CSS MASTERY

FROM FIRST SELECTOR TO PRODUCTION-READY STYLESHEETS

- LEARN THE FUNDAMENTALS FROM THE GROUND UP
- MASTER LAYOUT, THE CASCADE & BROWSER BEHAVIOR
- BUILD REAL-WORLD PROJECTS
- WRITE CLEAN, MAINTAINABLE, PERFORMANT CSS

RYAN BROOKS

CSS Mastery

From First Selector to Production-Ready Stylesheets

Steve Publications

This book is available at https://leanpub.com/cssmastery

This version was published on 2026-08-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

From First Selector to Production-Ready Stylesheets

This book takes you from zero CSS knowledge to professional-level mastery. You will learn not only the syntax and features of Cascading Style Sheets, but how browsers interpret them, why certain patterns work better than others, how to reason about layout and cascade behavior systematically, and how to write clean, maintainable, performant CSS for production websites. Every concept is built progressively with complete working examples that combine into real-world projects.

Introduction

Open any website in your browser and look at what you see: colored backgrounds, carefully spaced text, images arranged in grids, buttons that change color when you hover over them, smooth animations as menus slide open. None of this comes from HTML alone. Every visual decision you are looking at is controlled by CSS.

Cascading Style Sheets are the language browsers use to translate your markup into something visually compelling. They determine where elements appear on the page, how large they are, what colors and fonts they use, how they respond when the screen size changes or a user interacts with them. CSS is also responsible for making layouts responsive so that websites work equally well on phones, tablets, and desktop monitors.

This book treats CSS as more than a collection of properties to memorize. You will learn the underlying mechanisms: how browsers calculate styles, how the cascade resolves conflicts between competing rules, how layout algorithms position elements, and how your CSS choices affect performance. This deeper understanding is what separates developers who can style something from developers who can debug any styling problem, reason about complex layouts, and write CSS that stays maintainable as projects grow.

The book is organized into parts that build on each other. We start with the fundamentals of syntax, selectors, and the cascade. Then we move through visual styling like colors, typography, and backgrounds. The core layout chapters cover Flexbox, Grid, positioning, and responsive design. Later sections explore modern features like custom properties, container queries, animations, and cascade layers. Finally we examine browser rendering, developer tools, accessibility considerations, and architectural patterns for large codebases.

Every chapter includes complete working examples you can run in your own browser. Code is annotated with explanations of what each rule does and why it matters. Where choices exist between approaches, I explain the trade-offs so you can decide what fits your situation.

If you already know HTML basics such as elements, attributes, tags and document structure, you are ready to begin. If not, spend a short time reviewing those concepts first, then come back. Let us start with understanding what CSS actually is and how it works under the hood.

Chapter 1: What Is CSS and How Does It Work?

The Role of CSS in Web Pages

HTML provides structure. CSS provides presentation. This division of concerns is one of the foundational principles of web design, and it matters more than you might initially realize.

Consider a simple HTML document with no CSS at all:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Plain HTML</title>
</head>
<body>
  <h1>Welcome to My Site</h1>
  <p>This is a paragraph of text. Without CSS, it looks like this in every
  ↪  browser.</p>
  <ul>
    <li>Item one</li>
    <li>Item two</li>
    <li>Item three</li>
  </ul>
  <a href="#">A link</a>
</body>
</html>
```

Open this in any browser and you will see the same basic appearance: a heading, some text, a bulleted list, and a blue underlined link. The browser applies its own default styles known as the user-agent stylesheet. These defaults are intentionally minimal because HTML was originally designed for documents, not visual design.

Now imagine adding CSS to that same HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Styled with CSS</title>
  <style>
    body {
      font-family: system-ui, sans-serif;
      line-height: 1.6;
      max-width: 600px;
      margin: 2rem auto;
      color: #333;
    }

    h1 {
      color: #1a1a2e;
      font-size: 2rem;
      border-bottom: 3px solid #e94560;
      padding-bottom: 0.5rem;
    }

    ul {
      background: #f8f9fa;
      padding: 1rem 1rem 1rem 2rem;
      border-radius: 8px;
    }

    a {
      color: #e94560;
      text-decoration: none;
      font-weight: bold;
    }

    a:hover {
      text-decoration: underline;
    }
  </style>
</head>
<body>
  <h1>Welcome to My Site</h1>
  <p>This is a paragraph of text. Without CSS, it looks like this in every
  ↪  browser.</p>
  <ul>
    <li>Item one</li>
    <li>Item two</li>
    <li>Item three</li>
  </ul>
  <a href="#">A link</a>
</body>
</html>
```

The HTML is identical. The CSS transforms a plain document into something with personality, hierarchy, and visual clarity. You now have a custom font choice, controlled line spacing, a constrained content width centered on the page, color accents, rounded corners, and an interactive hover effect on the link. All of this without changing a single character of HTML.

This separation is powerful for several reasons:

- **Reusability**: One stylesheet can style hundreds of pages consistently.
- **Maintainability**: Change a color in one place and it updates everywhere that uses it.
- **Flexibility**: The same HTML content can be styled differently for different contexts such as mobile versus desktop, or print versus screen.
- **Accessibility**: CSS can enhance readability through spacing and contrast without altering the document structure that assistive technologies rely on.

