties that are used to apply masking images in a way that mirrors border-image properties. In fact, with one exception, the properties between border images and border masks are direct analogues, and the values the same. Refer to "Image Borders" on page 276 for a detailed explanation of how these work, but here are some quick recaps.

Remember that without having a border of some sort, none of these properties will have any visible effect. To apply a border and then mask it, you must first declare a border's style, at a minimum. If you intend your masked border to be 10 pixels wide, you would need something like this:

```
border: 10px solid;
```

Once that's been established, you can begin masking the border.



As of late 2022, all these properties are supported in Chromium and WebKit browsers as `-webkit-mask-box-image-*` instead of the names used in the specification. The actually supported names are noted in the property summary boxes that follow, but examples use the standard (unprefixed) property names. Also note: as of this writing, the Gecko (Firefox) family does not support border masks in any form.

mask-border-source

Values	none | <*image*>
Initial value	none
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Computed value	none, or the image with its URL made absolute
Inherited	No
Animatable	No
Note	Supported in Chromium and WebKit only as `-webkit-mask-box-image-source`

The `mask-border-source` property specifies the image to be used as a mask. This can be a URL, gradient, or other supported <*image*> value type. Once the masking image has been set up, you can move on to doing things like slicing it into sections, defining a distinct width for the mask, and so on.

mask-border-slice

Values	[<*number*> | <*percentage*>]{1,4} && `fill`?
Initial value	`100%`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Percentages	Refer to size of the border image
Computed value	As four values, each a number or percentage, and optionally the `fill` keyword
Inherited	No
Animatable	<*number*>, <*percentage*>
Note	Supported in Chromium and WebKit only as `-webkit-mask-box-image-slice`

The `mask-border-slice` property establishes a set of four slice-lines that are laid over the border, and where they fall determines how the mask will be sliced up for use in each of the eight parts of the border area: the top, right, bottom, and left edges, as well as the top-left, top-right, bottom-right, and bottom-left corners. The property takes up to four values, defining (in order) offsets from the top, right, bottom, and left edges.



As of late 2022, no logical-property equivalent exists for `mask-border-slice`. If the proposed addition of a `logical` keyword, or something equivalent, to this property is ever adopted and implemented, at that point it will be possible to use `mask-border-slice` in a writing-flow-relative fashion.

Consider the following, diagrammed in Figure 20-53:

```
#one {mask-border-slice: 25%;}
#two {mask-border-slice: 10% 20%;}
#thr {mask-border-slice: 10 20 15 30;}
```

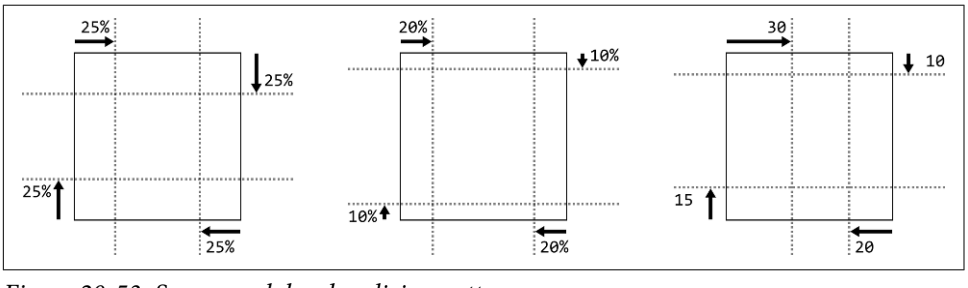


Figure 20-53. Some mask border-slicing patterns

You might think that numeric offsets need to be given a length unit to define a distance, but this is not so. Number values are interpreted in the coordinate system of the image used for the mask. With a raster image like a PNG, the coordinate system will be the pixels of the image. In an SVG image, the coordinate system defined by the SVG file is used.

Using the optional `fill` keyword causes the center portion of the mask image to be applied to the element inside the border area. By default, it is not used, allowing the element's padding and content to be fully seen. If you do use it by adding `fill`, the part of the mask image inside the four slice lines will be stretched over the element's content and padding, and applied to them. Consider the following, illustrated in Figure 20-54:

```
p {mask-border-image: url(circles.png);}
p.one {mask-border-slice: 33%;}
p.two {mask-border-slice: 33% fill;}
```

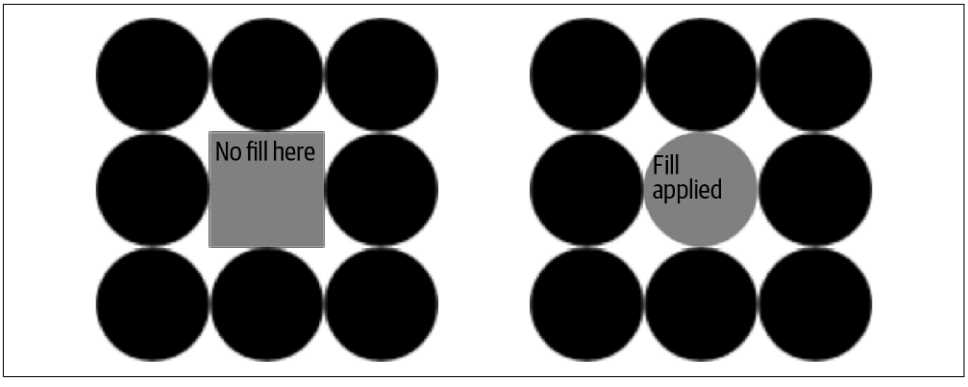


Figure 20-54. Applying the mask fill



As of late 2022, a bug in browsers that support the prefixed property causes the content and padding of an element to be completely hidden unless the `fill` keyword is used. Thus, in order to use border masks and show the content of an element, you need to fill the center of the mask image completely, and use `fill`.

mask-border-width

Values	[<*length*> | <*percentage*> | <*number*> | `auto`]{1,4}
Initial value	`1`
Applies to	All elements, except table elements when `border-collapse` is `collapse`
Percentages	Relative to width/height of the entire border image area—that is, the outer edges of the border box
Computed value	Four values: each a percentage, number, `auto` keyword, or <*length*> made absolute
Inherited	No
Animatable	Yes
Note	Values can never be negative; supported in Chromium and WebKit only as `-webkit-mask-box-image-width`

This property allows you to define a width (or individual widths) for the four edge slices of the border mask. If the slices are not actually the size(s) you declare, they will be resized to fit. For example, a masking image might be sliced and then sized as follows:

```
mask-border-slice: 33%; mask-border-width: 1em;
```

This allows you to slice up the masking image in one way, and then size it as needed for the context or define a universal size for masking image, regardless of the context in which it appears.

mask-border-outset

Values	[<*length*> | <*number*>]{1,4}
Initial value	`0`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Percentages	N/A
Computed value	Four values, each a number or <*length*> made absolute
Inherited	No
Animatable	Yes
Note	Supported in Chromium and WebKit only as `-webkit-mask-box-image-outset`

With `mask-border-outset`, you can push the mask outside the border area. This is useful only if you're already pushing a border image outside the border area with `border-image-outset` and want to also apply the mask to that border image, or if you've applied an outline to the element and want to mask that as well. If neither is true, the masked area outside the border will mask only the margin area, which is already transparent and so can't be visibly altered.



As of late 2022, browsers supporting the prefixed property not only push the slices outward, but also expand the center area by the given amount, scaling up the masked area covered by the center slice in the process. This behavior is not called for or apparently supported by the specifications current as of this writing, and is most likely a bug (unless the behavior is eventually made retroactively correct by a CSS Working Group decision).

mask-border-repeat

Values	[`stretch` | `repeat` | `round` | `space`]{1,2}
Initial value	`stretch`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Computed value	Two keywords, one for each axis
Inherited	No
Animatable	No
Note	Supported in Chromium and WebKit only as `-webkit-mask-box-image-repeat`

Thus far, our only example of border masking has used a masking image that is an exact fit for the element it's masking. This is unlikely to be the case, since elements can be resized by any number of factors. The default is to stretch each slice to fit its part of the border area, but other options are possible. Figure 20-55 illustrates the options (center areas have been removed for clarity).

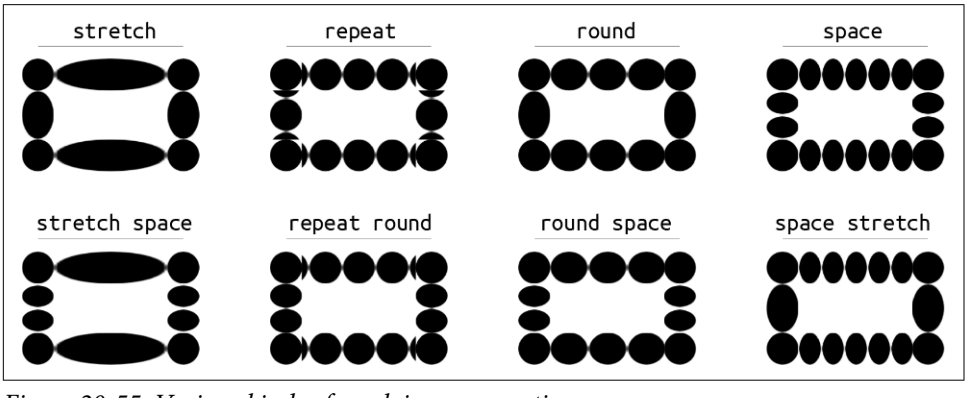


Figure 20-55. Various kinds of mask image repeating

As shown in Figure 20-55, `mask-border-repeat` can accept one or two repeat values. If one is given, it's applied to all sides of the border area. If two are given, the first applies to the horizontal sides of the border area, and the second to the vertical sides.

Border masks have one styling aspect that image borders do not, and it's set with the property `mask-border-mode`.

mask-border-mode

Values	`alpha` | `luminance`
Initial value	`alpha`
Applies to	All elements (in SVG, applies to all graphics elements and all container elements except the `<defs>` element, all graphics elements, and the `<use>` element)
Computed value	As specified
Inherited	No
Animatable	Discrete
Note	Not yet supported in any browser, even with a `-webkit-` prefix

The `mask-border-mode` property sets whether the masking mode is alpha based, or luminance based. For more details on the difference, see the `mask-mode` property discussed earlier in the chapter.

mask-border

Values	*<mask-border-source>* ‖ *<mask-border-slice>* [/ *<mask-border-width>*? [/ *<mask-border-outset>*]?]? ‖ *<mask-border-repeat>* ‖ *<mask-border-mode>*
Initial value	See individual properties
Applies to	See individual properties
Computed value	See individual properties
Inherited	No
Animatable	See individual properties
Note	Supported in Chromium and WebKit only as `-webkit-mask-box-image` without the `mask-border-mode` value

The property `mask-border` incorporates all of the previous border-masking properties into one convenient shorthand.

Object Fitting and Positioning

One more variety of masking applies solely to replaced elements like images. With `object-fit`, you can change the way the replaced element fills its element box—or even have it not fill that box completely.

object-fit

Values	`fill` | `contain` | `cover` | `scale-down` | `none`
Initial value	`fill`
Applies to	Replaced elements
Computed value	As declared
Inherited	No
Animatable	No

If you've ever worked with `background-size`, these values probably look familiar. They do similar things, too, only with replaced elements.

For example, assume a 50 × 50 pixel image. We can change its size via CSS with something like this:

```
img {width: 250px; height: 150px;}
```

The default expectation is that these style declarations will stretch the 50 × 50 image to be 250 × 150. And if `object-fit` is its default value, `fill`, that's exactly what happens.

Change the value of `object-fit`, however, and other behaviors occur. The following examples are illustrated in Figure 20-56:

```
img {width: 250px; height: 150px; background: silver; border: 3px solid;}
img:nth-of-type(1) {object-fit: none;}
img:nth-of-type(2) {object-fit: fill;}
img:nth-of-type(3) {object-fit: cover;}
img:nth-of-type(4) {object-fit: contain;}
```

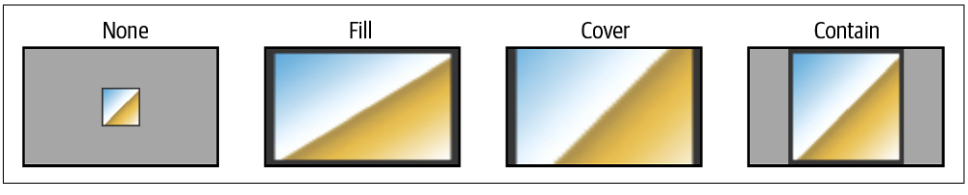


Figure 20-56. Four kinds of object fitting

In the first instance, `none`, the `<img>` element is drawn 250 pixels wide by 150 pixels tall. The image itself, however, is drawn 50 × 50 pixels—its intrinsic size—because it was directed to *not* fit the element box. The second instance, `fill`, is the default behavior, as mentioned. This is the only value that may distort the image, as the dimensions are the element's dimensions, not the image's intrinsic size.

In the third instance, `cover`, the image is scaled up until no part of the element box is left "uncovered"—but the image itself keeps its intrinsic aspect ratio. In other words, the image stays a square. In this case, the longest axis of the `<img>` element is `250px` long, so the image is scaled up to be 250 × 250 pixels. That 250 × 250 image is then placed in the 250 × 150 `<img>` element.

The fourth instance, `contain`, is similar, except the image is only big enough to touch two sides of the `<img>` element. This means the image is 150 × 150 pixels, and placed into the 250 × 150 pixel box of its `<img>` element.

To reiterate, what you see in Figure 20-56 is four `<img>` elements. There are no wrapper `<div>` or `<span>` or other elements around those images. The border and background color are part of the `<img>` element. The image placed inside the `<img>` element is fitted according to `object-fit`. The element box of the `<img>` element then acts rather like it's a simple mask for the fitted image inside it. (And then you can mask and clip the element box with the properties covered earlier in this chapter.)

A fifth value for `object-fit`, not represented in Figure 20-56, is `scale-down`. The meaning of `scale-down` is "do the same as either `none` or `contain`, whichever leads to a smaller size." This lets an image always be its intrinsic size unless the `<img>` element gets too small, in which case it's scaled down à la `contain`. This is illustrated in Figure 20-57,

where each <img> element is labeled with the height values it's been given; the width in each case is 100px.

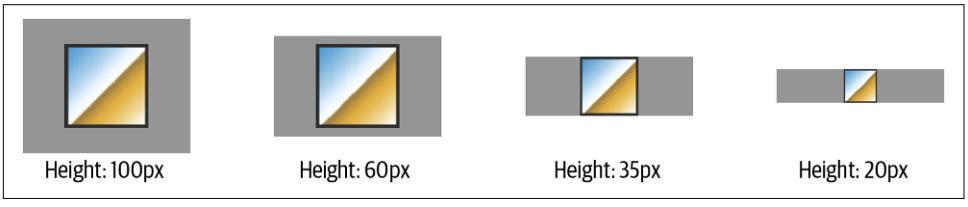


Figure 20-57. Various scale-down scenarios

So if a replaced element is bigger or smaller than the element box into which it's being fit, how can we affect its alignment within that box? Using object-position is the answer.

object-position

Values	*<position>*
Initial value	50% 50%
Applies to	Replaced elements
Computed value	As declared
Inherited	No
Animatable	Yes
Notes	*<position>* is exactly the same as the values permitted for background-position, and has the same behaviors

The value syntax here is just like that for mask-position or background-position, allowing you to position a replaced element within its element box if it isn't set to object-fit: fill. Thus, given the following CSS, we get the result shown in Figure 20-58:

```
img {width: 200px; height: 100px; background: silver; border: 1px solid;
     object-fit: none;}
img:nth-of-type(2) {object-position: top left;}
img:nth-of-type(3) {object-position: 67% 100%;}
img:nth-of-type(4) {object-position: left 142%;}
```

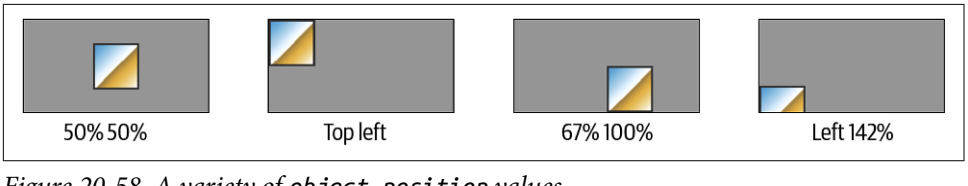


Figure 20-58. A variety of object-position values

Notice that the first example has a value of `50% 50%`, even though that isn't present in the CSS code. That illustrates that the default value of `object-position` is `50% 50%`. The next two examples show how various `object-position` values move the image around within the `<img>` element box.

As the last example shows, it's possible to move an unscaled replaced element like an image so that it's partly clipped by its element box. This is similar to positioning background images or masks so that they are clipped at the element boundaries.

It's also possible to position fitted elements that are larger than the element box, as can happen with `object-fit: cover`, although the results can be very different than with `object-fit: none`. The following CSS will have results like those shown in Figure 20-59:

```
img {width: 200px; height: 100px; background: silver; border: 1px solid;
     object-fit: cover;}
img:nth-of-type(2) {object-position: top left;}
img:nth-of-type(3) {object-position: 67% 100%;}
img:nth-of-type(4) {object-position: left 142%;}
```

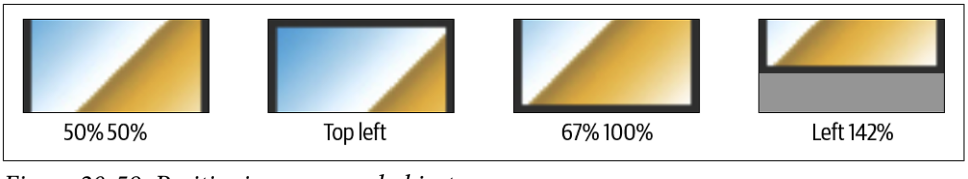


Figure 20-59. Positioning a covered object

If any of these results confuse you, review "Positioning Background Images" on page 319 for more details.

Summary

With all of the effects available to CSS authors, we have an infinite variety of outcomes and thus an infinite variety of creative presentation of elements. Whether it's altering elements' appearances with filters, changing how they're composited with their backdrops, clipping or masking parts of elements, or altering the way images fill out their element boxes, there have never been more options at your fingertips.

CHAPTER 21

CSS At-Rules

For 20 chapters now, we've explored the properties, values, and selectors that can be combined to create CSS rules. These are what we might call *normal rules* or *regular rules*, and they're powerful, but sometimes more is needed. Sometimes there needs to be a way to encapsulate certain styles in conditional blocks, such that styles can be applied at certain page widths or only if a given CSS feature is recognized by the browser processing the stylesheet.

These are almost invariably enclosed in *at-rules*, so called because they start with an at (`@`) symbol. You've seen some of these in previous chapters, such as `@font-face` and `@counter-style`, but there are still more that aren't so tightly bound to specifics of styling. This chapter explores the three powerful at-rules `@media`, `@container`, and `@supports`.

Media Queries

Thanks to the mechanisms defined in HTML and CSS called *media queries*, you can restrict any set of styles (including entire stylesheets) to a specific medium, such as screen or print, and to a specific set of media conditions. These mechanisms allow you to define a combination of media types and conditions such as display size or color depth, to pick two examples. We'll cover the basic form before exploring the more complex forms.

Basic Media Queries

For HTML-based stylesheets, you can impose medium restrictions through the `media` attribute. This works the same for both the `<link>` and `<style>` elements:

```
<link rel="stylesheet" media="print"
    href="article-print.css">
<style media="print">
    body {font-family: sans-serif;}
</style>
```

The `media` attribute can accept a single medium value or a comma-separated list of values. Thus, to link in a stylesheet that should be used in only the `screen` and `print` media, you would write this:

```
<link rel="stylesheet" media="screen, print"
    href="visual.css">
```

In a stylesheet itself, you can also impose medium restrictions on `@import` rules:

```
@import url(visual.css) screen;
@import url(article-print.css) print;
```

Remember that if you don't add medium information to a stylesheet, it will be applied in *all* media. Therefore, if you want one set of styles to apply only onscreen, and another to apply only in print, you need to add medium information to both stylesheets. For example:

```
<link rel="stylesheet" media="screen"
    href="article-screen.css">
<link rel="stylesheet" media="print"
    href="article-print.css">
```

If you were to remove the `media` attribute from the first `<link>` element in this example, the rules found in the stylesheet *article-screen.css* would be applied in *all* media.

CSS also defines syntax for `@media` blocks. This allows you to define styles for multiple media within the same stylesheet. Consider this basic example:

```
<style>
body {background: white; color: black;}
@media screen {
    body {font-family: sans-serif;}
    h1 {margin-top: 1em;}
}
@media print {
    body {font-family: serif;}
    h1 {margin-top: 2em; border-bottom: 1px solid silver;}
}
</style>
```

Here we see that in all media, the `<body>` element is given a white background and a black foreground by the first rule. This happens because its stylesheet, the one defined by the `style` attribute, has no `media` attribute and thus defaults to `all`. Next, a block of rules is provided for the `screen` medium alone, followed by another block of rules that applies only in the `print` medium.



The indentation shown in these blocks is solely for purposes of clarity. You don't have to indent the rules found inside an `@media` block, but you're welcome to do so if it makes your CSS easier to read.

The `@media` blocks can be any size, containing any number of rules. When authors have control over a single stylesheet, such as in a shared hosting environment or a CMS that restricts what users can edit, `@media` blocks may be the only way to define medium-specific styles. This is also the case when CSS is used to style a document using an XML language that does not contain a `media` attribute or its equivalent.

