

Rajmond JÁNÓ · Adelina Ioana ILIEȘ

WEB TECHNOLOGIES

PART TWO

CSS

HTML

CSS

JS



U.T.PRESS

Cluj-Napoca, 2026

ISBN 978-606-737-864-1

Rajmond JÁNÓ

Adelina Ioana ILIEȘ

WEB TECHNOLOGIES

Part two

CSS



U.T. PRESS
Cluj-Napoca, 2026
ISBN 978-606-737-864-1



Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Ș.l.dr.ing. Cosmina Corcheș

Ș.l.dr.ing. Ștefania Ramona Benea

Pregătire format electronic on-line: Gabriela Groza

Copyright © 2026 Editura U.T.PRESS

Reproducerea integrală sau parțială a textului sau ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-862-7

ISBN 978-606-737-864-1 vol. 2

Contents

- Contents 1
- Chapter 1. Introduction to CSS..... 4**
 - 1.1. What Is CSS? 4
 - 1.2. The Relationship Between HTML and CSS..... 5
 - 1.3. Evolution of CSS: From CSS1 to CSS3 and Beyond 7
 - 1.4. Overview of Modern CSS Development 8
- Chapter 2. Getting Started with CSS 10**
 - 2.1. Ways to Add CSS: Inline, Internal, and External Styles 10
 - 2.2. CSS Syntax: Rules, Selectors, and Declarations 12
 - 2.3. Linking Stylesheets and Organizing CSS Files.....14
 - 2.4. Using Browser Developer Tools to Inspect Styles 16
- Chapter 3. Selectors, Specificity, and the Cascade 19**
 - 3.1. Basic Selectors: Element, Class, and ID 19
 - 3.2. Attribute, Pseudo-Class, and Pseudo-Element Selectors 21
 - 3.3. Combinators and Grouping Selectors 23
 - 3.4. Understanding Specificity, the Cascade, and Inheritance..... 25
- Chapter 4. The CSS Box Model 29**
 - 4.1. Content, Padding, Border, and Margin 29
 - 4.2. The box-sizing Property.....31
 - 4.3. Display Types: Block, Inline, and Inline-Block..... 32
 - 4.4. Managing Overflow and Visibility..... 34
- Chapter 5. Colors, Backgrounds, and Borders..... 37**
 - 5.1. Color Formats: Named Colors, HEX, RGB, and HSL 37
 - 5.2. Background Images, Position, Size, and Repeat..... 39
 - 5.3. Gradients: Linear, Radial, and Conic 41
 - 5.4. Borders, Border-Radius, and Box Shadows..... 44

Chapter 6. Typography and Text Styling	47
6.1. Font Properties: Family, Size, Weight, and Style.....	47
6.2. Web Fonts and the @font-face Rule	49
6.3. Text Alignment, Decoration, and Transformation	51
6.4. Line Height, Letter Spacing, and Readability	54
Chapter 7. Layout with Flexbox	57
7.1. Flex Containers and the Main and Cross Axes	57
7.2. Flex Item Properties: Grow, Shrink, and Basis.....	59
7.3. Aligning and Justifying Content	61
7.4. Common Flexbox Layout Patterns	63
Chapter 8. Layout with CSS Grid	66
8.1. Defining Grid Containers, Rows, and Columns	66
8.2. Placing Items with Grid Lines and Areas.....	67
8.3. The fr Unit and the repeat() Notation.....	69
8.4. Common Grid Layout Patterns	71
Chapter 9. Responsive Design and Media Queries.....	74
9.1. The Viewport and Mobile-First Design	74
9.2. Writing Media Queries and Choosing Breakpoints	75
9.3. Responsive Units: %, vw/vh, em, and rem	78
9.4. Responsive Images and Adaptive Layouts.....	80
Chapter 10. Transitions, Transforms, and Animations	83
10.1. CSS Transitions and Easing Functions.....	83
10.2. 2D and 3D Transforms	85
10.3. Keyframe Animations with @keyframes.....	87
10.4. Performance Considerations for Animated CSS	89
Chapter 11. Modern CSS Features and Best Practices	92
11.1. Custom Properties (CSS Variables) and var()	92

11.2. Functional Notations: calc(), clamp(), min(), and max()	94
11.3. Organizing CSS and Naming Conventions	96
11.4. Cross-Browser Compatibility and Vendor Prefixes	97
Chapter 12. Hands-On Project: Styling a Complete Web Page	100
12.1. Planning a Visual Design from a Style Guide	100
12.2. Styling Layout, Typography, and Color	101
12.3. Adding Responsive Behavior and Interactivity	104
12.4. Testing and Refining Across Devices	106
Conclusion	109
A Recap of Key Concepts	109
Next Steps in Web Development	110
Additional Resources	110
Appendices	112
A. CSS Property Reference	112
B. Glossary of Terms	117
C. Further Reading and Online Resources	118

Chapter 1. Introduction to CSS

1.1. What Is CSS?

CSS, short for Cascading Style Sheets, is the language that controls how HTML content looks and is laid out on a page. Where HTML describes what a piece of content is, a heading, a paragraph, a list, an image, CSS describes how that content should be presented: its color, size, spacing, position, and behavior across different screen sizes. A single HTML document can look completely different depending on which stylesheet is attached to it, which is exactly the point: CSS keeps the meaning of a document separate from its appearance.

The name itself describes how the language works. A style sheet is a collection of formatting rules. "Cascading" refers to the way those rules combine: styles can come from the browser, the site's author, and even the reader, and CSS defines a precise, predictable order in which conflicting rules are resolved. That mechanism, the cascade, is one of the defining ideas of the language and is the subject of Chapter 3.

To see the difference CSS makes, consider a simple HTML fragment with no styling applied at all:

```
1 <h1>Welcome to My Site</h1>
2 <p>This paragraph has no styling applied to it yet.</p>
3 <button>Click Me</button>
```

Rendered on its own, this markup produces a plain, top-to-bottom stack of black text on a white background, using whichever default font the browser happens to use. Every visual decision beyond that, the font, the colors, the spacing around the button, whether the layout uses one column or three, is left entirely to CSS.

In broad terms, CSS is responsible for:

- Color and typography, text color, background color, font family, size, and weight;
- Box styling, borders, padding, margins, rounded corners, and shadows;

- Layout, how elements are arranged on the page, using tools such as Flexbox and Grid;
- Responsiveness, adapting a layout to different screen sizes, from phones to large desktop monitors;
- Motion, transitions and animations that respond to user interaction or run automatically.

This book, Part 2 of the Web Technologies series, covers plain CSS as defined by the W3C and WHATWG specifications, the same way Part 1 covered plain HTML without relying on any external library. Preprocessors such as Sass and LESS, and CSS frameworks such as Bootstrap or Tailwind, all compile down to or build on top of the CSS covered here; they are the subject of a later volume in this series.

A Brief Comparison with HTML

It can help to think of HTML and CSS as answering two different questions about the same document. HTML answers "what is this?", this text is a heading, this text is a list item, this element is a navigation region. CSS answers "how should this look?", this heading should be large and bold, this list should have no bullets, this navigation region should stretch across the top of the page. Neither language can fully replace the other: markup without styling is usually readable but visually plain, while styling rules have nothing to apply to without well-structured markup underneath them.

1.2. The Relationship Between HTML and CSS

CSS never exists on its own; it is always applied to an HTML document (or, less commonly, to other markup languages such as SVG or XML). Understanding how the two languages connect, and why they are kept separate in the first place, is essential before writing a single line of CSS.

Separation of Concerns

Modern web development follows a principle known as separation of concerns: structure, presentation, and behavior are each handled by a different language.

- **HTML** defines the structure and meaning of content.
- **CSS** defines how that content is presented.

- **JavaScript** defines how the page behaves and responds to interaction, the subject of Part 3 of this series.

Keeping these concerns apart, rather than mixing presentation directly into markup (as the deprecated `<font>` element once did), has practical benefits: the same HTML document can be restyled completely by swapping a single stylesheet, multiple pages can share one consistent look by linking to the same CSS file, and a page remains usable, if plain, even if its stylesheet fails to load.

How a Browser Combines the Two

When a browser loads a web page, it does not apply CSS as an afterthought. It builds two internal models in parallel: the DOM (Document Object Model), built from the HTML, and the CSSOM (CSS Object Model), built from every stylesheet that applies to the page, including the browser's own built-in default styles. The browser then combines the DOM and the CSSOM into a render tree, which it uses to calculate the exact position and appearance of every visible element before drawing it on screen.

This means every element always has some styling applied to it, even before an author writes a single CSS rule. Headings are bold and sized larger than paragraph text, links are blue and underlined, and lists have bullets or numbers, all because of the browser's built-in user-agent stylesheet. Authoring CSS means overriding and extending these defaults, not starting from nothing.

A First Look at Attaching CSS

The mechanics of attaching a stylesheet to an HTML document are covered in full in Chapter 2, but it is worth seeing the connection once here. The most common approach links an external `.css` file from the document's `<head>`:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>My Page</title>
6    <link rel="stylesheet" href="styles.css">
7  </head>
8  <body>
9    <h1>Welcome to My Site</h1>
10 </body>
11 </html>
```

with the rules themselves living in a separate file:

```
1 h1 {  
2   color: darkslateblue;  
3   font-family: Georgia, serif;  
4 }
```

The <link> element is the bridge between the two documents. Nothing about the <h1> element itself changed, the CSS file simply describes how any element matching the selector h1 should be rendered, anywhere it appears in the linked document.

1.3. Evolution of CSS: From CSS1 to CSS3 and Beyond

CSS has been developed and standardized by the World Wide Web Consortium (W3C) since the mid-1990s, growing from a simple way to set fonts and colors into the layout and animation engine that powers modern web design.

CSS1: The Beginning (1996)

CSS was proposed by Håkon Wium Lie in 1994 and became a W3C recommendation in December 1996 as CSS Level 1. It covered the fundamentals still in use today: font properties, text color and alignment, spacing via margins and padding, and basic positioning of elements, a first, modest alternative to styling pages with HTML tables and presentational markup.

CSS2 and CSS2.1 (1998-2011)

CSS Level 2, released in 1998, added support for positioning schemes (absolute, relative, and fixed positioning), media types that let a stylesheet target print output differently from screen output, and more complete control over typography. Because parts of CSS2 were never consistently implemented by browsers, the W3C later published CSS2.1, a revised recommendation in 2011 that documented the subset of CSS2 actually supported in practice, an early lesson in how much browser compatibility shapes the CSS specification process.

CSS3 and the Shift to Modules

Rather than publishing a single monolithic CSS3 specification, the CSS Working Group split CSS development into independent modules, each covering one area of functionality and progressing through the standardization process, and gaining browser support, on its own schedule. Selectors, Color, Backgrounds and Borders, Flexible Box Layout, Grid Layout, Transitions, and Animations are all separate modules rather than sections of one document.

This modular approach is why there is no single, universally agreed "CSS4": some modules, such as Color, have progressed to Level 4 or 5, while others are still at Level 1 or 2. Browsers implement individual modules as they stabilize, which is why a feature can be usable in one browser well before it appears in another.

CSS as a Living Standard

Today, CSS development resembles the same continuously evolving process described for HTML5 in Part 1 of this series. New modules and features, container queries, the `has()` relational selector, nesting, and nearly every layout and coloring feature used in this book, are developed openly, implemented incrementally by browser vendors, and refined based on real-world use, rather than released in large, infrequent versions.

1.4. Overview of Modern CSS Development

The CSS used to build professional websites today looks considerably different from the CSS of a decade ago, even though the fundamentals covered in this book, selectors, the box model, the cascade, remain unchanged. This section previews the ideas that the rest of the book builds toward.

From Hacks to Purpose-Built Layout Tools

For much of the language's history, CSS had no dedicated tool for two-dimensional or evenly distributed layout. Developers achieved multi-column designs with float and clear, workarounds originally intended for wrapping text around images, or resorted to laying out entire pages with HTML tables. Flexbox (Chapter 7) and CSS Grid (Chapter 8) replaced these workarounds with layout systems designed for the job from the start, and are now the default choice for arranging content on a modern page.

Responsive, Mobile-First Design

With the majority of web traffic now coming from phones and tablets, a layout that only works at one screen size is no longer acceptable. Media queries, flexible units, and mobile-first design, starting with a layout for the smallest screen and progressively enhancing it for larger ones, are covered in Chapter 9 and are treated as a default expectation throughout the rest of this book, not an advanced afterthought.

Custom Properties and Computed Values

CSS custom properties, commonly called CSS variables, let a value such as a brand color or a spacing unit be defined once and reused throughout a stylesheet, updated in one place when it changes. Paired with functions such as `calc()`, `clamp()`, `min()`, and `max()`, covered in Chapter 11, modern CSS can perform layout math that once required a preprocessor.

Scope of This Book

This volume focuses exclusively on plain, native CSS: the properties, selectors, and layout systems that ship in every modern browser without any build step or external tool. It does not cover CSS preprocessors such as Sass, SCSS, or LESS, nor CSS frameworks such as Bootstrap or Tailwind CSS. Those tools are built on top of the fundamentals taught here, and are the subject of a separate book later in this series; a solid grasp of plain CSS makes learning any of them considerably easier.

Conclusion

This chapter introduced CSS as the language that governs presentation on the web, distinct from, but always working alongside, the structural markup covered in Part 1. It traced CSS from its 1996 origins through the modular CSS3 era to the actively evolving language used today, and previewed the modern layout and styling tools explored throughout the rest of this book. Chapter 2 puts this into practice, showing exactly how to attach CSS to an HTML document and how a CSS rule is written.

Chapter 2. Getting Started with CSS

2.1. Ways to Add CSS: Inline, Internal, and External Styles

CSS can be attached to an HTML document in three ways: directly on an individual element with the style attribute, inside a `<style>` block in the document's `<head>`, or in a separate `.css` file linked with `<link>`. All three use the exact same CSS syntax; they differ only in where the rules live and how broadly they apply.

Inline Styles

An inline style is written directly on an element using the style attribute, and applies only to that one element.

```
1 <p style="color: crimson; font-weight: bold;">
2   This paragraph is styled inline.
3 </p>
```

Inline styles are convenient for a one-off test or a value generated dynamically by a script, but they do not scale: styling ten paragraphs the same way means repeating the same declarations ten times, and inline styles cannot be reused, cached separately by the browser, or easily overridden. As a general rule, inline styles are best avoided in favor of the other two approaches.

Internal Styles

An internal, or embedded, stylesheet is written inside a `<style>` element placed in the document's `<head>`. It applies to the entire document in which it appears.

```

1 <head>
2   <meta charset="UTF-8">
3   <title>My Page</title>
4   <style>
5     p {
6       color: crimson;
7       font-weight: bold;
8     }
9   </style>
10 </head>

```

Internal stylesheets keep all of a page's markup and styling in a single file, which can be useful for a quick prototype, a single standalone page, or an email template where external files are impractical. Their main drawback is that the styles cannot be shared with other pages; a second page needs its own copy of the same `<style>` block.

External Stylesheets

An external stylesheet lives in its own .css file and is attached to an HTML document with a `<link>` element in the `<head>`.