CSS does not replace HTML. It complements it. Good web design uses semantic HTML to describe what content is, then uses CSS to control how it looks. When you separate structure from presentation cleanly, both become easier to work with.

How Browsers Read and Apply Stylesheets

Understanding the browser rendering process helps you write better CSS because you start to see your code through the lens of the machine that interprets it.

When a browser loads a webpage, it follows a pipeline:

1. **HTML parsing**: The browser reads your HTML file and builds a Document Object Model (DOM), which is a tree representation of all elements on the page.
2. **CSS parsing**: Simultaneously, the browser fetches and parses any CSS files referenced by the HTML, building a CSS Object Model (CSSOM). This is a tree of all style rules organized for quick lookup.
3. **Render tree construction**: The DOM and CSSOM are combined into a render tree that contains only visible elements along with their computed styles. Elements set to `display: none` do not appear in the render tree because they are invisible.

4. **Layout (reflow)**: The browser calculates the exact position and size of every element in the render tree based on CSS layout rules, viewport dimensions, and the box model.
5. **Paint**: The browser fills in pixels for each element: text colors, background colors, borders, images, shadows.
6. **Composite**: If elements exist on different layers (for example due to `z-index`, `transform`, or `opacity`), the browser composites those layers into the final image you see on screen.

This pipeline matters because CSS choices affect each stage. A large stylesheet slows down parsing. Complex selectors slow down matching. Properties that change element dimensions trigger expensive layout recalculation when they change dynamically. Understanding this process lets you write CSS that is not only correct but efficient.

The browser starts rendering as soon as it has enough information, typically painting from the top of the page downward. This is why CSS in the `<head>` is important: if the browser encounters a `<style>` tag or `<link>` to a stylesheet while parsing HTML, it pauses HTML parsing until the CSS is downloaded and processed because styles can affect how elements are rendered. This makes CSS "render-blocking." Placing CSS in the head ensures visual consistency as the page loads rather than having content appear unstyled and then suddenly jump into position once CSS arrives.

Syntax: Rules, Selectors, Declarations, Properties, Values

CSS syntax is deceptively simple. A single rule consists of two parts: a selector and a declaration block.

```
selector {
  property: value;
}
```

- **Selector**: Specifies which HTML element or elements the rule applies to.
- **Declaration block**: Contains one or more declarations enclosed in curly braces `{}`.
- **Declaration**: A single `property: value` pair ending with a semicolon `;`.

- **Property**: The CSS feature you want to control (such as `color`, `font-size`, `margin`).
- **Value**: What you want that property to be (such as `red`, `16px`, `20px`).

Here is a real example:

```
h1 {
  color: #1a1a2e;
  font-size: 2rem;
}
```

The selector `h1` targets all heading level one elements. The declaration block contains two declarations: `color` set to the hex value `#1a1a2e`, and `font-size` set to `2rem`. Every `<h1>` element in the document will have dark text at twice the root font size.

Selectors can target multiple elements by separating them with commas:

```
h1, h2, h3 {
  color: #1a1a2e;
}
```

This single rule applies the same color to all three heading levels. It is equivalent to writing three separate rules but more concise.

Values in CSS come in many forms:

- **Keywords**: Predefined words like `none`, `auto`, `bold`, `center`.
- **Lengths**: Numbers with units like `16px`, `2rem`, `10em`, `50vw`.
- **Colors**: Formats like `red`, `#ff0000`, `rgb(255, 0, 0)`, `hsl(0, 100%, 50%)`.
- **Percentages**: Relative values like `50%` or `100vh`.
- **Functions**: Calculated values like `calc(100% - 20px)` or `clamp(1rem, 2vw + 1rem, 3rem)`.

Each property accepts only certain types of values. Setting `color: 20px` is invalid because color expects a color value, not a length. Browsers simply ignore invalid declarations without throwing errors, which is why careful attention to syntax and understanding what each property accepts matters.

A complete CSS rule with multiple declarations:

```css
.card {
  background-color: #ffffff;
  border: 1px solid #e0e0e0;
  border-radius: 8px;
  padding: 1.5rem;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}
```

The selector `.card` targets any element with `class="card"`. The declarations set a white background, a thin gray border, rounded corners, internal spacing, and a subtle shadow. Every element matching this selector receives all five of these styles.

Attaching CSS: Inline, Internal, External Stylesheets

You can attach CSS to an HTML document in three ways, each with different use cases and trade-offs.

External stylesheet: A separate `.css` file linked from the HTML using a `<link>` element in the head:

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

This is by far the most common approach for real projects. The CSS file can be shared across multiple pages, cached by browsers to speed up subsequent page loads, and kept separate from HTML markup for cleaner organization. For anything beyond a single-page demo or experiment, use external stylesheets.

Internal stylesheet: CSS placed directly in the HTML document inside a `<style>` element:

```
<head>
  <style>
    body {
      font-family: system-ui, sans-serif;
    }
  </style>
</head>
```

Internal styles are useful for page-specific CSS that does not apply elsewhere, or for embedding small amounts of critical CSS directly in the HTML to speed up initial rendering. They keep everything in one file during prototyping but become unwieldy as projects grow.