These are the three most widely recognized media types:

`all`
: Use in all presentational media.

`print`
: Use when printing the document for sighted users, and also when displaying a print preview of the document.

`screen`
: Use when presenting the document in a screen medium like a desktop computer monitor or a handheld device. All web browsers running on such systems are screen-medium user agents.

It's entirely possible that new media types will be added over time, so remember that this limited list may not always be so limited. It's fairly easy to imagine `augmented-reality` as a media type, for example, since text in AR displays would likely need to be of higher contrast in order to stand out against the background reality.

HTML4 defined a list of media types that CSS originally recognized, but most have been deprecated and should be avoided. These are `aural`, `braille`, `embossed`, `handheld`, `projection`, `speech`, `tty`, and `tv`. If you have old stylesheets that use these media types, they should almost certainly be converted to one of the three recognized media types, if possible.



As of 2022, a couple of browsers still support `projection`, which allows a document to be presented as a slideshow. Several mobile-device browsers also support the `handheld` type, but not in consistent ways.

It's possible in some circumstances to combine media types into comma-separated lists, though the rationale for doing so isn't terribly compelling, given the small number of media types currently available. For example, styles could be restricted to only screen and print media in the following ways:

```
<link rel="stylesheet" media="screen, print"
    href="article.css">

@import url(article.css) print, screen;

@media screen,print {
    /* styles go here */
}
```

Complex Media Queries

In the previous section, you saw how multiple media types could be chained together with a comma. We might call that a *compound media query*, because it allows us to address multiple media at once. There is a great deal more to media queries, though: it's possible to apply styles based not just media types, but also features of those media, such as display size or color depth.

This is a great deal of power, and it's not enough to rely on commas to make it all happen. Thus, CSS includes the logical operator `and` to pair media types with features of those media.

Let's see how this plays out in practice. Here are two essentially equivalent ways of applying an external stylesheet when rendering the document on a color printer:

```
<link href="print-color.css"
    media="print and (color)" rel="stylesheet">

@import url(print-color.css) print and (color);
```

Anywhere a media type can be given, a media query can be constructed. This means that, following on the examples of the previous section, it is possible to list more than one query in a comma-separated list:

```
<link href="print-color.css"
   media="print and (color), screen and (color)" rel="stylesheet">

@import url(print-color.css) print and (color), screen and (color);
```

If even one of the media queries evaluates to `true`, the associated stylesheet is applied. Thus, given the previous `@import`, *print-color.css* will be applied if rendering to a color printer *or* to a color screen environment. If printing on a black-and-white printer, both queries will evaluate to `false` and *print-color.css* will not be applied to the document. The same holds true in a grayscale screen environment, any speech media environment, and so forth.

Each media descriptor is composed of a media type and one or more listed media features, with each media feature descriptor enclosed in parentheses. If no media type is provided, it is assumed to be `all`, which makes the following two examples equivalent:

```
@media all and (min-resolution: 96dpi) {…}
@media (min-resolution: 96dpi) {…}
```

Generally speaking, a media feature descriptor is formatted like a property-value pair in CSS, only enclosed by parentheses. A few differences exist, most notably that some features can be specified without an accompanying value. For example, any color-based medium will be matched using `(color)`, whereas any color medium using a 16-bit color depth is matched using `(color: 16)`. In effect, the use of a descriptor without a value is a true/false test for that descriptor: `(color)` means "is this medium in color?"

Multiple feature descriptors can be linked with the `and` logical keyword. In fact, there are two logical keywords in media queries:

`and`
: Links together two or more media features in such a way that all of them must be true for the query to be true. For example, `(color) and (orientation: landscape) and (min-device-width: 800px)` means that all three conditions must be satisfied: if the media environment has color, is in landscape orientation, *and* the device's display is at least 800 pixels wide, then the stylesheet is used.

`not`
: Negates the entire query so that if all of the conditions are true, the stylesheet is not applied. For example, `not (color) and (orientation: landscape) and (min-device-width: 800px)` means that if the three conditions are satisfied, the statement is negated. Thus, if the media environment has color, is in landscape orientation, and the device's display is at least 800 pixels wide, then the stylesheet is *not* used. In all other cases, it will be used.

CSS has no `or` logical keyword, as its role is served by the comma, as shown previously.

Note that the `not` keyword can be used only at the beginning of a media query. It is not presently legal to write something like `(color) and not (min-device-width: 800px)`. In such cases, the entire query block will be ignored.

Let's consider an example of how all this plays out:

```
@media screen and (min-resolution: 72dpi) {
        .cl01 {font-style: italic;}
}
@media screen and (min-resolution: 32767dpi) {
        .cl02 {font-style: italic;}
}
@media not print {
        .cl03 {font-style: italic;}
}
@media not print and (monochrome) {
        .cl04 {font-style: italic;}
}
```

Figure 21-1 shows the result, but bear in mind that, even though you may be reading this on printed paper, the actual image was generated with a screen-medium browser (Firefox Nightly, as it happens) displaying an HTML document with the previous CSS applied to it. So everything you see in Figure 21-1 was operating under a `screen` medium.

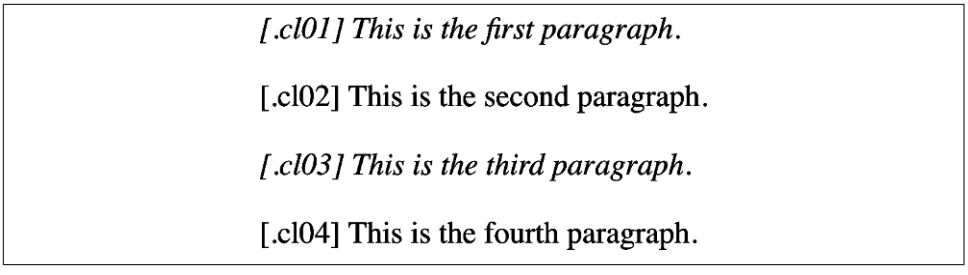


Figure 21-1. Logical operators in media queries

The first line is italicized because the screen on which the file was displayed had a resolution equal to or greater than 72 dots per inch. Its resolution was not, however, `32767dpi` or higher, so the second media block is skipped and thus the second line stays unitalicized. The third line is italicized because, being a screen display, it was `not print`. The last line is italicized because it was either not print or not monochrome—in this case, not monochrome.

Another keyword, `only`, was designed to create deliberate backward incompatibility. Yes, really.

`only`
: Used to hide a stylesheet from browsers old enough that they understand media queries but not media types. (This is almost never a problem in modern usage, but the capability was created and so we document it here.) In browsers that *do* understand media types, the `only` keyword is ignored and the stylesheet is applied. In browsers that do not understand media types, the `only` keyword creates an apparent media type of `only all`, which is not valid.

Special Value Types

Two value types were introduced by media queries. These types are used in conjunction with specific media features, which are explained later in the chapter:

<ratio>
: Two numbers separated by a forward slash (`/`), defined in Chapter 5.

<resolution>
: A resolution value is a positive *<integer>* followed by either of the unit identifiers `dpi` or `dpcm`. In CSS terms, a *dot* is any display unit, the most familiar of which is the pixel. As usual, whitespace is not permitted between the *<integer>* and the identifier. Therefore, a display that has exactly 150 pixels (dots) per inch is matched with `150dpi`.

Keyword Media Features

So far you've seen several media features in the examples, but not a complete list of the possible features and their values. Let's fix that now!

Note that none of the following values can be negative, and that media features are always enclosed in parentheses:

Media feature: `any-hover`

Values: `none` | `hover`
: Checks for any available input mechanism that can hover over elements (i.e., trigger a `:hover` state). The `none` value means there are no such mechanisms, or no mechanisms that can do so conveniently. Compare with the `hover` media feature, which restricts checking to the primary input mechanism.

Media feature: `any-pointer`

Values: `none` | `coarse` | `fine`
: Checks for an input mechanism that creates an onscreen pointer. The `none` values indicates no such devices, `coarse` indicates at least one device with limited accuracy (e.g., a finger), and `fine` indicates at least one device with high accuracy (e.g., a mouse). Compare with `pointer`, which restricts checking to the primary input mechanism.

Media feature: `color-gamut`

Values: `srgb` | `p3` | `rec2020`
: Tests the range of colors supported by both the browser and the output device. As of late 2022, the majority of displays support the `srgb` and `p3` gamuts. The `p3` value refers to the Display P3 color space, which is a superset of sRGB. The `rec2020` value refers to the gamut specified by the ITU-R Recommendation BT. 2020 Color Space, which is a superset of P3. The `color-gamut` media feature is not supported by Firefox as of late 2022.

Media feature: `display-mode`

Values: `fullscreen` | `standalone` | `minimal-ui` | `browser`
: Tests the display mode of the top-level browsing context and any child browsing contexts. This corresponds to the Web Application Manifest specification's `display` member, and is commonly used to check if a progressive web application visitor is browsing a website or on an installed application, but applies whether or not a manifest has been defined. See "Forced Colors, Contrast, and Display Mode" on page 1035 for details.

Media feature: `dynamic-range`

Values: `standard` | `high`
: Checks whether the browsing context supports a high dynamic range for visual output. The `high` value means the media environment supports high peak brightness, a high contrast ratio, and a 24-bit color depth or higher. There are no precisely defined values for high peak brightness or color contrast, so this is left to browsers to decide. Any device that matches `high` will also match `standard`. The `dynamic-range` media feature achieved widespread browser support in early 2022.

Media feature: `forced-colors`

Values: `none` | `active`
: Checks whether the browser is in *forced-color* mode, which forces browser-default values for a set of CSS properties such as `color` and `background-color`, and specific values for a handful of others, and may also trigger a `prefers-color-scheme` value. See "Forced Colors, Contrast, and Display Mode" on page 1035 for details. The `forced-colors` media feature is not supported by WebKit as of late 2022.

Media feature: `grid`

Values: `0` | `1`
: Refers to the presence (or absence) of a grid-based output device, such as a TTY terminal. This does *not* refer to CSS Grid. A grid-based device will return `1`; otherwise, `0` is returned. This media feature can be used in place of the old `tty` media descriptor.

Media feature: `hover`

Values: `none` | `hover`
: Checks whether the user's *primary* input mechanism can hover over elements. The `none` value means the primary mechanism cannot hover, or cannot do so conveniently; an example of the latter is a mobile device that pretends to hover when an inconvenient tap-and-hold action is performed. The `hover` value means hovering is convenient, such as with a mouse. Compare to `any-hover`, which checks whether any mechanism permits hovering, not just the primary.

Media feature: `inverted-colors`

Values: `none` | `inverted`
: Checks whether colors are being inverted by the underlying operating system. The `none` value means colors are being displayed normally; `inverted` means that all pixels in the display area are being inverted. The `inverted-colors` media feature is supported only in WebKit as of late 2022.

Media feature: `orientation`

Values: `portrait` | `landscape`
: Refers to the orientation of the user agent's display area, where `portrait` is returned if the media feature `height` is equal to or greater than the media feature `width`. Otherwise, the result is `landscape`.

Media feature: `overflow-block`

Values: `none` | `scroll` | `optional-paged` | `paged`
: Checks how the output device handles content that overflows along the block axis. The `none` value means the overflowed content cannot be accessed; `scroll` means the content can be accessed by scrolling to it in some way; `optional-paged` means the user can scroll to the content, but page breaks can be manually triggered using properties like `break-inside`; `paged` means overflowing content can be accessed only by "paging" to see the content, as in an ebook. The `overflow-block` media feature is supported only in Firefox as of late 2022.

Media feature: `overflow-inline`

Values: `none` | `scroll`
: Checks to see how the output device handles content that overflows along the inline axis. The `none` value means the overflowed content cannot be accessed; `scroll` means the content can be accessed by scrolling to it in some way. The `overflow-inline` media feature is supported only in Firefox as of late 2022.

Media feature: `pointer`

Values: `none` | `coarse` | `fine`
: Checks whether the *primary* input mechanism creates an onscreen pointer. The `none` value means the primary input device generates no pointer, `coarse` means it does but with limited accuracy, and `fine` means it does with high accuracy (e.g., a mouse). Compare to `any-pointer`, which checks whether any mechanism creates a pointer, not just the primary.

Media feature: `prefers-color-scheme`

Values: `light` | `dark`
: Checks which color scheme the user has selected at the browser or operating system level (i.e., Light mode or Dark mode). Thus, the author can define specific color values for, say, `prefers-color-scheme: dark`. Safari adds a `no-preference` value, but this has not been standardized or adopted by other browsers as of late 2022.

Media feature: `prefers-contrast`

Values: `no-preference` | `less` | `more` | `custom`
: Checks whether the user has set a preference for high-contrast output, at either the browser or operating system level (e.g., Windows High Contrast mode). See "Forced Colors, Contrast, and Display Mode" on page 1035 for details.

Media feature: `prefers-reduced-motion`

Values: `no-preference` | `reduce`
: Checks whether the user has set a preference regarding motion, at either the browser or operating system level. The `reduce` value means the user has indicated they wish motion to be reduced or eliminated, possibly because of vestibular disorders that create discomfort when viewing motion onscreen. Transitions and animations should most often be put into a `prefers-reduced-motion: reduce` block for accessibility reasons.

Media feature: `scan`

Values: `progressive` | `interlace`
: Refers to the scanning process used in an output device. The `interlace` value is the type generally used in CRT and some plasma displays. As of late 2022, all known implementations match the `progressive` value, making this media feature somewhat useless.

Media feature: `scripting`

Values: `none` | `initial-only` | `enabled`
: Checks whether a scripting language such as JavaScript is available. The `initial-only` value means scripting can be performed only at page load, but not thereafter. The `scripting` media feature is not supported by any browser as of late 2022.

Media feature: `update`

Values: `none` | `slow` | `fast`
: Checks whether the content's appearance can be changed after page load. The `none` value means no updates are possible, such as in print media. The `slow` value means changes are possible but cannot be animated smoothly because of device or browser constraints. The `fast` value means smooth animations are possible. The `update` media feature is supported only by Firefox as of late 2022.

Media feature: `video-dynamic-range`

Values: `standard` | `high`

Checks whether the browsing context supports a high dynamic range for visual output on videos. This is useful because some devices render video separately from other graphics, and so may support a different dynamic range for video than for other content. The `high` value means the media environment supports high peak brightness, a high contrast ratio, and a 24-bit color depth or higher. There are no precisely defined values for high peak brightness or color contrast, so this is left to browsers to decide. Any device that matches `high` will also match `standard`. The `video-dynamic-range` media feature achieved widespread browser support in early 2022.

Forced Colors, Contrast, and Display Mode

Three of the previously defined media features relate to user preference in their display, and allow you to detect those preferences so you may style accordingly. Two are closely intertwined, so we'll start with them.

If a user has gone to the effort of defining a specific set of colors to be used in the display of their content, such as with Windows High Contrast mode, then `forced-colors: active` will be matched, as will `prefers-contrast: custom`. You can use one or both of these queries to apply specific styles under such conditions.

If `forced-colors: active` returns true, the following CSS properties will be forced to use the browser (or operating system) default values, overriding any values you may have declared:

- `background-color`
- `border-color`
- `color`
- `column-rule-color`
- `outline-color`
- `text-decoration-color`
- `text-emphasis-color`
- `-webkit-tap-highlight-color`

Also, the SVG `fill` and `stroke` attributes will be ignored and set to their default values.

Additionally, the following property-value combinations are enforced over whatever the author has declared:

- `box-shadow: none`
- `text-shadow: none`
- `background-image: none` for values that are not URL-based (e.g., gradients)
- `color-scheme: light dark`
- `scrollbar-color: auto`

This means that, to pick one example, any element whose hover or focus styles depend on changing the color of a border will fail to have an effect. Thus, you could provide a change of font weight and border style (not color) instead:

```
nav a[href] {border: 3px solid gray;}
nav a[href]:is(:hover, :focus) {border-color: red;}

@media (forced-colors: active) {
        :hover {font-weight: bold; border-style: dashed;}
}
```

This is an example of the sorts of changes you should make to accommodate forced-color situations, providing greater usability through small changes. You *should not* use this query to set up an entire separate design for users who have forced certain colors.

As noted previously, if a user has set things up such that `forced-colors: active` is triggered, `prefers-contrast: custom` will also be triggered. The meanings of this media feature's values are as follows:

`no-preference`
: The browser and/or operating system are not aware of a user preference with regards to color contrast.

`less`
: The user has requested interfaces with less contrast than usual. Examples of this could be users with a propensity for migraine headaches or dyslexia, as some (not all) dyslexics find high-contrast text difficult to parse.

`more`
: The user has requested interfaces with more contrast than usual.

`custom`
: The user has defined a specific set of colors that are not matched by either `more` or `less`, such as the Windows High Contrast mode.

It is possible to query for any value by not supplying a value, which is especially useful in this scenario. You might cater to both low- and high-contrast users as follows:

```
body {background: url(/assets/img/mosaic.png) repeat;}

@media (prefers-contrast) {
        body {background-image: none;}
}
```

The `display-mode` media feature is entirely different from the previous two features. The `display-mode` media feature lets authors determine the kind of display environment being used and act accordingly.

First let's define what the various values mean:

`fullscreen`
: The application takes up the entire available display area and does not show any application chrome (e.g., address bar, back button, status bar, etc.).

`standalone`
: The application appears like a native standalone application. This removes application chrome such as address bar, but will make operating-system-derived navigation elements like back buttons available.

`minimal-ui`
: The application appears similar to a native standalone application, but provides a way to access application chrome for things like address bars, the application's navigation controls, and so on. System-specific interface controls for things like "share" or "print" may also be included.

`browser`
: The application appears as normal, showing the entire application chrome including things like the complete address bar with forward/back/home buttons, scrollbar gutters, and so on.

These various states can be triggered either by the user putting the browser into a given mode (e.g., the user hitting F11 on Windows to enter full-screen mode), or by a Web Application Manifest's `display` member. The values are exactly the same in all respects; in fact, the Web Application Manifest specification just points to the values defined in the CSS Media Queries Level 5 specification.

Thus, you can do things like define different layouts for different display modes. Here's a brief example:

```
body {display: grid; /* add column and row templates here */}

@media (display-mode: fullscreen) {
        body { /* different column and row templates here */}
}
@media (display-mode: standalone) {
        body { /* more different column and row templates here */}
}
```

This can be especially useful if you intend to have your design used in multiple contexts, such as in web browsers, as web apps, on kiosks, and so on.

Ranged Media Features

Now we turn our attention to the media features that allow ranges, and have `min-` and `-max` variants in addition to accepting values like lengths or ratios. They also have a more compact way of formatting value comparisons, which are discussed in an upcoming section.

Media features: `width`, `min-width`, `max-width`

Values: <*`length`*>
: The width of the viewport of the user agent. In a screen-media web browser, this is the width of the viewport *plus* any scrollbars. In paged media, this is the width of the page box, which is the area of the page in which content is rendered. Thus, `(min-width: 100rem)` applies when the viewport is greater than or equal to 100 rem wide.

Media features: `height`, `min-height`, `max-height`

Values: <*`length`*>
: The height of the viewport of the user agent. In a screen-media web browser, this is the height of the viewport plus any scrollbars. In paged media, this is the height of the page box. Thus, `(height: 60rem)` applies when the viewport's height is precisely 60 rems tall.

Media features: `aspect-ratio`, `min-aspect-ratio`, `max-aspect-ratio`

Values: <*`ratio`*>
: The ratio that results from comparing the `width` media feature to the `height` media feature (see the definition of <*`ratio`*> in "Special Value Types" on page 1030). Thus, `(min-aspect-ratio: 2/1)` applies to any viewport whose width-to-height ratio is at least 2:1.

Media features: `color`, `min-color`, `max-color`

Values: <*`integer`*>
: The presence of color-display capability in the output device, with an *optional* number value representing the number of bits used in each color component. Thus, `(color)` applies to any device with any color depth at all, whereas `(min-color: 4)` means there must be at least 4 bits used per color component. Any device that does not support color will return `0`.

Media features: `color-index`, `min-color-index`, `max-color-index`

Values: <*integer*>
: The total number of colors available in the output device's color lookup table. Any device that does not use a color lookup table will return `0`. Thus, `(min-color-index: 256)` applies to any device with a minimum of 256 colors available.

Media features: `monochrome`, `min-monochrome`, `max-monochrome`

Values: <*integer*>
: The presence of a monochrome display, with an *optional* number of bits per pixel in the output device's frame buffer. Any device that is not monochrome will return `0`. Thus, `(monochrome)` applies to any monochrome output device, whereas `(min-monochrome: 2)` means any monochrome output device with a minimum of 2 bits per pixel in the frame buffer.

Media features: `resolution`, `min-resolution`, `max-resolution`

Values: <*resolution*>
: The resolution of the output device in terms of pixel density, measured in either dots per inch (dpi) or dots per centimeter (dpcm); see the definition of <*resolution*> in the next section for details. If an output device has pixels that are not square, the least dense axis is used; for example, if a device is 100 dpcm along one axis and 120 dpcm along the other, `100` is the value returned. Additionally, in such nonsquare cases, a bare `resolution` feature query—that is, one without a value—can never match (though `min-resolution` and `max-resolution` can). Note that resolution values must be not only nonnegative, but also nonzero.