```

1 <link rel="stylesheet" href="styles.css">

```

with the corresponding file, `styles.css`, containing nothing but CSS rules:

```

1 p {
2   color: crimson;
3   font-weight: bold;
4 }

```

External stylesheets are, in almost every real project, the preferred approach, for several reasons:

- A single CSS file can style every page on a site, keeping the design consistent and easy to update from one place;

- Browsers cache external stylesheets separately from the HTML, so returning visitors do not re-download the CSS on every page;
- Keeping markup and styling in separate files makes both easier to read and maintain.

Comparing the Three Approaches

Inline, internal, and external styles are not mutually exclusive, a single page can use all three at once, but each is suited to a different situation:

- **Inline:** a single element, a one-off override, or a value set by JavaScript at runtime.
- **Internal:** styles specific to a single, standalone page.
- **External:** the default choice for any multi-page site or real project.

Unless a chapter says otherwise, every example in the rest of this book assumes an external stylesheet linked from the document's `<head>`.

2.2. CSS Syntax: Rules, Selectors, and Declarations

Regardless of where it lives, every piece of CSS is built from the same small set of building blocks. Learning to read this structure fluently makes every later chapter, which is really just an exploration of new selectors and new properties, far easier to follow.

The Anatomy of a CSS Rule

A CSS rule, also called a rule set, pairs a selector with a declaration block:

```
1 selector {  
2   property: value;  
3   property: value;  
4 }
```

The selector determines which HTML elements the rule applies to. The declaration block, wrapped in curly braces, contains one or more declarations, each pairing a property with a value and ending in a semicolon.

```
1 p {  
2   color: navy;  
3   font-size: 18px;  
4   line-height: 1.5;  
5 }
```

This rule selects every <p> element in the document and applies three declarations to each one: a text color, a font size, and a line height.

Declarations, Properties, and Values

A declaration is a single property-value pair, such as `color: navy;`. The property (`color`) names the aspect of the element being styled; the value (`navy`) specifies the setting for that property. Every declaration inside a declaration block must end with a semicolon; while CSS does not strictly require one after the very last declaration, including it consistently avoids errors when a new declaration is added later.

A missing semicolon is one of the most common CSS mistakes, and one of the hardest to spot: the browser silently merges the next line into the broken declaration instead of raising an error.

Whitespace and Formatting

CSS ignores extra whitespace, line breaks, and indentation between tokens, so a rule can technically be written on a single line:

```
1 p { color: navy; font-size: 18px; }
```

In practice, one declaration per line, indented inside the declaration block as shown in every example in this book, is far easier to read and to compare against a version control diff, and is the convention followed throughout the rest of this book.

Comments

CSS comments are written between `/*` and `*/` and can span multiple lines. Unlike HTML comments, CSS has no single-line comment syntax such as `//`.

```
1  /* Section: primary navigation */
2  nav {
3      background-color: #222;
4      /* TODO: revisit this color once the logo is final */
5      color: white;
6  }
```

Grouping Selectors

When several selectors share the exact same declarations, they can be grouped together in a single rule by separating them with commas, avoiding repetition.

```
1  h1,
2  h2,
3  h3 {
4      font-family: Georgia, serif;
5      color: #222;
6  }
```

This single rule is equivalent to writing three separate rules, one for each heading level, each with the same two declarations. Selectors themselves, the wide variety of ways an element or group of elements can be targeted, are covered in full in Chapter 3.

2.3. Linking Stylesheets and Organizing CSS Files

Real projects rarely consist of a single HTML file and a single CSS file sitting side by side. This section looks more closely at how the `<link>` element locates a stylesheet, and at how larger projects organize their CSS across multiple files.

The `<link>` Element in Detail

The `<link>` element that attaches an external stylesheet is placed in the document's `<head>` and takes two attributes: `rel`, which describes the relationship between the current document and the linked file, and `href`, which gives its location.

```
1 <link rel="stylesheet" href="css/styles.css">
```

A page can link more than one stylesheet, and the order matters: if two linked stylesheets both set a declaration for the same selector, the one linked later in the document wins, following the same cascade rules covered in Chapter 3.

```
1 <link rel="stylesheet" href="css/reset.css">
2 <link rel="stylesheet" href="css/styles.css">
```

Relative and Absolute Paths, Revisited

The href attribute of a <link> element follows the same rules for relative and absolute URLs introduced for the <a> element in Part 1. A path such as css/styles.css is resolved relative to the location of the current HTML file, while a path such as /assets/styles.css, starting with a forward slash, is resolved from the root of the site regardless of which page it appears on.

- href="styles.css", a file in the same folder as the current page;
- href="css/styles.css", a file inside a css subfolder;
- href="../../shared/styles.css", a file one folder up, then into a shared folder;
- href="/assets/styles.css", an absolute path from the site's root, valid from any page.

Organizing CSS Across Multiple Files

A small site can comfortably keep all of its rules in a single styles.css file. As a project grows, many teams split their CSS into several files with clear responsibilities, for example:

- reset.css or base.css, resets or normalizes browser default styling and sets base typography;
- layout.css, page-level structure: header, footer, main content regions;
- components.css, reusable pieces such as buttons, cards, and form controls;
- A page-specific file, such as home.css, for styles used on only one page.

Each file is linked with its own `<link>` element, in the order the styles should be applied:

```
1 <link rel="stylesheet" href="css/reset.css">
2 <link rel="stylesheet" href="css/layout.css">
3 <link rel="stylesheet" href="css/components.css">
4 <link rel="stylesheet" href="css/home.css">
```

The @import Rule

CSS also provides an `@import` rule, which loads one stylesheet from inside another rather than from the HTML document:

```
1 @import url("reset.css");
2
3 body {
4     font-family: sans-serif;
5 }
```

`@import` is convenient but has a real performance cost: an imported stylesheet cannot begin downloading until the file that imports it has already been fetched and parsed, creating a chain of sequential requests instead of the parallel downloads that multiple `<link>` elements allow. For this reason, linking each stylesheet directly from the HTML document, as shown above, is generally preferred over `@import`.

2.4. Using Browser Developer Tools to Inspect Styles

Every modern browser ships with a set of developer tools built in, and no CSS workflow is complete without them. They let you see, and temporarily change, exactly which styles are applied to any element on a page, which is often faster than editing a file, saving it, and reloading the browser to test a single value.

Opening Developer Tools

Developer tools can be opened in any major browser by pressing F12, or by right-clicking any element on the page and choosing "Inspect" from the context menu. The second method is usually more useful while learning CSS, since it opens the tools with that exact element already selected.

The Elements (or Inspector) Panel

The Elements panel, called Inspector in some browsers, shows the live DOM of the page as a navigable tree, matching the HTML source but reflecting any changes made by CSS or JavaScript after the page loaded. Clicking an element in this tree highlights it directly on the page, and clicking an element on the page highlights the corresponding line in the tree.

The Styles Pane

Alongside the element tree, the Styles pane lists every CSS rule that applies to the currently selected element, grouped by the stylesheet and selector each rule comes from, in the order the cascade applies them. Declarations that are overridden by a more specific or later rule are shown with a strikethrough, making it possible to see at a glance not just what a property's final value is, but which rule set it and which other rules lost out.

Every property and value in the Styles pane can be edited directly, and individual declarations can be toggled on or off with a checkbox, without touching the underlying CSS file. This is invaluable for experimenting: try a different color, nudge a margin, or disable a declaration entirely to see what it was actually doing, all without saving a single file.

Changes made in developer tools are temporary and exist only in the browser's current session; reloading the page discards them, so any change worth keeping still has to be copied back into the actual CSS file.

The Box Model Diagram and Computed Styles

Most browsers' developer tools also include a small diagram, usually in a "Computed" or "Layout" tab, showing the selected element's content, padding, border, and margin as nested rectangles with their exact pixel sizes labeled. This box model, introduced in Chapter 4, is one of the most useful diagnostic views in the entire toolset for tracking down unexpected spacing.

A Practical Workflow

A typical debugging session looks like this: right-click the element that looks wrong and choose Inspect, look at the Styles pane to see which rule is currently setting the property in question, and toggle or edit that declaration until the element looks the way it should. Only once the correct value has been found does it get copied into the actual .css file and saved.

- Use the element picker (usually an arrow-and-cursor icon in the top-left of the developer tools) to select an element directly from the page instead of hunting for it in the tree;
- Check the Styles pane for strikethrough declarations before assuming a property "isn't working", it may simply be overridden by a more specific rule;
- Use the responsive or device toolbar to preview the page at different screen widths, a preview of the responsive design techniques covered in Chapter 9.

Conclusion

This chapter covered the three ways CSS can be attached to an HTML document, the anatomy of a CSS rule, and how to organize and link stylesheets in a real project, closing with the browser developer tools that make every technique in the rest of this book faster to learn and easier to debug. With CSS properly connected to a document, Chapter 3 turns to selectors in depth: the many ways a rule can target exactly the elements it is meant to style.

Chapter 3. Selectors, Specificity, and the Cascade

3.1. Basic Selectors: Element, Class, and ID

A selector is the part of a CSS rule that determines which elements the accompanying declarations apply to. CSS offers a large vocabulary of selectors, from the simple to the highly specific; this section covers the three used in nearly every stylesheet.

The Type (Element) Selector

A type selector, also called an element selector, matches every instance of a given HTML tag by writing its name directly.

```
1 p {  
2   line-height: 1.6;  
3 }  
4 blockquote {  
5   font-style: italic;  
6 }  
7
```

This is the broadest kind of selector: it applies to every `<p>` or every `<blockquote>` in the document, regardless of where it appears or what other attributes it has.

The Class Selector

A class selector matches any element carrying a matching class attribute, and is written with a leading period.

```
1 <p class="highlight">This paragraph is highlighted.</p>  
2 <span class="highlight">So is this span.</span>
```

```
1 .highlight {  
2   background-color: yellow;  
3 }
```

Unlike an id, a class can be reused on as many elements as needed, and a single element can carry several classes at once, separated by spaces in the HTML.

```
1 <button class="btn btn-primary">Save</button>
```

This button matches both the `.btn` selector and the `.btn-primary` selector; each contributes its own declarations, a pattern commonly used to combine a generic component style with a specific variant.

The ID Selector

An id selector matches the single element carrying a matching id attribute, and is written with a leading hash symbol.

```
1 <header id="site-header">...</header>
```

```
1 #site-header {  
2   border-bottom: 1px solid #ddd;  
3 }
```

An id must be unique within a document, no two elements should share the same id, which makes id selectors precise but inflexible. As explained in the next section, id selectors also carry far more weight in the cascade than class selectors, which is one reason many style guides recommend styling with classes and reserving ids mainly for JavaScript hooks and in-page anchor links.

The Universal Selector

The universal selector, an asterisk, matches every element in the document. It is most often used in reset stylesheets to zero out default spacing before any other rule applies.

```
1 * {
2   margin: 0;
3   padding: 0;
4   box-sizing: border-box;
5 }
```

3.2. Attribute, Pseudo-Class, and Pseudo-Element Selectors

Beyond matching an element, a class, or an id, CSS can select elements based on their attributes, their state, or even a specific part of their content that has no corresponding HTML tag at all.

Attribute Selectors

An attribute selector matches elements that carry a given attribute, optionally with a specific value, written in square brackets.

```
1  /* Any element with a title attribute */
2  [title] {
3    cursor: help;
4  }
5
6  /* An input whose type is exactly "email" */
7  input[type="email"] {
8    border-color: steelblue;
9  }
10
11 /* An href that starts with "https://" */
12 a[href^="https://"] {
13   padding-right: 18px;
14 }
15
16 /* An href that ends with ".pdf" */
17 a[href$=".pdf"] {
18   color: firebrick;
19 }
```

The ^= and \$= operators match the start and end of an attribute's value; *= matches anywhere within it. These are especially useful for styling links differently based on their destination without adding an extra class to every one.

Pseudo-Class Selectors

A pseudo-class selects an element based on a state or position that isn't captured by the HTML itself, written with a single colon.

- :hover, while the pointer is over the element;
- :focus, while the element has keyboard focus;
- :active, while the element is being clicked or pressed;
- :first-child / :last-child, an element that is the first or last child of its parent;
- :nth-child(n), an element at a specific position among its siblings;
- :not(selector), an element that does not match the given selector;
- :checked, a checkbox or radio input that is currently checked;
- :disabled, a form control that is currently disabled.

```
1 a:hover {  
2   text-decoration: underline;  
3 }  
4 li:nth-child(odd) {  
5   background-color: #f7f7f7;  
6 }  
7  
8 button:not(:disabled):hover {  
9   cursor: pointer;  
10 }  
11
```

The last example combines two pseudo-classes: it applies only to a <button> that is currently hovered and is not disabled, showing that pseudo-classes can be chained directly onto a selector without any separator.

Pseudo-Element Selectors

A pseudo-element targets a specific part of an element's content, or generates content that has no element of its own in the HTML, written with a double colon.

- `::first-line`, the first rendered line of a block of text;
- `::first-letter`, the first letter of a block of text;
- `::before` and `::after`, insert generated content immediately before or after an element's actual content;
- `::placeholder`, the placeholder text of a form input.

```
1 blockquote::before {
2   content: "\201C";
3   font-size: 2em;
4   color: #999;
5 }
6 p::first-line {
7   font-weight: bold;
8 }
9
```

`::before` and `::after` require a content property, even if it is an empty string, or nothing is inserted. They are widely used for purely decorative content, quotation marks, icons, tooltip arrows, that should not exist as real, readable text in the HTML.

3.3. Combinators and Grouping Selectors

Combinators describe a relationship between two selectors, letting a rule target an element based on its position relative to another element, rather than on its own tag, class, or attributes alone.

The Descendant Combinator

A single space between two selectors is the descendant combinator: it matches any element matching the second selector that appears anywhere inside an element matching the first, at any depth of nesting.

```
1 article p {  
2   margin-bottom: 1em;  
3 }
```

This rule matches every `<p>`, however deeply nested, as long as it has an `<article>` ancestor somewhere above it.

The Child Combinator

A greater-than sign is the child combinator: it matches only a direct child of the first selector, not a more distant descendant.

```
1 nav > ul {  
2   display: flex;  
3 }
```

This matches a `<ul>` that is an immediate child of `<nav>`, but not a `<ul>` nested inside a `<div>` that is itself inside `<nav>`.

The Adjacent and General Sibling Combinators

A plus sign is the adjacent sibling combinator, matching an element that immediately follows another at the same level of nesting. A tilde is the general sibling combinator, matching any later sibling, not just the very next one.

```
1 /* A paragraph directly after an h2 */  
2 h2 + p {  
3   font-size: 1.1em;  
4 }  
5 /* Every paragraph that follows an h2, anywhere after it */  
6 h2 ~ p {  
7   color: #444;  
8 }
```

Grouping Selectors with Commas

As introduced in Chapter 2, a comma groups multiple selectors that should receive the exact same declarations, avoiding repetition. Combinators and grouping can be combined freely.

```
1 h1, h2, h3,  
2 .card-title,  
3 nav a:hover {  
4     color: darkslateblue;  
5 }
```

Each comma-separated entry in this list is evaluated completely independently; there is no relationship between `h1` and `.card-title` beyond sharing the same declaration block.

3.4. Understanding Specificity, the Cascade, and Inheritance

When more than one rule targets the same element and sets the same property, CSS needs a way to decide which rule wins. That decision is governed by three interacting mechanisms: specificity, the cascade, and inheritance.

Specificity

Specificity is a score assigned to every selector, based on the kinds of selectors it contains. When two rules conflict, the one with the higher specificity wins, regardless of which rule appears first in the stylesheet. Specificity is commonly expressed as three numbers:

Selector Type	Specificity Weight	Examples
ID selectors	1, 0, 0	<code>#site-header</code>
Class, attribute, and pseudo-class selectors	0, 1, 0	<code>.btn</code> , <code>[type="text"]</code> , <code>:hover</code>
Type selectors and pseudo-elements	0, 0, 1	<code>p</code> , <code>nav</code> , <code>::before</code>

The universal selector *	0, 0, 0	(adds no specificity)
--------------------------	---------	-----------------------

A selector's total specificity is found by counting how many of each type it contains. For example, `nav.primary a:hover` contains one class (`.primary`), one pseudo-class (`:hover`), and two type selectors (`nav` and `a`), giving it a specificity of 0-2-2.

