CSS COURSE -- OVERVIEW + DEEP DIVES

CSS Overview + Deep Dives

12 chapters covering the full modern CSS landscape -- cascade, layout, responsive design, animation, custom properties, and architecture.

Cascade • Box Model • Selectors • Flexbox • Grid • Responsive • Animations • Variables • Modern CSS • Architecture

CHAPTER 1

How CSS Works

Chapter 1 — How CSS Works: Cascade, Inheritance, and Specificity

CSS stands for *Cascading* Style Sheets — and that single word, cascading, describes the entire engine that decides which rule wins when multiple rules compete. Understanding this engine is what separates developers who fight CSS from those who reason about it. This chapter is a deep, authoritative treatment of all three mechanisms: the cascade, inheritance, and specificity.

1. The Cascade — The Full Algorithm

When the browser needs to apply a property to an element, it collects every declaration that targets that element and could set that property. It then runs those declarations through a filter in strict priority order. The *first filter that produces a single winner* stops — the rest are discarded. Only if all filters tie does the next one apply.

1 **Origin and importance**
Declarations are ranked by who wrote them and whether they carry `!important`. The full priority order (lowest to highest):

1. User-agent stylesheet (browser defaults)
2. User stylesheet (browser accessibility overrides)
3. Author stylesheet (your CSS)
4. Author `!important`
5. User `!important` (accessibility wins over author !important)
6. User-agent `!important`

Most of the time only levels 3 and 4 are in play. Level 4 beats level 3 for the same property.

↓ tie? continue

2 **CSS layers (`@layer`)**
If `@layer` is used, declarations in later-declared layers beat those in earlier-declared layers. Un-layered styles beat all layers. This is a relatively new addition to the cascade (covered in depth in the CSS Advanced course).

↓ tie? continue

3 **Scoping proximity**
With `@scope` (very new — late 2023+), rules in a more tightly scoped block win. In practice you'll rarely encounter this yet.

↓ tie? continue

4 **Specificity**
The selector with the highest specificity score wins. This is where most real-world cascade conflicts are resolved. Full details in section 3.

↓ tie? continue

5 **Order of appearance**
If everything else is equal, the declaration that appears later in the source wins. This is the final tiebreaker — and the reason linking a reset stylesheet before your own CSS is important.

> **The cascade is a filter, not a priority list.** Each stage is only reached if the previous stage couldn't resolve the conflict. Two declarations at different origins never compare specificities — origin always wins first. Two declarations from the same origin and with the same specificity are resolved purely by order, not by any property-level logic.

2. `!important` — What It Actually Does

`!important` bumps a declaration to a higher origin tier. It does *not* make a declaration "infinitely specific" — it moves it to a different layer of the cascade where the filter restarts. Two `!important` declarations from the same origin still compete on specificity, then order.

```
/* Author stylesheet */
.card {
    color: blue;         /* author normal — level 3 */
}

p {
    color: red !important;  /* author !important — level 4 */
}

/* Result on a <p class="card">: RED — !important beats specificity */
```

> **!important is a last resort, not a debugging shortcut.** Every `!important` you add makes the next conflict harder to reason about — you'll need another `!important` with higher specificity to override it. Legitimate uses: third-party stylesheet overrides, user accessibility stylesheets, and utility classes designed to always win (e.g. `.hidden { display: none !important; }`). If you're using it frequently to fix bugs, the real problem is a specificity architecture issue.

3. Specificity — The Scoring System

Specificity is a three-column score written as **(IDs, Classes, Elements)**. Each selector contributes points to exactly one column based on the type of selector it contains. Columns are compared left to right — a single ID beats any number of classes, regardless of how many classes are stacked.

Specificity score reference

Selector	Score (A-B-C)	Notes
`*`	0 0 0	Universal selector contributes nothing
`p`	0 0 1	One element type
`p::before`	0 0 2	Pseudo-elements count as element
`.card`	0 1 0	One class

Selector	Score (A-B-C)	Notes
`[type="text"]`	0 1 0	Attribute selectors count as class
`:hover`	0 1 0	Pseudo-classes count as class
`:is(h1, h2)`	0 0 1	:is() takes the specificity of its most specific argument
`:where(h1, h2)`	0 0 0	:where() always contributes zero — intentionally low-specificity
`:not(.active)`	0 1 0	:not() takes the specificity of its argument (.active = class)
`.card p`	0 1 1	One class + one element
`nav .card p`	0 1 2	One class + two elements
`#header`	1 0 0	One ID — beats any number of classes
`#header .nav a:hover`	1 2 1	1 ID + 2 classes/pseudoclasses + 1 element
`style=""`	1 0 0 × ∞	Inline styles have a separate column above IDs — they always beat selectors

Reading specificity scores — worked examples

```
/* Which rule wins on <p class="intro" id="lead">? */

#lead {
    color: red;       /* (1-0-0) — 1 ID */
}

.intro.intro.intro.intro.intro.intro.intro.intro.intro.intro.intro {
    color: blue;      /* (0-11-0) — 11 classes, still loses to 1 ID */
}

/* Result: RED — column A (IDs) is compared first. 1 > 0, full stop. */
/* Class column only matters if ID column is tied. */
```

```
/* Specificity tie broken by source order */

.card .title {
    font-size: 1.2rem;   /* (0-2-0) */
```

```
}

.article .title {
    font-size: 1.4rem;  /* (0-2-0) — same score */
}

/* On <h2 class="title"> inside both .card and .article: */
/* Both selectors match with equal specificity — LAST ONE WINS */
/* Result: 1.4rem (because .article .title appears later) */
```

Specificity and the :is(), :not(), :has() gotchas

```
/* :is() takes the highest specificity of its argument list */

:is(#header, .nav) a {
    color: blue;
    /* Score: (1-0-1) — #header pushes the whole :is() to ID level */
    /* Even if the element only matches .nav, the score is still 1-0-1 */
}

/* :where() always gives zero — great for utility/reset CSS */
:where(#header, .nav) a {
    color: blue;
    /* Score: (0-0-1) — :where() contributes nothing */
}

/* :has() takes the specificity of its argument */
.card:has(img) {
    padding: 0;
    /* Score: (0-1-1) — .card (class) + img (element) */
}
```

4. Inheritance

Some CSS properties automatically pass their computed value down to child elements if no other rule sets them. This is inheritance. It's what lets you set `font-family` on `body` and have it apply to every element on the page without repeating yourself.