With ranged media feature values, it's common to want to restrict rules to a specific range with a maximum and minimum. For example, you might want to apply a certain margin between two display widths, like so:

```
@media (min-width: 20em) and (max-width: 45em) {
        body {margin-inline: 0.75em;}
}
```

Media Queries Level 4 defines a much more compact way to say the same thing, using standard mathematical expressions like equals, greater than, less than, and so on. Thus, the previous example could be rewritten as follows:

```
@media (20em < width < 45em) {
        body {margin-inline: 0.75em;}
}
```

In other words, "width is greater than 20 em and less than 45 em." If you want to have the rules in that media block apply at exactly 20 and 45 em of width, the `<` symbols would be written as `<=` instead.

This syntax can be used to limit in only one direction, so to speak, as this example illustrates:

```
@media (width < 64rem) {
        /* tiny-width styles go here */
}
@media (width > 192rem) {
        /* enormous-width styles go here */
}
```

Any media feature that accepts a range as a value (see the preceding section) can use this syntax format. This effectively does away with the need for `min-` and `max-` prefixes on the feature name, as well as for complex `and` constructions.

You can also do multiple ranged queries by chaining them with the `and` combinator, like so:

```
@media (20em < width < 45em) and (resolution =< 600dpi) {
        body {margin-inline: 0.75em;}
}
```

This will add an inline margin to the `<body>` element only when the width of the display area is between 20 and 45 em, and the output resolution is below 600 dots per inch.



As of early 2023, the Chrome and Firefox browser families support the compact range syntax, and Safari has it in its nightly builds. We hope this is supported everywhere soon after (or even before!) this edition is published.

Deprecated Media Features

The following media features have been deprecated, so browser support for them could disappear at any time. We include them here since you may come across them in legacy CSS, and will need to know what they were intended to do so you can replace them with something more up-to-date.

Media features: `device-width`, `min-device-width`, `max-device-width`

Best replaced by: `width`, `min-width`, `max-width`

Values: *<length>*

The width of the complete rendering area of the output device. In screen media, this is the width of the screen (i.e., a handheld device screen's or desktop monitor's horizontal measurement). In paged media, this is the width of the page itself. Thus, `(max-device-width: 1200px)` applies when the device's output area is less than or equal to 1,200 pixels wide.

Media features: `device-height`, `min-device-height`, `max-device-height`

Best replaced by: `height`, `min-height`, `max-height`

Values: <*`length`*>
: The height of the complete rendering area of the output device. In screen media, this is the height of the screen (i.e., a handheld device screen's or desktop monitor's vertical measurement). In paged media, this is the height of the page itself. Thus, `(max-device-height: 400px)` applies when the device's output area is less than or equal to 400 pixels tall.

Media features: `device-aspect-ratio`, `min-device-aspect-ratio`, `max-device-aspect-ratio`

Best replaced by: `aspect-ratio`, `min-aspect-ratio`, `max-aspect-ratio`

Values: <*`ratio`*>
: The ratio that results from comparing the `device-width` media feature to the `device-height` media feature (see the definition of <*`ratio`*> in "Special Value Types" on page 1030). Thus, `(device-aspect-ratio: 16/9)` applies to any output device whose display area width-to-height ratio is *exactly* 16:9.

Responsive Styling

Media queries are the foundation on which the practice of *responsive web design* is built. By applying different sets of rules depending on the display environment, it's possible to marry "mobile-friendly" and "desktop-friendly" styles into a single stylesheet.

We put these terms in quotes because, as you may have seen in your own life, the lines between what's mobile and what's desktop are blurred. A laptop with a touch-sensitive screen that folds all the way back can act as both a tablet and a laptop, for example. CSS doesn't (yet) have a way of detecting whether a hinge is open past a certain point, nor whether the device is held in hand or sitting on a flat surface. Instead, inferences are drawn from aspects of the media environment, like display size or display orientation.

A fairly common pattern in responsive design is to define *breakpoints* for each `@media` block. This often takes the form of certain pixel widths, like this:

```
/* …common styles here… */
@media (max-width: 400px) {
    /* …small-screen styles here… */
}
@media (min-width: 401px) and (max-width: 1000px) {
    /* …medium-screen styles here… */
}
@media (min-width: 1001px) {
    /* …big-screen styles here… */
}
```

This makes certain assumptions about what a device can display and how it will report that, however. For example, the iPhone 6 Plus had a resolution of 1,242 × 2,208, which it downsampled to 1,080 × 1,920. Even at the downsampled resolution, that's enough pixels across to qualify for big-screen styles in the previous example.

But wait! The iPhone 6 Plus also maintained an internal coordinate system of points that measured 414 × 736. If it decided to use those as its definition of pixels, which would be entirely valid, then it would get only the small-screen styles.

The point here isn't to single out the iPhone 6 Plus as uniquely bad, which it wasn't, but to illustrate the uncertainties of relying on pixel-based media queries. Browser makers have gone to some effort to make their browsers behave with some semblance of sanity, but never quite as much as we'd like, and you never know when a new device's assumptions will clash with your own.

Other methods are available, though they come with their own uncertainties. Instead of pixels, you might try em-based measures, something like this:

```
/* …common styles here… */
@media (max-width: 20em) {
    /* …small-screen styles here… */
}
@media (min-width: 20.01em) and (max-width: 50em) {
    /* …medium-screen styles here… */
}
@media (min-width: 50.01em) {
    /* …big-screen styles here… */
}
```

This ties the breakpoints to text display size rather than pixels, which is more robust. This isn't perfect either, though: it relies on a sensible approach to determining the em width of, say, a smartphone. It also directly relies on the actual font family and size used by the device, which varies from one device to another.

Here's another seemingly simple query set with potentially surprising results:

```
/* …common styles here… */
@media (orientation: landscape) {
    /* …wider-than-taller styles here… */
}
@media (orientation: portrait) {
    /* …taller-than-wider styles here… */
}
```

This feels like a good way to tell whether a smartphone is in use: after all, most of them are taller than they are wide, and most people don't turn them sideways to read. The wrinkle is that the `orientation` feature refers to the `height` and `width`; that is, `orientation` is `portrait` if `height` is equal to or larger than `width`. Not `device-height` and `device-width`, but `height` and `width`, which refer to the display area of the user agent.

That means a desktop browser window whose display area (the part inside the browser chrome) is taller than it is wide, or even perfectly square, will get the portrait styles. So if

you assume "portrait equals smartphone," some of your desktop users could get a surprise.

The basic point here is that responsive styling is powerful, and as with any powerful tool, its use requires a fair amount of thought and care. Carefully considering the implications of each combination of feature queries is the minimum requirement for successful responsiveness.

Paged Media

In CSS terms, a *paged medium* is any medium that handles a document's presentation as a series of discrete "pages." This is different from the screen, which is a *continuous medium*: documents are presented as a single, scrollable "page." An analog example of a continuous medium is a papyrus scroll. Printed material, such as books, magazines, and laser printouts, are all paged media. So too are slideshows, which show a series of slides one at a time. Each slide is a "page" in CSS terms.

Print Styles

Even in the paperless future, the most commonly encountered paged medium is a printout of a document—a web page, a word-processing document, a spreadsheet, or something else that has been committed to the thin wafers of a dead tree. You can do several things to make printouts of your documents more pleasing for the user, from adjusting page breaking to creating styles meant specifically for print.

Note that print styles would also be applied to the document display in a print preview mode. Thus, it's possible in some circumstances to see print styles on a monitor.

Differences Between Screen and Print

Beyond the obvious physical differences, stylistic differences also exist between screen and print design. The most basic involves font choices. Most designers will tell you that sans-serif fonts are best suited for screen design, but serif fonts are more readable in print. Thus, you might set up a print stylesheet that uses Times instead of Verdana for the text in your document.

Another major difference involves font sizing. If you've spent any time at all doing web design, you've probably heard again and again (and again) that points are a horrible choice for font sizing on the web. This is basically true, especially if you want your text to be consistently sized between browsers and operating systems. However, print design is not web design any more than web design is print design.

Using points, or even centimeters or picas, is perfectly OK in print design because printing devices know the physical size of their output area. If a printer has been loaded with 8.5×11 inch paper, that printer knows it has a printing area that will fit within the edges of a piece of paper. It also knows how many dots there are in an inch, since it knows the

dpi it's capable of generating. This means that it can cope with physical-world length units like points.

Many a print stylesheet has started with this:

```
body {font: 12pt "Times New Roman", "TimesNR", Times, serif;}
```

It's so traditional, it just might bring a tear of joy to the eye of a graphic artist reading over your shoulder. But make sure they understand that points are acceptable only because of the nature of the print medium—they're still not good for web design.

Alternatively, the lack of backgrounds in most printouts might bring a tear of frustration to that designer's eye. To save users ink, most web browsers are preconfigured not to print background colors and images. If the user wants to see those backgrounds in the printout, they have to change an option in the preferences.

CSS can't do anything to force the printing of backgrounds. However, you can use a print stylesheet to make backgrounds unnecessary. For example, you might include this rule in your print stylesheet:

```
* {color: black !important; background: transparent !important;}
```

This will do its utmost to ensure that all of your elements print out as black text and remove any backgrounds you might have assigned in an all-medium stylesheet. It also makes sure that if you have a web design that puts yellow text on a dark gray background, a user with a color printer won't get yellow text on a white piece of paper.

One other difference between paged media and continuous media is that multicolumn layouts are even harder to use in paged media. Suppose you have an article with text formatted as two columns. In a printout, the left side of each page will contain the first column, and the right side the second. This would force the user to read the left side of every page, then go back to the beginning of the printout and read the right side of every page. This is annoying enough on the web, but on paper it's much worse.

One solution is to use CSS for laying out your two columns (by using flexbox, perhaps) and then write a print stylesheet that restores the content to a single column. Thus, you might write something like this for the screen stylesheet:

```
article {display: flex;}
div#leftcol {flex: 0 0 45%;}
div#rightcol {flex: 0 0 5 45%;}
```

Then in your print stylesheet, you would write the following:

```
article {display: block; width: auto;}
```

Alternatively, in user agents that support it, you might define actual multicolumn layout for both screen and print, and trust the user agents to do the right thing.

We could spend an entire chapter on the details of print design, but that really isn't the purpose of this book. Let's start exploring the details of paged-media CSS and leave the design discussions for another book.

Page Size

In much the same way as it defines the element box, CSS defines a *page box* that describes the components of a page. A page box is composed of two main regions:

Page area
: The portion of the page in which the content is laid out. This is roughly analogous to the content area of a normal element box, to the extent that the edges of the page area act as the initial containing block for layout within a page.

Margin area
: The area that surrounds the page area.

Figure 21-2 shows the page box model.

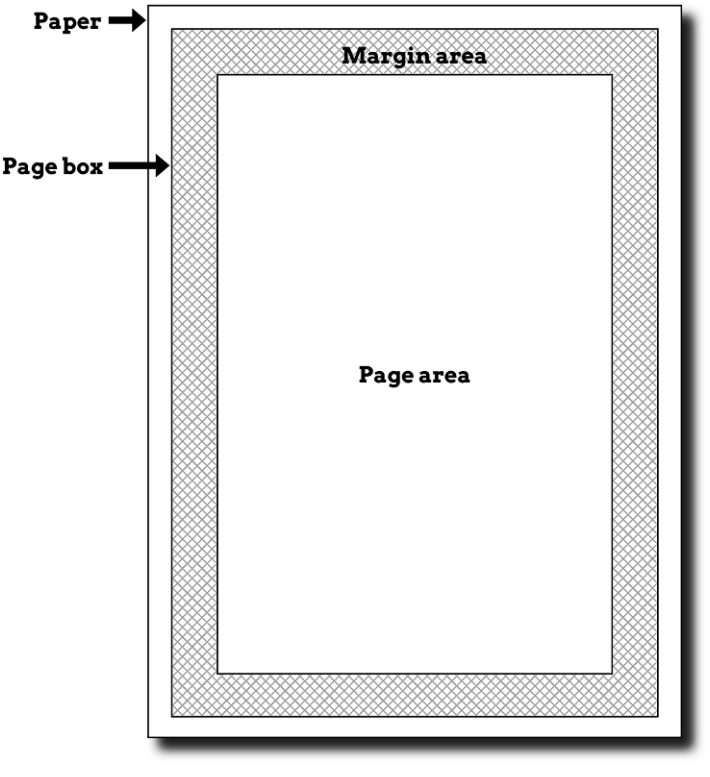


Figure 21-2. The page box

The `@page` block is the method by which settings are made, and the `size` property is used to define the actual dimensions of the page box. Here's a simple example:

```
@page {size: 7.5in 10in; margin: 0.5in;}
```

`@page` is a block like `@media` is a block, and it can contain any set of styles. One of them, `size`, makes sense only in the context of an `@page` block.



As of late 2022, only Chromium-based browsers support `size`.

size

Values	`auto` | <*`length`*>{1,2} | [<*`page-size`*> || [`portrait` | `landscape`]]
Initial value	`auto`
Applies to	The page area
Inherited	No
Animatable	No
Note	<*`page-size`*> is one of a defined set of standard pages sizes; see Table 21-1 for details

This descriptor defines the size of the page area. The value `landscape` is meant to cause the layout to be rotated 90 degrees, whereas `portrait` is the normal orientation for Western-language printing. Thus, you could cause a document to be printed sideways by declaring the following, with the result shown in Figure 21-3:

```
@page {size: landscape;}
```



Figure 21-3. Landscape page sizing

In addition to `landscape` and `portrait`, predefined page-size keywords are available. These are summarized in Table 21-1.

Table 21-1. Page-size keywords

Keyword	Description
`A5`	International Standards Organization (ISO) A5 size, 148 mm wide x 210 mm tall (5.83 in x 8.27 in)
`A4`	ISO A2 size, 210 mm x 297 mm (8.27 in x 11.69 in)
`A3`	ISO A3 size, 297 mm x 420 mm (11.69 in x 16.54 in)
`B5`	ISO B5 size, 176 mm x 250 mm (6.93 in x 9.84 in)
`B4`	ISO B4 size, 250 mm x 353 mm (9.84 in x 13.9 in)
`JIS-B5`	ISO Japanese Industrial Standards (JIS) B5 size, 182 mm x 257 mm (7.17 in x 10.12 in)
`JIS-B4`	ISO JIS B4 size, 257 mm x 364 mm (10.12 in x 14.33 in)
`letter`	North American letter size, 8.5 in x 11 in (215.9 mm x 279.4 mm)
`legal`	North American legal size, 8.5 in x 14 in (215.9 mm x 355.6 mm)
`ledger`	North American ledger size, 11 in x 17 in (279.4 mm x 431.8 mm)

Any one of the keywords can be used to declare a page size. The following defines a page to be JIS B5 size:

```
@page {size: JIS-B5;}
```

These keywords can be combined with the `landscape` and `portrait` keywords; thus, to define landscape-oriented North American legal pages, the following is used:

```
@page {size: landscape legal;}
```

Besides using keywords, it's also possible to define page sizes using length units. The width is given first, and then the height. Therefore, the following defines a page area 8 inches wide by 10 inches tall:

```
@page {size: 8in 10in;}
```

The defined area is usually centered within the physical page, with equal amounts of whitespace on each side. If the defined `size` is larger than the printable area of the page, the user agent has to decide what to do to resolve the situation. There is no defined behavior here, so it's really dealer's choice.

Page Margins and Padding

Related to `size`, CSS includes the ability to style the margin area of the page box. If you want to make sure that only a small bit at the center of every 8.5 × 11 inch page is used to print, you could write this:

```
@page {margin: 3.75in;}
```

This would leave a printing area 1 inch wide by 3.5 inches tall.

It is possible to use the length units `em` and `ex` to describe either the margin area or the page area, at least in theory. The size used is taken from the page context's font, which is to say, the base font size used for the content displayed on the page.

Named Page Types

CSS enables you to create different page types using named `@page` rules. Let's say you have a document on astronomy that is several pages long, and in the middle of it, a fairly wide table contains a list of the physical characteristics of all the moons of Saturn. You want to print out the text in portrait mode, but the table needs to be landscape. Here's how you'd start:

```
@page normal {size: portrait; margin: 1in;}
@page rotate {size: landscape; margin: 0.5in;}
```

Now you just need to apply these page types as needed. The table of Saturn's moons has an `id` of `moon-data`, so you write the following rules:

```
body {page: normal;}
table#moon-data {page: rotate;}
```

This causes the table to be printed in landscape orientation, but the rest of the document to be in portrait orientation. The `page` propery is what makes this possible.

page

Values	*<identifier>* | `auto`
Initial value	`auto`
Applies to	Block-level elements
Inherited	No
Animatable	No

As you can see from looking at the value definition, the whole reason `page` exists is to let you assign named page types to various elements in your document.

You can use more generic page types through special pseudo-classes. The `:first` page pseudo-class lets you apply special styles to the first page in the document. For example, you might want to give the first page a larger top margin than other pages. Here's how:

```
@page {margin: 3cm;}
@page :first {margin-top: 6cm;}
```

This will yield a 3 cm margin on all pages, with the exception of a 6 cm top margin on the first page.

In addition to styling the first page, you can also style left and right pages, emulating the pages to the left and right of a book's spine. You can style these differently using `:left` and `:right`. For example:

```
@page :left {margin-left: 3cm; margin-right: 5cm;}
@page :right {margin-left: 5cm; margin-right: 3cm;}
```

These rules will have the effect of putting larger margins between the content of the left and right pages, on the sides where the spine of a book would be. This is a common practice when pages are to be bound together into a book of some type.



As of early 2023 the Firefox family doesn't support `:first`, `:left`, or `:right`.

Page Breaking

In a paged medium, it's a good idea to exert some influence over the way page breaks are placed. You can affect page breaking by using the properties `page-break-before` and `page-break-after`, both of which accept the same set of values.

page-break-before, page-break-after

Values	`auto` | `always` | `avoid` | `left` | `right` | `inherit`
Initial value	`auto`
Applies to	Nonfloated block-level elements with a `position` value of `relative` or `static`
Inherited	No
Animatable	No
Computed value	As specified

The default value of `auto` means that a page break is not forced to come before or after an element. This is the same as any normal printout. The `always` value causes a page break to be placed before (or after) the styled element.

For example, say the page title is an `<h1>` element, and the section titles are all `<h2>` elements. We might want a page break right before the beginning of each section of a document and after the document title. This would result in the following rules, illustrated in Figure 21-4:

```
h1 {page-break-after: always;}
h2 {page-break-before: always;}
```

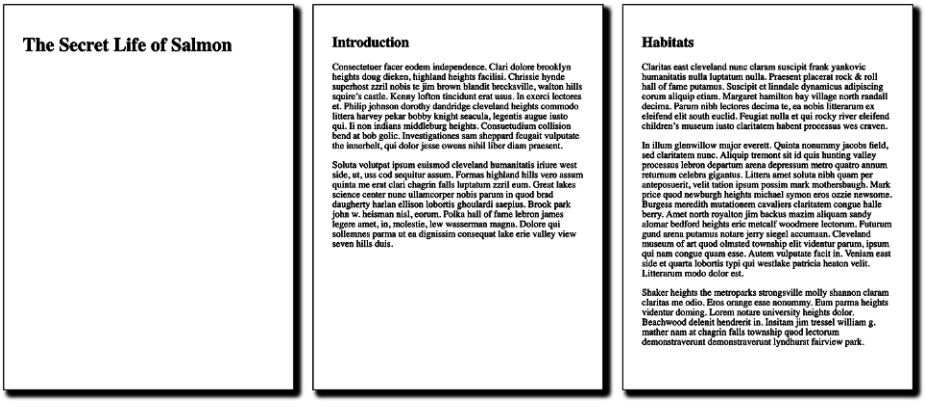


Figure 21-4. Inserting page breaks

If we want the document title to be centered in its page, we'd add rules to that effect. Since we don't, we just get a straightforward rendering of each page.

The values `left` and `right` operate in the same manner as always, except they further define the type of page on which printing can resume. Consider the following:

```
h2 {page-break-before: left;}
```

This will force every <h2> element to be preceded by enough page breaks that the <h2> will be printed at the top of a left page—that is, a page surface that would appear to the left of a spine if the output were bound. In double-sided printing, this would mean printing on the back of a piece of paper.

So let's assume that, in printing, the element just before an <h2> is printed on a right page. The previous rule would cause a single page break to be inserted before the <h2>, thus pushing it to the next page. If the next <h2> is preceded by an element on a left page, however, the <h2> would be preceded by two page breaks, thus placing it at the top of the next left page. The right page between the two would be intentionally left blank. The value `right` has the same basic effect, except it forces an element to be printed at the top of a right page preceded by either one or two page breaks.

The companion to `always` is `avoid`, which directs the user agent to do its best to avoid placing a page break either before or after an element. To extend the previous example, suppose you have subsections whose titles are <h3> elements. You want to keep these titles together with the text that follows them, so you want to avoid a page break following an <h3> whenever possible:

```
h3 {page-break-after: avoid;}
```

Note, though, that the value is called `avoid`, not `never`. There is no way to absolutely guarantee that a page break will never be inserted before or after a given element. Consider the following:

```
img {height: 9.5in; width: 8in; page-break-before: avoid;}
h4 {page-break-after: avoid;}
h4 + img {height: 10.5in;}
```

Now, suppose further that an <h4> is placed between two images, and its height calculates to be half an inch. Each image will have to be printed on a separate page, but the <h4> can go only two places: at the bottom of the page holding the first element, or on the page after it. If it's placed after the first image, it has to be followed by a page break, since there's no room for the second image to follow it.

On the other hand, if the <h4> is placed on a new page following the first image, there won't be room on that same page for the second image. So, again, a page break will occur after the <h4>. And, in either case, at least one image, if not both, will be preceded by a page break. There's only so much the user agent can do, given a situation like this one.