Inline styles: CSS applied directly to an individual element using the `style` attribute:

```
<p style="color: red; font-weight: bold;">This text is styled inline.</p>
```

Inline styles have the highest specificity in normal CSS, which means they override almost all other rules targeting that element. This makes them powerful for one-off overrides but terrible for maintainability because styles are scattered across HTML files instead of centralized. Avoid inline styles in production code unless you have a specific reason such as dynamically generating styles via JavaScript where no alternative exists.

Order of loading: When multiple stylesheets are present, they are applied in the order they appear in the HTML. If two rules target the same element with equal specificity, the one that appears later wins. This is called source order and it is one factor in how the cascade resolves conflicts:

```
<head>
  <link rel="stylesheet" href="base.css">
  <link rel="stylesheet" href="components.css">
  <link rel="stylesheet" href="overrides.css">
</head>
```

Here, `overrides.css` is loaded last. If it contains a rule that conflicts with an equally specific rule in `base.css`, the override wins simply because it appears later in source order. This ordering can be used intentionally to structure stylesheets, though modern CSS offers better tools for managing precedence as we will see later.

Best practice summary: Use external stylesheets for all production work. Use internal styles sparingly for page-specific or critical CSS. Avoid inline styles except in rare cases where they are genuinely the only practical option.

The Cascade: Order, Specificity, and Importance

The word "cascading" in Cascading Style Sheets refers to the algorithm that determines which style wins when multiple rules apply to the same element. This is the single most important concept in CSS because misunderstanding the cascade leads to confusing behavior, specificity wars, and fragile stylesheets full of `!important` hacks.

The cascade does not simply take the last rule it sees. It follows a defined priority system with several stages:

1. **Relevance**: The browser first filters out rules that do not apply due to conditions like media queries or feature queries. A rule inside `@media (min-width: 768px)` is irrelevant on a 400-pixel-wide screen, so it does not participate in the cascade at all.
2. **Origin and importance**: CSS declarations come from different origins with different priority levels. From lowest to highest precedence:

   - User-agent normal (browser defaults)
   - User normal (user's custom styles)
   - Author normal (your stylesheets)
   - Animations (`@keyframes`)
   - Author important (`!important` in your CSS)
   - User important (`!important` in user styles)
   - User-agent important (critical browser defaults)

   Within the same origin, declarations marked with `!important` take precedence over normal declarations. For example, `color: red !important` overrides `color: blue` even if the blue rule has higher specificity or appears later. However, overusing `!important` creates brittle styles that are difficult to override predictably. Use it only when necessary and understand why you need it before reaching for it.
3. **Cascade layers**: If cascade layers (`@layer`) are used, declarations in later-defined layers take precedence over earlier layers regardless of specificity. This is a modern feature designed specifically to give authors explicit control over style priority without relying on specificity tricks. We will cover this in depth in Chapter 14.

4. **Specificity**: When two rules have the same origin and importance level, the browser compares their specificity. The selector with higher specificity wins. Specificity is calculated based on what types of selectors are used: ID selectors contribute more weight than class selectors, which contribute more than element type selectors. We will explore this calculation in detail in Chapter 3.
5. **Scoping proximity**: If `@scope` rules are present, declarations from the scope closest to the target element take precedence. This is a newer feature for localized styling.
6. **Source order**: When everything else is equal (same origin, same importance, same layer, same specificity), the rule that appears later in the CSS wins. This is why moving a rule to the bottom of your stylesheet sometimes "fixes" a styling problem: it does not actually fix anything, it just changes which rule appears last.

How this plays out in practice: Consider this example:

```
/* Rule 1: element selector */
p {
  color: blue;
}

/* Rule 2: class selector (higher specificity) */
.text-primary {
  color: red;
}

/* Rule 3: same as rule 2 but appears later */
.text-secondary {
  color: green;
}
```

For an element `<p class="text-primary text-secondary">`:

- Rule 1 applies because it targets `p` elements.
- Rules 2 and 3 also apply because the element has both classes.
- Rules 2 and 3 have higher specificity than Rule 1, so blue is eliminated.
- Between Rules 2 and 3, they have equal specificity (both are single class selectors).
- Rule 3 appears later in source order, so green wins.

The final text color is green.

Why the cascade matters: The cascade is not a bug or an accident. It is the core design of CSS that allows styles to be layered and overridden predictably. A well-designed stylesheet leverages the cascade intentionally: base styles are defined broadly with low-specificity selectors, then progressively refined by more specific rules where needed. When you fight the cascade instead of working with it, your code becomes harder to understand and maintain.

Understanding how the cascade works gives you a mental model for predicting what any given CSS will do. Instead of guessing which rule wins or resorting to trial and error, you can reason about conflicts systematically. This is the foundation everything else in this book builds on.

Chapter 2: Selectors: Targeting Elements with Precision

Type, Class, ID, and Universal Selectors

Selectors are the targeting mechanism of CSS. They answer the question "which elements should this rule apply to?" Mastering selectors means being able to precisely target exactly what you need without accidentally styling the wrong things or writing overly complex rules.