Inheritance only kicks in when no other value is specified for the element — it is the last resort after the cascade finds nothing. The cascade always beats inheritance.

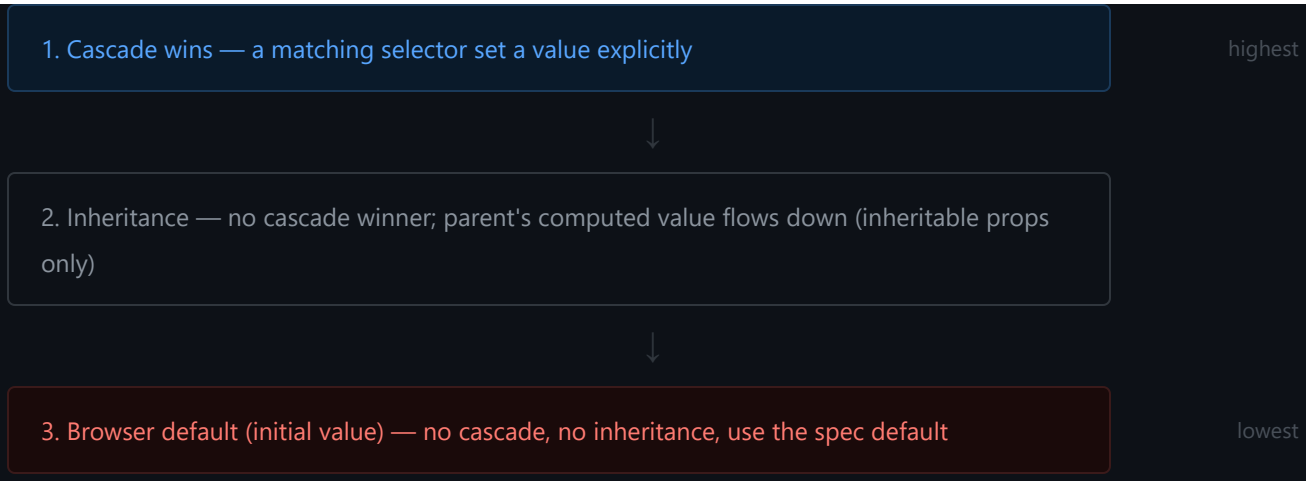
INHERITED BY DEFAULT	NOT INHERITED BY DEFAULT
color	margin
font-family	padding
font-size	border
font-weight	background
font-style	width / height
line-height	display
letter-spacing	position
text-align	box-shadow
text-transform	transform
visibility	overflow
cursor	flex / grid properties
list-style	animation

Controlling inheritance — the four keywords

```
.child {
    color: inherit;     /* Force inheritance — use parent's computed value */
    color: initial;     /* Reset to CSS spec default (often browser default) */
    color: unset;       /* If inheritable: inherit. If not: initial. Smart reset.
    color: revert;      /* Roll back to user-agent stylesheet value — more natura
}

/* Practical use: reset ALL properties on an element */
.isolated-widget {
    all: revert;        /* every property reverts to user-agent defaults */
}
```

Prefer `unset` over `initial` for resets. `initial` resets to the CSS specification default, which may differ from what the browser renders by default. `color: initial` sets colour to black regardless of the user's system theme. `unset` is smarter — it inherits for inheritable properties and uses initial for the rest, giving more predictable results in most reset scenarios.

Inheritance vs the cascade — order of resolution



5. Computed, Used, and Resolved Values

CSS has multiple stages of value processing. Understanding them clarifies why DevTools sometimes shows a different value than what you wrote:

```
/* You write:              font-size: 1.2em  (specified value) */
/* Browser resolves em:    font-size: 19.2px (computed value — em resolved to px)
/* After layout:           font-size: 19px   (used value — after rounding) */
/* DevTools shows:         19.2px            (resolved value — usually the comput

body  { font-size: 16px; }
.card { font-size: 1.2em; }   /* computed: 19.2px */
.card p { font-size: 1.2em; } /* computed: 23.04px — compounding! */
```

This compounding is why `em` for font-size can produce unexpected results in nested elements — each level multiplies the parent's computed value. Use `rem` (root em) to reference the root font-size consistently instead:

```
html   { font-size: 16px; }       /* root: 16px */
.card  { font-size: 1.2rem; }   /* 19.2px — always relative to root */
.card p { font-size: 1.2rem; }  /* 19.2px — same, no compounding */
```

6. Practical Strategies

Keep specificity low by default

```
/* Avoid: high-specificity selectors are hard to override */
nav ul li a.active {              /* (0-1-3) */
    color: blue;
}

/* Better: one class does the job with less weight */
.nav-link--active {               /* (0-1-0) — easy to override later */
    color: blue;
}
```

Use :where() to write zero-specificity base styles

```
/* A reset that any single class can override */
:where(h1, h2, h3, h4, h5, h6) {
    margin: 0;
    font-weight: bold;
    /* Score: (0-0-0) — any class beats this */
}

.hero-title {
    font-weight: 900;   /* (0-1-0) — wins easily */
}
```

Leverage inheritance instead of repeating values

```
/* Bad: repeating font declarations everywhere */
h1, h2, h3, p, li, td, label {
    font-family: 'Inter', sans-serif;
}

/* Good: set once on body, let inheritance do the work */
body {
    font-family: 'Inter', sans-serif;
    color: #1a1a1a;
    line-height: 1.6;
}
/* Every element inherits these automatically */
```

Debug cascade conflicts with DevTools

In Chrome/Firefox DevTools, select an element and open the **Styles** panel. You'll see every matching declaration, sorted by cascade priority. Overridden declarations are shown with a strikethrough. Hover over the selector to see its specificity score. This is the fastest way to diagnose "why isn't my CSS applying?"