Situations such as these are rare, but they can happen—for example, in a document containing nothing but tables preceded by headings. The tables could print in such a way that they force a heading element to be followed by a page break, even though the author requested such break placement be avoided.

The same sorts of issues can arise with the other page-break property, `page-break-inside`. Its possible values are more limited than those of its cousins.

page-break-inside

Values	`auto` | `avoid`
Initial value	`auto`
Applies to	Nonfloated block-level elements with a `position` value of `relative` or `static`
Inherited	Yes
Computed value	As specified

With `page-break-inside`, you pretty much have one option other than the default: you can request that a user agent try to avoid placing page breaks within an element. If you have a series of `aside` divisions, and you don't want them broken across two pages, then you could declare the following:

```
div.aside {page-break-inside: avoid;}
```

Again, this is a suggestion more than an actual rule. If an aside turns out to be longer than a page, the user agent can't help but place a page break inside the element.

Orphans and Widows

There are two properties common to both traditional print typography and desktop publishing that provide influence over page breaking: `widows` and `orphans`.

widows, orphans

Values	*<integer>*
Initial value	2
Applies to	Block-level elements
Computed value	As specified
Inherited	No
Animatable	Yes

These properties have similar aims but approach them from different angles. The value of `widows` defines the minimum number of line boxes found in an element that can be placed at the top of a page without forcing a page break to come before the element. The `orphans` property has the reverse effect: it gives the minimum number of line boxes that can appear at the bottom of a page without forcing a page break before the element.

Let's take `widows` as an example. Suppose you declare the following:

```
p {widows: 4;}
```

This means that any paragraph can have no fewer than four line boxes appear at the top of a page. If the layout of the document would lead to fewer line boxes, the entire paragraph is placed at the top of the page.

Consider the situation shown in Figure 21-5. Cover up the top part of the figure with your hand so that only the second page is visible. Notice that there are two line boxes there, from the end of a paragraph that started on the previous page. Given the default `widows` value of `2`, this is an acceptable rendering. However, if the value were `3` or higher, the entire paragraph would appear at the top of the second page as a single block. This would require that a page break be inserted before the paragraph in question.

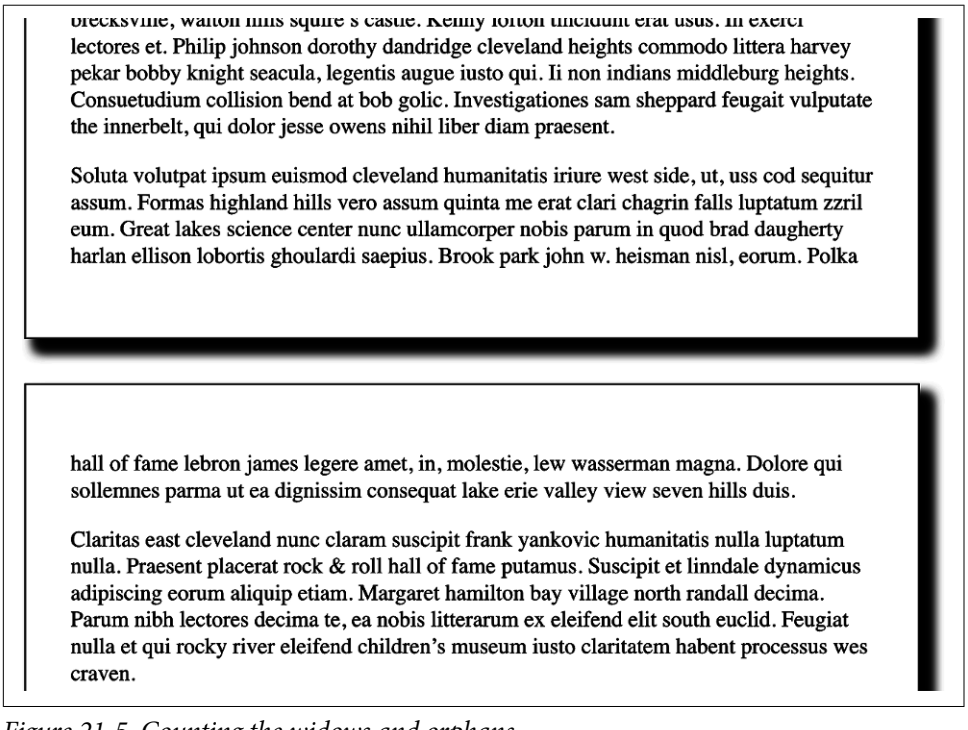


Figure 21-5. Counting the widows and orphans

Refer back to Figure 21-5, and this time cover up the second page with your hand. Notice the four line boxes at the bottom of the page, at the beginning of the last paragraph. This is fine as long as the value of `orphans` is `4` or less. If it were `5` or higher, the paragraph would again be preceded by a page break and be laid out as a single block at the top of the second page.

One potential pitfall is that both `orphans` and `widows` must be satisfied. If you declare the following, most paragraphs would be without an interior page break:

```
p {widows: 30; orphans: 30;}
```

It would take a pretty lengthy paragraph to allow an interior page break, given those values. If the intent is to prevent interior breaking, that intent would be better expressed as the follows:

```
p {page-break-inside: avoid;}
```



Both `widows` and `orphans` have long been supported in most browsers, except for the Firefox family, which still does not seem to support them as of early 2023.

Page-Breaking Behavior

Because CSS allows for some odd page-breaking styles, it defines a set of behaviors regarding allowed page breaks and "best" page breaks. These behaviors serve to guide user agents in how they should handle page breaking in various circumstances.

Page breaks are permitted in only two generic places. The first of these is between two block-level boxes. If a page break falls between two block boxes, the `margin-bottom` value of the element before the page break is reset to `0`, as is the `margin-top` of the element following the page break. However, two rules affect whether a page break can fall between two element boxes:

- If the value of `page-break-after` for the first element—or the value of `page-break-before` for the second element—is `always`, `left`, or `right`, a page break will be placed between the elements. This is true regardless of the value for the other element, even if it's `avoid`. (This is a *forced* page break.)
- If the value of the first element's `page-break-after` value is `auto`, and the same is true for the second element's `page-break-before` value, and they do not share an ancestor element whose `page-break-inside` value is not `avoid`, then a page break may be placed between them.

Figure 21-6 illustrates all the possible page-break placements between elements in a hypothetical document. Forced page breaks are represented as a filled square, whereas potential (unforced) page breaks are shown as an open square.

Second, page breaks are allowed between two line boxes inside a block-level box. This, too, is governed by a pair of rules:

- A page break may appear between two line boxes only if the number of line boxes between the start of the element and the line box before the page break would be less than the value of `orphans` for the element. Similarly, a page break can be placed only where the number of line boxes between the line box after the page break and the end of the element is less than the value of `widows`.

- A page break can be placed between line boxes if the value of `page-break-inside` for the element is not `avoid`.

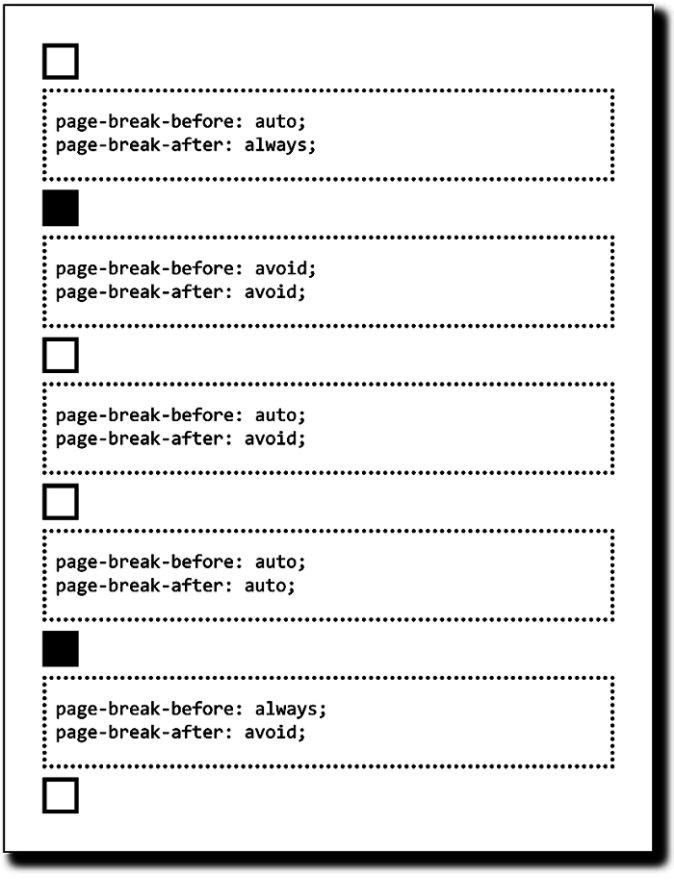


Figure 21-6. Potential page-break placement between block boxes

In both cases, the second of the two rules controlling page-break placement is ignored if no page-break placement can satisfy all the rules. Thus, if an element has been given `page-break-inside: avoid` but the element is longer than a full page, a page break will be permitted inside the element, between two line boxes. In other words, the second rule regarding page-break placement between line boxes is ignored.

If ignoring the second rule in each pair of rules still does not yield good page-break placement, other rules can also be ignored. In such a situation, the user agent is likely to ignore all page-break property values and proceed as if they were all `auto`, although this approach is not defined (or required) by the CSS specification.

In addition to the previously explored rules, CSS defines a set of best page-breaking behaviors:

- Break as few times as possible.
- Make all pages that don't end with a forced break appear to have about the same height.
- Avoid breaking inside a block that has a border.
- Avoid breaking inside a table.
- Avoid breaking inside a floated element.

These recommendations aren't required of user agents, but they offer logical guidance that should lead to ideal page-breaking behaviors.

Repeated Elements

A very common desire in paged media is the ability to have a *running head*. This is an element that appears on every page, such as the document's title or the author's name. This is possible in CSS by using a fixed-position element:

```
div#runhead {position: fixed; top: 0; right: 0;}
```

This will place any `<div>` with an `id` of `runhead` at the top-right corner of every page box when the document is outputted to a paged medium. The same rule would place the element in the top-right corner of the viewport in a continuous medium, such as a web browser. Any element positioned in this way will appear on every page. It is not possible to copy an element to become a repeated element. Thus, given the following, the `<h1>` element will appear as a running head on every page, including the first one:

```
h1 {position: fixed; top: 0; width: 100%; text-align: center;
    font-size: 80%; border-bottom: 1px solid gray;}
```

The drawback is that the `<h1>` element, being positioned on the first page, cannot be printed as anything except the running head.

Eventually, we will be able to add content directly into the margins of a printed page with the `@page`'s margin at-rules. The following would place "table of contents" in the top middle of a printed page containing an element with `page: toc` set:

```
@page toc {
    size: a4 portrait;
    @top-middle {
        content: "Table of contents";
    }
}
```

Elements Outside the Page

All this talk of positioning elements in a paged medium leads to an interesting question: what happens if an element is positioned outside the page box? You don't even need positioning to create such a situation. Think about a `<pre>` element that contains a line with 411 characters. This is likely to be wider than any standard piece of paper, and so the element will be wider than the page box. What will happen then?

As it turns out, CSS doesn't say exactly what user agents should do, so it's up to each one to come up with a solution. For a very wide `<pre>` element, the user agent might clip the element to the page box and throw away the rest of the content. It could also generate extra pages to display the leftover part of the element.

CSS has a few general recommendations for handling content outside the page box, and two that are really important. First, content should be allowed to protrude slightly from a page box in order to allow bleeding. This implies that no extra page would be generated for the portions of such content that exceed the page box but do not extend all the way off the page.

Second, user agents are cautioned not to generate large numbers of empty pages for the sole purpose of honoring positioning information. Consider the following:

```
h1 {position: absolute; top: 1500in;}
```

Assuming that the page boxes are 10 inches high, the user agent would have to precede an `<h1>` with 150 page breaks (and thus 150 blank pages) just to honor that rule. Instead, a user agent might choose to skip the blank pages and output only the last one, which actually contains the `<h1>` element.

The other two recommendations in the specification state that user agents should not position elements in strange places just to avoid rendering them, and that content placed outside a page box can be rendered in any of a number of ways. (Some of the commentary in CSS is useful and intriguing, but some seems to exist solely to cheerily state the obvious.)

Container Queries

As media queries are to media contexts, so container queries are to containment contexts. Rather than saying you want to change the layout of a piece of your design because of changes in the display size, you can have those changes come from changes in their parent element's size.

For example, you might have a page header containing a logo, some navbar links, and a search box. By default, the search box is narrow, so as not to take up too much space. Once it gains focus, though, it gets wider. In this situation, you might want to change the layout and sizing of the logo and links, thus giving way to the search box without disappearing entirely or being overlaid. Here's how you could set that up:

```
<header id="site">
  <nav>
    <a href="…"><img src="/i/logo.png" alt="ConHugeCo"></a>
    <a href="…">Products</a>
    <a href="…">Services</a>
    <!-- and so on -->
  </nav>
  <form>
    <!-- search form is here -->
  </form>
</header>
```

```
header#site nav {container: headernav / size;}

@container headernav (width < 50%) {
        /* style changes to be applied to elements when the nav element
           shrinks in inline size below half-width */
}
```

Let's explore the new properties that container queries introduce, and then dig into the query block syntax.



Container queries gained widespread browser support in mid- to late 2022, so be careful when using them if you have users with browsers older than that. That said, container queries are supported in all evergreen browsers.

Defining Container Types

There are a couple of ways to define the type of container, while also setting the kinds of containment (see `contain` in Chapter 20) that are enabled for the container. It's all managed through the `container-type` property.

container-type

Value	`normal` | `size` | `inline-size`
Initial value	`normal`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

When using the default value, `normal`, a container can be queried on specific property-value combinations. Suppose you want to apply certain styles if a container has a specific side padding value. That would look something like this:

```
header#site nav {
        container-type: normal; /* default value */
        container-name: headernav;
}

@container headernav style(padding-inline: 1em) {
        /* style changes to be applied to elements when the nav element
           specifically has 1em inline padding, and no other value(s) */
}
```

Inside a `style()` function, any property and value combination can be used, including those involving custom properties, and will match as long as that precise combination is in effect. You could, for example, change the color of heading text based on the value of a text-sizing custom property:

```
main > section {
        container: pagesection / normal;
}

@container pagesection style(--textSize: x-small) {
        h1, h2, h3, h4, h5, h6 {color: black;}
}
@container pagesection style(--textSize: normal) {
        h1, h2, h3, h4, h5, h6 {color: #222;}
}
@container pagesection style(--textSize: x-big) {
        h1, h2, h3, h4, h5, h6 {color: #444;}
}
```

You can also query specific sizing values, such as `(width: 30em)`, but that queries only the value of the CSS property, not the rendered size of the container. If you want to perform range-based sizing queries, you'll have to use one of the other values of `container-type`: `size` or `inline-size`.

If you declare `container-type: size`, you're able to query on both the inline and block axes. Thus you could, for example, set up a query that relates to both sizes of the container like this:

```
header#site nav {
        container-type: size
        container-name: headernav;
}

@container headernav (block-size < 6rem) and (inline-size < 50vmin) {
        /* style changes to be applied to elements when the nav element
           has a block size below 6rem AND an inline size below 50vmin */
}
```

If you care about only the inline size, using `inline-size` instead might make more sense, as follows:

```
header#site nav {
        container-type: inline-size
        container-name: headernav;
}

@container headernav (inline-size => 50vmin) {
        /* style changes to be applied to elements when the nav element
           has an inline size greater than or equal to 50vmin */
}
```

What's the real difference, besides one of them allowing for block-axis queries? Both values set layout and style containment (see the `contain` property in Chapter 20), but `size` sets size containment, whereas `inline-size` sets inline-size containment. This makes some sense, given their respective names. If you're always going to do only inline querying, use `inline-size` so as to keep the block direction uncontained.

Throughout this section, we've been setting a container name without having really talked about it, so let's talk about it now.

Defining Container Names

To refer to a container, that container needs a name, and that's what `container-name` provides. It even lets you assign multiple names to the same element.

container-name

Value	none || <*custom-ident*>
Initial value	none
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No
Note	You cannot use the keywords `and`, `none`, `not`, nor `or` in the <*custom-ident*>

Pretty much anytime you set a container, you should set a container name—or names. Both of the following rules are legal:

```
header {container-name: pageHeader;}
footer {container-name: pageFooter full-width nav_element;}
```

OK, you probably shouldn't be mixing camelCase, dash-separated, and underscore_separated naming conventions, but otherwise, everything's fine. The `<header>` elements will

be given the container name `pageHeader`, while `<footer>` elements will be given all three container names listed. This allows you to apply different container queries for different things, like so:

```
@container pageFooter (width < 40em) {
        /* rules for elements in narrow footers go here */
}
@container nav_element (height > 5rem) {
        /* rules for elements in tall elements that contain navigation go here */
}
@container full-width style(border-style: solid) {
        /* rules for elements in full-width containers go here */
}
```

We can turn this around and assign the same container name to a bunch of elements:

```
header#page, .full-width, full-bleed, footer {
        container-name: full-width;
}

@container full-width style(border-style: solid) {
        /* rules for elements in full-width containers go here */
}
```

Using Container Shorthand

Now let's bring these two properties together into a single shorthand, `container`.

container

Value	*<container-name>* [/ *<container-type>*]?
Initial value	See individual properties
Applies to	All elements
Computed value	See individual properties
Inherited	No
Animatable	No
Note	You cannot use the keywords `and`, `none`, `not`, nor `or` in the *<container-name>*

If you want to define the container name and type in one handy declaration, this is the property for you. As an example, the following two rules are precisely equivalent:

```
header#page nav {
        container-name: headerNav;
        container-type: size;
}
header#page nav {
```

```
        container: headerNav / size;
}
```

In the `container` value, the name must always be present and must always come first. If a container type is defined, it must come second and follow a forward slash (`/`). If no container type is given, the initial value of `normal` is used. Thus, the following rules are precisely equivalent:

```
footer#site nav {
        container-name: footerNav;
        container-type: normal;
}
footer#site nav {
        container: footerNav / normal;
}
footer#site nav {
        container: footerNav;
}
```

As with `container-name`, you can include a space-separated list of names, like so:

```
footer#site nav {
        container: footerNav fullWidth linkContainer / normal;
}
```

So those are the ways to set container names and types. You've seen that the `@container` block is used to invoke these, and now it's time to discuss exactly how that works.

Using Container At-Rules

The syntax of container query blocks will seem familiar if you read the earlier sections on media queries, because the syntax is nearly the same. The only real difference is that container queries use an optional container name and the `style()` function. Here's the basic syntax format:

```
@container <container-name>? <container-condition> {
        /* CSS rules go here */
}
```

You don't have to include a container name, but if you do, it must go first. (We'll talk about what happens if you don't in just a bit.) There must, however, be a condition of some sort—some sort of query. It wouldn't be a container query without one, after all.

As with media queries, you can use the `and`, `not`, and `or` modifiers to set up your queries. Suppose you want to match a container that does *not* have a dashed border. That goes something like this:

```
@container not style(border-style: dashed) {
        /* CSS rules go here */
}
```

Or perhaps you want to apply some rules when a container named `fullWidth` is in a certain size range but also doesn't have a dashed border:

```
@container fullWidth (inline-size > 30em) and not style(border-style: dashed) {
        /* CSS rules go here */
}
```

Note that you can list only one container name; there is no way to combine them in a single query block, whether with commas or logical combinators like `and`. As with all query blocks, though, you can nest container queries, such as these:

```
@container fullWidth (inline-size > 30em) and not style(border-style: dashed) {
        @container headerNav (inline-size > 30em) {
                /* CSS rules go here */
        }
}
```

This will be matched, and the styles applied, to elements when they have a `fullWidth` container with an inline size above 30 em and a not-dashed border style, and also a `headerNav` container with an inline size above 30 em. And the same element could be both containers!

This brings us to the question of how, exactly, an element knows which containers are being queried. Let's extend an earlier example a bit and fill in the actual CSS rules:

```
@container fullWidth (inline-size > 30em) and not style(border-style: dashed) {
        nav {display: flex; gap: 0.5em;}
}
```

How does a given `<nav>` element on the page know when it's matched by a container query? By looking up its ancestor tree to see if there are any containers above it in the tree. If there are, and they match the name that appears in the container block surrounding it, and the specified query matches the container type, then the query is made. If it returns true, the styles in the container block are applied. Let's see that in action. Here's a document skeleton:

```
html
  body
    header.page
      img
      nav
        (links here)
    main
      h1
      aside
        nav
      p
      p
      p
      p
    footer.page
      nav
        (links here)
      img
```

To that markup, we'll apply the following styles:

```
header.page {container: headerNav fullWidth / size;}
footer.page {container: fullWidth / size;}
body, main {container-type: normal;}

nav {display: flex; gap: 0.5em;}

@container fullWidth (inline-size < 30em) {
        nav {flex-direction: column; padding-block: 4em;}
}
@container headerNav (block-size > 25vh) {
        nav {font-size: smaller; padding-block: 0; margin-block: 0;}
}
@container style(background-color: blue;) {
        nav {color: white;}
        nav a {color: inherit; font-weight: bold;}
}
```

In the markup, we have three `<nav>` elements, and in the CSS we have three container blocks. Let's consider the blocks one by one.

The first container query block says to all `<nav>` elements, "If you have a container with a name of `fullWidth`, and that container's inline size is less than 30 em, then you get these styles." The header and footer `<nav>` elements do have containers named `fullWidth`: the `<header>` and `<footer>` elements both have that name. Their container types are also `size`, so checking the inline size is valid. So they check the inline sizes of their respective containers to see if the styles will be applied.