The four fundamental selector types form the building blocks for everything else:

Type (element) selector: Matches all elements of a given tag name.

```
h1 { color: navy; }
p { line-height: 1.6; }
a { text-decoration: none; }
```

Type selectors are simple and have low specificity, which makes them good for applying broad base styles to all instances of an element. Use them when you genuinely want every `<h1>` or every `<a>` on the page to share the same styling. However, be cautious with type selectors in complex layouts because they can unintentionally affect elements nested inside components where different behavior is needed.

Class selector: Matches any element that has a specific class attribute value. Written with a dot prefix.

```
.btn {
  padding: 0.5rem 1rem;
  border-radius: 4px;
}

.alert-error {
  background-color: #fee2e2;
  color: #991b1b;
}
```

Classes are the workhorse of CSS selectors. They let you apply styles based on semantic meaning rather than element type or position in the document. A class name like `.btn` can style a `<button>`, an `<a>`, or any other element that should look and behave like a button. This flexibility is why classes are preferred over IDs or type selectors for most styling needs.

Classes also support multiple values on a single element:

```
<button class="btn btn-primary btn-large">Click me</button>
```

Each class can contribute its own styles, which combine through the cascade. Here, `.btn` provides base button styling, `.btn-primary` adds a primary color theme, and `.btn-large` increases the size. This composable approach is fundamental to scalable CSS architecture.

ID selector: Matches the single element with a specific ID attribute value. Written with a hash prefix.

```
#main-navigation {
  position: sticky;
  top: 0;
}

#site-footer {
  margin-top: 4rem;
}
```

IDs must be unique within a document; only one element can have `id="main-navigation"`. They carry high specificity, which creates problems when styles need to be overridden later. For this reason, modern CSS practice generally recommends avoiding IDs for styling and reserving them for JavaScript hooks and fragment identifiers (URL anchors). Use classes instead for visual styling; they are more flexible and easier to work with in the cascade.

Universal selector: Matches every element on the page. Written as an asterisk.

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

The universal selector is most commonly seen in CSS resets and normalization styles that establish a consistent baseline across browsers. It has the lowest possible specificity, which makes it safe to use for broad defaults because any more specific rule will override it. However, * selectors can have performance implications on very large documents because the browser must evaluate them against every element. Use them sparingly and understand what they are doing.

Combining selector types: You can chain selectors together to create more specific targeting:

```
/* Type + class: only <p> elements with class "lead" */
p.lead {
  font-size: 1.25rem;
}

/* This would NOT match a <div class="lead"> */
```

Here, `p.lead` is more specific than either `p` alone or `.lead` alone because it requires both conditions to be true. The element must be a paragraph AND have the lead class.

Combinators: Descendant, Child, Adjacent, General Sibling

Combinators let you select elements based on their relationship to other elements in the document tree. They connect two or more simple selectors and define how those elements relate to each other.

Descendant combinator: A space between two selectors matches any element that is nested anywhere inside another element.

```
nav a {
  color: white;
}

article p {
  line-height: 1.8;
}
```

The rule `nav a` matches every `<a>` element that is a descendant of a `<nav>`, whether it is a direct child or nested several levels deep:

```
<nav>
  <a href="#">Direct child - matched</a>
  <ul>
    <li><a href="#">Grandchild - also matched</a></li>
  </ul>
</nav>
```

Descendant selectors are powerful but can be overly broad. If you have deeply nested content, a descendant selector might match elements you did not intend to style. Be explicit about when broad matching is what you actually want versus when you need more precise targeting.

Child combinator: The `>` symbol matches only direct children, not deeper descendants.

```
ul > li {
  list-style-type: square;
}

.container > .card {
  margin: 1rem 0;
}
```

The rule `ul > li` matches `<li>` elements that are immediate children of a `<ul>`, but not `<li>` elements nested inside other lists:

```
<ul>
  <li>Direct child  -  matched</li>
  <li>
    Nested item
    <ul>
      <li>Grandchild  -  NOT matched by ul > li</li>
    </ul>
  </li>
</ul>
```

Child selectors are more predictable than descendant selectors because they have a clear boundary. Use them when you want to style only the first level of nesting and avoid accidentally affecting deeper levels.

Adjacent sibling combinator: The `+` symbol matches an element that immediately follows another element as a sibling (same parent).

```
h2 + p {
  font-style: italic;
}

.input-error + .error-message {
  color: red;
}
```

The rule `h2 + p` matches only the first `<p>` that comes immediately after an `<h2>` with no other elements between them:

```
<h2>Chapter One</h2>
<p>This paragraph is matched  -  it follows h2 directly.</p>
<p>This paragraph is NOT matched  -  another p is in between.</p>
```

Adjacent sibling selectors are useful for styling patterns where one element's presence affects the next element, such as showing an error message right after a form field that has an error.

General sibling combinator: The `~` symbol matches all elements that follow another element as siblings, not just the immediate next one.

```
h2 ~ p {
  color: #555;
}

.checkbox:checked ~ .options-panel {
  display: block;
}
```

The rule `h2 ~ p` matches every `<p>` that comes after an `<h2>` as a sibling, regardless of how many elements are between them:

```
<h2>Chapter One</h2>
<p>Matched  -  follows h2.</p>
<div>A div in between.</div>
<p>Also matched  -  still a sibling after h2.</p>
```

General sibling selectors are particularly powerful for creating interactive components without JavaScript. The checkbox example shows a common pattern: when a hidden checkbox is checked, all following `.options-panel` elements become visible. This works because `:checked` is a pseudo-class that reflects the checkbox state, and `~` targets siblings that come after it in the markup.