Chapter Summary

Concept	Key point
Cascade	A five-stage filter: origin → @layer → scope → specificity → order. First stage that produces a clear winner stops. !important changes origin tier, not specificity.
Specificity	Three-column score (IDs, Classes, Elements). Compared left to right — 1 ID beats infinite classes. Inline styles are above IDs. !important is above inline styles.
:is() / :has()	Take specificity of their most specific argument. :where() always contributes zero — use for intentionally low-specificity base styles.
Inheritance	Typographic and text properties inherit by default; box model and layout properties do not. Cascade always beats inheritance.
inherit / initial / unset / revert	Keywords to explicitly control inheritance. Prefer `unset` over `initial` for resets. `all: revert` resets every property at once.
em vs rem	em is relative to the parent's computed font-size and compounds. rem is always relative to the root — predictable and non-compounding.
Specificity strategy	Keep selectors at one-class specificity where possible. Use :where() for base/reset styles. Avoid IDs in CSS. Reserve !important for genuine utility overrides.

EXERCISES

1. **Specificity scoring:** Without a browser, work out the specificity score for: `body main article.feature > p:first-child`. Then check your answer in DevTools.
2. **Inheritance experiment:** Set `color: hotpink` on `body` and `border: 2px solid hotpink` on `body`. Which elements inherit the colour? Which inherit the border? Why?
3. **:where() rewrite:** Take this selector — `header nav ul li a` — and rewrite it using `:where()` so it has zero specificity but still targets the same elements.
4. **The !important trap:** Write two rules targeting the same element with `!important` on both. Observe which wins and explain why using cascade step 1 and step 4.
5. **em compounding:** Set `html { font-size: 16px }`, then nest three elements each with `font-size: 1.25em`. Calculate the final font-size at the deepest level. Rewrite using rem and confirm the size stays constant.

Next: Chapter 2 — The Fundamentals: Box Model, Display, and Positioning. A deep dive into how CSS sizes and positions elements — the box model in all its detail, every value of `display`, and all five positioning modes with practical use cases for each.

CHAPTER 2

Box Model, Display and Positioning

Chapter 2 — The Fundamentals: Box Model, Display, and Positioning

Every element on a page is a rectangular box. The box model defines how that rectangle is sized. The `display` property defines how boxes relate to one another in flow. The `position` property defines whether a box participates in normal flow at all, and if not, where it lands instead. These three systems interact constantly — understanding each one precisely is the foundation of layout work.

1. The Box Model

Every box has four concentric layers. From inside out:

- **Content** — where text and child elements render. Sized by `width` / `height`.
- **Padding** — transparent space inside the border. Inherits the element's background.
- **Border** — a drawn edge around the padding. Can have colour, style, width, radius.
- **Margin** — transparent space outside the border. Always transparent — cannot have a background.

box-sizing — the most important CSS reset

By default, `width` and `height` size only the content area (`box-sizing: content-box`). Adding padding and border makes the element larger than the declared width — a constant source of layout bugs. The fix is universal:

CONTENT-BOX (DEFAULT — AVOID)

You declare `width: 300px` + `padding: 20px` + `border: 2px`

`Total width = 300 + 40 + 4 = 344px`

The box is wider than you said. Every time you add padding or border you must mentally subtract from width. Breaks percentage-based layouts constantly.

BORDER-BOX (USE THIS — ALWAYS)

You declare `width: 300px` + `padding: 20px` + `border: 2px`

`Total width = 300px (padding + border eat into it)`

The box is exactly as wide as you said. Content shrinks to accommodate padding and border. Layouts become predictable.

```
/* The universal box-sizing reset — include in every project */
*, *::before, *::after {
    box-sizing: border-box;
}
```

Margin collapse

Vertical margins between adjacent block elements collapse to the larger of the two values — they do not add together. This is intentional (consistent paragraph spacing) but surprises developers constantly.

```
p {
    margin-top:    24px;
    margin-bottom: 16px;
}

/* Two adjacent <p> elements: gap between them = 24px, not 40px */
/* The 24px "wins" over the 16px — they collapse */
```

Margin collapse rules:

- Only happens **vertically** (top/bottom margins). Horizontal margins never collapse.
- Only between **block-level elements** in normal flow. Flex/grid children never collapse margins.
- Parent and first/last child margins collapse if no border, padding, or overflow separates them.
- An empty block's top and bottom margins collapse into one.

- Margins of **different signs** are added (a positive and negative cancel partially).

> **Stop margin collapse without hacks:** Adding even `padding-top: 1px` or `overflow: hidden` to a parent prevents parent–child collapse. But the cleaner modern approach is to use Flexbox or Grid on the parent — flex/grid containers never collapse their children's margins.

Negative margins

```
/* Negative margins are valid and pull elements toward each other */
.sidebar {
    margin-top: -24px;   /* pulls element UP 24px */
}

/* Classic full-bleed trick: break out of a padded parent */
.full-bleed {
    margin-left:  -32px;  /* matches parent's padding-left */
    margin-right: -32px;
}
```

2. The `display` Property

`display` controls two things simultaneously: how the element *itself* participates in its parent's layout (the outer display type), and how it lays out its own children (the inner display type). The two-value syntax makes this explicit:

```
/* Two-value syntax (modern) */
.box {
    display: block flow;       /* outer: block  | inner: flow (normal flow) */
    display: inline flow;      /* outer: inline | inner: flow */
    display: inline flow-root; /* outer: inline | inner: block formatting contex
    display: block flex;       /* outer: block  | inner: flex */
    display: inline flex;      /* outer: inline | inner: flex */
    display: block grid;       /* outer: block  | inner: grid */
}

/* Shorthands (still fully supported) */
.box {
    display: block;          /* = block flow */
```

```
    display: inline;          /* = inline flow */
    display: inline-block;    /* = inline flow-root */
    display: flex;            /* = block flex */
    display: inline-flex;     /* = inline flex */
    display: grid;            /* = block grid */
    display: inline-grid;     /* = inline grid */
}
```

Every display value explained

`block`

Takes up full parent width. **Starts a new line.** Respects width, height, margin, padding in all directions. Default for div, p, h1–h6, section, article, ul, ol.

`inline`

Flows with text. **No line break.** Width/height ignored. Horizontal margin/padding work; vertical margin/padding don't affect layout flow. Default for span, a, strong, em.

`inline-block`

Inline on the outside (no line break, flows with text), block on the inside (respects width, height, all margin/padding). Old-school approach for horizontal nav items.

`flex`

Block-level flex container. Children become flex items, laid out in one axis. The modern replacement for inline-block horizontal layouts.

`inline-flex`

Inline-level flex container. Useful for icon+text combos inside text flow — the container stays inline but its children are flex-laid-out.

`grid`

Block-level grid container. Children become grid items placed on a two-dimensional grid. The most powerful CSS layout tool.