```

1  /* Specificity 0-0-1 */
2  p { color: black; }
3
4  /* Specificity 0-1-0 (wins: higher specificity) */
5  .warning { color: red; }
6
7  <p class="warning">This text is red.</p>
```

An inline style attribute has a specificity higher than any selector, and !important, covered below, overrides specificity entirely, both should be used sparingly, since they make later overrides considerably harder to reason about.

The Cascade

When two rules have equal specificity, the cascade falls back to source order: the rule that appears later, whether later in the same file or in a stylesheet linked after an earlier one, wins.

```

1  p {
2    color: navy;
3  }
4
5  p {
6    color: darkgreen; /* wins: same specificity, appears later */
7  }
```

The cascade also weighs where a rule comes from: author styles (the site's own CSS) normally override the browser's default user-agent styles, and a declaration marked

!important is moved to the top of this ordering, overriding normal author-style rules regardless of specificity.

```
1 p {  
2   color: navy !important;  
3 }  
4 .warning {  
5   color: red; /* loses to the !important rule above, even though .warning has higher specificity */  
6 }  
7
```

Because !important short-circuits the normal rules, overriding it later requires either another !important declaration with equal or higher specificity, or removing it altogether, for this reason it is best treated as a last resort rather than a routine tool.

Inheritance

Some CSS properties are inherited by default: if not explicitly set on an element, its value is taken from its parent. Typography-related properties, color, font-family, font-size, line-height, are inherited, which is why setting a font once on <body> typically applies it to the entire page.

Box-model and layout properties, margin, padding, border, width, display, are not inherited, since inheriting them would rarely make sense: a child element automatically taking on its parent's exact margin or border would usually produce broken layouts rather than useful ones.

```
1 body {  
2   font-family: system-ui, sans-serif;  
3   color: #222;  
4 }  
5 /* No font-family or color rule needed here, both are inherited from body */  
6 p {  
7   margin-bottom: 1em;  
8 }  
9
```

Any property can be forced to inherit explicitly with the keyword value inherit, and the keyword value initial resets a property to its specification-defined default, ignoring both inheritance and the cascade.

Conclusion

This chapter covered the full range of CSS selectors, from simple element, class, and id selectors through attribute, pseudo-class, and pseudo-element selectors, and the combinators that relate one selector to another. It closed with the rules that resolve conflicts between them: specificity, the cascade, and inheritance. With selectors covered in depth, Chapter 4 turns to the box model, the system that governs the size and spacing of every element these selectors can target.

Chapter 4. The CSS Box Model

4.1. Content, Padding, Border, and Margin

Every element on a web page is, from CSS's point of view, a rectangular box. The CSS box model describes how the size of that box is calculated, from four concentric layers: content, padding, border, and margin.

The Four Layers

- Content, the actual text, image, or other content of the element, sized by properties such as width and height;
- Padding, transparent space between the content and the border, inside the element's background;
- Border, a line drawn around the padding, which can be styled with width, style, and color;
- Margin, transparent space outside the border, separating the element from its neighbors, and never filled by the element's own background.

Each of these can be set individually per side (top, right, bottom, left) or with shorthand properties that set all four sides, or opposite pairs, at once.

```
1 .card {  
2   padding: 20px;  
3   border: 1px solid #ddd;  
4   margin: 16px;  
5 }
```

Per-Side and Shorthand Syntax

Padding and margin accept one, two, three, or four values, each interpreted differently.

```

1  /* One value: all four sides */
2  p { margin: 10px; }
3
4  /* Two values: top/bottom, then left/right */
5  p { margin: 10px 20px; }
6
7  /* Three values: top, left/right, bottom */
8  p { margin: 10px 20px 15px; }
9
10 /* Four values: top, right, bottom, left (clockwise) */
11 p { margin: 10px 20px 15px 5px; }

```

Each side can also be set individually with its own property, such as `margin-top`, `padding-left`, or `border-bottom`, which is often clearer than shorthand when only one side needs a specific value.

Border Properties

The border shorthand sets width, style, and color in a single declaration, in any order.

```

1  .card {
2    border: 2px dashed steelblue;
3  }

```

`border-style` accepts values such as `solid`, `dashed`, `dotted`, and `double`; without an explicit style, a border does not render at all, even if a width and color are set, style is the one sub-property with no default other than `none`.

Margin Collapsing

Vertical margins between two block-level elements do not always add together. When the bottom margin of one element and the top margin of the next both apply, they collapse into a single margin equal to the larger of the two, not their sum, a behavior unique to vertical margins and one of the more surprising parts of the box model for newcomers.

```
1 p {
2   margin-top: 20px;
3   margin-bottom: 30px;
4 }
5 /* Two stacked <p> elements end up 30px apart, not 50px */
```

4.2. The box-sizing Property

By default, the width and height properties in CSS set the size of an element's content box only, padding and border are added on top, making the element's true rendered size larger than the width that was set. The box-sizing property controls whether this is true.

content-box: The Default Behavior

```
1 .box {
2   box-sizing: content-box; /* default */
3   width: 300px;
4   padding: 20px;
5   border: 5px solid black;
6 }
7 /* Rendered width: 300 + 20 + 20 + 5 + 5 = 350px */
```

With the default content-box model, adding padding or a border to an element makes it visibly wider or taller than the width or height that was declared, which can make layouts built from several boxes of a supposedly equal width drift out of alignment as soon as one of them gets padding the others don't have.

border-box: The Modern Default

```
1 .box {  
2   box-sizing: border-box;  
3   width: 300px;  
4   padding: 20px;  
5   border: 5px solid black;  
6 }  
7 /* Rendered width: exactly 300px; padding and border eat into the content area instead of adding to it */
```

With border-box, the width and height properties set the total size of the element, padding and border included; the content area simply shrinks to make room. This matches how most designers and developers intuitively expect sizing to work, which is why it has become the de facto standard.

Applying border-box Globally

Because content-box is rarely the desired behavior, most stylesheets today open with a rule that switches every element to border-box in one place.

```
1 *, *::before, *::after {  
2   box-sizing: border-box;  
3 }
```

Setting box-sizing: border-box globally, near the very top of a stylesheet, is standard practice in essentially every modern CSS codebase and is assumed for every layout example later in this book.

4.3. Display Types: Block, Inline, and Inline-Block

The display property determines how an element participates in the page's layout: whether it starts on its own line, how it sizes itself, and whether width and height even apply to it. Every element has a default display value defined by the browser's user-agent stylesheet, which CSS can override.

Block-Level Elements

A block-level element always starts on a new line and stretches to fill the full width of its container by default. width, height, margin, and padding all apply to it exactly as described

in the box model. `<div>`, `<p>`, `<h1>` through `<h6>`, `<ul>`, and `<section>` are all block-level by default.

```
1 div {  
2   display: block; /* already the default for <div> */  
3 }
```

Inline Elements

An inline element flows within a line of text rather than starting a new one, and only takes up as much horizontal space as its content requires. Critically, width and height have no effect on an inline element, and only left and right margin and padding are respected, top and bottom padding may visually overlap surrounding lines without pushing them away. `<span>`, `<a>`, `<strong>`, and `<em>` are inline by default.

```
1 span {  
2   display: inline; /* already the default for <span> */  
3   width: 200px; /* has no effect */  
4 }
```

Inline-Block Elements

inline-block combines behavior from both: the element flows inline with surrounding content, sitting on the same line as neighboring elements, but width, height, and all four sides of margin and padding are fully respected, exactly as they would be on a block-level element.

```
1 .badge {  
2   display: inline-block;  
3   width: 80px;  
4   padding: 4px 8px;  
5   text-align: center;  
6 }
```

This makes inline-block a common choice for small components, badges, buttons, tags, that need a fixed size but should still flow alongside other inline content, such as within a line of text.

display: none

A fourth commonly used value, none, removes the element from the page entirely: it is not rendered, takes up no space, and is skipped by screen readers, unlike the visibility property covered next.

```
1 .tooltip {  
2   display: none;  
3 }  
4 .tooltip.is-visible {  
5   display: block;  
6 }
```

Chapters 7 and 8 introduce two further display values, flex and grid, which turn an element into a container that arranges its children using the Flexbox and Grid layout systems rather than the normal block/inline flow described here.

4.4. Managing Overflow and Visibility

Content does not always fit neatly inside the box that contains it. The overflow property controls what happens when it doesn't, and visibility offers a second way to hide an element without removing it from the page's layout.

The overflow Property

When an element's content is larger than its box, often because a fixed height was set, or a long unbroken string doesn't wrap, overflow decides how the excess is handled.

- visible (the default), overflowing content spills outside the box, remaining fully visible;
- hidden, overflowing content is clipped and simply not shown;
- scroll, the box always shows scrollbars, letting the user reveal the hidden content;

- auto, the browser adds scrollbars only if the content actually overflows.

```
1 .log-panel {  
2   height: 200px;  
3   overflow-y: auto;  
4 }
```

overflow-x and overflow-y control horizontal and vertical overflow independently; the overflow shorthand sets both at once. auto is the most commonly used value in practice, since it avoids showing an empty scrollbar on a box that doesn't actually need one.

Truncating Text with Ellipsis

A common pattern combines overflow: hidden with two other properties to truncate a single line of text with an ellipsis (...) instead of letting it overflow or wrap.

```
1 .card-title {  
2   white-space: nowrap;  
3   overflow: hidden;  
4   text-overflow: ellipsis;  
5 }
```

All three declarations are required together: white-space: nowrap keeps the text on one line, overflow: hidden clips what doesn't fit, and text-overflow: ellipsis replaces the clipped end with

visibility vs. display: none

visibility: hidden hides an element visually, but unlike display: none, the element still occupies its normal space in the layout, surrounding content does not shift to fill the gap. It is also still present for some assistive technologies, unlike display: none, though behavior here varies and neither property should be relied on as an accessibility technique on its own.

```
1 .placeholder-icon {  
2   visibility: hidden; /* reserves the space, hides the icon */  
3 }
```

A child of a `visibility: hidden` element can restore its own visibility with `visibility: visible`, which is not possible with `display: none`, once a parent is removed from layout, nothing inside it can opt back in.

Conclusion

This chapter covered the CSS box model in full: the content, padding, border, and margin layers that determine an element's size and spacing; the box-sizing property that decides whether padding and border add to or eat into a declared width; the display values that govern how an element participates in the page's flow; and the overflow and visibility properties that handle content too large for its box. Chapter 5 moves from structure to appearance, covering color, backgrounds, and borders in depth.

Chapter 5. Colors, Backgrounds, and Borders

5.1. Color Formats: Named Colors, HEX, RGB, and HSL

Almost every visual property in CSS, text color, backgrounds, borders, shadows, accepts a color value, and CSS offers several different notations for specifying one. Each has its own strengths, and modern stylesheets often mix more than one.

Named Colors

CSS defines a fixed list of 147 named colors, from common ones like red, white, and black to more specific names like cornflowerblue and rebeccapurple.

```
1 h1 {  
2   color: darkslateblue;  
3 }
```

Named colors are readable and convenient for quick examples and prototypes, but the fixed list cannot express every possible color, and offers no way to control opacity, for either reason, most production stylesheets rely on one of the numeric formats below instead.

Hexadecimal (HEX) Colors

A hex color specifies red, green, and blue as two-digit hexadecimal numbers, prefixed with a hash symbol, giving access to over sixteen million colors.

```
1 .card {  
2   background-color: #3d8bff;  
3   border-color: #1f3e73;  
4 }
```

A three-digit shorthand is also valid when each pair of digits repeats: #fff is shorthand for #ffffff (white), and #333 is shorthand for #333333 (a dark gray). An optional fourth pair of digits (or fourth shorthand digit) sets alpha transparency, from 00 (fully transparent) to ff (fully opaque).

```
1 .overlay {  
2   background-color: #00000080; /* black at 50% opacity */  
3 }
```

RGB and RGBA

The `rgb()` function specifies the same red, green, and blue channels as decimal numbers from 0 to 255, which many people find more intuitive to reason about or adjust than hexadecimal.

```
1 .card {  
2   background-color: rgb(61, 139, 255);  
3 }  
4 .overlay {  
5   background-color: rgb(0 0 0 / 50%);  
6 }  
7
```

Modern CSS allows a fourth, optional alpha value, either as a percentage or a decimal between 0 and 1, separated from the color channels by a forward slash (the once-required `rgba()` function and comma-separated syntax still work identically, for compatibility with older code).

HSL and HSLA

`hsl()` specifies a color by hue, saturation, and lightness rather than by color channel, which often maps more directly onto how people actually think about color: hue is a position on the color wheel from 0 to 360 degrees, saturation is a percentage from gray to fully vivid, and lightness is a percentage from black to white.

```
1 .card {  
2   background-color: hsl(215, 100%, 62%);  
3 }
```

HSL makes it straightforward to build a set of related colors by changing just one component, for example, keeping the same hue and saturation while producing five progressively lighter shades by adjusting only the lightness percentage, a common technique for building a color palette by hand.

5.2. Background Images, Position, Size, and Repeat

Beyond a solid `background-color`, CSS can place an image behind an element's content, with fine control over how that image is positioned, sized, and repeated.

`background-image`

```
1 .hero {  
2   background-image: url("hero-banner.jpg");  
3 }
```

The `url()` function accepts a relative or absolute path, following the exact same rules covered for linking stylesheets in Chapter 2. A `background-image` never affects the size of its element the way an `<img>` element does; the element's size still comes from its content, its width and height, or its layout context.

`background-repeat`

By default, a background image tiles both horizontally and vertically to fill its element. `background-repeat` controls this.

```
1 .pattern {
2   background-image: url("dots.png");
3   background-repeat: repeat-x; /* tile horizontally only */
4 }
5
6 .hero {
7   background-image: url("hero-banner.jpg");
8   background-repeat: no-repeat;
9 }
```

background-position

background-position sets where a non-repeating (or partially repeating) image starts, using keywords (top, bottom, left, right, center) or length and percentage values for horizontal and vertical position.

```
1 .hero {
2   background-image: url("hero-banner.jpg");
3   background-repeat: no-repeat;
4   background-position: center top;
5 }
```

background-size

background-size controls the rendered size of the image, and accepts explicit lengths or two especially useful keywords:

- cover, scales the image to completely fill the element, cropping whatever overflows, with no empty space;
- contain, scales the image to fit entirely within the element, without cropping, which may leave empty space on two sides.

```
1 .hero {  
2   background-image: url("hero-banner.jpg");  
3   background-repeat: no-repeat;  
4   background-position: center;  
5   background-size: cover;  
6 }
```

`background-size: cover` is the standard choice for full-width hero banners and photographic backgrounds, since it guarantees the image fills its container at every screen size without distortion.

The background Shorthand

All of the properties above, along with `background-color`, can be combined into a single background shorthand declaration.

```
1 .hero {  
2   background: #222 url("hero-banner.jpg") no-repeat center / cover;  
3 }
```

Notice the forward slash between the position (`center`) and the size (`cover`), this is the syntax the shorthand requires whenever both are specified together.

5.3. Gradients: Linear, Radial, and Conic

A gradient is a smooth transition between two or more colors, generated entirely by CSS with no image file required. Gradients are used anywhere `background-image` is accepted, and can be combined with an actual image as well.

Linear Gradients

`linear-gradient()` transitions between colors along a straight line, in a direction set by the first argument.

```

1 .banner {
2   background: linear-gradient(to right, #3d8bff, #1f3e73);
3 }
4 .banner-diagonal {
5   background: linear-gradient(135deg, #3d8bff, #1f3e73);
6 }
7

```

The direction can be a keyword (to right, to bottom left) or an angle in degrees, measured clockwise from the top. Any number of color stops can be listed, optionally with a position for each, to control exactly where each color begins.