Combinator specificity: Combinators themselves do not add to specificity. Only the selectors they connect contribute weight. Both `nav a` and `nav > a` have the same specificity because the space and `>` symbols are neutral; what matters is that both use one type selector (`nav`) and one type selector (`a`).

Practical example combining multiple selectors:

```
/* Base styles for all buttons */
.btn {
  padding: 0.5rem 1rem;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

/* Primary variant */
.btn-primary {
  background-color: #2563eb;
  color: white;
}

```

```
/* Secondary variant */
.btn-secondary {
  background-color: #e5e7eb;
  color: #1f2937;
}

/* Only buttons inside a form footer get extra spacing */
.form-footer > .btn {
  margin-left: 0.5rem;
}

/* The first button in any group has no left margin */
.btn-group > .btn:first-child {
  margin-left: 0;
}
```

This example shows how different selector types and combinators work together to create a flexible button system with base styles, variants, and contextual overrides.

Attribute Selectors

Attribute selectors match elements based on the presence or value of HTML attributes. They provide precise targeting without requiring additional classes and are especially useful for styling form inputs, links, and data-driven content.

Presence selector: Matches any element that has a given attribute, regardless of its value.

```
input[required] {
  border-left: 3px solid #ef4444;
}

a[href] {
  color: blue;
}
```

The rule `input[required]` styles any input field that has the required attribute, making it visually clear which fields must be filled in. This is more maintainable than adding a separate class because the styling automatically reflects the actual form validation state.

Exact value match: Matches elements where the attribute equals a specific value exactly.

```
input[type="text"] {
  padding: 0.5rem;
}

a[target="_blank"] {
  text-decoration: underline wavy;
}

[data-theme="dark"] body {
  background-color: #111827;
}
```

The rule `input[type="text"]` targets only text inputs, leaving password fields, email fields, and other input types unaffected. This is useful for applying different styles to different input kinds without adding extra markup.

Partial value matches: Several operators let you match substrings within attribute values:

- `[attr^="value"]`: Starts with the specified value.
- `[attr$="value"]`: Ends with the specified value.
- `[attr*="value"]`: Contains the specified value anywhere.

```
/* Links starting with https get a secure indicator */
a[href^="https://"]::after {
  content: " 🔒";
}

/* Style all image file links */
a[href$=".png"],
a[href$=".jpg"],
a[href$=".gif"] {
  font-weight: bold;
}

/* Style any element with "active" anywhere in its data-state */
[data-state*="active"] {
  background-color: #dbeafe;
}
```

These partial match selectors are powerful for patterns where attribute values follow predictable conventions. For example, you might use `[class*="btn-"]` to apply shared styles to all button variants, though using a common parent class is usually cleaner.

Word match: The `[attr~="value"]` operator matches when the attribute value contains a specific word separated by spaces.

```
/* Matches lang="en", lang="en-US", lang="fr en" */
[lang~="en"] {
  font-family: "Georgia", serif;
}
```

This is most commonly used with the `class` or `lang` attributes where multiple space-separated values are valid. However, for classes specifically, a regular class selector `.classname` is simpler and more readable than `[class~="classname"]`.

Hyphen match: The `[attr|="value"]` operator matches when the attribute value equals the specified value or starts with that value followed by a hyphen.

```
/* Matches lang="en", lang="en-US", lang="en-GB" */
[lang|="en"] {
  direction: ltr;
}
```

This is particularly useful for language attributes where you want to target all English variants with a single selector.

Case sensitivity: Attribute selectors are case-sensitive by default for most HTML attributes. However, the `i` flag makes matching case-insensitive:

```
/* Case-insensitive match for data-type */
[data-type="warning" i] {
  background-color: #fef3c7;
}
```

This is useful when attribute values might be entered with inconsistent casing.

Attribute selectors and specificity: Each attribute selector contributes to specificity the same way a class selector does (0-1-0 in the three-column system). So `[type="text"]` has the same specificity as `.classname`, which is higher than a type selector like `input`.

Pseudo-Classes: State-Based Selection

Pseudo-classes select elements based on their state or position rather than their markup. They are written with a colon prefix and are essential for creating interactive interfaces and targeting structural relationships.

User action pseudo-classes: These respond to how the user interacts with an element.

```
/* Link states */
a:link { color: blue; }       /* Unvisited link */
a:visited { color: purple; }  /* Visited link */
a:hover { color: red; }       /* Mouse pointer over element */
a:active { color: darkred; }  /* Element being clicked */
a:focus { outline: 2px solid blue; } /* Element has keyboard focus */

/* Form elements */
input:focus {
  border-color: #2563eb;
  box-shadow: 0 0 0 3px rgba(37, 99, 235, 0.2);
}

button:disabled {
  opacity: 0.5;
  cursor: not-allowed;
}

input:checked + label {
  font-weight: bold;
}
```

The order `:link`, `:visited`, `:hover`, `:active` matters when styling links because of the cascade. A common mnemonic is "LoVe HAte": Link, Visited, Hover, Active. If you put `:hover` before `:link` in your CSS, the link rule will override hover styles for unvisited links because it appears later with equal specificity.