`inline-grid`

Inline-level grid container. Uncommon — used when a grid needs to size to its content and flow inline with surrounding text.

`flow-root`

Creates a new block formatting context (BFC) without being flex or grid. Reliably contains floats and prevents margin collapse from leaking out. Replaced the old `overflow: hidden` clearfix hack.

`contents`

The element itself generates no box — its children are treated as if they are direct children of the parent. Useful for semantic wrapper elements in flex/grid layouts. **Accessibility warning:** removes the element from the accessibility tree in some browsers.

`none`

Removes the element from the document entirely — no box, no space, invisible to accessibility tools. Different from `visibility: hidden` which hides visually but preserves the

`table / table-*`

Emulates HTML table behaviour on non-table elements. Rarely needed in modern CSS — grid handles these patterns better — but useful for legacy email templates.

`list-item`

Block with a marker box (bullet/number). Default for li. Rarely set explicitly but useful when you want non-li elements to render like list items.

space and keeps the element in the accessibility tree.		

Block Formatting Context (BFC)

A BFC is an independent layout region. Elements inside don't affect elements outside, and vice versa. Understanding what creates a BFC explains many CSS "quirks":

```
/* Things that create a new BFC */
/* - display: flow-root                  (cleanest) */
/* - overflow: hidden / scroll / auto     (legacy clearfix) */
/* - display: flex / grid / inline-block  (on the container) */
/* - position: absolute / fixed           (implicit BFC) */
/* - float: left / right                  (implicit BFC) */

/* BFC use cases: */
/* 1. Contain floated children */
.container { display: flow-root; }

/* 2. Prevent margin collapse escaping the parent */
.card { overflow: hidden; }  /* older approach */
.card { display: flow-root; }  /* better */

/* 3. Stop text from wrapping around a float */
.text-col { overflow: hidden; }
```

3. Positioning

The `position` property removes an element from normal flow (or modifies its position within it) using the offset properties `top`, `right`, `bottom`, `left`, and `inset`.

Value	In normal flow?	Offset reference	Creates stacking context?	Typical use
`static`	Yes	None — offsets ignored	No	Default. Every element is static unless changed.

Value	In normal flow?	Offset reference	Creates stacking context?	Typical use
`relative`	Yes (holds its space)	Its own normal-flow position	If z-index ≠ auto	Nudge from normal position. More importantly: establish a containing block for absolute children.
`absolute`	No (removed)	Nearest positioned ancestor (non-static)	If z-index ≠ auto	Tooltips, dropdowns, badges, overlays — anything that needs to float over other content.
`fixed`	No (removed)	Viewport (initial containing block)	Yes (always)	Sticky nav bars, cookie banners, floating chat buttons. Unaffected by scroll.
`sticky`	Yes — until threshold	Nearest scrolling ancestor	Yes (always)	Table headers, section headers that stick on scroll. Behaves like relative until the offset threshold is hit.

The containing block — the key to understanding absolute positioning

An absolutely positioned element is placed relative to its *containing block* — the nearest ancestor that has a `position` value other than `static`. If no such ancestor exists, it falls back to the initial containing block (the viewport). This is why `position: relative` is added to a parent even when you don't want to move the parent itself:

```
.card {
    position: relative;    /* makes .card the containing block */
}

.card__badge {
    position: absolute;
    top:    -8px;
    right: -8px;
    /* Positioned relative to .card's border edge, not the viewport */
}
```

> **transform creates a containing block too.** Any element with a `transform`, `filter`, or `perspective` value other than `none` becomes the containing block for absolutely positioned

> descendants — even without `position: relative`. This catches developers off guard when a `fixed` element stops being fixed because an ancestor has a CSS transform applied to it.

The inset shorthand

```
/* inset is a shorthand for top / right / bottom / left */
.overlay {
    position: absolute;
    inset: 0;          /* top:0; right:0; bottom:0; left:0 — fills container */
}

.tooltip {
    position: absolute;
    inset: auto 0 0 auto;  /* top:auto right:0 bottom:0 left:auto */
}
```

Centering with absolute positioning

```
/* Classic approach — requires known dimensions */
.modal {
    position: absolute;
    top:    50%;
    left:   50%;
    transform: translate(-50%, -50%);  /* pull back by 50% of OWN size */
}

/* Modern approach with inset + margin: auto */
.modal {
    position: absolute;
    inset:    0;
    margin:   auto;
    width:    400px;   /* must have explicit dimensions */
    height:   fit-content;
}
```

sticky — how it actually works

```
.section-header {
    position: sticky;
```

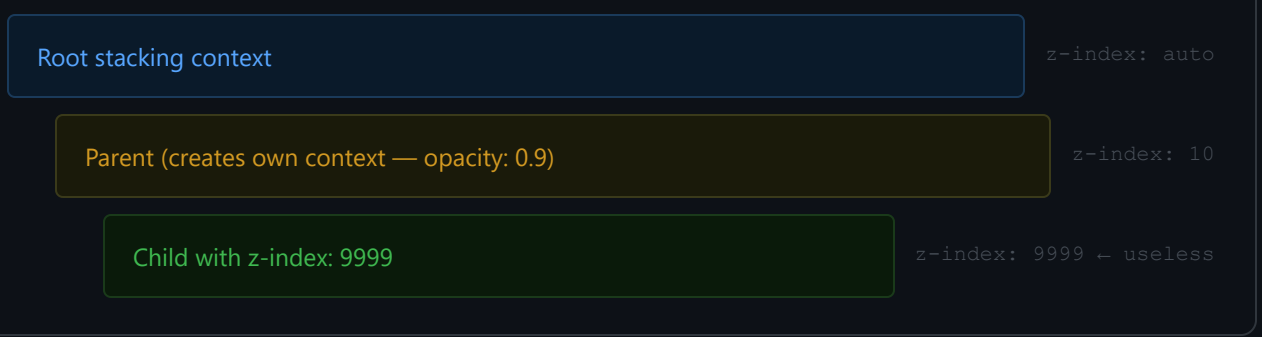
```
    top: 0;     /* sticks when this distance from the scroll container top is rea
}

/* Common sticky failures: */
/* 1. No offset defined — sticky with no top/bottom/left/right never sticks */
/* 2. overflow: hidden/auto on any ancestor — clips sticky and breaks it */
/* 3. Parent is too short — sticky only works within its parent's bounds */
/* 4. height: 100% on the parent — parent is the full page, sticky never fires
```

4. z-index and Stacking Contexts

`z-index` controls which element appears on top when boxes overlap. But it only works on elements that are *positioned* (anything except static) or are flex/grid items. The critical detail most developers miss: z-index values only compare within the same *stacking context*.