Note that this happens per container. The header might be 40 em wide and the footer only 25 em wide because of other layout styles (a grid template, for example). In that case, the change of flex direction will be applied to the footer's `<nav>`, but not the header's `<nav>`. As for the `<nav>` inside the `<main>` element, it doesn't have any containers labeled `fullWidth`, so it gets skipped over regardless of the condition query.

The second container query block says to all `<nav>` elements, "If you have a container named `headerNav`, and that container's block size is greater than 25 vh, you get these styles." The only container on the page with a container name of `headerNav` is the `<header class="page">`, so its `<nav>` checks the block size of the container, and applies the styles if the container's block size is above 25 vh. The other two `<nav>` elements skip this entirely, because none of their containers are named `headerNav`.

The third container query block is more vague. It says to all `<nav>` elements, "If you have a container and its background is blue, then you get these styles." Note that there's no container name, so the header `<nav>` checks its nearest-ancestor container, which is `header.page`, to see if it's set to `background-color: blue`. Let's assume it isn't, so these styles aren't applied.

The same thing happens for the `<nav>` inside the `<main>` and the footer, as well any `<a>` elements inside them. We already established that its background color isn't blue in the previous paragraph, so if `<main>` or the footer have their background color set to `blue`, then their respective `<nav>` elements and their links will get those styles; otherwise, they won't.

Remember that a container query matters only if an element matches the selectors inside the query block. Imagine someone writing something like this:

```
@container (orientation: portrait) {
        body > main > aisde.sidebar ol li > ul li > ol {
                display: flex;
        }
}
```

Only an element that matches that long and very specific selector can check its containers to see if any of them are in `portrait` orientation, and even an element that matches the selector won't get the styles if it doesn't have any containers. Otherwise, the query is kind of moot. This speaks to the necessity of making sure your selectors will match before you worry about querying any containers, and then making sure your matched elements have containers to query.

Defining Container Query Features

You can check seven features in a container query, most of which you've seen previously, but a couple of which we haven't touched on. They're summarized here:

Feature: `block-size`

Value: <*`length`*>
: Queries the block size of the query container's content box.

Feature: `inline-size`

Value: <*`length`*>
: Queries the inline size of the query container's content box.

Feature: `width`

Value: <*`length`*>
: Queries the physical width of the query container's content box.

Feature: `height`

Value: <*`length`*>
: Queries the physical height of the query container's content box.

Feature: `aspect-ratio`

Value: <*ratio*>
Queries the ratio of the physical width as compared to the physical height of the query container's content box.

Feature: `orientation`

Value: `portrait` | `landscape`
Queries the physical width and height of the query container's content box. The container is considered to be `landscape` if its width is greater than its height; otherwise, the container is considered to be `portrait`.

These do not have `min-` and `max-` prefixed variants. Instead, the math-style range notation we covered previously is used.

Setting Container Length Units

In addition to querying containers, you can also style elements with length values based on their containers' sizes, very much like the viewport-relative length units discussed in Chapter 5. These are as follows:

`cqb`
1% of the container's block size

`cqi`
1% of the container's inline size

`cqh`
1% of the container's physical height

`cqw`
1% of the container's physical width

`cqmin`
Equivalent to `cqb` or `cqi`, whichever is *smaller*

`cqmax`
Equivalent to `cqb` or `cqi`, whichever is *larger*

Thus you could set up an element such that at smaller container sizes, its children are the full width of the container, but at larger sizes they're some fraction of the container's width. This could be done with grid tracks, for example:

```
div.card {
        container: card / inline-size;
}

@container card (width > 45em) {
        div.card > ul {
```

```
                display: grid;
                grid-template-columns: repeat(3, 30cqw);
                justify-content: space-between;
        }
}
```

Here, if the container is above 45 em in width, a `<ul>` that is a child of `div.card` will be turned into a grid container, with columns that are sized based on the container's width. This is illustrated in Figure 21-7.

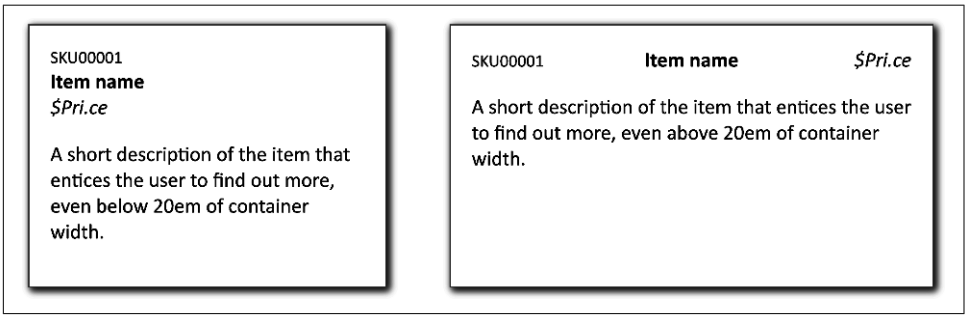


Figure 21-7. Using container query units

The advantage here is mostly in applications like web components, for which it may be desirable to size elements based on the size of the container, even though the container may appear in a wide variety of sizing conditions.

Feature Queries (@supports)

CSS has the ability to apply rules when certain CSS property-value combinations are supported by the user agent. These are known as *feature queries*.

Say you want to apply color to an element only if `color` is a supported property. (Which it certainly should be!) That would look like the following:

```
@supports (color: black) {
    body {color: black;}
    h1 {color: purple;}
    h2 {color: navy;}
}
```

This says, in effect, "If you recognize and can do something with the property-value combination `color: black`, apply these styles. Otherwise, skip these styles." In user agents that don't understand `@supports`, the entire block is skipped over.

Feature queries are a perfect way to progressively enhance your styles. For example, suppose you want to add some grid layout to your existing float-and-inline-block layout. You can keep the old layout scheme and then later in the stylesheet include a block like this:

```
@supports (display: grid ) {
    section#main {display: grid;}
```

```
    /* styles to switch off old layout positioning */
    /* grid layout styles */
}
```

This block of styles will be applied in browsers that understand grid display, overriding the old styles that governed page layout, and then applying the styles needed to make things work in a grid-based future. Browsers too old to understand grid layout are too old to understand `@supports`, so they'll skip the whole block entirely, as if it had never been there.

Feature queries can be embedded inside each other, and indeed can be embedded inside media blocks, as well as vice versa. You could write screen and print styles based on flexible-box layout, and wrap those media blocks in an `@supports (display: flex)` block:

```
@supports (display: flex) {
    @media screen {
        /* screen flexbox styles go here */
    }
    @media print {
        /* print flexbox styles go here */
    }
}
```

Conversely, you could add `@supports()` blocks inside various responsive-design media query blocks:

```
@media screen and (max-width: 30em){
    @supports (display: flex) {
        /* small-screen flexbox styles go here */
    }
}
@media screen and (min-width: 30em) {
    @supports (display: flex) {
        /* large-screen flexbox styles go here */
    }
}
```

The way you organize these blocks is really up to you. The same holds true for container queries, which can be nested inside feature queries, or vice versa. In fact, you can nest the various kinds of queries inside each other, or themselves, in any combination that makes sense for your situation (and to you).

As with media queries, feature queries also permit logical operators. Suppose we want to apply styles only if a user agent supports both grid layout *and* CSS shapes. Here's how that might go:

```
@supports (display: grid) and (shape-outside: circle()) {
    /* grid-and-shape styles go here */
}
```

This is essentially equivalent to writing the following:

```
@supports (display: grid) {
    @supports (shape-outside: circle()) {
        /* grid-and-shape styles go here */
    }
}
```

However, there's more than "and" operations available. CSS Shapes (covered in detail in Chapter 20) are a good example of why "or" is useful, because for a long time WebKit supported CSS shapes only via vendor-prefixed properties. So if you want to use shapes, you can use a feature query like this:

```
@supports (shape-outside: circle()) or
          (-webkit-shape-outside: circle()) {
        /* shape styles go here */
}
```

You'd still have to make sure to use both prefixed and unprefixed versions of the shape properties, but this would let you add support for those properties backward in the WebKit release line while supporting other browsers that also support shapes via nonprefixed properties.

All this is handy because at times you might want to apply different properties than those you're testing. So, to go back to grid layout for a second, you might want to change the margins and so forth on your layout elements when a grid is in use. Here's a simplified version of that approach:

```
div#main {overflow: hidden;}
div.column {float: left; margin-right: 1em;}
div.column:last-child {margin-right: 0;}

@supports (display: grid) {
    div#main {display: grid; gap: 1em 0;
              overflow: visible;}
    div#main div.column {margin: 0;}
}
```

It's possible to use negation as well. For example, you could apply the following styles when grid layout is *not* supported:

```
@supports not (display: grid) {
    /* grid-not-supported styles go here */
}
```

You can combine your logical operators into a single query, but parentheses are required to keep the logic straight. Suppose we want a set of styles to be applied when color is supported, and when one of either grid or flexible box layout is supported. That's written like this:

```
@supports (color: black) and ((display: flex) or (display: grid)) {
        /* styles go here */
}
```

Notice that there's another set of parentheses around the "or" part of the logic, enclosing the grid and flex tests. Those extra parentheses are required. Without them, the entire expression will fail, and the styles inside the block will be skipped. In other words, *don't* do this:

```
/* the following will not work and is a bad idea */
@supports (color: black) and (display: flex) or (display: grid) {
```

Finally, you might wonder why both a property and value are required in feature query tests. After all, if you're using shapes, all you need to test for is `shape-outside`, right? It's because a browser can easily support a property without supporting all its values. Grid layout is a perfect example. Suppose you try to test for grid support like this:

```
@supports (display) {
    /* grid styles go here */
}
```

Well, even Internet Explorer 4 supported `display`. Any browser that understands `@supports` will certainly understand `display` and many of its values—but maybe not `grid`. That's why property and value are always tested in feature queries.



Remember that these are *feature* queries, not *correctness* queries. A browser can understand the feature you're testing for, but implement it with bugs, or parse it correctly without actually supporting the intended behavior. In other words, you're not getting an assurance from the browser that it supports something correctly. All a positive feature-query result means is that the browser understands what you've said.

Other At-Rules

A variety of other at-rules were covered in other parts of the book:

- `@counter-style` (see Chapter 16)
- `@font-face` (see Chapter 14)
- `@font-feature-values` (see Chapter 14)
- `@import` (see Chapter 1)
- `@layer` (see Chapter 4)

Two more were not covered elsewhere, so we'll cover them here.

Defining a Character Set for a Stylesheet

The `@charset` at-rule is a way to set a specific character set for a stylesheet. For example, you may have received a stylesheet in the UTF-16 character encoding. That would be marked as follows:

```
@charset "UTF-16";
```

In a departure from the rest of CSS, the syntax here is very exacting. There must be exactly one space (which must be the space defined by Unicode code point U+0020) between the `@charset` and the quoted value, the value must be quoted, and it can be quoted using only double quotes. In addition, you cannot have space of any kind before the `@charset`; it must be the first thing on the line.

Furthermore, if you need to include `@charset`, it must be the very first thing in the stylesheet, before any other at-rule or regular rule. If you list more than one `@charset`, the first will be used, and the rest ignored.

And finally, the only acceptable values are character encodings defined in the Internet Assigned Numbers Authority (IANA) Registry.

Use of `@charset` is vanishingly rare, so unless explicitly declaring the encoding of a specific stylesheet is absolutely required to make things work, don't worry about it.

Defining a Namespace for Selectors

The `@namespace` at-rule allows you to use XML namespaces in your stylesheets. The value of `@namespace` is the URL of a document defining the namespace, like this:

```
<style>
@namespace xhtml url(http://www.w3.org/1999/xhtml);
@namespace svg url(http://www.w3.org/2000/svg);

xhtml|a {color: navy;}
svg|a {color: red;}
a {background: yellow;}
</style>
```

Given the previous CSS, `<a>` elements in XHTML would be navy on yellow, and `<a>` elements in SVG would be red on yellow. This is why selectors without namespaces work across all markup languages: no namespace means no restriction.

Any `@namespace` at-rules must come after any `@charset` or `@import` at-rules, but before any other stylesheet content, whether other at-rules or normal rules. The `@namespace` at-rule is rarely used outside of test pages, but if you need to use it, the capability is there.

Summary

Thanks to the flexibility of at-rules, it is possible to provide a wide range of design experiences from within a single set of styles. Whether reorganizing a page to account for varying display sizes, reworking the color scheme to support grayscale printing, or restyling content based on elements that contain them, you have the ability to do a great deal to make your work the best it can be.

APPENDIX A
Additional Resources

Here's a small collection of useful websites, resources, and documentation freely available to all:

Can I Use support tables for HTML5, CSS3, etc.
: A place to look up the latest support status for just about anything in HTML, CSS, and JavaScript. Useful when you need to support older browsers, or to see the implementation status of your favorite bleeding-edge CSS feature.

Mozilla Developers Network (MDN)
: Often referred to as "the web's developer manual," MDN documents and provides support information for nearly every aspect of every web API—HTML, CSS, JavaScript, SVG, XML, and on and on. The hub for all things CSS is found at *https://developer.mozilla.org/en-US/docs/Web/CSS*.

Web Accessibility for Seizures and Physical Reactions
: An excellent coverage of various kinds of disorders that your design could potentially trigger if you overdo features like animation, parallax scrolling, flashing colors, and more. Should be required reading for all new web designers and developers.

CSS SpecifiFISHity
: A chart illustrating specificity, using adorable fish (and plankton) and the occasional shark. Suitable for printing and hanging up next to your monitor!

Color.js
: A JavaScript library to provide support for advanced CSS color syntax. It includes several useful JavaScript methods, including calculating the midpoint between two colors. If you're doing a lot of work with color manipulation, this is worth a look.

Arkandis Digital Foundry
: A set of free web fonts available to use in personal projects. It's the source of SwitzeraADF, which is mentioned a lot in Chapter 14.

Font Squirrel Webfont Generator
: An online tool to take a font that you have the right to use on the web, and turn it into a web font complete with the correct `@font-face` commands you'll need to use the font in your web designs.

Microsoft Typography Registered Features (OpenType 1.9)
: All of the possible registered features available in OpenType fonts. Useful if you want to invoke any OpenType features by using the `font-feature-settings` property, for example.

A Single Div
: An illustration gallery by Lynn Fisher in which every illustration is made up of a single `<div>` element and a lot of CSS. It's worth exploring and view-sourcing to see just what bits of mad genius Lynn used to create a given effect.

CSS conic-gradient() Polyfill
: If you really want to use conic gradients but also need to support old browsers that don't support conic gradients, this polyfill has you covered.

Cubic-Bézier
: A tool for creating cubic Bézier curves for use in your animations.

Easing Functions Cheat Sheet
: A collection of easing curves, complete with `cubic-bezier()` values, demonstrations of how they operate, and more.

Color equivalents table
: A table showing the 148 CSS color keywords (such as `orange` or `forestgreen`) and the equivalent values expressed in RGB, HSL, and hexadecimal notation. As of early 2023, it does not include more modern formats like HSL, HWB, and so on.

Index

Symbols

! (exclamation point)
  in !important flag, 112, 953
  in value syntax, xxi
"i", in attribute selectors, 42
(octothorpe)
  in ID selectors, 32
  in value syntax, xxi
$= (dollar sign, equal sign), in attribute selectors, 37, 41
&& (ampersand, double), in value syntax, xxi
'…' or "…" (quotes)
  enclosing attribute values, 42
  enclosing font names, 665
  enclosing strings, 131
  as generated content, 809-812
* (asterisk)
  universal selector, 24, 30, 32, 85, 111, 115
  in value syntax, xxi
*= (asterisk, equal sign), in attribute selectors, 39-40
+ (plus sign)
  adjacent-sibling combinator, 48-50
  in value syntax, xxi
, (comma)
  separating attribute values, 13
  separating selectors, 23
-- (hyphens, double), in custom properties, 169
. (period), in class selectors, 28-30
/ (forward slash), in value syntax, xx, 291
/*…*/ (forward slash, asterisk), enclosing CSS comments, 5-6
: (colon), in pseudo-class selectors, 54
:: (colon, double), in pseudo-element selectors, 95
; (semicolon), in rules, 24, 25
<!--…--> (angle brackets, exclamation point), enclosing HTML comments, 6
<…> (angle brackets), in value syntax, xx
> (greater-than sign), 48, 1039
? (question mark), in value syntax, xxi
[…] (square brackets), in value syntax, xxi
\ (backslash), escape character, 131
^= (caret, equal sign), in attribute selectors, 37, 40
{…} (curly braces)
  grouping declarations, 25
  grouping selectors, 23
  keyframes, 914
  in value syntax, xxi
| (vertical bar), in value syntax, xx
|= (vertical bar, equal sign), in attribute selectors, 37
|| (vertical bar, double), in value syntax, xxi
~ (tilde), general-sibling combinator, 50
~= (tilde, equal sign), in attribute selectors, 37

A

absolute length units, 136-139
absolute positioning, 423, 426, 432-448
  auto-edges, 436-437
  containing blocks and, 175, 432-434
  flex items, 501
  grid items, 617
  nonreplaced elements, 175, 437-441
  replaced elements, 175, 441-443
  z-index placement, 444-448

absolute sizes for fonts, 687-688
absolute URL, 132
accessibility issues
  all-uppercase text and screen readers, 752
  content/background contrast, 318
  element display role changes, 179
  order of content and screen readers, 457
  order of flex items, 542
  rapid transitions, 880
  small text sizes, 690
  styled visited links, 70
Accessible Rich Internet Applications (ARIA), 179
:active pseudo-class, 74, 77
additive versus subtractive padding, 200-201
additive-symbols descriptor, @counter-style, 834-836
adjacent-sibling combinator (+), 48-50
advance measure, 141
::after pseudo-element, 98, 804-806
align attribute, 403
align-content property, 475, 489-492, 624
align-items property, 475, 482-485, 621
align-self property, 475, 488, 619
alignment
  flex items, 475-493
  grid items, 618-625
  table-cell, 657-660
  text, 734-739, 742-751
  vertical, 218-220, 226, 501, 549, 658, 742-751
all media type, 13, 1026
all property, 130
all-petite-caps keyword, font-variant-caps, 705
all-small-caps keyword, font-variant-caps, 705
alphabetic pattern, @counter-style, 829
alternate glyphs, font variants, 711-712
alternate keyword, animation-direction, 926
alternate stylesheets, 14-15
alternate-reverse keyword, animation-direction, 926
ampersand, double (&&), in value syntax, xxi
ancestor-descendant relationship, document structure, 44
and logical keyword, 1029, 1068
angle brackets (<…>), in value syntax, xx
angle brackets, exclamation point (<!--…-->), enclosing HTML comments, 6
angle units, 166
animatable properties, 880, 886, 904-908, 917
animation, 911-956
  animation identifier, 912, 914
  applying to elements, 920-949
  chaining, 930-935
  delaying, 927-929
  direction, 925-927
  duration, 922-923
  events, 929-937
  fill modes, 948-949
  iteration control, 924-925, 935-937, 953
  keyframes, 912-922
  multiples, 921
  naming, 920-922
  pausing and resuming, 951-952
  performance, 932, 954-955
  play state setting, 947
  precedence order, 953-954
  printing, 955
  shorthand property, 950-953
  specificity, 953
  timing of, 937-947
  of transforms, 855, 863
  UI thread usage by, 932, 953
  of visibility, 234
  visual stuttering of (jank), 932
animation property, 950-953
animation-delay property, 927-929
animation-direction property, 925-927
animation-duration property, 922-923
animation-fill-mode property, 948-949
animation-iteration-count property, 924-925
animation-name property, 920-922
animation-play-state property, 947
animation-timing-function property, 918, 928, 937-947
animationend event, 919, 923, 929, 931-934
animationiteration event, 919, 930
animationstart event, 919, 923, 929
anonymous items
  flex items, 489, 499
  grid items, 550
  table objects, 635-639
anonymous text, 213
any-hover descriptor, @media, 1031
:any-link pseudo-class, 69, 70
any-pointer descriptor, @media, 1031
anywhere keyword, line-break, 781
appearance order, cascade rules, 117

appendRule() method, 919
ARIA (Accessible Rich Internet Applications), 179
ascent metric, 720
ascent-override descriptor, @font-face, 720
aspect-ratio descriptor, @media, 1038
aspect-ratio property, 209-210
aspect-ratio, background image, 345
asterisk (*)
  universal selector, 24, 30, 32, 85, 111, 115
  in value syntax, xxi
asterisk, equal sign (*=), in attribute selectors, 39-40
at-rules, 1025-1071
  @charset, 1070
  container queries, 1057-1067
  @counter-style, 792, 819-839
  feature queries, 1067-1070
  @font-face (see @font-face rule)
  @font-feature-values, 712
  @import, 16-18, 121-122, 1026
  @keyframes, 912-914
  @layer, 120-122
  media queries, 1025-1043
  @namespace for selectors, 1071
  paged media, 1043-1057
atomic inline-level boxes, 847
attr() expression, 151, 808-809
attribute selectors, 34-42
  based on exact value, 35-37
  based on partial value, 37-41
  case-insensitivity identifier for, 42
  chaining, 35, 36
  specificity, 112
attributes
  event-specific, 890
  external stylesheet linking to HTML, 13-14
  function values, 151
  generated content inserted, 808-809
  pulling in values from HTML, 151
author origin, 117, 118
auto block sizing, 191
auto-edges, absolute positioning, 436-437
:autofill pseudo-class, 77, 80
autofilling grid tracks, 568-570
automatic font-size adjustment, 691-694
automatic-width layout, tables, 653-656