The `:focus` pseudo-class is critical for accessibility. Keyboard users navigate through focusable elements and need visible indication of which element currently has focus. Never remove focus outlines without providing an alternative visual indicator. The default browser outline exists for a reason.

Structural pseudo-classes: These target elements based on their position in the document tree.

```css
/* First and last children */
li:first-child { font-weight: bold; }
li:last-child { border-bottom: none; }

/* Only child (element is the sole child of its parent) */
p:only-child { text-align: center; }

/* Nth child patterns */
tr:nth-child(even) { background-color: #f9fafb; }  /* Zebra striping */
li:nth-child(3n) { color: red; }                   /* Every third item */
li:nth-last-child(2) { font-style: italic; }       /* Second from end */

/* First of a specific type */
p:first-of-type { margin-top: 0; }
img:last-of-type { display: block; margin: 1rem auto; }
```

The `:nth-child()` function accepts several patterns:

- A number like `3` matches only the third child.
- The keyword `even` matches every even-numbered child (2, 4, 6...).
- The keyword `odd` matches every odd-numbered child (1, 3, 5...).
- The formula `an+b` where `a` and `b` are numbers: `3n+1` matches the 1st, 4th, 7th elements; `2n` is equivalent to `even`.

Important distinction: `:nth-child()` counts all children regardless of type, while `:nth-of-type()` counts only siblings of the same element type. Consider this HTML:

```html
<div>
  <p>Paragraph 1</p>
  <span>Span</span>
  <p>Paragraph 2</p>
</div>
```

- `p:nth-child(2)` matches nothing because the second child is a span, not a paragraph.
- `p:nth-of-type(2)` matches "Paragraph 2" because it is the second paragraph.

Negation pseudo-class: The `:not()` pseudo-class excludes elements that match its argument.

```css
/* All buttons except the one with class "ghost" */
button:not(.ghost) {
  background-color: #2563eb;
}

/* All paragraphs except the first one in each section */
p:not(:first-of-type) {
  margin-top: 1rem;
}

/* All inputs except submit buttons */
input:not([type="submit"]) {
  width: 100%;
}
```

The specificity of `:not()` itself is zero. Only the selector inside contributes to specificity. So `button:not(.ghost)` has the same specificity as `button.ghost`: one type selector and one class selector.

Other useful pseudo-classes:

```css
/* Empty elements (no children at all) */
div:empty { display: none; }

/* Elements being hovered by pointer (broader than :hover) */
.card:hover { transform: translateY(-2px); }

/* Read-only vs read-write form controls */
input:read-only { background-color: #f3f4f6; }
input:read-write { background-color: white; }

/* Valid and invalid inputs based on HTML validation */
input:valid { border-color: #10b981; }
input:invalid { border-color: #ef4444; }

/* Language targeting */
:lang(fr) { font-family: "Georgia", serif; }
```

Pseudo-classes are one of CSS's most powerful features because they let you style based on behavior and structure without modifying HTML. A well-designed interface uses pseudo-classes extensively for hover states, focus indicators, form validation feedback, and structural patterns like zebra-striped tables or highlighted first items.

Pseudo-Elements: Styling Parts of Elements

Pseudo-elements let you style specific parts of an element or insert generated content without adding extra HTML markup. They are written with a double colon prefix (`::`) to distinguish them from pseudo-classes, though single-colon syntax is still supported for historical pseudo-elements like `:before` and `:after`.

Generated content with ::before and ::after: These create anonymous elements before or after an element's content. They require a `content` property or they do not render.

```
/* Add an icon before external links */
a[href^="https://"]:not([href*="yoursite.com"])::after {
  content: " ↗";
  font-size: 0.8em;
}

/* Create a decorative quote mark */
blockquote::before {
  content: """;
  font-size: 4rem;
  color: #e5e7eb;
  line-height: 0;
  vertical-align: -0.5rem;
}

/* Tooltip pattern */
.tooltip::after {
  content: attr(data-tooltip);
  position: absolute;
  bottom: 100%;
  left: 50%;
  transform: translateX(-50%);
  background: #1f2937;
  color: white;
  padding: 0.25rem 0.5rem;
  border-radius: 4px;
  font-size: 0.8rem;
  white-space: nowrap;
  opacity: 0;
  pointer-events: none;
  transition: opacity 0.2s;
}

.tooltip:hover::after {
```

```
  opacity: 1;
}
```

The `attr()` function inside `content` pulls a value from an HTML attribute. In the tooltip example, `data-tooltip="Helpful text"` on the element becomes the tooltip content. This pattern lets you add tooltips without extra markup or JavaScript.

Important characteristics of `::before` and `::after`:

- They are inline by default but can be changed to block, flex, grid, etc.
- They participate in layout like regular elements; they take up space and can have margins, padding, borders.
- They cannot contain other elements or text nodes directly (only the content you specify).
- They are useful for decorative elements, icons, arrows, clearing floats, and visual effects.