What creates a stacking context

- The root element (`<html>`) — the base stacking context
- `position: relative/absolute` + `z-index` other than `auto`
- `position: fixed` or `sticky` (always)
- `opacity` less than 1
- `transform`, `filter`, `perspective`, `clip-path` (non-none)
- `isolation: isolate` — the clean, explicit way to create one
- `will-change` with certain properties
- Flex/grid items with `z-index` other than auto



In the diagram above, the child's `z-index: 9999` only competes within the parent's stacking context. The parent itself has `z-index: 10` in the root context. No matter how high the child's z-index goes, it cannot render above something at the root context level with `z-index: 11`.

```
/* The stacking context trap */
.parent {
    position: relative;
    z-index:   1;          /* creates a stacking context */
    opacity:   0.99;       /* ALSO creates a stacking context */
}

.parent .dropdown {
    position: absolute;
    z-index:   9999;       /* trapped inside parent's context */
    /* Will never render above another element at parent's peer level with z-inc
}

/* Fix: use isolation: isolate to clearly control context creation */
.parent {
    isolation: isolate;    /* explicit, readable, no side effects */
}
```

> **Debugging z-index issues:** In Chrome DevTools, the Layers panel (More Tools → Layers) shows every stacking context as a separate layer. When a z-index value seems to have no effect, open the Layers panel and find the element — it will show you which context it belongs to and who its siblings are in that context.

5. overflow

`overflow` controls what happens when content is larger than its box. It is a shorthand for `overflow-x` and `overflow-y`.

```
.box {
    overflow: visible;  /* default — content spills out, no scroll, no clip */
    overflow: hidden;   /* clips content — also creates a BFC */
```

```
    overflow: scroll;   /* always shows scrollbars (even if not needed) */
    overflow: auto;     /* scrollbar only when needed — almost always what you w
    overflow: clip;     /* clips like hidden but does NOT create a BFC */

    /* Axis-specific */
    overflow-x: auto;
    overflow-y: hidden;
}
```

> **overflow: hidden breaks position: sticky.** If any ancestor of a sticky element has `overflow: hidden`, `overflow: auto`, or `overflow: scroll`, the sticky element will not stick. This is the most common reason sticky mysteriously stops working. Use `overflow: clip` if you need to clip without creating a scroll container.

Chapter Summary

Concept	Key point
box-sizing	Always set `*, *::before, *::after { box-sizing: border-box }`. With border-box, declared width includes padding and border — no mental arithmetic needed.
margin collapse	Vertical block margins collapse to the larger value. Flex/grid containers prevent collapse. `display: flow-root` prevents parent–child collapse without side effects.
display	Controls outer type (how the element sits in its parent's flow) and inner type (how it lays out its own children). `flex`/`grid` set both at once.
flow-root	Creates a BFC cleanly. Use it to contain floats or prevent margin collapse escaping a parent, instead of the old `overflow: hidden` hack.
position: relative	Doesn't move the element visually unless you add offsets. Main purpose: establish a containing block for absolutely positioned children.
position: absolute	Removed from flow. Positioned relative to nearest non-static ancestor. If none, falls back to viewport.
transform breaks fixed	Any ancestor with a CSS `transform` becomes the containing block for fixed/absolute children. Fixed elements stop being fixed relative to the

Concept	Key point
	viewport.
`sticky failures`	Must have an offset. Any ancestor with `overflow: hidden/auto/scroll` breaks it. Only works within its parent's bounds.
`Stacking contexts`	z-index only compares within the same stacking context. An element can't escape its parent's context no matter how high its z-index. Use `isolation: isolate` to create contexts deliberately.

EXERCISES

1. **Box model maths:** Create a div with `width: 200px`, `padding: 20px`, `border: 5px solid`. Without `box-sizing: border-box`, how wide is it? With it? Verify in DevTools by hovering the element in the Elements panel.
2. **Margin collapse:** Create two adjacent `<p>` elements each with `margin: 24px 0`. Measure the gap. Now wrap them in a `<div>` and add `display: flex; flex-direction: column` to the wrapper. Measure again. Explain the difference.
3. **Positioned badge:** Build a card with a "NEW" badge positioned at the top-right corner using `position: absolute`. Ensure the badge stays anchored to the card, not the viewport.
4. **Sticky header:** Create a page with multiple sections each 400px tall. Give each section a heading with `position: sticky; top: 0`. Then add `overflow: hidden` to the section. Observe what breaks and fix it.
5. **Stacking context trap:** Create two overlapping absolutely positioned divs. Give one `z-index: 1` and its child `z-index: 999`. Give the other `z-index: 2`. Confirm that `z-index: 999` cannot beat `z-index: 2`.

Next: Chapter 3 — Typography, Color, and Backgrounds. A deep dive into font loading and the font stack, all font properties, colour systems (hex, rgb, hsl, oklch), gradients, background-image, background-size and position, and blending modes.

CHAPTER 3

Typography, Color and Backgrounds

Chapter 3 — Typography, Color, and Backgrounds

Typography and colour are where design meets CSS. Most developers know the basic properties, but the details — font loading performance, colour space maths, gradient syntax edge cases, layered backgrounds — separate clean production work from fragile one-offs. This chapter is a complete reference for all three systems.

1. Typography

The font stack

A font stack is a comma-separated list of font families. The browser tries each in order, using the first one it finds installed. The last entry should always be a CSS generic family as a final fallback.

SYSTEM UI (MODERN)

```
system-ui, -apple-system, BlinkMacSystemFont, 'Segoe UI', sans-serif
```

Native OS font on every platform. Zero load time. Used by GitHub, Linear, Notion.

GEOMETRIC SANS

```
'Inter', 'Helvetica Neue', Arial, sans-serif
```

Load Inter from Google Fonts or self-host. Fallback to system Helvetica/Arial.

READABLE SERIF

```
Georgia, 'Times New Roman', serif
```

Georgia is universally available. Good for long-form reading.