```

1 .rainbow-bar {
2   background: linear-gradient(
3     to right,
4     red 0%,
5     yellow 25%,
6     green 50%,
7     blue 75%,
8     violet 100%
9   );
10 }

```

Radial Gradients

radial-gradient() transitions between colors outward from a central point, by default forming an ellipse that stretches to fit the element.

```

1 .spotlight {
2   background: radial-gradient(circle, #ffffff, #1f3e73);
3 }

```

Adding the `circle` keyword, as above, forces a perfectly round gradient regardless of the element's aspect ratio; without it, the gradient stretches into an ellipse to match the element's width and height.

Conic Gradients

`conic-gradient()` transitions between colors around a center point, like the hands of a clock sweeping around, rather than outward from it. It is commonly used for color wheels and pie-chart-like visuals.

```
1 .color-wheel {  
2   background: conic-gradient(  
3     red, yellow, lime, aqua, blue, magenta, red  
4   );  
5   border-radius: 50%;  
6 }
```

Gradients as Overlays

Because a gradient is just another `background-image` value, it can be layered on top of an actual photo by listing both, separated by a comma, the first layer listed renders on top.

```
1 .hero {  
2   background:  
3     linear-gradient(rgb(0 0 0 / 40%), rgb(0 0 0 / 40%)),  
4     url("hero-banner.jpg") center / cover no-repeat;  
5 }
```

This is a common technique for darkening a photo enough that white text placed over it stays readable, without editing the image file itself.

5.4. Borders, Border-Radius, and Box Shadows

Chapter 4 introduced the border property as part of the box model. This section covers two properties closely related to borders that shape an element's visual edge: rounded corners and drop shadows.

border-radius

border-radius rounds an element's corners, and accepts the same one-to-four-value shorthand syntax as margin and padding, applied to each corner instead of each side.

```
1 .card {
2   border: 1px solid #ddd;
3   border-radius: 8px;
4 }
5
6 .avatar {
7   width: 64px;
8   height: 64px;
9   border-radius: 50%; /* a perfect circle on a square box */
10 }
```

Setting border-radius to 50% on a perfectly square element is the standard technique for a circular avatar or icon; on a non-square element, the same value produces an ellipse rather than a circle.

Individual corners can be targeted with their own longhand properties when only some of the four need rounding.

```
1 .dropdown-panel {
2   border-bottom-left-radius: 8px;
3   border-bottom-right-radius: 8px;
4 }
```

box-shadow

`box-shadow` adds one or more drop shadows around an element's box, specified as a horizontal offset, a vertical offset, an optional blur radius, an optional spread radius, and a color, in that order.

```
1 .card {  
2   box-shadow: 0 4px 12px rgb(0 0 0 / 15%);  
3 }
```

In this example, the shadow has no horizontal offset, sits 4px below the element, blurs over a 12px radius, and is a semi-transparent black. Larger blur values produce a softer, more diffuse shadow; a larger blur combined with a low opacity is typical for a subtle "elevated card" effect.

Multiple and Inset Shadows

Several shadows can be applied to the same element by separating them with commas, and the keyword `inset` turns a shadow inward, appearing to carve into the element rather than lift it off the page.

```
1 .button {  
2   box-shadow:  
3     0 1px 2px rgb(0 0 0 / 20%),  
4     0 0 0 3px rgb(61 139 255 / 30%); /* a focus ring */  
5 }  
6 .input-pressed {  
7   box-shadow: inset 0 2px 4px rgb(0 0 0 / 25%);  
8 }  
9
```

Layering a thin, low-opacity shadow with a wider, more transparent one, as in the button example, is a common way to build a focus ring or an outer glow without reaching for an actual outline or border.

Conclusion

This chapter covered the four ways CSS represents color, the properties that control background images (position, repeat, and size), the three types of gradients CSS can generate without any image file, and the border-radius and box-shadow properties that shape and lift an element's edges. Chapter 6 turns from an element's box to the text inside it, covering typography in depth.

Chapter 6. Typography and Text Styling

6.1. Font Properties: Family, Size, Weight, and Style

Typography is one of the areas where CSS has the most direct impact on readability. This section covers the core properties that choose a typeface and control its basic appearance.

font-family and Font Stacks

font-family specifies which typeface an element should use, as a comma-separated list of fallbacks tried in order, ending in a generic family the browser is guaranteed to support.

```
1 body {  
2   font-family: "Helvetica Neue", Arial, sans-serif;  
3 }
```

The browser uses the first font in the list that is actually installed or available, falling back to the next. A font name containing a space, such as "Helvetica Neue", must be quoted; single-word names like Arial do not require quotes, though quoting consistently is a common style choice. The final generic keyword, serif, sans-serif, monospace, or cursive, is not optional in a well-formed font stack, since it is the only guaranteed fallback if every named font is unavailable.

font-size

font-size accepts an absolute unit such as px, a relative unit such as em or rem, or a percentage. Relative units, covered fully in Chapter 9, are generally preferred for accessibility, since they scale correctly when a reader increases their browser's default font size.

```

1  html {
2    font-size: 100%; /* respects the user's browser default */
3  }
4  h1 {
5    font-size: 2.5rem;
6  }
7
8  p {
9    font-size: 1rem;
10 }
11

```

font-weight

font-weight controls how bold text appears, either as a keyword (normal, bold) or a numeric value from 100 to 900 in steps of 100, which allows finer control when a typeface provides several weights.

```

1  p {
2    font-weight: 400; /* equivalent to normal */
3  }
4  strong {
5    font-weight: 700; /* equivalent to bold */
6  }
7
8  .subtitle {
9    font-weight: 300; /* a lighter weight, if the font supports it */
10 }
11

```

A numeric weight only has a visible effect if the loaded font actually provides that weight; if it doesn't, the browser substitutes the closest weight it does have, or synthesizes a bold or light appearance that rarely looks as good as a true matching weight.

font-style

font-style switches between normal and italic (or, rarely, oblique, a slanted variant used when a typeface has no true italic design of its own).

```
1 em {  
2   font-style: italic; /* already the default for <em> */  
3 }
```

The font Shorthand

font combines style, weight, size, line-height, and family into a single declaration, though its strict argument order makes it less commonly used than the individual longhand properties shown above.

```
1 p {  
2   font: italic 400 16px/1.5 Georgia, serif;  
3 }
```

6.2. Web Fonts and the @font-face Rule

font-family only works if the named font is actually available on the reader's device. Web fonts solve this by letting a font file be downloaded and used directly by the browser, guaranteeing the exact same typeface for every visitor.

Self-Hosted Fonts with @font-face

The @font-face rule defines a custom font by pointing to one or more font files, giving it a name that can then be used anywhere in font-family, exactly like a system font.

```

1 @font-face {
2   font-family: "Inter";
3   src: url("fonts/inter-regular.woff2") format("woff2");
4   font-weight: 400;
5   font-style: normal;
6 }
7 @font-face {
8   font-family: "Inter";
9   src: url("fonts/inter-bold.woff2") format("woff2");
10  font-weight: 700;
11  font-style: normal;
12 }
13 body {
14   font-family: "Inter", sans-serif;
15 }
16
17

```

Each weight and style of a font family is typically its own separate file and its own `@font-face` rule, as shown above for the regular and bold weights of the same typeface; the `font-weight` and `font-style` declared inside `@font-face` tell the browser which file to load when `body { font-weight: 700; }` is requested elsewhere in the stylesheet.

WOFF2 is the modern, widely supported, and most compressed web font format, and is generally the only format needed today; older formats such as WOFF, TTF, and EOT exist mainly for legacy browser support.

Third-Party Font Services

Services such as Google Fonts host font files and generate the necessary `@font-face` rules automatically, provided either as a `<link>` to a hosted stylesheet or as an `@import` rule.

```

1 <link rel="preconnect" href="https://fonts.googleapis.com">
2 <link rel="stylesheet"
3   href="https://fonts.googleapis.com/css2?family=Inter:wght@400;700&display=swap">

```

This is often faster to set up than self-hosting, at the cost of a small external request and, depending on the service and jurisdiction, a data-privacy consideration worth checking before use on a production site.

font-display and Loading Behavior

Because a web font must be downloaded before it can be used, the browser needs a strategy for what to show while it loads. The `font-display` property, set inside `@font-face`, controls this.

```
1 @font-face {  
2   font-family: "Inter";  
3   src: url("fonts/inter-regular.woff2") format("woff2");  
4   font-display: swap;  
5 }
```

`font-display: swap` tells the browser to render text immediately in a fallback font, then swap to the custom font once it finishes downloading, avoiding invisible text while the page loads. It is the most commonly recommended value for body text.

6.3. Text Alignment, Decoration, and Transformation

Beyond choosing a typeface, CSS controls how text is aligned within its container, whether it carries decorations such as underlines, and whether its letter case is transformed for display.

text-align

`text-align` sets the horizontal alignment of text within its containing block.

```
1 h1 {
2   text-align: center;
3 }
4 blockquote {
5   text-align: right;
6 }
7
8 .justified {
9   text-align: justify;
10 }
11
```

text-align: justify stretches each line to fill the full width of its container by adjusting the spacing between words, which can produce uneven, hard-to-read gaps, especially in narrow columns, it is generally used sparingly in web typography compared to print.

text-decoration

text-decoration adds or removes lines associated with text, most commonly used to remove the default underline from links or to strike through outdated content.

```
1 a {
2   text-decoration: none;
3 }
4 .original-price {
5   text-decoration: line-through;
6 }
7
```

The full text-decoration shorthand also accepts a line style and color, useful for a custom underline that doesn't match the text color exactly.

```
1 a.styled-link {  
2   text-decoration: underline wavy #3d8bff;  
3 }
```

Removing the underline from a link with `text-decoration: none` makes it harder for readers to identify as clickable; when doing so, make sure some other visual signal, a distinct color, a hover state, still marks the text as a link.

text-transform

`text-transform` changes the capitalization of text as it is displayed, without altering the actual underlying text in the HTML.

```
1 .section-label {  
2   text-transform: uppercase;  
3   letter-spacing: 2px;  
4 }  
5 .book-title {  
6   text-transform: capitalize;  
7 }  
8
```

- `uppercase`, ALL CAPS;
- `lowercase`, all lowercase;
- `capitalize`, Capitalizes The First Letter Of Each Word;
- `none`, no transformation, the default.

Because `text-transform` only changes how text is displayed, the underlying HTML content can still be written in normal sentence case, which keeps it readable in the page source and correctly capitalized for screen readers and search engines.

6.4. Line Height, Letter Spacing, and Readability

A few final typography properties have an disproportionately large effect on how comfortable a block of text is to read, even though each one only adjusts a small amount of spacing.

line-height

line-height sets the vertical space allocated to each line of text, and is most often given as a unitless number, interpreted as a multiple of the element's font size.

```
1 p {  
2   font-size: 16px;  
3   line-height: 1.5; /* 24px between baselines */  
4 }
```

A unitless line-height is strongly preferred over a fixed pixel or em value, because it is inherited as a ratio rather than a computed pixel value; a fixed value inherited by a child element with a different font size produces cramped or excessively loose lines instead of scaling correctly with it.

A line-height between about 1.4 and 1.6 is a common starting point for body text; headings are often set tighter, closer to 1.1 to 1.3, since large text needs proportionally less vertical breathing room between lines to stay readable.

letter-spacing and word-spacing

letter-spacing adjusts the horizontal space between characters, and word-spacing adjusts the space between words. Both accept a length value, including negative values to tighten spacing.

```
1 .eyebrow {  
2   text-transform: uppercase;  
3   letter-spacing: 0.1em;  
4   font-size: 0.85rem;  
5 }  
6 h1 {  
7   letter-spacing: -0.02em; /* slightly tighter at large sizes */  
8 }  
9
```

A small positive letter-spacing, as shown for the uppercase `.eyebrow` label, is a common technique for improving the legibility of all-caps text, which otherwise tends to look visually cramped. A small negative letter-spacing is sometimes applied to very large headings for a similar reason in reverse.

Measuring Line Length

Readability research consistently points to an ideal line length of roughly 45 to 75 characters for body text; lines that are too long make it easy to lose your place when your eye returns to the start of the next line. Because CSS has no unit for "number of characters" that adapts across different fonts, the `ch` unit, the width of the "o" character in the current font, is commonly used to approximate it directly on the `max-width` property.

```
1 article p {  
2   max-width: 65ch;  
3 }
```

Bringing It Together

A well-set block of body text typically combines several of the properties from this chapter at once: a readable font stack, a comfortable size, a unitless line-height, and a constrained line length.

```
1 article p {
2   font-family: Georgia, "Times New Roman", serif;
3   font-size: 1.125rem;
4   line-height: 1.6;
5   max-width: 65ch;
6   color: #222;
7 }
```

Conclusion

This chapter covered choosing and sizing a typeface, loading custom web fonts with `@font-face`, aligning and decorating text, and the fine-grained spacing properties, line-height and letter-spacing chief among them, that most directly affect how comfortable a page is to read. With colors, boxes, and typography covered, Chapter 7 turns to layout, starting with Flexbox.

Chapter 7. Layout with Flexbox

7.1. Flex Containers and the Main and Cross Axes

Flexbox, formally the Flexible Box Layout module, is a one-dimensional layout system designed for arranging items in a row or a column, distributing space between them and aligning them within their container. It replaced older techniques based on float for most everyday layout tasks.

Creating a Flex Container

Flexbox is activated by setting `display: flex` on a container element. Every direct child of that container automatically becomes a flex item, without needing any property set on the children themselves.

```
1 .toolbar {  
2   display: flex;  
3 }
```

```
1 <div class="toolbar">  
2   <button>Save</button>  
3   <button>Cancel</button>  
4   <button>Delete</button>  
5 </div>
```

With nothing more than `display: flex`, the three buttons already arrange themselves in a single row, each sized to fit its own content, instead of stacking vertically the way block-level `<button>` elements normally would.

The Main Axis and the Cross Axis

Every flex container has two axes. The main axis runs in the direction items are laid out, horizontally by default, and the cross axis runs perpendicular to it. Nearly every Flexbox property either distributes space along the main axis or aligns items along the cross axis,

so keeping the two straight is the single most useful mental model for the rest of this chapter.

flex-direction

flex-direction sets which way the main axis runs, which in turn determines whether justify-content (covered in 7.3) distributes space horizontally or vertically.

```
1 .toolbar {
2   display: flex;
3   flex-direction: row; /* default: left to right */
4 }
5
6 .sidebar-menu {
7   display: flex;
8   flex-direction: column; /* top to bottom */
9 }
```

row-reverse and column-reverse are also available, reversing the order items are laid out in without changing their order in the HTML source, useful when a visual reordering has no semantic meaning worth reflecting in the markup itself.

flex-wrap

By default, flex items are forced onto a single line, shrinking if necessary to fit. flex-wrap: wrap allows items to flow onto multiple lines instead, once they no longer fit on one.

```
1 .card-grid {
2   display: flex;
3   flex-wrap: wrap;
4   gap: 16px;
5 }
```

The gap property, shown above, sets consistent spacing between flex items, both between items on the same line and between wrapped lines, without needing margin on individual

items, a considerably simpler approach than the margin-based spacing hacks Flexbox layouts once required.

7.2. Flex Item Properties: Grow, Shrink, and Basis

While the properties in 7.1 are set on the flex container, three closely related properties are set on individual flex items, controlling how each one grows or shrinks to fill the available space along the main axis.

flex-basis

flex-basis sets an item's initial size along the main axis, before any growing or shrinking is applied. It behaves like width in a row layout, or height in a column layout, but is specific to Flexbox and takes priority over those properties when both are set.