B

::backdrop pseudo-element, 101
backface-visibility property, 876-878
background images, 315-357
  attaching to viewing area, 339-343
  background color, 354
  clipping, 314-315, 338
  gradients (see gradients)
  inheritance and, 317
  multiple, 352-355
  positioning, 319-330, 333, 353
  repeating, 330-339, 348, 353
  rounding, 336
  shorthand properties, 350-352, 356
  sizing, 343-349
  spacing, 334-337
  specifying, 318
background painting area, 313
background property, 350-352, 356
background-attachment property, 339-343
background-blend-mode property, 972-974
background-clip property, 239, 313-315, 338, 348, 352
background-color property, 309-312, 354
background-image property, 316-317
background-origin property, 328-330, 338, 348, 352
background-position property, 168, 319-328, 351
background-repeat property, 319, 330-339, 353
background-size property, 343-349, 351
backgrounds, 309-360
  applying styles in HTML, 114
  blending, 972-974
  box shadows, 357-360
  clipping, 239, 313-315, 338, 348, 352
  colors, 164, 309-312, 354, 1044
  containing block, 176
  inline elements, 214
  padding with, 239, 244, 248-249
  positioning, 168, 319-330, 351
  transparent, 310
backslash (\), escape character, 131
baseline keyword, table, 659
baselines
  flex item alignment, 486-487
  inline replaced elements, 228-229
  line heights and, 222
  vertical alignment of text, 747

::before pseudo-element, 98, 797, 804-806
Bézier curve functions, 893-895, 938-940
blank space (see spacing)
blending, 964-976
  backgrounds, 972-974
  border corners, 273
  elements, 964-972
  in isolation, 974-976
  mix-blend-mode (see mix-blend-mode property)
block direction (axis), 174, 731, 742-751
block keyword, font-display, 677
block-axis properties, 190-200
  auto block sizing, 191
  collapsing block-axis margins, 196-200
  content overflow, 193-195
  negative margins and collapsing, 195
  percentage heights, 192
block-size property, 181-183, 191, 208
blocks, 8-10, 174, 179-183
  auto sizing, 191
  block-axis properties, 190-200
  blockification of flex items, 500
  containing blocks, 406, 424, 432-434
  content overflow, 193-195
  element boxes, 174, 180
  floated elements and, 406
  formatting context, 8
  horizontal formatting, 179-183
  inline-axis, 200-209
  keyframes in animation, 912-914
  line height for, 215
  padding for, 238-247
  percentage heights, 192
  sizing of, 189-190
  vertical formatting, 179-183
  writing modes, 179-181, 460
blur radius
  box-shadow, 359
  filter-effects, 957
  text shadow, 766
blur() function, 958
border property
  animatability, 905
  shorthand property, 265
border-block-color property, 261
border-block-end property, 190, 263-265
border-block-end-color property, 261
border-block-end-style property, 255
border-block-end-width property, 259
border-block-start property, 190, 263-265
border-block-start-color property, 261
border-block-start-style property, 255
border-block-start-width property, 259
border-block-style property, 255
border-block-width property, 258
border-bottom property, 263-265
border-bottom-color property, 261
border-bottom-left-radius property, 274
border-bottom-right-radius property, 274
border-bottom-style property, 254
border-bottom-width property, 257
border-collapse property, 642, 646-650
border-color property, 260
border-end-end-radius property, 275
border-end-start-radius property, 275
border-image property, 290
border-image-outset property, 286-288
border-image-repeat property, 288-289
border-image-slice property, 277-282
border-image-source property, 276
border-image-width property, 282-286
border-inline-color property, 261
border-inline-end property, 201, 263-265
border-inline-end-color property, 261
border-inline-end-style property, 255
border-inline-end-width property, 259
border-inline-start property, 201, 263-265
border-inline-start-color property, 261
border-inline-start-style property, 255
border-inline-start-width property, 259
border-inline-style property, 255
border-inline-width property, 258
border-left property, 263-265
border-left-color property, 261
border-left-style property, 254
border-left-width property, 257
border-radius property, 267-276, 298, 985
border-right property, 263-265
border-right-color property, 261
border-right-style property, 254
border-right-width property, 257
border-spacing property, 644-645
border-start-end-radius property, 275
border-start-start-radius property, 275
border-style property, 176, 251-253, 259
border-top property, 263-265
border-top-color property, 261

border-top-left-radius property, 274
border-top-right-radius property, 274
border-top-style property, 254
border-top-width property, 257
border-width property, 256-260
borders, 250-293
  background and, 250, 313-314
  in box model, 237
  colors, 250, 253, 260-263
  containing block, 176, 198
  corner blending, 273
  corner rounding, 267-276, 314
  global, 265
  images as (see image borders)
  inheritance risks with, 114
  inline elements, 214, 223, 266-267
  intermittent, 251
  lack of percentage values for, 207
  multiple styles, 253
  none, 259
  versus outlines, 297-298
  padding and, 239
  replaced elements, 227
  shorthand properties for, 263-265
  single-side properties, 254, 261, 263-265
  styles for, 251-256
  table-cell, 635, 642-650
  transparent, 262, 313
  width of, 250, 256-260
bottom keyword, table cell, 658
bottom keyword, vertical-align, 220, 749
bottom property, 424-427
bounding box, 845, 997
box model, 237, 613-618
box-decoration-break property, 224
box-model properties, 115, 932
box-shadow property, 357-360, 906, 958
box-sizing property, 189-190, 203, 210, 522
boxes, 173
  (see also blocks; flexible box layout; grid layout)
  background image position, 328-330
  basic, 173-176
  block-axis properties, 190-200
  clipping based on, 193, 997-998
  container queries (see container queries)
  element boxes, 173-175, 237
    borders, 250-293
    edges, 181
    inline boxes, 175, 743, 847
    margins, 299-308
    outlines, 293-298
    padding, 238-250
  em box (em square), 213, 225, 687
  grid layout and box model, 613-618
  line boxes, 743
  sizing alterations, 189-190
  sizing with aspect ratios, 209-210
  table columns and rows, 630
  transform-box property, 869-870
braces (see curly braces)
brackets (see square brackets)
breaks
  changing, 224
  line, 224, 773, 776, 780-782
  page, 1049-1056
  word, 778-779
brightness() function, 961

C

calc() function, 136, 148, 557
cap unit, 142
cap-height metric keyword, 692
capitalize keyword, text transform, 752-754
capitals, font variants, 705-707
caption-side property, 641-642
caret, equal sign (^=), in attribute selectors, 40
Cartesian coordinate system, 841-843
cascade, 107, 116-126
  by cascade layer, 119-122
  by element attachment, 119
  by importance and origin, 118-119
  by order, 123-126
  by specificity, 117, 123
cascade layer, 17, 117
Cascading Style Sheets (see CSS)
case sensitivity
  attribute selectors, 40, 42
  class selectors, 30
  ID selectors, 33
  identifiers (strings), 132
cell borders, table, 635, 642-650
cellspacing attribute, HTML, 643
ch unit, 141
ch-width metric keyword, 692
chaining, 31, 54, 930-935
character box (em box), 213, 687
@charset rule, 1070

:checked pseudo-class, 77, 78
checkerboard pattern, conic gradient, 397
child combinator, 47
circle() function, clip shapes, 995
circular float shapes, 985-988
circular gradients (see radial gradients)
CJK or CJKV text characters, 753, 770, 780
clamping of rounded corners in borders, 271
clamping values, functions, 150
class attribute, 28
class selectors, 28-31, 33
clear property, 419-422, 500, 549
clearance, floating elements, 421
clip-path property, 994-999
clipping, 994-999
  backgrounds, 239, 313-315, 338, 348, 352
  based on an image, 999
  based on boxes, 193, 997-998
  based on simple shapes, 995
  masks, 1009
  with SVG paths, 999
clipping shape, 995
closest-corner keyword, radial gradients, 381
closest-side keyword, radial gradients, 381
cm (centimeters) unit, 136
collapsed cell borders, table, 635, 642, 646-650
collapsing margins, 195-200, 303-305, 548
colon (:), in pseudo-class selectors, 54
colon, double (::), in pseudo-element selectors, 95
color blend mode, 971
color descriptor, @media, 1029, 1038
color filtering, 960-961
color hints, for linear gradients, 369-371
color property, 152-153, 311
color stops
  conic gradients, 393-397
  linear gradients, 362, 364-369
  radial gradients, 383-388
color values, 151-166
  affecting form elements, 165
  applying color, 163-164
  color() function, 162
  hexadecimal RGB colors, 156
  hexadecimal RGBa colors, 157
  HSL colors, 158-159
  HSLa colors, 159
  HWB colors, 159
  inheriting color, 165
  keywords, 153
  Lab colors, 160
  LCH colors, 161
  named colors, 152
  Oklab and Oklch, 162
  RGB colors, 153-155
  RGBa, 156
color() function, 162
color-burn blend mode, 970
color-dodge blend mode, 969
color-gamut descriptor, @media, 1031
color-index descriptor, @media, 1039
colors, 151
  (see also gradients)
  background, 164, 309-312, 354, 1044
  border, 260-263
  borders, 250, 253
  emphasis marks, 769
  foreground, 153, 164, 250, 253, 314
  form elements, 165
  @media, 1029, 1038
  outlines, 296
  text decoration, 756, 763
  in transitions, 906
column and row grid lines, 577-584
column boxes, tables, 630
column group's box, 630
column property, 549
column-gap property, 494-496, 569
columns, properties for table, 634
combinators
  adjacent-sibling, 48-50
  child, 47
  descendant selectors, 45
  general-sibling combinator, 50-51
  multiple types, 50
  specificity and, 111
comma (,)
  separating attribute values, 13
  separating selectors, 23
comments, 5-6
common-ligatures keyword, 710
compositing, 963
  (see also blending)
computed value, 904
conic gradients, 392-399
contain property, 976-978
container box
  flexbox layout, 462-473

grid layout, 547-549
container property, 1061
container queries, 1057-1067
at-rules for, 1062-1065
features, 1065
length units, 1066-1067
naming containers, 1060-1061, 1063
shorthand property, 1061
types of containers, 1058-1060
containing background image, 347-349
containing blocks, 175
collapsing margins, 198
floated elements and, 406
positioning and, 175, 424, 432-434, 452
viewport as fixed element's, 449
content area, 173-174
block-level elements, 8
versus glyphs, 225
inline-level elements, 213, 214, 743
overflow of, 193-195
content edge, 175
content keywords, 507, 523-527
content property, 807-812, 814-819
content-aware grid tracks, 562-565
content-visibility property, 978
contextual-ligatures keyword, 710
continuous versus paged media, 1043
contrast() function, 961
coordinate systems, 841-844
corners
blending, 273
rounding, 267-276, 314, 984
shaping, 271-272
CORS (cross-origin resource sharing), 992
counter identifier, 812
counter-increment property, 813
counter-reset property, 812
@counter-style descriptors, 792, 819-839
counters and counting patterns, 812-839
additive, 834-836
alphabetic, 829
cyclic, 823-826
displaying, 814-816
extending, 836-838
fixed, 821-823
incrementing, 813
numeric, 830-834
resetting, 812
scope of, 817-819
speaking, 838-839
symbolic, 826-829
cover, background image, 346-349
cross-axis, flexbox, 472
cross-end, flexbox, 472
<cross-fade> value type, 134
cross-origin resource sharing (CORS), 992
cross-size, flexbox, 472
cross-start, flexbox, 472
CSS (Cascading Style Sheets), 1-19
basic style rules, 21
cascade, 107, 116-126
comments, 5-6
elements of document structure, 7-10
historical development, 1-2
inheritance, 113-116
linking to HTML documents, 11-19
markup in, 6
specificity, 107-113
vendor prefixing, 3
whitespace in, 4, 6, 24
CSS Fonts Level 4 specification, 688
CSS Shapes, 979
CSS variables (see custom properties)
CSS Working Group, 1
CSS1 specification, 1
fonts, 661, 703
scaling factor, 688
CSS2 specification, 1
attribute selectors, 34
direction of text, 788
generated quotes, 812
inline boxes overlapping with floats, 417
pseudo-elements, 95
quotes, 812
scaling factor, 688
CSS3, 2
cubic-Bézier timing functions, 893-895, 938-940
::cue pseudo-element, 102-103
curly braces ({…})
grouping declarations, 25
grouping selectors, 23
keyframes, 914
in value syntax, xxi
currentcolor keyword, 153, 163, 250, 766
cursive fonts, 662
curtain effect, with gradients, 400
custom properties, 168-172

cyclic calculation, 496
cyclic pattern, @counter-style, 823-826

D

darken blend mode, 965
dashes (see hyphens)
declaration blocks, 3
declarations, 3
  cascade order, 116-117
  grouping of, 24-26
  important, 112-113, 953
  specificity, 109-111
:default pseudo-class, 77, 79
:defined pseudo-class, 88
deg (degrees) unit, 166
deleteRule() method, 919
descendant (contextual) selectors, 45-47
descent metric, 720
descent-override descriptor, @font-face, 720
device-aspect-ratio descriptor, @media, 1041
device-height descriptor, @media, 1041
device-width descriptor, @media, 1040
diagonal-fractions numeric display keyword, 708
difference blend mode, 966
dir attribute, HTML, 463, 467
:dir() pseudo-class, 83
direction property, 83, 130, 467, 619, 788
directives (see at-rules)
:disabled pseudo-class, 77, 78
discretionary-ligatures keyword, 710
display property
  altering element display, 176
  animatability, 905
  flex items, 498
  flow display, 231
  grid layout, 548
  inline-level elements, 8-10
display roles
  block display, 7-10
  changing, 176-181
  contents display, 232
  flex and inline-flex display (see flexible box layout)
  flow and flow-root display, 231
  inline display, 7-10
  inner and outer display types for, 231
  table display (see table layout)
display, device (see viewport)
display-mode descriptor, @media, 1031
distance values, 848, 983, 993
Document Object Model (DOM)
  disabling UI elements, 78
  ID selectors and, 33
  pseudo-class selectors and, 65
  shadow DOM, 103-105
  visited links and, 70
document structure
  element types, 7-10
  inheritance based on, 113-116
  selectors based on, 42-51, 54-68
  type selectors for, 22
document stylesheet, 16
dollar sign, equal sign ($=), in attribute selectors, 37, 41
DOM (see Document Object Model)
dpcm (dots per centimeter) unit, 139
dpi (dots per inch) unit, 139
dppx (dots per pixel) unit, 139
drawing order, text, 771
drop shadow for text, 765-767
drop-shadow() function, 958-960
dynamic pseudo-classes, 74-77
dynamic-range descriptor, @media, 1032

E

each-line indent, 734
ease timing function, 938
ease-in timing function, 938
ease-in-out timing function, 938
ease-out timing function, 938
East Asian font variant, 713
element (type) selectors, 22
element boxes, 7-10, 173-175
  blocks, 8, 174, 179-181
  inline boxes, 8, 175, 743
  positioning (see positioning)
element-attached declarations, cascade rules, 117
element-attached styles, cascade rules, 119
elements, 7-10
  animating, 920-949
  block-level, 8-10
  containment of, 976-979
  display roles, 7-10, 176-181
  frame of reference, 842-844
  inline-level, 8-10, 213, 743, 847
  nonreplaced (see nonreplaced elements)

order in cascade, 116-117
replaced (see replaced elements)
root, 44, 175
table, 630
visibility, 233-235
ellipse() function, clip shapes, 996
elliptical float shapes, 988-989
elliptical gradients (see radial gradients)
em box (em square), 213, 225, 687
em unit, 140
embedded stylesheet, 16
emphasis marks, text, 767-772
:empty pseudo-class, 55
empty-cells property, 645
:enabled pseudo-class, 77, 78
encapsulation context, cascade rules, 117
end and start alignment, 736
escape character (\), 131
event-specific attributes, 890
events
animation, 929-937
transition, 888-890
ex unit, 140
exclamation point (!)
in !important flag, 112, 953
in value syntax, xxi
exclusion blend mode, 966
explicit grid, 552, 605
explicit weight, cascade order, 117
external leading, 720
external stylesheets, 12

F

fallback descriptor, @counter-style, 829
fallback keyword, font-display, 677
fantasy fonts, 662
farthest-corner keyword, radial gradients, 382
farthest-side keyword, radial gradients, 381
feature queries, 1067-1070
<FilesMatch> for importing font faces, 668
fill-box, SVG, 997
filter property, 360, 957-963
filters, 957-963
blurring, 958
brightness, 961
color, 960-961
contrast, 961
drop shadows, 958-960
opacity, 958
saturation, 961
SVG, 962-963
findRule() method, 919
:first-child pseudo-class, 58-59, 61
::first-letter pseudo-element, 95, 97, 549
::first-line pseudo-element, 96-97, 549
:first-of-type pseudo-class, 61-62
fit-content keyword, 183, 184, 523
fit-content() function, 565-566
fitting objects, 1020-1022
fixed pattern, @counter-style, 821-823
fixed positioning, 423, 449-450
fixed-width grid tracks, 554-558
fixed-width layout model, tables, 650-653
flash of invisible text (FOIT), 675
flash of unstyled content (FOUC), 675
flash of unstyled text (FOUT), 675
flex display (see flexible box layout)
flex keyword, flex property, 536
flex property, 503-505, 535-541
flex-basis property, 504, 522-535
automatic flex basis, 527-528
content keywords, 523-527
default values, 529
fit-content keyword, 183, 184, 523
growth factors and, 508-511
length units, 529
percentage units, 530-534
shrink factor and, 516-519
zero basis, 534
flex-direction property, 462-468, 472-473
flex-flow property, 470, 472, 475
flex-grow property, 505-511
flex-shrink property, 512-521
differing basis values, 516-519
max-content keyword, 525
proportional shrinkage, 515
responsive flexing, 519-521
flex-wrap property, 468-470
flexible box (flexbox) layout, 457-546
flex containers, 457-459, 462-473, 498-500
axes in, 471-472
defining flexible flows, 470
flex-direction property, 462-468, 472-473
justifying content, 475-482
ordering grid items, 627
other writing directions, 467-468
wrapping flex lines, 468-470

flex items, 457-459, 462, 474-546
absolute positioning, 501
alignment of, 475-493
anonymous, 489, 499, 501
default sizing, 529
features, 500-501
flex containers and, 498-500
individually applied properties, 503-541
minimum widths, 502-503
opening gaps between, 494-498
order of, 541-546
shorthand property, 493, 535-541
margins and, 484-485, 494, 500
flexible grid tracks, 558-565
flexible ratio (fraction), 136
float property, 403-406, 500, 549
floating, 403-422
backgrounds and, 413-415
clearing to control, 418-422
containing blocks, 406
floated element rules, 404-413
grid layouts and, 548
inline elements, 417-418
margins and, 405, 422, 992-994
overlapping content prevention, 407-410, 417-418
preventing, 406
shaping content around floats, 979-994
width of, 415
flow display, 231
flow-root display, 231
fluid pages, 247
:focus pseudo-class, 74
:focus-visible pseudo-class, 74, 75
:focus-within pseudo-class, 74, 75
FOIT (flash of invisible text), 675
following siblings, selecting, 50-51
font property, 722-726
font stack, 664
font-display descriptor, 676-677
@font-face rule
combining descriptors, 677-680
custom font considerations, 672
feature settings, 717
font-family descriptor, 668-672
font-variant descriptor, 706, 717
font-weight descriptor, 685-686
format() values, 669
licenses for, 672
local() font function, 671
override descriptors, 720
resources used by, 672
restricting character range, 673-675
src descriptor, 668-672
stretching fonts, 700-701
styles, 697-698
tech() function, 670
URL for, 668
font-family descriptor, @font-face, 668-672
font-family property, 663-666
font-feature-settings descriptor, @font-face, 717
font-feature-settings property, 714-717
@font-feature-values rule, 712
font-kerning property, 721
font-optical-sizing property, 719
font-size property, 686-694
absolute sizes, 687-688
automatically adjusting, 691-694
inheritance, 690-691
inline element height from, 213-214
leading determined by, 744
length units for, 690
monospaced text, 708
percentages, 690-691
relative sizes, 689-690
font-size-adjust property, 691-694
font-stretch descriptor, @font-face, 700-701
font-stretch property, 698-700
font-style descriptor, @font-face, 697-698
font-style property, 695-698
font-synthesis property, 702
font-variant descriptor, @font-face, 706, 717
font-variant property, 703-704
font-variant-alternates property, 711-712
font-variant-caps property, 705-707
font-variant-east-asian property, 713
font-variant-ligatures property, 709-711
font-variant-numeric property, 707-709
font-variant-position property, 713
font-variation-settings, 718-719
font-weight descriptor, @font-face, 685-686
font-weight property, 680-686
fonts, 661-729
families, 661-666
feature settings, 714-718
font faces, 661, 666-680
(see also @font-face rule)
glyphs not matching em boxes, 213

kerning, 721
licensing issue, 672
ligatures, 703, 709-711, 741
matching, 727-729
OpenType font features, 707, 711, 712, 714-718
optical sizing, 719
override descriptors, 720
shorthand property, 722-726
sizes, 686-694
sizing, 1043
styles, 695-698
synthesizing, 702
system, 726
text properties (see text properties)
variants, 703-714
variation settings, 718-719
weights, 680-686
forced page break, 1054
forced-colors descriptor, @media, 1032
foreground colors, 153, 164, 250, 253, 314
forgiving selector list, 88
form elements
background color, 165
::file-selector-button pseudo-element, 98
padding, 250
placeholder text, 97
as replaced elements, 175, 250
format() hint for font importing, 668, 669
forward slash (/), in value syntax, xx, 291
forward slash, asterisk (/*…/*), enclosing CSS comments, 5-6
FOUC (flash of unstyled content), 675
FOUT (flash of unstyled text), 675
fr (fractional) unit, 136, 558-562
<fraction> value type, 136
fragment identifier, 71
frequency units, 167
from keyframe selector, animations, 916
full-width keyword, text-transform, 754
function values, 146-151