CODE / MONOSPACE

```
'JetBrains Mono', 'Fira Code', 'Courier New', monospace
```

Load a web font, fall back to universally available Courier New.

Loading web fonts — @font-face

```
/* Self-hosting a web font — maximum performance control */
@font-face {
```

```
    font-family:  'Inter';
    src:          url('/fonts/inter-var.woff2') format('woff2');
    font-weight: 100 900;      /* variable font range */
    font-style:  normal;
    font-display: swap;        /* show fallback immediately, swap when loaded */
}

/* font-display values: */
/* auto     — browser decides (usually block) */
/* block    — invisible text for ~3s, then swap */
/* swap     — fallback immediately, swap when ready (FOUT) */
/* fallback — 100ms invisible, then fallback, swap within 3s */
/* optional — 100ms invisible, then fallback, no swap if slow */
```

> **Use `font-display: swap` for body text, `optional` for decorative fonts.** `swap` avoids invisible text (FOIT) at the cost of a layout shift (FOUT) when the font loads. `optional` avoids both — if the font doesn't load within 100ms, the browser uses the fallback permanently for that page load. Good for non-critical decorative fonts.

Variable fonts

```
/* Variable fonts expose axes — weight, width, slant, optical size */
h1 {
    font-family:   'Inter';
    font-weight:   650;       /* any value between 100-900 in a variable font */
    font-variation-settings: 'wght' 650, 'opsz' 32;
    /* font-variation-settings overrides standard properties — use as last reso
}

/* Animating variable fonts */
@keyframes weight-pulse {
    from { font-variation-settings: 'wght' 300; }
    to   { font-variation-settings: 'wght' 800; }
}
```

All typography properties

Property	Key values	Notes
font-size	px · em · rem · % · clamp()	Use rem for consistent sizing. clamp() for fluid type. Avoid px on body — it overrides user browser settings.
font-weight	100–900 · bold · normal	Numeric values work with variable fonts. bold = 700, normal = 400. The browser synthesises bold if the weight isn't available — looks terrible.
font-style	normal · italic · oblique <angle>	oblique with an angle only works on variable fonts with a slant axis.
line-height	unitless · px · em · %	Prefer unitless (e.g. 1.6) — it's relative to the element's own font-size and inherits cleanly. 1.4–1.7 for body, 1.1–1.3 for headings.
letter-spacing	em · px · normal	Use em so spacing scales with font-size. Positive = wider, negative = tighter. Common: 0.01em–0.05em for body, 0.1em+ for uppercase labels.
word-spacing	em · px · normal	Rarely needed — use sparingly and only with em units.
text-align	left · right · center · justify · start · end	Prefer start/end over left/right — they respect writing direction (RTL support). justify on screen often creates awkward rivers of whitespace.
text-transform	uppercase · lowercase · capitalize · none	capitalize uppercases the first letter of each word. Use uppercase + letter-spacing for labels/badges.
text-decoration	none · underline · line-through · overline	Shorthand for text-decoration-line, -color, -style, -thickness. Can set underline color independently of text color — great for styled links.
text-underline-offset	px · em	Moves underline away from baseline. 3–4px feels more intentional than the default browser position.
text-indent	px · em · %	Indents first line. Negative values with matching left padding create hanging indents.
white-space	normal · nowrap · pre · pre-wrap · pre-line	nowrap prevents line breaks. pre preserves whitespace + line breaks. pre-wrap wraps at container edge while preserving spaces.
overflow-wrap	normal · break-word · anywhere	break-word breaks long words to prevent overflow. anywhere is more aggressive. Use on containers with user-generated content.

Property	Key values	Notes
text-overflow	clip · ellipsis	Only works when overflow: hidden and white-space: nowrap are also set.
font-variant	small-caps · numeric · ligatures	font-variant is a shorthand. font-variant-numeric controls number rendering (lining, oldstyle, tabular). font-variant-ligatures controls fi/fl ligatures.

Fluid typography with clamp()

```
/* clamp(minimum, preferred, maximum) */
h1 {
    font-size: clamp(1.75rem, 4vw + 1rem, 3.5rem);
    /* Never smaller than 1.75rem, never larger than 3.5rem */
    /* Scales smoothly between viewport widths */
}

/* Line length control — the most important typography rule */
.prose {
    max-width: 65ch;    /* ch = width of the "0" character — ~60-75ch is ideal
}
```

Single-line truncation vs multi-line clamping

```
/* Single-line ellipsis — requires all three */
.truncate {
    white-space:   nowrap;
    overflow:      hidden;
    text-overflow: ellipsis;
}

/* Multi-line clamp — limit to N lines */
.clamp-3 {
    display:            -webkit-box;
    -webkit-line-clamp: 3;
    -webkit-box-orient: vertical;
    overflow:           hidden;
    /* Still the correct approach despite the -webkit- prefix — fully supported
}
```

2. Color

Color systems — a complete picture

Hex

`#4f8ef7` · `#4f8ef7ff`

Most common. 6-digit RGB hex, optional 2-digit alpha. 3-digit shorthand (#abc = #aabbcc). Opaque by default.

rgb() / rgba()

`rgb(79 142 247)` · `rgb(79 142 247 / 0.8)`

Modern syntax uses spaces, not commas. Alpha with / separator. rgba() is legacy — just use rgb() with 4 values.

hsl()

`hsl(220 90% 64%)` · `hsl(220 90% 64% / 0.8)`

Hue (0–360 degrees), Saturation %, Lightness %. Human-readable — easy to create colour palettes by adjusting lightness.

oklch() — modern

`oklch(65% 0.18 260)` · `oklch(65% 0.18 260 / 0.8)`

Perceptually uniform — equal lightness steps look equally different. Better for accessible colour palettes and gradients. Wide gamut capable (P3 and beyond).

color() — wide gamut

`color(display-p3 0.3 0.56 0.97)`

Access wide gamut colour spaces (display-p3, rec2020). Vivid colours outside sRGB. Falls back to sRGB on unsupported displays.

color-mix()

`color-mix(in oklch, blue 60%, red)`

Mix two colours in any colour space. oklch gives the most visually pleasing result. Great for generating tints and shades programmatically.