```
1 .sidebar {  
2   flex-basis: 240px;  
3 }
```

flex-grow

flex-grow controls how an item shares any extra space left over after all items have been laid out at their basis size. It is a unitless number acting as a ratio: an item with flex-grow: 2 grows twice as much as a sibling with flex-grow: 1, in proportion to the leftover space, not to the item's own size.

```
1 .main-content {  
2   flex-grow: 1; /* takes up all remaining space */  
3 }  
4 .sidebar {  
5   flex-grow: 0; /* stays at its flex-basis size */  
6   flex-basis: 240px;  
7 }  
8
```

This pairing, a fixed-width sidebar with `flex-grow: 0`, next to a main content area with `flex-grow: 1`, is one of the most common Flexbox layout patterns: the sidebar stays a constant width, and the main content area automatically fills whatever space remains.

flex-shrink

`flex-shrink` controls how an item shrinks when the container is too small to fit every item at its `flex-basis` size, using the same ratio logic as `flex-grow`. The default value, `1`, allows every item to shrink; setting it to `0` prevents an item from ever shrinking below its basis size, even if that means other items shrink further to compensate.

```
1 .logo {  
2   flex-shrink: 0; /* never let the logo get squeezed */  
3   flex-basis: 120px;  
4 }
```

The flex Shorthand

`grow`, `shrink`, and `basis` are almost always set together using the flex shorthand, in that order.

```
1 .main-content {  
2   flex: 1 1 0; /* grow, shrink, basis */  
3 }
```

Two shorthand keywords cover the most common cases directly: `flex: 1` is shorthand for `flex: 1 1 0%`, an item that grows and shrinks freely and ignores its own content size when dividing space; `flex: auto` is shorthand for `flex: 1 1 auto`, an item that grows and shrinks but starts from its natural content size instead of zero.

```
1 .column {  
2   flex: 1; /* three columns, each taking an equal third */  
3 }
```

```
1 <div style="display: flex; gap: 16px;">
2   <div class="column">A</div>
3   <div class="column">B</div>
4   <div class="column">C</div>
5 </div>
```

7.3. Aligning and Justifying Content

Beyond sizing items, Flexbox provides a set of alignment properties that distribute and position items within their container along both axes.

justify-content: Aligning Along the Main Axis

`justify-content` distributes space along the main axis, controlling how items are positioned when they don't fill the full width (or height, in a column layout) of the container.

- `flex-start` (default), items pack toward the start of the main axis;
- `flex-end`, items pack toward the end;
- `center`, items are centered along the main axis;
- `space-between`, equal space is placed between items, with none at the outer edges;
- `space-around`, equal space surrounds each item, so edge items get half as much outer space as the space between items;
- `space-evenly`, perfectly equal space everywhere, including the outer edges.

```
1 .navbar {
2   display: flex;
3   justify-content: space-between;
4 }
```

```
1 <nav class="navbar">
2   <div class="logo">MySite</div>
3   <div class="nav-links">...</div>
4 </nav>
```

`justify-content: space-between`, pushing the logo to one edge and the navigation links to the other, is one of the most frequently used Flexbox patterns for site headers.

align-items: Aligning Along the Cross Axis

`align-items` positions items along the cross axis, typically vertically in a row layout.

- `stretch` (default), items stretch to fill the container's cross-axis size;
- `flex-start`, items align to the start of the cross axis;
- `flex-end`, items align to the end of the cross axis;
- `center`, items are centered along the cross axis;
- `baseline`, items align along their text baselines.

```
1 .navbar {
2   display: flex;
3   justify-content: space-between;
4   align-items: center;
5 }
```

The combination of `justify-content` and `align-items` shown above, center on the cross axis, some distribution on the main axis, is also the single most common way to perfectly center one element inside another with Flexbox: `display: flex; justify-content: center; align-items: center;` on the parent centers a child both horizontally and vertically, regardless of the child's size.

align-self: Overriding a Single Item

`align-self` overrides `align-items` for one specific flex item, letting a single item break from how its siblings are aligned.

```
1 .card-footer-cta {  
2   align-self: flex-end;  
3 }
```

align-content: Aligning Wrapped Lines

When flex-wrap creates multiple lines, align-content distributes space between those lines along the cross axis, using the same set of values as justify-content. It has no effect on a single-line flex container.

```
1 .card-grid {  
2   display: flex;  
3   flex-wrap: wrap;  
4   align-content: flex-start;  
5 }
```

7.4. Common Flexbox Layout Patterns

With the individual properties covered, this section brings them together into a handful of layout patterns that appear, in some form, on nearly every website.

A Sticky Footer

A common layout goal is a footer that stays at the bottom of the viewport on short pages, without floating in the middle of the screen, while still being pushed down normally on pages with enough content to fill the screen.

```
1 body {  
2   display: flex;  
3   flex-direction: column;  
4   min-height: 100vh;  
5 }  
6 main {  
7   flex: 1;  
8 }
```

Making `<body>` a column flex container at least as tall as the viewport, with `<main>` set to `flex: 1`, means `<main>` always expands to absorb any leftover space, pushing a footer that follows it in the markup to the very bottom.

Equal-Height Cards

Flex items automatically stretch to match the tallest item in the same row by default (`align-items: stretch`), which makes a row of cards with varying amounts of content naturally line up at equal height with no extra CSS required.

```
1 .card-row {
2   display: flex;
3   gap: 16px;
4 }
5
6 .card {
7   flex: 1;
8   display: flex;
9   flex-direction: column;
10 }
11
12 .card-body {
13   flex: 1; /* pushes the footer of each card to the bottom */
14 }
```

Nesting Flexbox inside each card, making the card itself a column flex container, with a `flex: 1` body, is what aligns each card's footer (a price, a button) to the same bottom edge, regardless of how much text sits above it.

A Media Object (Icon or Image Beside Text)

A frequently reused pattern pairs a fixed-size image or icon with a block of text beside it, the text taking up the remaining width.

```
1 .media {  
2   display: flex;  
3   gap: 12px;  
4   align-items: flex-start;  
5 }  
6 .media-icon {  
7   flex-shrink: 0;  
8   width: 40px;  
9 }  
10 .media-body {  
11   flex: 1;  
12 }  
13  
14
```

When to Reach for Grid Instead

Flexbox is a one-dimensional model: it excels at laying out a single row or a single column, and wrapping that row or column onto multiple lines when needed. Once a layout genuinely needs to align content along rows and columns at the same time, a photo gallery, a dashboard, an entire page template, CSS Grid, covered in the next chapter, is usually a better fit.

Conclusion

This chapter introduced Flexbox: activating it with `display: flex`, the main and cross axes it lays items out along, the `grow/shrink/basis` properties that size individual items, the alignment properties that position and distribute them, and several layout patterns built from these pieces. Chapter 8 introduces CSS Grid, a complementary layout system built for two-dimensional layout.

Chapter 8. Layout with CSS Grid

8.1. Defining Grid Containers, Rows, and Columns

CSS Grid Layout is a two-dimensional layout system: unlike Flexbox, it lays out items along rows and columns simultaneously, making it well suited to whole-page layouts and any design that needs items to align both horizontally and vertically at once.

Creating a Grid Container

Grid is activated with `display: grid` on a container element, exactly as `display: flex` activates Flexbox.

```
1 .layout {  
2   display: grid;  
3 }
```

On its own, `display: grid` produces a single-column grid, with every child stacked in its own row, the grid only becomes useful once explicit columns and rows are defined.

grid-template-columns and **grid-template-rows**

These two properties define the grid's structure: how many columns and rows it has, and how large each one is, as a space-separated list of sizes.

```
1 .layout {  
2   display: grid;  
3   grid-template-columns: 240px 1fr;  
4   grid-template-rows: auto 1fr auto;  
5 }
```

This example defines a two-column, three-row grid: a fixed 240px first column and a flexible second column, with a header row and footer row sized to their content (auto) and

a middle row that absorbs remaining vertical space (1fr), the fr unit is covered in full in section 8.3.

Placing Items in the Grid

By default, grid items are placed automatically, one per cell, in source order, following the grid's flow direction. Nothing more than display: grid and a template is required for this automatic placement to work.

```
1 <div class="layout">
2   <header>Header</header>
3   <main>Main content</main>
4   <footer>Footer</footer>
5 </div>
```

Explicit placement, where an item is assigned to a specific row and column rather than the next available cell, is covered in section 8.2.

The gap Property

As with Flexbox, gap sets consistent spacing between grid cells, and can be set for rows and columns independently with row-gap and column-gap.

```
1 .layout {
2   display: grid;
3   grid-template-columns: 240px 1fr;
4   gap: 24px;
5 }
```

8.2. Placing Items with Grid Lines and Areas

Grid's real power comes from precisely controlling where each item sits, either by referencing the numbered lines between grid cells or by naming areas of the grid directly.

Grid Lines

Every row and column in a grid is bordered by numbered lines, starting at 1. A three-column grid has four vertical grid lines; a three-row grid has four horizontal grid lines. `grid-column` and `grid-row` place an item by specifying which lines it starts and ends at.

```
1 .hero-banner {
2   grid-column: 1 / 3; /* from line 1 to line 3, spanning two columns */
3 }
```

The `span` keyword is often more convenient than an explicit end line, since it describes how many tracks to cover rather than requiring the exact line number to be counted out.

```
1 .hero-banner {
2   grid-column: 1 / span 2; /* start at line 1, span 2 columns */
3 }
4 .sidebar {
5   grid-row: 1 / span 3; /* spans three rows */
6 }
7
```

Named Grid Areas

`grid-template-areas` offers a distinctly visual way to lay out a grid: each row of the property's value is written as a quoted string, and each word names the area that cell belongs to, directly sketching the shape of the layout inside the CSS itself.

```
1 .page {
2   display: grid;
3   grid-template-columns: 240px 1fr;
4   grid-template-rows: auto 1fr auto;
5   grid-template-areas:
6     "sidebar header"
7     "sidebar main"
8     "sidebar footer";
9 }
10 header { grid-area: header; }
11 main { grid-area: main; }
12 footer { grid-area: footer; }
13 .sidebar { grid-area: sidebar; }
```

Each element is assigned to an area with the single `grid-area` property, matching one of the names used in `grid-template-areas`. Here, the word "sidebar" spans all three rows of the first column simply by being repeated in that position on every row of the template string, letting the sidebar run the full height of the layout without any manual line-number math.

A period (.) in a `grid-template-areas` string marks an empty cell, one deliberately left without any assigned item.

8.3. The `fr` Unit and the `repeat()` Notation

Two features specific to Grid make defining flexible, evenly distributed tracks considerably less verbose than listing out fixed sizes by hand.

The `fr` Unit

`fr` stands for "fraction" and represents a share of the leftover space in the grid container, after any fixed-size tracks have been subtracted, conceptually similar to `flex-grow` in Flexbox, but expressed directly as a track size.

```
1 .layout {
2   display: grid;
3   grid-template-columns: 1fr 1fr 1fr; /* three equal columns */
4 }
5
6 .layout-sidebar {
7   display: grid;
8   grid-template-columns: 240px 2fr 1fr; /* fixed, then a 2:1 split of the rest */
9 }
```

In the second example, the `240px` column is subtracted first, and the remaining space is then split so the second column gets twice as much as the third, the same proportional logic as `flex-grow`, applied to grid tracks instead of flex items.

The `repeat()` Function

`repeat()` avoids writing out a repeated pattern of track sizes by hand, taking a count and a size (or pattern of sizes) to repeat.

```

1 .gallery {
2   display: grid;
3   grid-template-columns: repeat(4, 1fr); /* four equal columns */
4   gap: 16px;
5 }

```

`repeat(4, 1fr)` is exactly equivalent to writing `1fr 1fr 1fr 1fr`, and `repeat()` can just as easily repeat a multi-value pattern.

```

1 .timeline {
2   grid-template-columns: repeat(3, 100px 20px); /* 100px, 20px, repeated three times */
3 }

```

Auto-Fill and Auto-Fit Responsive Grids

Combining `repeat()` with the auto-fill or auto-fit keyword, instead of a fixed count, along with `minmax()`, produces a grid that automatically fits as many columns as the available width allows, without a single media query.

```

1 .card-grid {
2   display: grid;
3   grid-template-columns: repeat(auto-fit, minmax(220px, 1fr));
4   gap: 16px;
5 }

```

`minmax(220px, 1fr)` tells each column to never shrink below 220px, but to grow and share leftover space equally (1fr) when there is room. `auto-fit` then fits as many 220px-or-wider columns as will comfortably fit in the container, automatically producing more columns on a wide screen and fewer on a narrow one, a genuinely responsive grid from three lines of CSS, previewing the responsive design techniques covered in full in Chapter 9.

8.4. Common Grid Layout Patterns

This closing section applies the properties from this chapter to two layouts that appear constantly in real-world interfaces: a full page shell, and a responsive card gallery.

A Classic Page Layout

The header/sidebar/main/footer layout sketched in section 8.2 is one of the most common full-page Grid layouts, and is worth seeing assembled in full.

```
1 body {
2   display: grid;
3   min-height: 100vh;
4   grid-template-columns: 240px 1fr;
5   grid-template-rows: auto 1fr auto;
6   grid-template-areas:
7     "sidebar header"
7     "sidebar main"
8     "sidebar footer";
9 }
10 header { grid-area: header; }
11 main    { grid-area: main; }
12 footer  { grid-area: footer; }
13 .sidebar { grid-area: sidebar; }
14
15
```

Setting `min-height: 100vh` on the grid container guarantees the layout always fills at least the full viewport height, so the footer never rides up in the middle of the screen on a page with little content, the same underlying goal as the Flexbox sticky-footer pattern from Chapter 7, solved here as a natural side effect of the grid's row sizing instead.

A Responsive Card Gallery

The auto-fit pattern from section 8.3, combined with `gap`, is enough on its own to build a fully responsive photo or product gallery.

```

1  .gallery {
2    display: grid;
3    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
4    gap: 20px;
5  }
6  .gallery img {
7    width: 100%;
8    aspect-ratio: 1 / 1;
9    object-fit: cover;
10   border-radius: 8px;
11 }
12

```

aspect-ratio: 1 / 1 forces each image into a square regardless of its original dimensions, and object-fit: cover, the background-size: cover equivalent for the element, crops the image to fill that square without distorting it.

Choosing Between Flexbox and Grid

Flexbox and Grid are not competitors; they solve different problems and are routinely used together on the same page, Grid for the overall page structure and Flexbox for the one-dimensional arrangement of items inside each section of it.

Situation	Better Fit
Arranging items in a single row or column	Flexbox
Aligning content along both rows and columns at once	Grid
A component whose size is driven by its content (a toolbar, a tag list)	Flexbox
An overall page or app layout with named regions	Grid
A responsive image or card gallery	Grid

Conclusion

This chapter introduced CSS Grid: defining rows and columns with `grid-template-columns` and `grid-template-rows`, placing items precisely with grid lines or named `grid-template-areas`, the `fr` unit and `repeat()` notation that make flexible tracks concise to write, and two layout patterns, a full-page shell and a responsive gallery, built entirely from them. With both major CSS layout systems covered, Chapter 9 turns to responsive design more broadly: media queries, breakpoints, and the units that make a layout adapt to any screen.

Chapter 9. Responsive Design and Media Queries

9.1. The Viewport and Mobile-First Design

Responsive design is the practice of building a single layout that adapts to whatever screen it is viewed on, rather than building separate sites for desktop and mobile. It rests on a handful of ideas that need to be in place before a single media query is written.

The Viewport Meta Tag

By default, mobile browsers render a page at a fixed desktop-like width, typically around 980px, and then zoom the whole result out to fit the screen, producing tiny, unreadable text rather than an actually responsive layout. The viewport meta tag, placed in the document's <head>, disables this behavior.