Text-related pseudo-elements: These target specific parts of text content.

```
/* First letter of an element (drop cap effect) */
article p:first-of-type::first-letter {
  font-size: 3rem;
  float: left;
  margin-right: 0.1em;
  line-height: 1;
}

/* First line of an element */
.intro::first-line {
  font-weight: bold;
  color: #1f2937;
}

/* Selection highlight styling */
::selection {
  background-color: #fef08a;
  color: #1f2937;
}
```

The `::first-letter` pseudo-element applies only to block-level containers and affects the first letter of text content, ignoring any leading whitespace or generated content. It is commonly used for drop caps in editorial design.

The `::selection` pseudo-element styles the appearance of highlighted text when a user selects it with their mouse or keyboard. Only `background-color` and `color` are reliably supported across browsers.

Other pseudo-elements:

```
/* Placeholder text in form inputs */
::placeholder {
  color: #9ca3af;
  font-style: italic;
}

/* File input button styling (limited browser support) */
::-webkit-file-upload-button {
  background-color: #2563eb;
  color: white;
  border: none;
  padding: 0.5rem 1rem;
  border-radius: 4px;
}

/* Scrollbar styling (vendor-prefixed, non-standard) */
::-webkit-scrollbar {
  width: 8px;
}

::-webkit-scrollbar-thumb {
  background-color: #d1d5db;
  border-radius: 4px;
}
```

Note that scrollbar pseudo-elements and some form control pseudo-elements use vendor prefixes (`-webkit-`) because they are not part of the standard specification. Support varies across browsers, so test carefully if you use them.

Pseudo-element specificity: Each pseudo-element adds to specificity like a type selector (0-0-1). So `p::before` has higher specificity than `p` alone but lower specificity than `.classname`.

Modern Selectors: :is(), :where(), :not(), :has()

CSS Selectors Level 4 introduced several powerful new selectors that solve real problems in complex stylesheets. Understanding these modern tools lets you write cleaner, more maintainable CSS.

The :is() pseudo-class: Matches any element that matches at least one selector in its argument list. It is also known as the "matches" or ":any" selector.

```
/* Apply the same style to multiple heading levels */
:is(h1, h2, h3, h4) {
  font-family: "Georgia", serif;
  color: #1f2937;
}

/* Style any focusable element */
:is(a, button, input, textarea, select, [tabindex]):focus {
  outline: 2px solid #2563eb;
  outline-offset: 2px;
}
```

Before `:is()`, you would need to repeat the declarations for each selector or list all selectors at the start of a rule. With `:is()`, you group them cleanly in one place.

The specificity of `:is()` equals the specificity of its most specific argument. So `:is(h1, .title, #main)` has the same specificity as `#main` alone (the ID selector). This behavior is intentional but can be surprising: if you mix low and high specificity selectors inside `:is()`, the whole thing takes on the highest weight.

The :where() pseudo-class: Works exactly like `:is()` but always has zero specificity regardless of its arguments.

```
/* These rules have specificity 0-0-0, easy to override */
:where(h1, h2, h3) {
  margin-bottom: 1rem;
}

:where(.card, .panel, .modal) {
  border-radius: 8px;
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
}
```

Use `:where()` when you want to group selectors without accidentally creating high-specificity rules that are hard to override later. It is particularly useful in base stylesheets, resets, and utility frameworks where rules should be easily overridden by more specific styles.

Comparison: `:is(h1, .title)` has specificity equal to `.title` (0-1-0). `:where(h1, .title)` always has specificity 0-0-0. Choose based on whether

you want the grouped selectors to carry their natural weight or be effectively neutral.

The :has() pseudo-class: This is the long-awaited "parent selector." It matches an element if it contains at least one element matching the relative selector list in its argument.

```
/* Style a card differently when it contains an image */
.card:has(> img) {
  padding-top: 0;
}

/* Highlight a form field's label when the input has an error */
.form-group:has(input:invalid) label {
  color: #ef4444;
}

/* Style a list item differently if it contains a nested list */
li:has(> ul) {
  font-weight: bold;
}

/* Style a section when it has an active tab */
.tabs:has(.tab[aria-selected="true"]) {
  border-bottom: 2px solid #2563eb;
}
```

Before `:has()`, styling a parent based on its children required JavaScript. You could not say "style this div if it contains an image" or "change the label color if the input is invalid." The `:has()` selector closes this gap and enables powerful pure-CSS patterns for form validation feedback, conditional component states, and dynamic layouts.

Important notes about `:has()`:

- It can also target previous siblings using combinators inside its argument: `div:has(+ p.error)` matches a div that is immediately followed by a paragraph with class error.
- The specificity of `:has()` equals the specificity of its most specific inner selector, similar to `:is()`.
- Performance: Because `:has()` requires the browser to check descendants or siblings when evaluating a rule, it can be more expensive than simple selectors. Avoid using broad `:has()` rules on large documents or in frequently updated sections of the page.

- Browser support is now excellent across all major browsers as of 2024, making it safe for production use with fallbacks for very old browsers if needed.