> **Why oklch beats hsl for design systems.** In hsl, colours at the same declared lightness (e.g. yellow at 50% and blue at 50%) look wildly different in perceived brightness — yellow looks far lighter. oklch corrects for this: equal lightness values produce colours that actually look equally light. When building a design token system, defining your palette in oklch means accessible contrast ratios are predictable.

currentColor — the CSS colour inheritance keyword

```
/* currentColor equals the element's computed color value */
.icon {
    color:  #58a6ff;
```

```
    border: 2px solid currentColor;  /* same blue */
    box-shadow: 0 0 8px currentColor;  /* same blue */
}

/* In SVGs: fill="currentColor" makes the SVG adopt its CSS color */
/* Essential for icon systems — one SVG, any colour via CSS */
```

Transparency and the alpha channel

```
/* Multiple ways to express transparency */
.el {
    color: rgb(0 0 0 / 0.5);          /* 50% transparent black */
    color: hsl(0 0% 0% / 50%);        /* same */
    color: #00000080;                 /* hex with alpha (80 hex = ~50%) */
    color: transparent;               /* fully transparent (= rgb(0 0 0 / 0)) */

    /* opacity vs alpha — critical difference: */
    opacity: 0.5;       /* affects the element AND all its children */
    background: rgb(0 0 0 / 0.5);  /* affects only this property — children unai
}
```

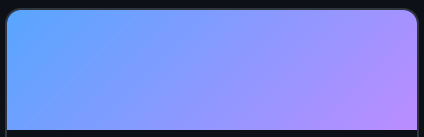
3. Backgrounds

All background properties

Property	Values	Notes
background-color	any colour value	Rendered below background-image. Useful as a fallback when the image fails to load.
background-image	url() · linear-gradient() · radial-gradient() · conic-gradient() · none	Multiple images layer on top of each other (first listed = top layer). Gradients are treated as images.
background-size	auto · cover · contain · px/% pairs	**cover** fills the box (may crop). **contain** fits entire image (may leave gaps). Use cover for hero images, contain for logos.
background-position	center · top left · 50% 25% · px values	Sets where the image is anchored. Percentage values are relative to both container and image

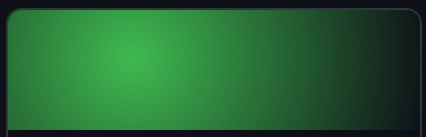
Property	Values	Notes
		size (not just container).
background-repeat	repeat · no-repeat · repeat-x · repeat-y · space · round	**space** distributes copies evenly. **round** scales images to fit whole copies. Always use no-repeat with background-size: cover.
background-attachment	scroll · fixed · local	**fixed** = parallax effect (relative to viewport). **local** scrolls with content inside the element. **scroll** = default.
background-origin	padding-box · border-box · content-box	Where the background image starts. Default is padding-box (starts at the inner edge of the border).
background-clip	border-box · padding-box · content-box · text	**text** clips the background to the text shape — used for gradient text effects.
background	shorthand	Order: color image position/size repeat attachment origin clip. Resets all sub-properties — use longhand when layering multiple backgrounds.

Gradients



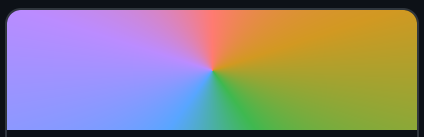
linear-gradient

`linear-gradient(135deg, #58a6ff, #bc8cff)`



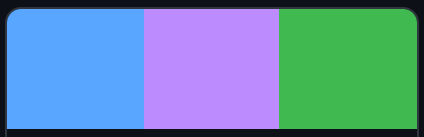
radial-gradient

`radial-gradient(circle at 30% 40%, #3fb950, #0d1117)`



conic-gradient

`conic-gradient(from 0deg, red, yellow, green, blue, red)`



hard stops

`same position twice = hard edge`

```
/* Gradient syntax in full */

/* linear-gradient — direction then colour stops */
```

```css
.box {
    background: linear-gradient(to right, #58a6ff, transparent);
    background: linear-gradient(135deg, #58a6ff 0%, #bc8cff 50%, #3fb950 100%);
    background: linear-gradient(in oklch, hsl(220 80% 60%), hsl(280 80% 60%));
    /* "in oklch" interpolates colours through oklch space — avoids muddy midpoi
}

/* radial-gradient — shape, size, position */
.hero {
    background: radial-gradient(ellipse 80% 60% at center top, #1a2a4a, #0d1117)
}

/* conic-gradient — pie charts, angle wheels, starburst patterns */
.pie {
    background: conic-gradient(from 0deg, #58a6ff 0% 40%, #3fb950 40% 70%, #d29
    border-radius: 50%;
}

/* Hard colour stop — same position twice */
.stripe {
    background: linear-gradient(90deg,
        #58a6ff 0% 33%,
        #bc8cff 33% 66%,
        #3fb950 66% 100%
    );
}
```

Layered backgrounds

```css
/* Multiple background layers — first listed renders on top */
.card {
    background-image:
        url('/icons/pattern.svg'),                /* top: SVG pattern */
        linear-gradient(135deg, #1a2a3a, #0d1117);  /* bottom: gradient base */

    background-size:    40px 40px, cover;
    background-repeat:  repeat, no-repeat;
    background-position: center, center;
}
```

Gradient text

```
/* Text with gradient fill — three properties required */
.gradient-text {
    background:            linear-gradient(90deg, #58a6ff, #bc8cff);
    -webkit-background-clip: text;
    background-clip:        text;
    color:                  transparent;  /* make text transparent to show bg */
}
```

mix-blend-mode and background-blend-mode

```
/* mix-blend-mode: how the element blends with what's behind it */
.overlay {
    mix-blend-mode: multiply;   /* darkens — good for image colour overlays */
    mix-blend-mode: screen;     /* lightens */
    mix-blend-mode: overlay;    /* contrast boost */
    mix-blend-mode: color;      /* applies hue+saturation, keeps luminosity */
}

/* background-blend-mode: how an element's own bg layers blend together */
.duotone {
    background-image:
        linear-gradient(#58a6ff, #bc8cff),
        url('/photo.jpg');
    background-blend-mode: multiply;   /* duotone photo effect */
}
```

Chapter Summary

Concept	Key point
`Font stacks`	Always end with a generic family. System-ui is the fastest — no loading needed. Use font-display: swap to avoid invisible text during web font load.
`Variable fonts`	One file for all weights/widths. Use font-weight with numeric values (100–900). Avoid font-variation-settings except for non-standard axes.