```
1 <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Without this exact meta tag, media queries and any other responsive CSS on the page will not work correctly on real mobile devices, even though the layout may look perfectly responsive when the desktop browser window is resized, it is one of the most common causes of "my responsive site doesn't work on phones."

width=device-width sets the viewport's width to match the device's actual screen width in CSS pixels, and initial-scale=1.0 sets the initial zoom level to 100%, so 1 CSS pixel corresponds to a predictable, consistent physical size across devices.

Mobile-First Design

Mobile-first design means writing the base, unqualified CSS for the smallest expected screen first, then layering on additional styles for larger screens with media queries, the opposite order from designing for desktop first and cramming the layout into a phone screen afterward.

```

1  /* Base styles: apply to every screen, mobile included */
2  .card-grid {
3      display: grid;
4      grid-template-columns: 1fr;
5      gap: 16px;
6  }
7  /* Enhancement: applies only from 768px upward */
8  @media (min-width: 768px) {
9      .card-grid {
10         grid-template-columns: repeat(3, 1fr);
11     }
12 }
13

```

This approach tends to produce simpler, more robust CSS: the base styles handle the most constrained case, and each media query only has to add or adjust what changes for a larger screen, rather than undoing an entire desktop layout to squeeze it onto a phone.

Fluid, Not Fixed, Layouts

A responsive layout also avoids fixed pixel widths wherever a flexible alternative works just as well: percentages, fr units, and flex-grow, all covered in earlier chapters, already adapt to the available space without a media query at all. Media queries are best reserved for changes a fluid layout genuinely cannot express on its own, such as switching from one column to three.

9.2. Writing Media Queries and Choosing Breakpoints

A media query applies a block of CSS only when a given condition about the browser or device is true, most often, the width of the viewport.

Basic Media Query Syntax

```

1 @media (min-width: 768px) {
2   .container {
3     max-width: 720px;
4     margin: 0 auto;
5   }
6 }

```

@media is followed by one or more conditions and a block of ordinary CSS rules, which only take effect while the condition holds. min-width matches when the viewport is at least that wide; max-width matches when it is at most that wide.

```

1 /* Applies only on narrow screens, up to 599px wide */
2 @media (max-width: 599px) {
3   .sidebar {
4     display: none;
5   }
6 }

```

Combining Conditions

The and keyword combines multiple conditions in a single media query, useful for targeting a specific range of screen widths.

```

1 /* Applies only between 600px and 899px */
2 @media (min-width: 600px) and (max-width: 899px) {
3   .layout {
4     grid-template-columns: repeat(2, 1fr);
5   }
6 }

```

Choosing Breakpoints

A breakpoint is the viewport width at which a media query changes the layout. Rather than targeting specific devices, whose exact screen sizes change every year, breakpoints are best

chosen based on where a given design actually starts to look cramped or overly sparse, genuinely letting the content decide.

Common Breakpoint	Roughly Targets
480px	Small phones
768px	Tablets and large phones
1024px	Small laptops and tablets in landscape
1280px	Standard desktop monitors

These widths are a common starting point, not a rule: the right breakpoints for a real project are the ones where that project's own layout needs to change, found by resizing the browser and watching where it breaks, not copied blindly from a table.

Where Media Queries Live

A media query can wrap any number of rules, and can appear anywhere in a stylesheet, though grouping all of a component's responsive adjustments together, directly after that component's base styles, is a common and easy-to-maintain convention.

```
1 .nav-links {
2   display: none;
3 }
4 @media (min-width: 768px) {
5   .nav-links {
6     display: flex;
7     gap: 24px;
8   }
9 }
10
```

9.3. Responsive Units: %, vw/vh, em, and rem

Choosing the right unit is as important to a responsive layout as media queries themselves. CSS offers several relative units, each tied to a different reference size.

Percentages

A percentage is always relative to some property of the element's containing block, most often its width, though which property depends on which CSS property the percentage is used on.

```
1 .sidebar {  
2   width: 30%; /* 30% of its parent's width */  
3 }
```

Viewport Units: vw and vh

vw and vh are relative to the viewport itself: 1vw is 1% of the viewport's width, and 1vh is 1% of its height, regardless of the element's parent.

```
1 .hero {  
2   height: 60vh; /* 60% of the viewport height */  
3 }  
4 h1 {  
5   font-size: 5vw; /* scales continuously with viewport width */  
6 }  
7
```

Sizing text purely in vw units scales it continuously with the viewport, but with nothing to stop it shrinking to the point of illegibility on a narrow phone, or growing enormous on a wide monitor, `clamp()`, covered in Chapter 11, solves this by pairing a vw-based value with a minimum and maximum.

em: Relative to Font Size

1em is equal to the font-size of the current element, or, when used to set font-size itself, equal to the font-size of the parent element. Because em compounds through nested

elements that each set their own font-size, it can produce unexpectedly large or small results several levels deep in a document.

```
1 .card {
2   font-size: 1.2em; /* 1.2 x parent's font-size */
3   padding: 1em; /* 1.2em's worth of pixels, since padding uses this element's own font-size */
4 }
```

rem: Relative to the Root

rem ("root em") is always relative to the font-size of the document's root <html> element, regardless of how deeply nested the current element is, avoiding the compounding behavior of em entirely, which is why rem is generally preferred for font sizes throughout a stylesheet.

```
1 html {
2   font-size: 100%; /* respects the browser/user default, usually 16px */
3 }
4 h1 {
5   font-size: 2.5rem; /* always 2.5 x the root size, wherever it appears */
6 }
7
8 p {
9   font-size: 1rem;
10 }
11
```

Because rem is relative to the root font-size, and the root font-size in turn respects a reader's browser settings, using rem for font sizes is one of the simplest ways to keep a site accessible to readers who have increased their browser's default text size, a fixed px value on <html> would override that preference entirely.

Choosing the Right Unit

Unit	Relative To	Typically Used For
%	The containing block	Fluid widths

vw / vh	The viewport	Full-bleed sections, fluid type
em	The current or parent font-size	Spacing that should scale with local text size
rem	The root font-size	Font sizes, and most spacing throughout a stylesheet
px	An absolute pixel	Borders, and anything that should never scale

9.4. Responsive Images and Adaptive Layouts

Layout is only half of responsive design; images need their own set of techniques to scale correctly and avoid wasting bandwidth on a screen too small to benefit from a full-resolution file.

The Fluid Image Pattern

The single most important responsive image rule prevents an image from ever overflowing its container, regardless of the image's native dimensions.

```

1  img {
2    max-width: 100%;
3    height: auto;
4  }
```

`max-width: 100%` lets an image shrink to fit a narrow container while never stretching wider than its natural size, and `height: auto` preserves the image's aspect ratio as its width changes, without it, an explicit height set elsewhere would leave the image looking squashed or stretched at other widths.

object-fit for Cropped Images

When an image must fill a container of a fixed aspect ratio, such as a square thumbnail grid, `object-fit` controls how it is cropped or scaled to fit, exactly as `background-size` does for background images.

```
1 .thumbnail {  
2   width: 100%;  
3   aspect-ratio: 1 / 1;  
4   object-fit: cover;  
5 }
```

- `cover`, fills the box completely, cropping whatever overflows;
- `contain`, fits the whole image inside the box, without cropping;
- `fill` (default), stretches the image to fill the box exactly, distorting its aspect ratio if needed.

Serving Different Images per Screen Size

The `<picture>` element, part of the HTML specification covered in Part 1 of this series, lets the browser choose between multiple image sources based on media conditions, for example, serving a cropped, portrait-oriented image on narrow screens and a wider landscape image on desktop.

```
1 <picture>  
2   <source media="(min-width: 768px)" srcset="hero-wide.jpg">  
3     
4 </picture>
```

This is a layout and art-direction decision, made in HTML rather than CSS, but it works hand in hand with the responsive layout techniques in this chapter and is worth knowing about alongside them.

Hiding and Showing Content Responsively

Beyond images, entire sections of a layout are often shown or hidden depending on screen size, a condensed mobile navigation menu instead of a full desktop one, for instance.

```
1 .mobile-menu-button {  
2   display: block;  
3 }  
4 .desktop-nav-links {  
5   display: none;  
6 }  
7  
8 @media (min-width: 768px) {  
9   .mobile-menu-button {  
10    display: none;  
11  }  
12  .desktop-nav-links {  
13    display: flex;  
14  }  
15 }  
16  
17
```

Toggling `display: none` based on screen size hides content visually but still downloads it (including any images inside it); it is a layout technique, not a way to reduce how much data a mobile visitor downloads.

Conclusion

This chapter covered the viewport meta tag and mobile-first design as the foundation of a responsive layout, writing and combining media queries, choosing sensible breakpoints, the relative units that let a layout scale smoothly between them, and the patterns that keep images fluid and well-cropped at any size. Chapter 10 shifts from static layout to motion: transitions, transforms, and keyframe animations.

Chapter 10. Transitions, Transforms, and Animations

10.1. CSS Transitions and Easing Functions

A transition smoothly animates a property from one value to another over a set duration, whenever that property's value changes, most commonly in response to a pseudo-class like `:hover` or `:focus`, or a class toggled by JavaScript.

The transition Property

```
1 .button {  
2   background-color: #3d8bff;  
3   transition: background-color 0.2s ease;  
4 }  
5 .button:hover {  
6   background-color: #1f3e73;  
7 }  
8
```

The transition is declared on the button's base state, not on `:hover`, it describes how any change to background-color on this element should animate, regardless of what triggers that change. transition accepts, in order, the property to animate, a duration, an optional easing function, and an optional delay.

Transitioning Multiple Properties

Several properties can be transitioned at once, each with its own duration if needed, by separating them with commas, or, using the keyword `all`, every animatable property that changes will transition with the same settings.

```

1 .card {
2   transform: translateY(0);
3   box-shadow: 0 1px 3px rgb(0 0 0 / 10%);
4   transition:
5     transform 0.2s ease,
6     box-shadow 0.2s ease;
7 }
8
9 .card:hover {
10  transform: translateY(-4px);
11  box-shadow: 0 8px 20px rgb(0 0 0 / 15%);
12 }

```

transition: all is convenient but transitions every property that changes, including ones a later code change might inadvertently affect; naming the specific properties to transition, as shown above, is usually the more predictable and better-performing choice.

Easing Functions

The easing function (also called a timing function) controls the rate of change over the transition's duration, rather than moving at a constant speed.

- linear, constant speed throughout;
- ease (default), starts slowly, speeds up, then slows down again near the end;
- ease-in, starts slowly and accelerates toward the end;
- ease-out, starts quickly and decelerates toward the end;
- ease-in-out, a more pronounced slow start and slow finish than ease;
- cubic-bezier(x1, y1, x2, y2), a fully custom easing curve.

ease-out is a common choice for elements entering the screen or responding to a user action, since a quick start feels immediately responsive, while ease-in suits elements leaving the screen.

10.2. 2D and 3D Transforms

The transform property repositions, resizes, rotates, or skews an element without affecting the normal document flow around it, unlike changing an element's actual position or width, neighboring elements do not shift to accommodate a transformed element.

translate: Moving an Element

```
1 .tooltip {  
2   transform: translate(10px, -50%);  
3 }
```

translate() shifts an element horizontally and vertically from its normal position. translateX() and translateY() move along a single axis; a percentage value in translate is relative to the element's own size, which is what makes translateY(-50%) a common way to vertically center an absolutely positioned element regardless of its height.

scale: Resizing an Element

```
1 .card:hover {  
2   transform: scale(1.03);  
3 }
```

scale() resizes an element by a multiplier: 1 is its normal size, 1.03 is 3% larger, and 0.9 is 10% smaller. scaleX() and scaleY() scale a single axis independently, which can stretch or squash an element deliberately.

rotate: Rotating an Element

```
1 .icon-chevron.is-open {  
2   transform: rotate(180deg);  
3 }
```

`rotate()` turns an element around its center point by default, by a given angle; positive values rotate clockwise.

Combining Transforms

Multiple transform functions can be listed together, space-separated, within a single transform declaration, and are applied in the order written.

```
1 .card:hover {  
2   transform: translateY(-4px) scale(1.02) rotate(-1deg);  
3 }
```

Setting `transform: translateY(-4px)` and then, separately, `transform: scale(1.02)` in two different rules does not combine them, the second declaration replaces the first entirely, since `transform` is a single property. All desired transform functions must be listed together in one declaration.

transform-origin

By default, `scale()` and `rotate()` transform an element around its exact center. `transform-origin` moves that pivot point, useful for effects like a menu that appears to unfold from one of its corners.

```
1 .menu {  
2   transform-origin: top left;  
3   transform: scale(0);  
4 }  
5 .menu.is-open {  
6   transform: scale(1);  
7 }  
8
```

A Note on 3D Transforms

CSS also supports 3D transforms, `rotateX()`, `rotateY()`, `translateZ()`, and the `perspective` property that gives them a vanishing point, which apply the same underlying ideas along a third axis. They are used more sparingly than 2D transforms in most interfaces and are

mentioned here mainly so the terms are recognizable; the 2D transforms covered above account for the large majority of real-world use.

10.3. Keyframe Animations with @keyframes

Transitions animate between exactly two states: before and after a change. @keyframes defines a sequence with any number of intermediate steps, and can run automatically, repeat, and reverse, without needing a triggering state change at all.

Defining a Keyframe Animation

```
1 @keyframes fade-in {
2   from {
3     opacity: 0;
4   }
5   to {
6     opacity: 1;
7   }
8 }
```

from and to are shorthand for the 0% and 100% keyframes; more complex animations use explicit percentages to define any number of intermediate steps.

```
1 @keyframes pulse {
2   0% {
3     transform: scale(1);
4   }
5   50% {
6     transform: scale(1.08);
7   }
8   100% {
9     transform: scale(1);
10  }
```

Applying an Animation

Defining `@keyframes` on its own does nothing; the `animation` property applies a defined animation to an element and controls how it plays.

```
1 .notification-badge {  
2   animation: pulse 1.5s ease-in-out infinite;  
3 }
```

This example names the pulse animation defined above, runs it over 1.5 seconds with an ease-in-out timing function, and repeats it indefinitely with the `infinite` keyword.

Animation Sub-Properties

Property	Purpose
<code>animation-name</code>	Which <code>@keyframes</code> rule to run
<code>animation-duration</code>	How long one cycle takes
<code>animation-timing-function</code>	The easing curve, same values as transition
<code>animation-delay</code>	How long to wait before starting
<code>animation-iteration-count</code>	How many times to repeat, or infinite
<code>animation-direction</code>	normal, reverse, or alternating each cycle
<code>animation-fill-mode</code>	Whether the element keeps the first/last keyframe's styles outside the animation's active time

As with transition, these can all be set individually or combined into the animation shorthand, in roughly the order listed above.

```
1 .toast {  
2   animation: fade-in 0.3s ease-out forwards;  
3 }
```

animation-fill-mode: forwards, commonly abbreviated just forwards in the shorthand as above, keeps the element in its final keyframe's state once the animation finishes, instead of snapping back to its un-animated appearance, essential for an element like a toast notification that should stay visible, not fade back out immediately after fading in.

10.4. Performance Considerations for Animated CSS

Not every CSS property animates equally cheaply. Understanding why some animations feel smooth while others stutter helps avoid a common source of janky, unpolished-feeling interfaces.

Why Some Properties Are Cheaper Than Others

Animating properties like width, height, top, left, or margin forces the browser to recalculate the layout of the animated element and, often, its surrounding elements, on every single frame, a relatively expensive operation called layout or reflow. Animating transform and opacity, by contrast, can typically be handled by the browser's compositor without recalculating layout at all, and in many cases without even repainting the element's pixels, making them dramatically cheaper to animate smoothly.