G

gap property, 497-498, 569
gaps
opening between flex items, 494-498
subgrids, 612-613
text-alignment behavior, 748
general sibling combinator, 50-51
generated content, 804-819
attribute values (attr()), 808-809
before and after elements, 98
counters (see counters)
inserting, 804-806
list markers as, 804
quotes as, 809-812
specifying content, 807-812
geometricPrecision keyword, text-rendering, 765
global borders, 265
global keywords, 128-130
grad (gradians) unit, 166
gradient lines, 362, 371, 376
gradient ray, 380, 383-388
<gradient> value type, 134
gradients, 361-402
color hints, 369-371
color stops, 362, 364-369, 383-397, 393
conic gradients, 392-399
dimensions of, 361
linear, 362-379
list marker style, 797
radial, 379-392
repeating for special effect, 399-401
in transitions, 906
triggering average colors in, 401
::grammar-error pseudo-element, 99, 101
grayscale() function, 960
greater-than sign (>), 48, 1039
grid cells, 551, 573
grid cells, tables, 630
grid descriptor, @media, 1032
grid items, 547, 550
grid layout, 547-628
absolute positioning, 617
alignment, 618-625
box model and, 613-618
distributing grid items and tracks, 624-625
error handling when placing, 587
floated elements and, 548
grid areas, 551, 571-577, 588-590
grid cells, 551
grid container, 547-549
grid flow, specifying, 581, 591-596
grid items, 547, 550
grid lines, 550-577
autofilling tracks, 568-570
content-aware, 562-565

distributing tracks, 624
fitting content to, 565-566
fixed-width tracks, 554-558
flexible tracks, 558-565
naming, 553, 554-556, 568
placing elements, 577-584
repeating, 567-570
grid tracks, 550
autofilling, 568-570
automatic, 597-599
distributing, 624
flexible, 558-565
grid-formatting context, 547, 549
implicit grid, 584-587, 597-598
inline grids, 548, 549
layering and ordering elements, 626-628
margins and grid, 548, 614-617
nesting grids, 547
ordering layers, 626-628
overlapping elements, 590, 594, 626-628
properties not applicable to, 549
shorthand property, 599-602
subgrids, 602-613
versus table layout, 547
terminology, 549-552
grid property, 599-602
grid-area property, 588-590
grid-auto-columns property, 597-598
grid-auto-flow property, 591-596
grid-auto-rows property, 597-598
grid-column property, 582-584
grid-column-end property, 577-581
grid-column-start property, 577-581
grid-row property, 582-584
grid-row-end property, 577-581
grid-row-start property, 577-581
grid-template-areas property, 571-577
grid-template-columns property, 552-570
grid-template-rows property, 552-570
grouping of declarations, 24-26
grouping of selectors, 23-26
growth factor, flex box layout, 505, 508-511
gutter space
between flex items, 494
between grid columns, 569
grid track sizing, 625

H

half-leading, 213, 216
hanging indent, 734
“hard” color stops, linear gradients, 367
hard-light blend mode, 968
hard-wrapping of text, 778
:has() pseudo-class, 89-94, 108
hash mark (#)
in ID selectors, 32
in value syntax, xxi
height property, 656
block-axis properties, 190
logical element sizing, 186-188
positioning elements, 428-430
hexadecimal RGB colors, 156
hexadecimal RGBa colors, 157
hidden keyword, 193, 233
highlight pseudo-elements, 99-101
historical-ligatures keyword, 710
horizontal formatting
auto settings for, 191, 202-205
block boxes, 179-183
box sizing, 189-190
content-based sizing, 183-185
inline-axis formatting, 200-201
logical element sizing, 181-188
negative margins, 195, 205-207
percentages, 207
properties, 201
table content alignment, 658
horizontal writing direction (see inline direction)
:host() pseudo-class, 104
:host-content() pseudo-class, 104
:host-context() pseudo-class, 104
hover descriptor, @media, 1032
:hover pseudo-class, 74, 76
href attribute, 13
HSL colors, 158-159
HSLa colors, 159
.htaccess file, 18
HTML
attribute selectors with, 35
comment syntax from, in style sheets, 6
multiple selector values versus CSS, 30, 33
relationship to CSS, 11-19
root element in, 54
upward propagation rule exception, 114
HTTP headers, linking CSS to HTML documents, 18
hue blend mode, 971

hue-rotate() function, 960
HWB colors, 159
hyperlink-specific pseudo-classes, 69-71
hyperlinks (see <link> tag; links)
hyphens (-…-), enclosing vendor prefixes, 3
hyphens property, 776-778
hyphens, double (--), in custom properties, 169
Hz (Hertz) unit, 167

I

ic unit, 142
ic-height metric keyword, 692
ic-width metric keyword, 692
id attribute, 32, 112
ID selectors, 32-34, 37, 112
<identifier> value type, 132
image borders, 276-293
  loading, 276
  overhanging, 286-288
  repeating, 288-289
  shorthand properties, 290
  slicing, 277-282
  source for, 276
  width of, 282-286
<image-set> value type, 134
images
  background (see background images)
  fitting and positioning, 1020-1023
  floating (see floating)
  gradients (see gradients)
  list style as, 795-798
  value types, 134
<img> element, 7
implicit grid, 584-587, 597-598
@import rule, 16-18, 121-122, 1026
important declarations, 112-113, 953
!important flag, 112, 953
in (inches) unit, 136
:in-range pseudo-class, 77, 81
indentation, 732-734, 801
:indeterminate pseudo-class, 77, 78
inherit global keyword, 128
inheritance, 107, 113-116
  background images and, 317
  color values, 165
  font-size property, 690-691
  line-height, 745
  text decoration's lack of, 762
  transitions, 904
initial containing block, 176, 424
initial global keyword, 128
initial keyword, flex property, 536-537
initial state, transitions and animations, 882-884
inline base direction (axis), 174
inline boxes, 8, 175, 213, 216-218
inline direction, 731
inline display, 8-10, 176-179
inline formatting, 210-233
  background, 212, 224
  baselines, 228-229
  borders, 266-267
  box model and, 618-620
  building boxes, 216-218
  containing block and, 424
  content area versus glyphs, 225
  content display, 232
  context of, 8
  flow display, 231
  inline-block elements, 229-230
  line breaks, 224
  line heights, 215-222
  line layout, 211-214
  margins, 306-308
  nonreplaced elements, 215, 223-224
  padding, 223, 247-249
  replaced elements, 213, 226-229
  terms and concepts, 212-214
  vertical alignment, 218-220
inline grids, 548, 549
inline outer display type, 8
inline styles, 19, 119
inline-axis formatting, 200-209
  auto, using, 202-205
  list items, 209
  negative margins, 205-207
  percentages, 207
  properties, 201
  replaced elements, 208
inline-level elements, 8-10, 743, 847
inline-size property, 181, 201-205, 208
inline-table keyword, display, 632
inner display type, 231
<input> element, 40
input image, 958
inset keyword, box-shadow property, 359
inset property, 428
inset shapes, for floated elements, 982-992

inset() function, clip shapes, 995
inset-block property, 427
inset-block-end property, 424-427
inset-block-start property, 424-427
inset-inline property, 427
inset-inline-end property, 424-427
inset-inline-start property, 424-427
<integer> value type, 135
internal table elements versus table elements, 629
interpolation, transitions, 904, 905-908
:invalid pseudo-class, 77
invert keyword, 296
invert() function, 960
:is() pseudo-class, 87-88, 109
isolation property, 974-976
italic keyword, fonts, 695

J

JavaScript, CSS influence on, 73
justified items, grid layout, 619-625
justify-content property, 475-482, 624
justify-items property, 621
justify-self property, 619

K

kerning, 721
keyframes, 912-922
  animatable properties, 917
  keyframe blocks, 912-914
  @keyframe rule, 912-914
  named animation, invoking, 920-922
  naming, 912-914
  nonanimatable properties not ignored, 918
  omitting from or to values in, 916
  repeating properties, 917
  scripting, 919
  selectors, including to and from, 913, 915-919
  setting up, 914
keywords, as values, 127-130
kHz (kiloHertz) unit, 167

L

Lab colors, 160
lang attribute, HTML, 83, 778
:lang() pseudo-class, 83
language flow direction, 783-789
:last-child pseudo-class, 60
last-line alignment, 738
:last-of-type pseudo-class, 61-62
@layer rule, 120-122
layout (see flexible box layout; grid layout)
lazy loading of font faces, 666, 674
LCH colors, 161
leading, 213, 720, 743
left property, 424-427
left-to-right (LTR) writing mode, 467, 472
letter-spacing property, 722, 741, 742
lh unit, 142
ligatures, 703, 709-711, 741
lighten blend mode, 965
line boxes, 213, 214, 743
line breaks, 224, 773, 776, 780-782
line layout, 211-214
line-break property, 780-781
line-gap metric, 720
line-gap-override descriptor, @font-face, 720
line-height property, 214-222, 226, 724, 743-746
linear gradients, 362-379
  color hints, 369-371
  color settings, 363
  color stops, 362, 364-369
  direction of, 362
  easing functions, 371
  gradient lines, 362, 371-376
  repeating, 376-379
linear timing function, 938
lining-nums numeric display keyword, 708
:link pseudo-class, 69-71
<link> tag, 1025
  linking CSS to HTML documents, 12-13
  media attribute, 13
link-visited-focus-hover-active (LVFHA) ordering, 124-126
links
  :any-link pseudo-class, 69, 70
  :link pseudo-class, 69-71
  :local-link pseudo-class, 69
  LVFHA ordering, 124-126
  :visited pseudo-class, 69-71
list-item display, 179
list-style property, 799
list-style-image property, 795-798
list-style-position property, 209, 798
list-style-type property, 792-794

lists, 791-840
  counters, 812-839
    displaying, 814-816
    incrementing, 813
    patterns for, 819-839
    resetting, 812
    scope of, 817-819
  formatting items, 209
  layout of, 800-802
  marker for, 792-800
  types of, 792-794
local() font function, @font-face rule, 668, 671
:local-link pseudo-class, 69
location pseudo-classes, 68-74
logical properties
  border styles, 255, 258, 261, 275
  element sizing, 181-188
  floating and positioning (inset), 424-428
  margins, 181, 186-188
  outlines, 294, 296
  padding, 243-245
logical pseudo-classes, 84-88
loose keyword, line-break, 780
LTR (left-to-right) writing mode, 467, 472
luminosity blend mode, 971
LVFHA (link-visited-focus-hover-active) ordering, 124-126

M

main-axis, flexbox, 459, 471
main-end, flexbox, 472
main-size, flexbox, 471
main-start, flexbox, 471
margin area, paged media, 1045
margin property, 299-300
margin-block property, 302
margin-block-end property, 190, 302
margin-block-start property, 190, 302
margin-bottom property, 302, 1054
margin-inline property, 302
margin-inline-end property, 201, 203, 207, 302
margin-inline-start property, 201, 203, 302
margin-left property, 302
margin-right property, 302
margin-top property, 302, 1054
margins, 299-308
  in box model, 237
  collapsing, 195-200, 303-305, 500, 548
  flexbox layout, 484-485, 494, 500
  floating images, 405, 415-416, 422, 992-994
  grid layout, 548, 614-617
  inline elements, 306-308
  length values, 300
  negative, 195, 198-200, 205-207, 305-306, 415-416
  padding and, 241
  paged media, 1048
  percentage values, 301
  replaced elements, 227
  single-side properties, 302
::marker pseudo-element, 799, 802-803
markers, list, 791-800
  as generated content, 804
  images as, 795-798
  positioning, 798
  shorthand property, 799
  strings as, 795
markup (see HTML)
mask property, 1012
mask-border property, 1019
mask-border-mode property, 1019
mask-border-outset property, 1017
mask-border-repeat property, 1018
mask-border-slice property, 1015
mask-border-source property, 1014
mask-border-width property, 1017
mask-clip property, 1009
mask-composite property, 1010-1012
mask-image property, 1000-1003
mask-mode property, 1003-1005
mask-origin property, 1008
mask-position property, 1008-1009
mask-repeat property, 1006
mask-size property, 1005-1006
mask-type property, 1013
masks, 1000-1020
  border-image masking, 1014-1020
  changing mode, 1003-1005
  clipping, 994-999, 1009
  compositing, 1010-1012
  image for, 1000-1003
  origin, 1008
  positioning, 1008-1009
  repeating, 1006
  setting types, 1013
  shorthand property, 1012
  sizing, 1005-1006
match-parent keyword, text-align, 738

mathematical expressions (calc() function), 136, 148
matrix() function, 861-864
matrix3d() function, 862
max() function, 150
max-block-size property, 185
max-content keyword, 183, 523, 525, 563
max-height property, 430-432
max-inline-size property, 185
max-width property, 430-432
maximum values, function, 149
measurement units (see units of measure)
media attribute, 13, 1026-1027
media queries (@media), 1025-1043
  complex, 1028-1030
  deprecated media features, 1040
  media attribute, 13, 1025-1027
  media feature descriptors and values, 1031-1035
  ranged media features, 1038-1040
  for reduced-motion animations, 912
  responsive styling using, 1041
  special value types, 1030
  user preference and media features, 1035-1038
middle keyword, table cell, 658
middle keyword, vertical-align, 220, 750
min() function, 149
min-block-size property, 185
min-content keyword, 183, 523, 526, 564
min-height property, 430-432
min-inline-size property, 185
min-width property, 430-432, 502-503
minified CSS, 5
minimum values, function, 150
minmax() function, 557, 561, 563, 565-566
mirrored linear gradient, 389
mix-blend-mode property, 964-972
  color blend mode, 971
  color-burn blend mode, 970
  color-dodge blend mode, 969
  darken blend mode, 965
  difference blend mode, 966
  exclusion blend mode, 966
  hard-light blend mode, 968
  hue blend mode, 971
  lighten blend mode, 965
  luminosity blend mode, 971
  multiply blend mode, 967
  non-mode categories, 965
  overlay blend mode, 967
  saturation blend mode, 971
  screen blend mode, 967
  soft-light blend mode, 968
mm (millimeters) unit, 137
monochrome descriptor, @media, 1039
monospace fonts, 662
ms (milliseconds) unit, 167
multiple border styles, 253
multiple class selectors, 30-31
multiply blend mode, 967
mutability pseudo-classes, 82

N

@namespace rule, 1071
navigation bars (see flexible box layout)
negation pseudo-class, 84-87
negative counter values, 813
negative delay values, transition delays, 900
negative descriptor, @counter-style, 831
negative margins, 195, 198-200, 205-207, 227, 305-306, 415-416
nesting grids, 547
newline character, in strings, 131
no-common-ligatures keyword, 710
no-contextual-ligatures keyword, 710
no-discretionary-ligatures keyword, 710
no-historical-ligatures keyword, 710
non-CSS presentational hints, 126
non-hyperlink location pseudo-classes, 71-74
nonanimatable properties, using, 918
nonreplaced elements, 7, 175, 216-218
normal flow, 174
not logical keyword, 1029, 1062, 1069
:not() pseudo-class, 84-87, 88, 108
:nth-child() pseudo-class, 62-66
:nth-last-child() pseudo-class, 65-66
:nth-last-of-type() pseudo-class, 66
:nth-of-type() pseudo-class, 66
null cell tokens, grid layout, 574
number sign (#)
  in ID selectors, 32
  in value syntax, xxi
<number> value type, 135
numeric font variants, 707-709
numeric pattern, @counter-style, 830-834

O

object bounding box, 845
object-fit property, 1020-1022
object-position property, 1022-1023
oblique font style, 695, 698
octothorpe (#)
  in ID selectors, 32
  in value syntax, xxi
offsets
  absolute positioning, 423, 426, 432-448
  basic properties, 424-427
  box-shadow, 359
  changing edges, 326-328
  fixed positioning, 423, 449-450
  length values, 323, 428-432
  relative positioning, 423, 450-452
  shorthand properties, 427
  static positioning, 423, 436
  sticky positioning, 423, 452-455
  subgrids, 605-609
  text-underline-offset, 759
Oklab and Oklch, 162
oldstyle-nums numeric display keyword, 708
only keyword, 1030
:only-child pseudo-class, 56-57, 66
:only-of-type pseudo-class, 57-58, 62, 68
opacity() function, 958
OpenType font features, 707, 711, 712, 714-718
operating system fonts, 726
optical sizing, fonts, 719
optimizeLegibility keyword, text-rendering, 764
optimizeSpeed keyword, text-rendering, 764
optional keyword, font-display, 677
:optional pseudo-class, 77, 80
or logical keyword, 1029, 1070
order property, 501, 541-546, 627-628
order, cascade by, 123-126
ordinal group, 542
ordinal numeric display keyword, 708
orientation descriptor, @media, 1033
origin, in cascade rules, 118-119
orphans property, 1052-1054
:out-of-range pseudo-class, 77, 81
outer display type, 8, 231
outline property, 296
outline-color property, 296
outline-style property, 294
outline-width property, 295
outlines, 293-298
  versus borders, 297
  colors, 296
  shorthand property, 296
  styles for, 294
  width of, 295
overconstraining
  formatting properties, 203
  relative positioning, 452
overflow property, 193-195
overflow-block descriptor, @media, 1033
overflow-inline descriptor, @media, 1033
overflow-wrap property, 781-782
overflow-x property, 194
overflow-y property, 194
overflowing flex container, 477-479, 485, 492
overhanging image borders, 286-288
overlapping elements
  floated, 407-410, 417-418
  grid items, 590, 594, 626-628
overlay blend mode, 967

P

pad descriptor, @counter-style, 832-834
padding, 196, 198, 238-250
  additive versus subtractive, 200-201
  in box model, 237
  inline elements, 223, 247-249
  logical, 243-245
  margins and, 241
  percentage values, 245-247
  replaced elements, 227, 249
  replicating values, 240-241
  single-side, 242-243
padding property, 238-241
padding-block property, 244
padding-block-end property, 190, 243
padding-block-start property, 190, 243
padding-bottom property, 242
padding-inline property, 244
padding-inline-end property, 201, 243
padding-inline-start property, 201, 243
padding-left property, 242
padding-right property, 242
padding-top property, 242
page area, 1045
@page block rule, 1046-1049
page property, 1048
page-break-after property, 1049-1051, 1054
page-break-before property, 1049-1051, 1054

page-break-inside property, 1051, 1055
paged media, 1043-1057
  elements outside the page, 1057
  margins, 1048
  named page types, 1048-1049
  orphan and widow handling, 1052-1054
  page breaking, 1049-1056
  page size, 1045-1048
  print styles for, 1043
  repeated elements on every page, 1056
  screen versus print, 1043
parent-child relationship, document structure, 43
pc (picas) unit, 137
<percentage> value type, 135
performance issues
  animations, 932
  custom fonts, 672
  text shadows, 767
period (.), in class selectors, 28-30
perspective property, 873-875
perspective() function, 864-866, 873-875
perspective-origin property, 875-876
petite-caps keyword, font-variant-caps, 705
phantom classes (see pseudo-class selectors)
physical properties, 187
pixels, 138, 139
place-content property, 493
place-items property, 623
place-self property, 620
::placeholder pseudo-element, 97
:placeholder-shown pseudo-class, 77, 79
plus sign (+)
  adjacent-sibling combinator, 48-50
  in value syntax, xxi
pointer descriptor, @media, 1033
polygon float shapes, 989-992
polygon() function, clip shapes, 996
position property, 422-423
position values, 168
positional numbering, 830
positioning, 422-455
  absolute (see absolute positioning)
  auto edges and, 436-437
  background images, 315-330, 333, 353
  backgrounds, 168, 315-330, 351
  containing blocks and, 175, 424, 432-434, 452
  fixed, 423, 449-450
  gradient, 364-369, 382-388
  images, 1022-1023
  inset shorthand properties, 427-428
  list markers, 798
  masks, 1008-1009
  nonreplaced elements, 175, 437-441
  object, 1022-1023
  offset properties, 424-427
  relative, 423, 450-452
  replaced elements, 175, 441-443
  static, 423, 436
  sticky, 423, 424, 452-455
  transforms (see transforms)
  types, 422-423
  width and height, 428-432
pound sign (#)
  in ID selectors, 32
  in value syntax, xxi
precedence rules (see cascade)
preferred stylesheet, 15
prefers-color-scheme descriptor, @media, 1033
prefers-contrast descriptor, @media, 1034
prefers-reduced-motion descriptor, @media, 1034
prefers-reduced-motion media query, 912
prefix descriptor, @counter-style, 824-826
presentational hints, non-CSS, 126
print media type, 1027
printing
  animations, 955
  paged media styles, 1043
  transitions and, 908
privacy issue
  ::selection pseudo-element and, 100
  visited links and, 70
properties, 3
  (see also specific properties by name)
  animatable, 880, 886, 904-908, 917
  custom, 168-172
  logical, 181-188, 243-245, 255, 258, 261, 275
  repeating in keyframes, 917
proportional fonts, 662
proportional-nums numeric display keyword, 708
pseudo-class selectors, 53-94
  chaining pseudo-classes, 54, 60
  hyperlink pseudo-classes, 69-71
  location pseudo-classes, 68-74
  logical pseudo-classes, 84-88

shadow pseudo-classes, 103
structural pseudo-classes, 54-68
UI-state pseudo-classes, 77-83
user action pseudo-classes, 74-77
pseudo-element selectors, 95-103
::backdrop, 101
::file-selector-button pseudo-element, 98
::first-letter, 95
::first-line, 96-97
::first-letter, 97
highlight pseudo-elements, 99-101
nesting issue with :has pseudo-class, 94
::placeholder pseudo-element, 97
shadow DOM pseudo-elements, 104
::video-cue, 102-103
pt (points) unit, 137
px (pixels) unit, 137