Practical example combining modern selectors:

```
/* Base card styles  -  low specificity via :where() */
:where(.card) {
  background: white;
  border-radius: 8px;
  padding: 1.5rem;
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
}

/* Cards with images get different layout */
.card:has(> .card-image) {
  padding: 0;
  overflow: hidden;
}

.card:has(> .card-image) > .card-image {
  width: 100%;
  height: 200px;
  object-fit: cover;
}

.card:has(> .card-image) > .card-content {
  padding: 1.5rem;
}

/* Highlight cards that are featured */
:is(.card-featured, [data-featured="true"]) {
  border: 2px solid #fbbf24;
}

/* Hover effect on any card with a link */
.card:has(> a) {
  cursor: pointer;
  transition: transform 0.2s ease;
}

.card:has(> a):hover {
  transform: translateY(-2px);
}
```

This example demonstrates how modern selectors enable expressive, maintainable component styles without JavaScript. The card adapts its layout based

on whether it contains an image, highlights when featured, and shows hover effects only when clickable: all through CSS alone.

Browser support notes: All the modern selectors discussed here (`:is()`, `:where()`, `:not()` with multiple arguments, `:has()`) are supported in Chrome 105+, Firefox 121+, Safari 15.4+, and Edge 105+. For projects that must support older browsers, use feature detection with `@supports selector(:has(.test))` to conditionally apply rules that depend on these selectors.

Chapter 3: The Cascade Explained: Specificity, Inheritance, and Source Order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

How the Cascade Resolves Conflicts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Calculating Specificity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Inheritance: What Inherits and What Does Not

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Source Order Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Importance and !important

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Initial, Inherited, Unset, Revert Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 4: The Box Model: Everything Is a Box

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Understanding Boxes: Content, Padding, Border, Margin

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Width and Height: How They Really Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

The box-sizing Property

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Margins: Collapse, Negative Values, and Edge Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Overflow: Controlling Content That Does Not Fit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Borders and Rounded Corners

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 5: Typography: Working with Text

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Font Families and the Web Safe Font Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Font Size Units and Scaling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Font Weight, Style, and Variant

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Line Height, Letter Spacing, and Word Spacing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Text Alignment, Decoration, and Transformation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Web Fonts: @font-face and System Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 6: Colors, Backgrounds, and Visual Surfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Color Formats: Named, Hex, RGB, HSL, OKLCH

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Background Colors and Images

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Multiple Backgrounds and Layering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Gradients: Linear and Radial

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Box Shadows and Text Shadows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Opacity vs. Transparent Colors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 7: Normal Flow, Positioning, and Stacking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Normal Flow and Document Order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Containing Blocks and Formatting Contexts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

The position Property: Static, Relative, Absolute, Fixed, Sticky

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Floats and Their Modern Role

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Stacking Contexts and z-index

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Multi-Column Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 8: Flexbox: One-Dimensional Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

When to Use Flexbox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Flex Container Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Flex Item Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Alignment and Distribution in Flexbox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Wrapping and Multi-Line Flex Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Real-World Flexbox Patterns: Navbars, Cards, Centering

Chapter 9: CSS Grid: Two-Dimensional Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

When to Use Grid Over Flexbox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Defining Tracks: Columns, Rows, and Gaps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Placing Items on the Grid

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Named Grid Areas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Auto Placement and Dense Packing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Subgrid and Nested Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Intrinsic Sizing: min-content, max-content, fit-content

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Complete Grid Layout Example: Dashboard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 10: Responsive Layout Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

The Philosophy of Responsive Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Media Queries: Syntax and Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Container Queries: Component-Level Responsiveness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Fluid Typography with clamp()

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Breakpoint Strategy and Mobile-First Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Logical Properties for International Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Complete Responsive Layout Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 11: Custom Properties, Calculations, and Dynamic Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

CSS Custom Properties: Definition and Usage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Cascading Variables: Scope and Inheritance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Mathematical Functions: calc(), min(), max()

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Clamping Values with clamp()

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Environment Variables for Safe Areas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Building Design Tokens with Custom Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 12: Transitions, Animations, and Motion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Transitions: Smooth Property Changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Transform: Moving, Scaling, Rotating Elements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

3D Transforms and Perspective

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Animations and Keyframes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Easing Functions and Timing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Scroll-Driven Animations and Performance Tips

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 13: Advanced Visual Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Filters: Blur, Contrast, Hue Rotate, and More

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Blend Modes for Colors and Images

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Backdrop Filters for Glass Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Masks and Masking Images

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Clipping Paths and clip-path

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Shapes and Float Shapes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Generated Content, Counters, and Advanced Pseudo-Elements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 14: CSS Architecture and Maintainability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

The Problem of Growing Stylesheets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Naming Conventions: BEM and Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Cascade Layers (@layer)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Utility Classes and Atomic CSS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Component Architecture and Scoping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Design Systems and Token-Based Theming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 15: Accessibility, Compatibility, and Production Readiness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Styling for Accessibility: Focus and Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Respecting User Preferences: Reduced Motion, High Contrast

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Dark Mode and Color Schemes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Print Stylesheets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Browser Compatibility and Feature Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Vendor Prefixes and Autoprefixer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Production CSS Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 16: How Browsers Render CSS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

The Rendering Pipeline: Style, Layout, Paint, Composite

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Reflow and Repaint: What Triggers Them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Compositor-Only Properties for Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Diagnosing Rendering Problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Optimizing CSS for Speed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Critical CSS and Loading Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Chapter 17: Mastering Browser Developer Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Inspecting Elements and Styles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

The Computed Tab and Cascade Panel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Layout and Box Model Visualization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Responsive Design Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Animations and Performance Panels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Accessibility Audits in DevTools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Practical Debugging Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/cssmastery.