As a rule of thumb, prefer animating transform and opacity over layout-affecting properties whenever the same visual effect can be achieved either way, moving an element with transform: translateX() instead of animating its left property is the single most common example.

```

1  /* Prefer this: */
2  .drawer {
3      transform: translateX(-100%);
4      transition: transform 0.3s ease;
5  }
6  .drawer.is-open {
7      transform: translateX(0);
8  }
9  /* Over this, which is more expensive to animate: */
10 .drawer {
11     left: -100%;
12     transition: left 0.3s ease;
13 }
14 .drawer.is-open {
15     left: 0;
16 }
17

```

will-change

The will-change property hints to the browser that an element is about to be animated, giving it a chance to prepare an optimization, such as promoting the element to its own compositor layer, ahead of time.

```

1  .card {
2      will-change: transform;
3  }

```

will-change is a hint for elements about to animate soon, not a general performance switch to leave on permanently, applying it to many elements, or to elements that rarely actually animate, can consume extra memory and paradoxically hurt performance instead of helping it.

Respecting Reduced-Motion Preferences

Some readers set a system-level preference to reduce or disable animation, often for reasons related to motion sensitivity or vestibular disorders. The `prefers-reduced-motion` media feature detects this preference, letting a stylesheet scale back or remove animation for readers who have asked for it.

```
1 @media (prefers-reduced-motion: reduce) {  
2   * {  
3     animation-duration: 0.001ms !important;  
4     animation-iteration-count: 1 !important;  
5     transition-duration: 0.001ms !important;  
6   }  
7 }
```

This pattern, applied broadly with the universal selector, effectively disables animation and transition durations sitewide for anyone who has requested reduced motion at the operating system level, while leaving every other user's experience untouched.

Conclusion

This chapter covered CSS transitions and their easing functions, 2D transforms for moving, scaling, and rotating elements, keyframe animations for multi-step and looping motion, and the performance and accessibility considerations that separate a smooth, comfortable animation from a janky or disorienting one. Chapter 11 covers the modern CSS features and best practices that tie the rest of this book together, including custom properties and the `calc()` family of functions.

Chapter 11. Modern CSS

Features and Best Practices

11.1. Custom Properties (CSS Variables) and var()

Custom properties, commonly called CSS variables, let a value be defined once and reused throughout a stylesheet, updated in a single place whenever it changes, and even overridden differently for different parts of a page.

Defining and Using a Custom Property

A custom property name always begins with two hyphens, and is typically defined on `:root`, a pseudo-class matching the document's root element that makes the property available everywhere.

```
1 :root {  
2   --brand-color: #3d8bff;  
3   --spacing-unit: 8px;  
4 }  
5 .button {  
6   background-color: var(--brand-color);  
7   padding: calc(var(--spacing-unit) * 2);  
8 }  
9
```

The `var()` function reads a custom property's value wherever a normal value would be expected. Changing `--brand-color` in the single `:root` declaration updates every rule that references it, without needing to find and replace the color throughout the whole file.

Fallback Values

`var()` accepts an optional second argument, used if the referenced custom property has not been defined or is currently invalid.

```
1 .badge {  
2   color: var(--badge-color, white);  
3 }
```

Custom Properties Cascade and Inherit

Unlike a preprocessor variable, which is resolved once when a stylesheet is compiled, a CSS custom property is a genuine part of the cascade: it can be inherited, and it can be redefined at any level of the document to change its value for that element and everything inside it.

```
1 :root {  
2   --card-bg: white;  
3 }  
4 .theme-dark {  
5   --card-bg: #1a1a1a;  
6 }  
7  
8 .card {  
9   background-color: var(--card-bg);  
10 }  
11
```

Any `.card` inside an element carrying the `.theme-dark` class automatically picks up the redefined `--card-bg`, without `.card` itself needing to know anything about theming, this cascading, context-sensitive behavior is what makes custom properties especially well suited to building light/dark themes and component variants.

Custom Properties vs. Preprocessor Variables

Sass and other preprocessors, covered in a later volume of this series, have offered variables for longer than native CSS has. The key difference is that a preprocessor variable is replaced with a fixed value at build time and cannot change afterward, while a native custom property remains live in the browser and can be read or changed at runtime,

including from JavaScript, a capability a compiled preprocessor variable simply cannot offer.

11.2. Functional Notations: `calc()`, `clamp()`, `min()`, and `max()`

CSS provides several functions for computing a value from other values, rather than hardcoding a single fixed number, letting a stylesheet express relationships that would otherwise need JavaScript or a preprocessor.

`calc()`: Mixing Units in One Expression

`calc()` performs arithmetic directly in CSS, and, unlike plain arithmetic, can freely mix units that would otherwise be impossible to combine, such as a percentage and a fixed pixel value.

```
1 .main-content {
2   width: calc(100% - 240px); /* full width, minus a fixed sidebar */
3 }
4 .sticky-header-offset {
5   margin-top: calc(var(--header-height) + 1rem);
6 }
7
```

Spaces around the `+` and `-` operators inside `calc()` are required; `calc(100% - 240px)` is parsed as a single, invalid value rather than a subtraction, since `- 240px` would be read as a negative value with nothing to combine it with.

`min()` and `max()`

`min()` returns the smallest of the values passed to it, and `max()` returns the largest, each evaluated independently and re-evaluated as the page resizes.

```
1 .container {
2   width: min(90%, 1100px); /* never wider than 1100px, never wider than 90% of its parent */
3 }
```

This single declaration replaces what would otherwise require a media query: the container fills 90% of its parent on narrow screens, and stops growing past 1100px once the viewport is wide enough for that cap to take over.

clamp(): A Value with a Floor and a Ceiling

clamp() takes three arguments, a minimum, a preferred value, and a maximum, and returns the preferred value, unless doing so would go below the minimum or above the maximum, in which case it returns that limit instead.

```
1 h1 {  
2   font-size: clamp(1.75rem, 5vw, 3.5rem);  
3 }
```

This heading scales fluidly with the viewport (5vw) on medium screens, exactly the concern raised about pure vw sizing in Chapter 9, but never shrinks below 1.75rem on a narrow phone or grows past 3.5rem on a very wide monitor, a single line of CSS standing in for what used to require several font-size declarations across multiple media queries.

Combining These Functions

calc(), min(), max(), and clamp() can all be nested inside one another and can reference custom properties, making it possible to build fairly sophisticated fluid values from simple building blocks.

```
1 :root {  
2   --space: clamp(1rem, 2vw, 2rem);  
3 }  
4 .section {  
5   padding: var(--space) calc(var(--space) * 1.5);  
6 }  
7
```

11.3. Organizing CSS and Naming Conventions

As a stylesheet grows from a handful of rules to thousands of lines across a real project, how it is organized starts to matter as much as any individual property. This section covers conventions that keep a large CSS codebase navigable.

The BEM Naming Convention

BEM (Block, Element, Modifier) is a widely used class naming convention that encodes a component's structure directly into its class names, making relationships between classes clear without needing to trace through the CSS or the markup.

```
1 <div class="card card--featured">
2   <h3 class="card__title">Featured Post</h3>
3   <p class="card__body">...</p>
4   <button class="card__button card__button--primary">
      Read more
5   </button>
6 </div>
7
```

- Block, a standalone component, such as card;
- Element, a part of a block, written with a double underscore, such as card__title;
- Modifier, a variant of a block or element, written with a double hyphen, such as card--featured.

```
1 .card { }
2 .card--featured { }
3 .card__title { }
4 .card__button { }
5 .card__button--primary { }
```

Because every class name is effectively self-scoped to its block, BEM keeps selectors flat and low in specificity (almost always a single class selector), which sidesteps many of the specificity conflicts described in Chapter 3.

Organizing Files by Responsibility

Building on the file-splitting approach introduced in Chapter 2, larger projects commonly separate CSS by responsibility, often loaded in this order:

- Settings, custom properties: color palette, spacing scale, font stacks;
- Base/reset, default element styling and browser-inconsistency resets;
- Layout, page-level structure: grid shells, containers, headers, footers;
- Components, individually reusable pieces: cards, buttons, form controls;
- Utilities, small, single-purpose classes such as `.text-center` or `.mt-4`.

Writing Maintainable CSS

- Prefer classes over ids for styling, reserving ids for JavaScript hooks and in-page anchors, as noted in Chapter 3;
- Avoid deeply nested descendant selectors such as `.sidebar ul li a span`, which are both fragile and hard to override;
- Keep selectors as low in specificity as the design allows, reaching for `!important` only as a genuine last resort;
- Group related declarations (positioning, box model, typography, visual) consistently within a rule, so similar rules are easy to compare at a glance;
- Comment on why a non-obvious value was chosen, not what the property obviously already says.

11.4. Cross-Browser Compatibility and Vendor Prefixes

Not every CSS feature is supported identically, or at all, in every browser. This closing section covers how to check support and handle the gaps gracefully.

Checking Feature Support

Before relying on a newer CSS feature in a real project, it is worth checking its current browser support, most commonly on a site such as caniuse.com, which tracks support across every major browser and version.

Vendor Prefixes

In the past, browsers implemented experimental CSS features behind a vendor-specific prefix before the feature was finalized and widely supported unprefixed.

```
1 .box {  
2   -webkit-transform: rotate(5deg); /* older WebKit/Chrome/Safari */  
3   -moz-transform: rotate(5deg);    /* older Firefox */  
4   transform: rotate(5deg);         /* standard, listed last */  
5 }
```

Vendor prefixes are far less necessary today than they once were, since most CSS features in this book ship unprefixed across current browser versions; when a prefix genuinely is still required for a specific feature, it should always be listed before the standard, unprefixed property, which is listed last so it takes precedence in any browser that supports both.

Rather than writing prefixes by hand, most real projects generate them automatically as part of a build process using a tool such as Autoprefixer, based on a project's declared browser support, build tooling of this kind is outside the scope of this book, which focuses on the CSS language itself.

The @supports Rule

@supports lets a stylesheet apply a block of CSS only if the browser actually supports a given property and value, providing a clean way to offer an enhanced experience without breaking browsers that lack support for it.

```

1  .gallery {
2    display: flex;
3    flex-wrap: wrap;
4  }
5  @supports (display: grid) {
6    .gallery {
7      display: grid;
8      grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
9    }
10 }
11

```

This pattern, called progressive enhancement and already introduced for HTML in Part 1 of this series, provides a reasonable Flexbox-based fallback layout first, then upgrades to Grid only in browsers that support it, rather than requiring every visitor to have the newest browser just to see a usable layout.

Graceful Degradation for Newer Features

Many modern CSS features degrade gracefully on their own, without needing `@supports` at all: a browser that doesn't recognize a property or value simply ignores that one declaration and continues applying every other rule normally. This means it is often safe to use a newer property directly, as long as the fallback behavior, usually, the property simply not applying, is an acceptable, if less polished, experience.

Conclusion

This chapter covered CSS custom properties and how they differ from preprocessor variables, the `calc()`, `min()`, `max()`, and `clamp()` functions for computing fluid values, conventions such as BEM and file organization for keeping a large stylesheet maintainable, and how to handle differences in browser support gracefully. With the full CSS toolkit covered, Chapter 12 closes the book with a hands-on project that applies it to a complete page.

Chapter 12. Hands-On Project: Styling a Complete Web Page

12.1. Planning a Visual Design from a Style Guide

This closing chapter applies every topic covered so far to a single, complete project: styling a personal portfolio landing page, structured in the HTML markup from Part 1 of this series. Before writing any CSS, it helps to plan the design deliberately, the way a real project would.

Defining a Style Guide

A small style guide, agreed on before writing CSS, keeps a project visually consistent and avoids repeatedly making the same small decisions, an exact shade of blue, a spacing value, differently in different places. For this project, the style guide is expressed directly as a block of custom properties, using the technique from Chapter 11.

```
1 :root {
2   /* Color */
3   --color-bg: #ffffff;
4   --color-text: #1f2430;
5   --color-muted: #6b7280;
6   --color-brand: #3d8bff;
7   --color-brand-dark: #1f3e73;
8
9   /* Typography */
10  --font-body: "Inter", system-ui, sans-serif;
11  --font-heading: "Inter", system-ui, sans-serif;
12
13  /* Spacing scale */
14  --space-1: 0.5rem;
15  --space-2: 1rem;
16  --space-3: 1.5rem;
17  --space-4: 2.5rem;
18
19  /* Shape */
20  --radius: 10px;
21 }
```

Every color, spacing value, and font used later in this chapter comes from this single block, following the settings-layer convention introduced in Chapter 11, changing the entire page's palette or spacing rhythm later would mean editing only these lines.

The Page Structure

The project assumes a semantic HTML skeleton already in place, built using the elements covered throughout Part 1: a `<header>` with a name and navigation, a `<main>` containing an introductory hero section and a grid of project cards, and a `<footer>` with contact links.

```
1 <body>
2   <header class="site-header">...</header>
3   <main>
4     <section class="hero">...</section>
5     <section class="projects">
6       <div class="project-grid">
7         <article class="project-card">...</article>
8         <!-- more project cards -->
9       </div>
10    </section>
11  </main>
12  <footer class="site-footer">...</footer>
13 </body>
```

With a style guide defined and the markup's structure settled, the remaining sections style each part of the page in turn.

12.2. Styling Layout, Typography, and Color

With the plan from 12.1 in place, this section builds the page's base styles and overall layout.

Base and Reset Styles

```
1 *, *::before, *::after {
2   box-sizing: border-box;
3   margin: 0;
4   padding: 0;
5 }
6 body {
7   font-family: var(--font-body);
8   color: var(--color-text);
9   background-color: var(--color-bg);
10  line-height: 1.5;
11 }
12 img {
13   max-width: 100%;
14   display: block;
15 }
```

This opening block applies border-box sizing globally (Chapter 4), zeroes out the browser's inconsistent default margins, and sets the page's base font and color from the style guide's custom properties.

The Header

```
1 .site-header {
2   display: flex;
3   align-items: center;
4   justify-content: space-between;
5   padding: var(--space-3) var(--space-4);
6 }
7 .site-header nav ul {
8   display: flex;
9   gap: var(--space-3);
10  list-style: none;
11 }
12 .site-header a {
13   color: var(--color-text);
14   text-decoration: none;
15   font-weight: 600;
16 }
17 .site-header a:hover {
18   color: var(--color-brand);
19 }
```

The header reuses the Flexbox space-between pattern from Chapter 7 to place the site name and navigation at opposite ends, and `list-style: none` removes the default bullet points from the navigation's `<ul>`.

The Hero Section

```
1 .hero {
2   padding: var(--space-4) var(--space-4) calc(var(--space-4) * 2);
3   text-align: center;
4 }
5
6 .hero h1 {
7   font-family: var(--font-heading);
8   font-size: clamp(2rem, 5vw, 3.5rem);
9   margin-bottom: var(--space-2);
10 }
11
12 .hero p {
13   color: var(--color-muted);
14   max-width: 60ch;
15   margin: 0 auto var(--space-3);
16   font-size: 1.125rem;
17 }
```

The heading's font-size uses `clamp()`, from Chapter 11, to scale fluidly with the viewport while staying within a comfortable minimum and maximum, and the introductory paragraph's `max-width: 60ch` applies the readable-line-length technique from Chapter 6.

Call-to-Action Buttons

```
1 .button {
2   display: inline-block;
3   padding: var(--space-2) var(--space-3);
4   border-radius: var(--radius);
5   background-color: var(--color-brand);
6   color: white;
7   text-decoration: none;
8   font-weight: 600;
9   transition: background-color 0.2s ease, transform 0.2s ease;
10 }
11
12 .button:hover {
13   background-color: var(--color-brand-dark);
14   transform: translateY(-2px);
15 }
```

12.3. Adding Responsive Behavior and Interactivity

With the base layout in place, this section makes the page adapt across screen sizes and adds the small interactive touches, hover effects, subtle motion, that make an interface feel finished.