Q

q (quarter-millimeters) unit, 137
question mark (?), in value syntax, xxi
quotes ('…' or "…")
enclosing attribute values, 42
enclosing font names, 665
enclosing strings, 131
as generated content, 809-812

R

rad (radians) unit, 166
radial gradients, 379-392
color stop positioning, 383-388
colors in, 379, 383, 386-388
degenerate (edge) cases, 388-391
gradient rays, 380, 381, 383-388
positioning, 382-388
radial-gradient() function, 379
repeating, 391
shape of, 380
size of, 380-382
range pseudo-classes, 81
<ratio> value type, 167, 1030
:read-only pseudo-class, 77, 82
:read-write pseudo-class, 77, 82
reader origin, 117, 118
reduced-motion animations, 912
rel attribute (HTML), 13
relative font sizes, 689-690
relative length units, 139-144
relative positioning, 423, 450-452
relative URLs, 132-133
rem unit, 143
rendering speed and legibility of text, 764
repeat() function, 567-570
repeating elements
animation keyframe properties, 917
background images, 330-339, 348
gradients, 376-379, 391, 397-401
grid tracks, 567-570
image borders, 288-289
replaced elements, 7, 175
borders for, 227, 267
inline, 213, 226-229
inline-axis formatting, 208
margins, 227, 308
padding, 227, 249
positioning, 175, 441-443
replicating values, padding, 240-241
:required pseudo-class, 77, 80
resolution descriptor, @media, 1039
resolution units, 139
<resolution> value type, 1030
resources, xxii, 1073-1074
responsive flexing, 519-521
responsive styling, media features, 1041
reverse keyword, animation-direction, 926
revert keyword, 129
revert-layer keyword, 129
RGB colors, 153-155
RGBa colors, 156
right property, 424-427
right-to-left (RTL) writing mode, 467, 472
Roman numeral counting, generated, 836
root elements, 44, 175
:root pseudo-class, 54
root-relative values, 142-144
rotate property, 858-859
rotate() function, 854
rotate3d() function, 855-858
rotateX() function, 854
rotateY() function, 854
rotateZ() function, 854
rounding corners, 267-276, 336, 984
row boxes, tables, 630
row group boxes, tables, 630
row primacy, tables, 634
row-gap property, 494-497, 569
RTL (right-to-left) writing mode, 467, 472
rules, 2, 12-16, 21

(see also at-rules)
cascade, 107, 110, 116-126
declarations, 19, 24-26, 112-113, 953
inheritance, 107, 113-116
vendor prefixes, 3

S

s (seconds) unit, 167
safe and unsafe flex item alignment, 487
sans-serif fonts, 662
saturate() function, 961
saturation blend mode, 971
scale property, 853, 859
scale() function, 852
scale3d() function, 852
scaleX() function, 851
scaleY() function, 847, 851
scaleZ() function, 851
scaling factor, 222, 337, 688
scan descriptor, @media, 1034
:scope pseudo-class, 71, 73
scoped styles, 977
scoping root, 73
screen blend mode, 967
screen media type, 1027
(see also viewport)
scripting @keyframes animations, 919
scripting descriptor, @media, 1034
scrolling
absolutely positioned elements, 434
attaching background image and, 339
block box overflow, 193, 194
sticky positioning and, 452-455
seizure disorders, animations affecting, 912
::selection pseudo-element, 99
selectors, 3, 21-51
adjacent-sibling combinator, 48-50
attribute, 34-42
child combinator for, 47
class, 28-31, 33
descendant (contextual), 45-47
document structure and, 42-51
grouping of, 23-26
ID, 32-34
@namespace rule for, 1071
pseudo-class (see pseudo-class selectors)
pseudo-element, 95-103, 104-105
specificity, 107-113
type, 22
universal, 24, 30, 32, 85, 111, 115
semicolon (;), in rules, 24, 25
separated cell borders, table, 643-646
sepia() function, 960
serif fonts, 662
shadow DOM pseudo-elements, 104
shadow pseudo-classes, 103
shadows
box-shadow property, 357-360, 906, 958
drop shadow filter, 958-960
text, 765-767
shape-image-threshold property, 981
shape-margin property, 992-994
shape-outside property, 980-981
shaping content around floats, 979-994, 981
with image transparency, 981
inset shapes, 982-992
margins, 992-994
shaping of rounded corners in borders, 271-272
shrink factor, for flexbox layout, 509, 512-521
sibling elements, document structure, 43, 48-51
simple alpha compositing, 963
single-axis overflow, 194-195
single-side border styles, 254, 261
single-side margins, 302
single-side padding, 242-243
size property, 1046
size-adjust descriptor, for fonts, 694
skew() function, 860
skewX() function, 860
skewY() function, 860
skipping ink, text decoration, 761-762
slashed-zero numeric display keyword, 708
slashes
backslash, 131
forward slash, xx, 5-6, 291
:slotted() pseudo-class, 104
::slotted() pseudo-element, 105
small-caps keyword, font-variant, 705
soft-light blend mode, 968
soft-wrapping of text, 778, 781
space-separated list of words, 38
spacing, 212
(see also whitespace)
aligning content, 490
background images, 334-337
border-spacing property, 644-645
flex items, 494-498
justifying content, 479

letter, 722, 741-742
margins and blank space, 299
word, 739-740
spanning grid lines, 578-580
speak-as descriptor, @counter-style, 838-839
specificity, 107-113
animation, 953
in cascade rule order, 117, 123
declarations, 109-111
ID selector and attribute, 112
importance of, 112-113
lack of in universal selector, 115
multiple match resolution, 111
zeroed selector, 111
::spelling-error pseudo-element, 99, 101
spherical coordinate system, 843
square brackets ([…]), in value syntax, xxi
src descriptor, @font-face, 668-672
src() function, 133
stacking of elements, 445-448, 472
starburst pattern, conic gradient, 398
start and end alignment, 736
state, pseudo-classes based on (see UI-state pseudo-classes)
static positioning, 423, 436
step timing functions, 896-898, 941-944
step-end timing function, 941-944
step-start timing function, 941-944
sticky positioning, 423, 424, 452-455
sticky-constraint rectangle, 424
stretching
aligning content, 491
of fonts, 700-701
strict keyword, line-break, 781
string values, 131, 807
stroke-box, SVG, 997
structural pseudo-classes, 54-68
style attribute (HTML), 19
<style> element, 1025-1027
linking CSS to HTML documents, 16
media attribute, 6
style sheets (see CSS)
sub keyword, vertical-align, 220
subgrids
explicit tracks, 605
gaps for, 612-613
naming lines, 610-612
offsets, 605-609
subscripts, 714, 748
subsetting fonts, 672
substring matching, for attribute selectors, 37-41
suffix descriptor, @counter-style, 824-826
super keyword, vertical-align, 220
superscripts, 714, 748
@supports (feature query) rule, 1067-1070
SVG format
clip paths and, 995, 999
filtering, 962-963
object bounding box, 845, 997
text rendering and, 765
swap keyword, font-display, 677
symbolic pattern, @counter-style, 826-829
symbols descriptor, @counter-style, 821
syntax conventions used in this book, xx-xxii
synthesizing fonts, 702
system descriptor, @counter-style, 821
system fonts, 726

T

tab-size property, 775
table cells versus grid cells, 630
table keyword, display, 632
table layout, 629-660
alignment within cells, 657-660
anonymous objects, 635-639
arrangement rules, 630
bounding box, 845
display values, 631-635
versus grid layout, 547
height of, 656
layers in, 639
row primacy, 634
sizing, 650-660
table versus grid cells, 630
visual arrangement, 629
table-caption keyword, display, 633
table-cell keyword, display, 633
table-column keyword, display, 633
table-column-group keyword, display, 633
table-footer-group keyword, display, 633
table-header-group keyword, display, 633
table-layout property, 653-656
table-row keyword, display, 632
table-row-group keyword, display, 632
tabular-nums numeric display keyword, 708
:target pseudo-class, 71-72
::target-text pseudo-element, 99, 100

:target-within pseudo-class, 71, 72
tech() function, @font-face rule, 670
text nodes
  :empty pseudo-class using, 55
  in flex items, 501
text properties, 731-790
  alignment of lines in an element, 742-751
  alignment within a line, 734-739, 742
  block direction, 731
  capitalization, 752-754
  emphasis marks, 767-772
  fonts (see fonts)
  gap behavior in, 748
  hyphenation, 776-778
  indentation, 732-734
  inline direction, 731
  letter spacing, 741-742
  line breaks, 773, 776, 780-782
  line height, 743-746
  rendering speed and legibility, 764
  shadows, 765-767
  shaping content around floats, 979-994
  text decoration, 754-764
  transformations, 751-754
  vertical alignment, 742-751
  whitespace, handling, 772-776
  word breaking, 778-779
  word spacing, 739-740
  wrapping text, 781
  writing modes (flow direction), 783-789
text shadows, transitions, 906
text-align property, 658, 734-738, 742
text-align-last property, 738
text-bottom keyword, vertical-align, 220
text-combine-upright property, 786-788
text-decoration property, 758
text-decoration-color property, 756
text-decoration-line property, 754-755
text-decoration-skip-ink property, 761-762
text-decoration-style property, 757
text-decoration-thickness property, 756
text-emphasis property, 771
text-emphasis-color property, 769
text-emphasis-position property, 770
text-emphasis-style property, 767-769
text-indent property, 732-734
text-orientation property, 467, 784
text-rendering property, 764-765
text-shadow property, 765-767
text-top keyword, vertical-align, 220
text-transform property, 751-754
text-underline-offset, 759
tilde (~), general-sibling combinator, 50
tilde, equal sign (~=), in attribute selectors, 37
tiling background images, 330-339
time units, 167
timing functions
  animations, 937-947
  transitions, 892-898
title attribute, 14-15
titling-caps keyword, font-variant-caps, 705
to keyframe selector, animations, 916
top keyword, table cell, 658
top keyword, vertical-align, 219, 749
top property, 424-427
transform property, 845-848
transform-box property, 869-870
transform-origin property, 866-869
transform-style property, 870-873
transformation, text properties, 751-754
transforms, 841-878
  3D style for, 842-844, 870-873
  animated, 855, 863
  backface visibility, 876-878
  bounding box for, 845
  coordinate systems used by, 841-844
  end-state equivalence, 863
  functions used with, 848-866
  order of individual properties, 859
  origin of, moving, 866-869
  perspective change, 873-876
transition property, 901-903
transition-delay property, 898-901
transition-duration property, 890-892
transition-property property, 885-890
transition-timing-function property, 892-898
transitionend event, 888-890
TransitionEvent Interface, 888
transitions, 879-909
  animatable properties for, 880, 886, 905
  border corner rounding, 273
  delaying, 898-901
  duration, 890-892, 904
  events, 888
  initial state, 882-884
  limiting effects by property, 885-890
  printing, 908
  reversing interrupted, 903

shorthand property, 901-903
suppressing, 888
timing of, 892-898, 903
on transform property, 881-884
translate property, 851, 859
translate() function, 849
translate3d() function, 849
translateX() function, 848
translateY() function, 848
translateZ() function, 849
transparent backgrounds, 310, 363
transparent borders, 262
transparent keyword, 153
turn unit, 166
type selectors, 22
typographic character unit, 771

U

UI-state pseudo-classes, 77-83
underlining text, 755
unicase keyword, font-variant-caps, 706
Unicode encoding, 132
unicode-bidi property, 130, 788
unicode-range descriptor, @font-face, 673-675
units of measure
absolute length, 136-139
angle, 166
fractional, 558-562
frequency, 167
lengths for container queries, 1066
with linear gradients, 363
relative length, 139-144
resolution, 139
time, 167
universal selector (*), 24
class selector and, 30, 32
inheritance and), 115
zeroed selector specificity), 111
unset keyword, 129
update descriptor, @media, 1034
URI values, generated content, 808
<url> value type, 134
url() function, 962
url(), using to import font faces, 668
URLs, images specified by, 132-133
user action pseudo-classes, 74-77
user agent origin, 117, 118

V

:valid pseudo-class, 77, 81
values, 3
(see also specific values by name)
all property, 130
color (see color values)
custom properties, 168-172
function, 146-151
identifiers, 132
images, 134
interpolated, 905-908
keywords, 127-130
mixed-value padding, 240
numbers, 134-136
percentages, 135
position, 168
ratio, 167
replicating in borders, 258
replicating in padding, 240-241
specifying multiple, 23
strings, 131
syntax conventions, xx-xxii
URLs, 132-133
var() function, 169-172
variable font files, 662
variables (see custom properties)
vb (viewport block) unit, 145
vendor prefixes, 3
vertical bar (|), in value syntax, xx
vertical bar, double (||), in value syntax, xxi
vertical bar, equal sign (|=), in attribute selectors, 37
vertical formatting
alignment setting, 218-220, 226, 501, 549, 658, 742-751
auto block sizing, 191-192
block boxes, 179-183
box sizing, 189-190
collapsing margins, 196-200, 303-305, 500, 548
content-based sizing, 183-185
line heights, inline formatting, 214
logical element sizing, 181-188
negative margins, 195
overflowing content, handling, 193-195
percentage heights, 192
properties, 201
vertical writing direction, 174, 731, 742-751
vertical-align property

flex items, 501
grid items and, 549
line heights, 214
setting, 218-220
in table cells, 658
in text, 746-751, 763
vestibular disorders, animations affecting, 912
vh (viewport height) unit, 144
vi (viewport inline) unit, 145
::video-cue pseudo-element, 102-103
video-dynamic-range descriptor, @media, 1035
view-box, SVG, 997
viewport (browser display)
as container for fixed positioning, 449
resolution units, 139
viewport-relative units, 139, 144-146
visibility property
animatability, 905
animatability and, 918
elements, 233
table columns, 635
visible keyword, 193, 233
visited links and privacy, 70
:visited pseudo-class, 69-71, 76
visual formatting, 173-235
block flow direction, 174, 423, 432-434
borders, 176, 189-191, 196, 198
boxes (see boxes)
content area, 173-174
display roles, 7-10
element display, altering, 176-181
horizontal formatting, 180-188, 194
inline base direction, 174
inline formatting, 210-233
inline-axis formatting, 200-209
list items, 209
margins (see margins)
normal flow, 174
outlines, 176, 293-298
padding (see padding)
sizing of logical element, 181-188
vertical formatting (see vertical formatting)
visibility of elements, 233-235
vmax (viewport maximum) unit, 145
vmin (viewport minimum) unit, 145
vw (viewport width) unit, 144

W

weight, in cascade rules, 116-117
:where() pseudo-class, 87-88, 111
white-space property, 772-775, 781
whitespace
calc() function and, 149
CSS comments not considered, 6
handling of, 4, 6
as ignored by flex container, 499
separating value keywords, 24
in text, 772-776, 781
widows property, 1052-1054
width descriptor, @media, 1038
width property
logical element sizing, 186-188
positioning elements, 428-430
table columns, 635
will-change property, 954-955
word-break property, 778-779
word-spacing property, 739-740
word-wrap property (see overflow-wrap property)
wrapping flex lines, 468-470
wrapping text, 773, 776
writing modes (flow direction), 180-188
block box handling, 179-181, 460
changing for languages, 463, 467-468
flexbox as agnostic to, 460
grid flow and, 595
inline layout's horizontal bias, 460
logical element sizing, 181-188
text alignment, 735-736
writing-mode property, 467, 783-784

X

x-height, 691
x/y/z coordinate system, 841-843
XML
attribute selectors with, 35, 36
class selector support, 34
dot-class notation issue for, 34
root element and, 54
type selectors for, 22

Z

z-index property, 444-448, 455
zero size situation, radial gradients, 388-391
zeroed selector specificity, 111

About the Authors

Eric A. Meyer has been working with the web since late 1993 and is an internationally recognized expert on the subjects of HTML, CSS, and web standards. A widely read author, he joined Igalia in 2021 as a developer advocate and standards evangelist, a role he originally performed for Netscape Communications in 2001.

Beginning in early 1994, Eric was the visual designer and campus web coordinator for the Case Western Reserve University website, where he also authored a widely acclaimed series of three HTML tutorials and was project coordinator for the online version of the *Encyclopedia of Cleveland History* and the *Dictionary of Cleveland Biography*, the first encyclopedia of urban history published fully and freely on the web.

He is the author of *Design for Real Life* (A Book Apart), *Eric Meyer on CSS* and *More Eric Meyer on CSS* (New Riders), *CSS: The Definitive Guide*, 4e (O'Reilly), and *CSS2.0 Programmer's Reference* (Osborne/McGraw-Hill), as well as numerous articles for A List Apart, Net Magazine, Netscape DevEdge, UX Booth, UX Matters, the O'Reilly Network, Web Techniques, and Web Review. Eric also created the classic CSS Browser Compatibility Charts (a.k.a. "The Mastergrid") and coordinated the authoring and creation of the W3C's first official CSS Test Suite.

He has conducted customized training for a wide variety of organizations through his independent consulting, and has delivered keynotes and technical talks at numerous conferences around the world. In 2006, he was inducted into the International Academy of Digital Arts and Sciences for "international recognition on the topics of HTML and CSS" and helping to "inform excellence and efficiency on the Web." In December 2014, he accidentally touched off Slate's Internet Outrage of the Day.

In his personal time, Eric acts as list chaperone of the css-discuss mailing list, which he cofounded with John Allsopp of Western Civilisation, and which is now supported by *evolt.org*. Eric lives in Cleveland, Ohio, which is a much nicer city than you've been led to believe. For nine years he was the host of "Your Father's Oldsmobile," a big-band radio show heard weekly on WRUW 91.1 FM in Cleveland. He's a staunch defender of the Oxford comma and the right of everyone everywhere to follow a sentence with however many spaces they deem proper. He enjoys a good meal whenever he can and considers almost every form of music to be worthwhile.

You can find more detailed information on Eric's personal web page.

How does someone get to be the author of *Flexbox in CSS*, *Transitions and Animations in CSS*, and *Mobile HTML5* (O'Reilly), and coauthor of *CSS3 for the Real World* (SitePoint) and *CSS: The Definitive Guide*? For **Estelle Weyl**, the journey was not a direct one. She started out as an architect, used her master's degree in health and social behavior from the Harvard School of Public Health to lead teen health programs, and then began dabbling in website development. By the time Y2K rolled around, she had become somewhat known as a web standardista at *http://www.standardista.com*.

Today, she writes a technical blog that pulls in millions of visitors, and speaks about CSS, HTML, JavaScript, accessibility, and web performance at conferences around the world. In addition to sharing esoteric programming tidbits with her reading public, Estelle has consulted for Kodak Gallery, SurveyMonkey, Visa, Samsung, Yahoo!, Apple, Williams-Sonoma, and Google Chrome's Web.Dev, where she wrote Learn HTML, among others. She is currently a technical writer for Open Web Docs working on MDN.

When not coding, she spends her time cooking, gardening, and doing construction, striving to update her 1910s-throwback home. Basically, it's just one more way Estelle is working to bring the world into the 21st century.

Colophon

The animals on the cover of *CSS: The Definitive Guide* are salmon (*salmonidae*), which is a family of fish consisting of many species. Two of the most common salmon are the Pacific salmon and the Atlantic salmon.

Pacific salmon live in the northern Pacific Ocean off the coasts of North America and Asia. There are five subspecies of Pacific salmon, with an average weight of 10 to 30 pounds. Pacific salmon are born in the fall in freshwater stream gravel beds, where they incubate through the winter and emerge as inch-long fish. They live for a year or two in streams or lakes and then head downstream to the ocean. There they live for a few years, before heading back upstream to their exact place of birth to spawn and then die.

Atlantic salmon live in the northern Atlantic Ocean off the coasts of North America and Europe. There are many subspecies of Atlantic salmon, including the trout and the char. Their average weight is 10 to 20 pounds. The Atlantic salmon family has a life cycle similar to that of its Pacific cousins, and also travels from freshwater gravel beds to the sea. A major difference between the two, however, is that the Atlantic salmon does not die after spawning; it can return to the ocean and back to spawn again, usually two or three times.

Salmon, in general, are graceful, silver-colored fish with spots on their backs and fins. Their diet consists of plankton, insect larvae, shrimp, and smaller fish. Their unusually keen sense of smell is thought to help them navigate from the ocean back to the exact spot of their birth, upstream past many obstacles. Some species of salmon remain landlocked, living their entire lives in fresh water.

Salmon are an important part of their ecosystems; their decaying bodies provide fertilizer for streambeds. Their numbers have been dwindling over the years, however. Factors in the declining salmon population include habitat destruction, fishing, dams that block spawning paths, acid rain, droughts, floods, and pollution.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Dover's Animals*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

Learn from experts.
Become one yourself.

Books | Live online courses
Instant answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.

©2023 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. 175 7x9.1975