A Responsive Project Grid

```
1 .project-grid {
2   display: grid;
3   grid-template-columns: repeat(auto-fit, minmax(260px, 1fr));
4   gap: var(--space-3);
5   padding: var(--space-4);
6 }
7 .project-card {
8   border: 1px solid #e5e7eb;
9   border-radius: var(--radius);
10  overflow: hidden;
11  background-color: var(--color-bg);
12  box-shadow: 0 1px 3px rgb(0 0 0 / 6%);
13  transition: transform 0.2s ease, box-shadow 0.2s ease;
14 }
15 .project-card:hover {
16   transform: translateY(-4px);
17   box-shadow: 0 12px 24px rgb(0 0 0 / 10%);
18 }
19 .project-card img {
20   aspect-ratio: 16 / 9;
21   object-fit: cover;
22 }
23 .project-card-body {
24   padding: var(--space-3);
25 }
```

The auto-fit and minmax() combination from Chapter 8 produces a fully responsive grid without a single media query: one column on a narrow phone, two or three as the viewport widens, always keeping each card at least 260px wide.

A Responsive Navigation Toggle

On narrow screens, the header's navigation collapses behind a menu button, reusing the show/hide pattern from Chapter 9.

```
1  .nav-toggle {
2    display: block;
3  }
4  .site-header nav {
5    display: none;
6  }
7
8  .site-header nav.is-open {
9    display: block;
10 }
11 @media (min-width: 768px) {
12   .nav-toggle {
13     display: none;
14   }
15   .site-header nav {
16     display: block;
17   }
18 }
19
20
21
```

The .is-open class itself is toggled by a small amount of JavaScript, the subject of Part 3 of this series; this chapter's CSS only defines what the collapsed and expanded states look like, following the same separation-of-concerns

principle from Chapter 1, behavior lives in JavaScript, presentation stays in CSS.

A Fade-In on Page Load

```
1 @keyframes fade-in-up {
2   from {
3     opacity: 0;
4     transform: translateY(12px);
5   }
6   to {
7     opacity: 1;
8     transform: translateY(0);
9   }
10 }
11
12 .hero {
13   animation: fade-in-up 0.6s ease-out;
14 }
15
16 @media (prefers-reduced-motion: reduce) {
17   .hero {
18     animation: none;
19   }
20 }
```

The hero section eases in on load using the @keyframes technique from Chapter 10, and the reduced-motion media query from that same chapter disables it entirely for readers who have requested less motion.

12.4. Testing and Refining Across Devices

A layout is not finished the moment it looks right in one browser window. This closing section walks through checking the finished page the way a real project would before shipping it.

Testing with Developer Tools

Using the device toolbar in the browser developer tools introduced in Chapter 2, check the page at several representative widths, a narrow phone (around 360px), a tablet (around

768px), and a typical laptop (around 1280px), watching for text that overflows its container, images that lose their aspect ratio, and spacing that feels cramped or excessive at the extremes.

A Pre-Launch Checklist

- Does every interactive element (links, buttons, the nav toggle) have a visible `:hover` and `:focus` state, not just a default appearance?
- Does the page remain readable and usable with the browser's text size increased, given the rem-based sizing from Chapter 9?
- Do images have appropriate alt text, carried over from the HTML in Part 1, and does every project card's aspect-ratio hold at every grid width?
- Does `prefers-reduced-motion` actually disable the hero animation, tested by enabling that setting at the operating system level?
- Is color used with enough contrast against its background to remain readable, particularly `--color-muted` text on `--color-bg`?
- Does the layout still make sense with the browser window resized to an unusual width between the tested breakpoints, not just at the breakpoints themselves?

Validating and Linting CSS

The W3C offers a CSS validator, mirroring the Markup Validator introduced for HTML in Part 1, which checks a stylesheet against the CSS specification and flags typos, unsupported values, and syntax errors that a browser might otherwise silently ignore. A linter goes a step further, also flagging style issues such as duplicate selectors or overly specific rules, and is commonly run automatically as part of a project's development workflow.

Iterating from Feedback

The final step of any real project is showing the result to someone else, a classmate, a colleague, an actual prospective user, and watching where they hesitate or get confused, rather than relying solely on personal judgment. Small friction points, a button that doesn't look clickable, text that is technically legible but uncomfortable to read for more than a sentence or two, are often invisible to the person who built the page and immediately obvious to someone seeing it for the first time.

Conclusion

This chapter brought together nearly every technique covered in this book: a style guide built from custom properties, a responsive layout combining Flexbox and Grid, fluid typography with `clamp()`, transitions and keyframe animation applied with restraint and respect for motion preferences, and a testing checklist for validating the result across devices. That is also the end of this volume: the Conclusion that follows reflects on the journey from Chapter 1 to here, and Part 3 of this series picks up where it leaves off, adding JavaScript to bring interactivity and behavior to the pages styled throughout this book.

Conclusion

A Recap of Key Concepts

This book began with a single question: what is CSS, and how does it relate to the HTML covered in Part 1 of this series? Twelve chapters later, that question has a considerably richer answer than it did at the start.

We began with the fundamentals: what CSS is, how it attaches to an HTML document, and the anatomy of a CSS rule. From there, we built outward, chapter by chapter:

- selectors, specificity, and the cascade, the mechanism that decides which rule wins when several apply to the same element;
- the box model, and the box-sizing property that makes it predictable;
- color, backgrounds, gradients, and the border-radius and box-shadow properties that shape an element's edges;
- typography: choosing and loading fonts, and the spacing properties that make text comfortable to read;
- Flexbox and CSS Grid, the two layout systems behind nearly every modern interface;
- responsive design, media queries, and the relative units that let a layout adapt to any screen;
- transitions, transforms, and keyframe animations, used with an eye toward performance and motion accessibility;
- custom properties, the calc() family of functions, and the naming and organization conventions that keep a large stylesheet maintainable;
- and finally, a complete hands-on project, applying every one of these techniques to a single page.

If a single idea runs through all twelve chapters, it is this: CSS is not a collection of unrelated tricks, it is one coherent system. The cascade from

Chapter 3 explains why a later rule wins; the box model from Chapter 4 explains why that padding value changed an element's rendered size; inheritance explains why setting a font once on `<body>` was enough. Understanding these underlying mechanisms, rather than memorizing individual properties in isolation, is what makes it possible to predict how an unfamiliar CSS rule will behave before testing it in a browser.

Next Steps in Web Development

Having built a solid foundation in styling a well-structured HTML document, the next steps most readers of this book will want to take are:

- JavaScript, for adding interactivity and dynamic behavior on top of the structure and styling covered in Parts 1 and 2, and the subject of Part 3 of this series;
- CSS preprocessors, such as Sass, SCSS, and LESS, which add variables, nesting, and other authoring conveniences on top of the plain CSS covered here;
- CSS frameworks, such as Bootstrap and Tailwind CSS, which provide pre-built components and utility classes built on the same fundamentals as this book, both preprocessors and frameworks are the subject of a later volume in this series;
- CSS architecture at scale, extending the BEM and file-organization conventions from Chapter 11 to large, multi-developer codebases;
- web accessibility in greater depth, building on the reduced-motion, contrast, and focus-state considerations touched on throughout this book.

None of these subjects require abandoning what this book has covered; each one builds directly on top of it. A preprocessor still compiles down to the selectors, box model, and cascade covered here, and a framework's components still rest on Flexbox, Grid, and the same specificity rules from Chapter 3.

Additional Resources

Appendix C, immediately following this conclusion, lists a curated set of official references, learning platforms, and tools for continuing to build on the material covered here. Appendix A provides a quick-reference table of the CSS properties used throughout

this book, and Appendix B collects the key terms introduced along the way into a single glossary.

Thank you for working through this volume. CSS continues to evolve, with new layout tools and convenience features added regularly, but the fundamentals covered here, the box model, the cascade, and the two major layout systems, remain the foundation that everything newer is built upon.

Appendices

A. CSS Property Reference

A quick-reference summary of the CSS properties used throughout this book, organized by category and cross-referenced to the section where each was introduced.

Box Model

Property	Purpose	Section
width, height	The size of an element's content (or full box, with border-box)	4.2
padding	Space between content and border	4.1
border	A line drawn around the padding	4.1
margin	Space outside the border, between elements	4.1
box-sizing	Whether width/height include padding and border	4.2
display	How an element participates in layout: block, inline, inline-block, flex, grid, none	4.3
overflow	How content larger than its box is handled	4.4
visibility	Hides an element while preserving its layout space	4.4

Color, Backgrounds, and Borders

Property	Purpose	Section
color	The text color of an element	5.1
background-color	An element's solid background color	5.1
background-image	A background image or gradient	5.2
background-position	Where a background image is placed	5.2
background-size	The rendered size of a background image	5.2
linear-gradient(), radial-gradient(), conic-gradient()	Generated color gradients, used as a background-image	5.3
border-radius	Rounds an element's corners	5.4
box-shadow	Adds one or more drop shadows around an element	5.4

Typography

Property	Purpose	Section
font-family	The typeface, as an ordered list of fallbacks	6.1
font-size	The size of text	6.1
font-weight	How bold text appears	6.1
font-style	Normal or italic text	6.1

@font-face	Defines a custom, downloadable web font	6.2
text-align	Horizontal alignment of text	6.3
text-decoration	Underlines, strikethroughs, and other text lines	6.3
text-transform	Displayed capitalization (uppercase, capitalize)	6.3
line-height	Vertical space allocated to each line of text	6.4
letter-spacing	Horizontal space between characters	6.4

Flexbox

Property	Purpose	Section
display: flex	Activates Flexbox on a container	7.1
flex-direction	Row or column main axis	7.1
flex-wrap	Whether items wrap onto multiple lines	7.1
flex-grow, flex-shrink, flex-basis (flex)	How a flex item grows or shrinks to fill space	7.2
justify-content	Distributes items along the main axis	7.3
align-items, align-self	Aligns items along the cross axis	7.3

gap	Spacing between flex (or grid) items	7.1
-----	--------------------------------------	-----

CSS Grid

Property	Purpose	Section
display: grid	Activates Grid on a container	8.1
grid-template-columns, grid-template-rows	Defines a grid's tracks	8.1
grid-column, grid-row	Places an item by grid line	8.2
grid-template-areas, grid-area	Places items by named area	8.2
fr unit / repeat()	Flexible track sizing and repeated patterns	8.3
minmax()	A track size with a minimum and maximum, often paired with auto-fit/auto-fill	8.3

Responsive Design

Property	Purpose	Section
@media	Applies CSS only under a given condition, such as a minimum viewport width	9.2
%, vw, vh, em, rem	Relative units used for fluid, adaptive sizing	9.3

object-fit	How a replaced element (image, video) fills its box	9.4
------------	---	-----

Transitions, Transforms, and Animation

Property	Purpose	Section
transition	Smoothly animates a property between two values on change	10.1
transform	Moves, scales, rotates, or skews an element without affecting layout	10.2
@keyframes / animation	Defines and plays a multi-step, optionally looping animation	10.3
will-change	Hints that an element is about to animate	10.4

Modern Features

Property	Purpose	Section
Custom properties (--name) / var()	Reusable, cascading values defined once and referenced elsewhere	11.1
calc()	Arithmetic, including across mixed units	11.2
min(), max(), clamp()	Fluid values with an optional floor and ceiling	11.2

@supports	Applies CSS only if the browser supports a given property/value	11.4
-----------	---	------

This table lists the properties this book covers directly; it is not exhaustive. The Mozilla Developer Network (MDN) CSS Reference, linked in Appendix C, documents the complete, current set of CSS properties.

B. Glossary of Terms

Key terms used throughout this book, collected here for quick reference.

Box model. The system of nested content, padding, border, and margin layers that determines an element's rendered size and spacing (Section 4.1).

box-sizing. The property that determines whether padding and border are added to, or subtracted from, a declared width and height (Section 4.2).

Breakpoint. The viewport width at which a media query changes a layout (Section 9.2).

Cascade. The set of rules CSS uses to resolve conflicts between rules of equal specificity, based on source order and origin (Section 3.4).

CSSOM. CSS Object Model; the browser's in-memory representation of every applicable stylesheet, combined with the DOM to render a page (Section 1.2).

Custom property. A reusable, author-defined value (commonly called a CSS variable), declared with a `--` prefix and read with `var()` (Section 11.1).

Easing function. The rate of change over a transition or animation's duration, such as ease or linear (Section 10.1).

Flex container. An element with `display: flex`, whose direct children become flex items arranged along a main axis (Section 7.1).

fr unit. A CSS Grid unit representing a fraction of a grid container's leftover space (Section 8.3).

Grid container. An element with `display: grid`, whose children are placed along explicitly defined rows and columns (Section 8.1).

Inheritance. The mechanism by which certain property values, mainly typographic ones, are passed from a parent element to its children unless overridden (Section 3.4).

Media query. A conditional block of CSS, introduced with `@media`, that applies only when a condition such as viewport width is met (Section 9.2).

Mobile-first design. Writing base CSS for the smallest expected screen first, then layering on enhancements for larger screens (Section 9.1).

Pseudo-class. A selector prefixed with a single colon that matches an element based on state or position, such as `:hover` or `:first-child` (Section 3.2).

Pseudo-element. A selector prefixed with a double colon that targets a specific part of an element's content, such as `::before` (Section 3.2).

Selector. The part of a CSS rule that determines which elements a declaration block applies to (Section 2.2).

Specificity. A score assigned to a selector, based on the kinds of selectors it contains, that determines which of two conflicting rules takes precedence (Section 3.4).

Style sheet. A collection of CSS rules, whether inline, internal, or in an external file (Section 1.1).

Vendor prefix. A browser-specific prefix, such as `-webkit-`, once used for experimental CSS features before standardization (Section 11.4).

Viewport. The visible area of a web page within the browser window, controlled on mobile devices by the viewport meta tag introduced in Part 1 (Section 9.1).

C. Further Reading and Online Resources

The following resources are useful both while working through this book and afterward, as CSS continues to evolve.

Official Specifications

- CSS Working Group specifications ([w3.org/Style/CSS](https://www.w3.org/Style/CSS/)), the authoritative, module-by-module CSS specifications described in Section 1.3;

- WHATWG (whatwg.org), the working group behind the HTML Living Standard referenced throughout this series.

Reference and Learning Platforms

- MDN Web Docs (developer.mozilla.org), comprehensive, example-rich documentation for every CSS property, including up-to-date browser compatibility data;
- CSS-Tricks (css-tricks.com), long-running, practical guides on Flexbox, Grid, and nearly every other CSS topic, including the widely referenced "A Complete Guide to Flexbox";
- web.dev (web.dev), Google's collection of guides on modern CSS, with a strong focus on performance and responsive design.

Tools

- W3C CSS Validation Service (jigsaw.w3.org/css-validator), validates a stylesheet against the official CSS specification, mirroring the HTML Markup Validator from Part 1;
- Can I Use (caniuse.com), up-to-date browser support tables for CSS features, referenced in Section 11.4;
- Browser developer tools, built into every major browser, covered in Section 2.4 and used throughout this book.

Accessibility and Motion

- Web Content Accessibility Guidelines, WCAG (w3.org/WAI/standards-guidelines/wcag), including color contrast requirements relevant to the color choices in Chapter 5;
- WebAIM Contrast Checker (webaim.org/resources/contrastchecker), checks text and background color combinations against WCAG contrast requirements;
- "An Introduction to Reduced Motion" and similar guides referenced from MDN, further reading on the prefers-reduced-motion media feature from Section 10.4.

As with any list of online resources, addresses and specific content may change over time; searching for the resource by name will generally lead to its current location even if a specific address above becomes outdated.