Concept	Key point
`line-height`	Use unitless values (1.5–1.7 for body). They inherit correctly and scale with any font-size change. Never use px for line-height.
`clamp()`	clamp(min, preferred, max) gives fluid type without breakpoints. 65ch max-width for prose line length.
`Color systems`	hex/rgb for compatibility, hsl for readability, oklch for perceptual accuracy and design systems. color-mix() for programmatic tints/shades.
`opacity vs alpha`	opacity affects the whole element tree. Alpha in a colour value (rgb / hsl / oklch with /) affects only that one property — children unaffected.
`currentColor`	Equals the element's computed color. Use in SVG fill, border, box-shadow to keep them in sync with one property change.
`background-size`	cover fills and may crop. contain fits whole image and may leave gaps. Always use no-repeat with cover.
`Gradients`	linear, radial, conic. Hard stops = same position twice. Use "in oklch" to avoid muddy midpoints. Layer gradients for complex backgrounds.
`Gradient text`	background-clip: text + color: transparent. Requires both -webkit- and unprefixed background-clip.

EXERCISES

1. **Font loading:** Add a Google Font to a page using a `<link>` tag, then convert it to a self-hosted `@font-face` with `font-display: swap`. Compare load behaviour using DevTools Network tab (throttle to Slow 3G).
2. **Fluid type scale:** Create a typographic scale — h1 through h3 and body — using `clamp()` for each, so sizes scale smoothly from mobile (320px) to desktop (1200px) with no breakpoints.
3. **Colour palette in oklch:** Define a 5-step tint scale for one hue using `oklch()` — keep hue and chroma constant, vary lightness from 20% to 80% in equal steps. Compare the same steps in `hsl()`. Which looks more perceptually even?
4. **Hero section:** Build a hero with a layered background: a semi-transparent gradient overlay on top of a photo, with a repeating SVG dot pattern at 10% opacity on top of both. All three layers in one `background-image` declaration.

5. **Gradient text heading:** Create a heading that displays a three-colour gradient across the text using `background-clip: text`. Then animate the gradient position using `@keyframes` and `background-position`.

Next: Chapter 4 — Deep Dive: Selectors. Every CSS selector type in one place — combinators, attribute selectors, all pseudo-classes (structural, state, functional), pseudo-elements, and the modern relational selectors `:is()`, `:where()`, `:not()`, and `:has()`.

CHAPTER 4

Deep Dive: Selectors

Chapter 4 — Deep Dive: Selectors

CSS has far more selectors than most developers use day to day. Knowing the full set means writing less JavaScript for DOM queries, producing more resilient stylesheets, and understanding exactly what specificity each rule carries. This chapter covers every selector type — simple, combinators, attribute, structural pseudo-classes, state pseudo-classes, functional pseudo-classes, pseudo-elements, and the modern relational selectors — with practical examples and specificity scores throughout.

1. Simple Selectors

Selector	Specificity	What it matches
SIMPLE		
`*`	`0-0-0`	Universal — every element. Zero specificity. Use for resets and box-sizing only.
`div`	`0-0-1`	Type selector — matches any element of that tag name. Prefer classes over element selectors for styling.
`.class`	`0-1-0`	Class selector — the workhorse. Can be chained: `.card.featured` matches elements with both classes.
`#id`	`1-0-0`	ID selector. High specificity — avoid in CSS. IDs should drive JavaScript and URL fragments, not styles. Each ID should appear once per page.

2. Combinators

Combinators express relationships between selectors. They don't add their own specificity — only the selectors on either side contribute.

(space)

DESCENDANT

```
nav a
```

>

CHILD

```
ul > li
```

Matches `a` anywhere inside `nav` — any depth. The most common combinator. Greedy: hits deeply nested elements too.

Matches `li` that is a **direct child** of `ul` only. Does not match nested lists' items. More precise than descendant.

+

ADJACENT SIBLING

`h2 + p`

Matches the **immediately following** sibling. The `p` must come right after `h2` with no elements in between.

~

GENERAL SIBLING

`h2 ~ p`

Matches **all subsequent** siblings. The `p` elements can be anywhere after `h2` within the same parent.

||

COLUMN

`col.selected || td`

Targets cells belonging to a column. Experimental — limited browser support. For table column styling.

```css
/* Combinator patterns in practice */

/* Lobotomised owl — add space above every element that follows another */
.stack > * + * {
    margin-top: 1rem;
}

/* Adjacent sibling to style the paragraph after a heading differently */
h2 + p {
    font-size:   1.1em;
    color:       #79c0ff;  /* lead paragraph style */
}

/* General sibling — all checked radio's subsequent labels */
input[type="radio"]:checked ~ label {
    font-weight: bold;
}
```

3. Attribute Selectors

Attribute selectors match elements based on the presence or value of HTML attributes. All carry class-level specificity **(0-1-0)**.

Selector	Matches when...
`[attr]`	Attribute exists (any value, even empty)
`[attr="val"]`	Attribute equals exactly "val"
`[attr~="val"]`	Attribute is a space-separated list containing "val" (class-like matching)
`[attr|="val"]`	Attribute equals "val" or starts with "val-" (language code matching: `[lang|="en"]` matches en, en-US, en-GB)
`[attr^="val"]`	Attribute **starts with** "val"
`[attr$="val"]`	Attribute **ends with** "val" — great for file extension targeting
`[attr*="val"]`	Attribute **contains** "val" anywhere
`[attr="val" i]`	Case-insensitive match (append `i` before closing bracket)
`[attr="val" s]`	Case-sensitive match (append `s` — useful in SVG/XML contexts)

```
/* Attribute selector real-world patterns */

/* Icon for external links */
a[href^="http"]::after {
    content: ' \2197';   /* ↗ arrow */
}

/* Style PDF links differently */
a[href$=".pdf"] {
    color: #ff7b72;
}

/* Style disabled form elements */
input[disabled],
button[disabled] {
    opacity: 0.4;
    cursor:  not-allowed;
}

/* Target ARIA roles for styling — ties CSS to semantics */
[role="alert"] {
```

```
    border: 2px solid #ff7b72;
    padding: 1rem;
}

/* Language-aware typography */
:lang(ja) {
    font-family: 'Noto Sans JP', sans-serif;
    line-height: 1.9;
}
```

4. Pseudo-Classes

Pseudo-classes select elements based on state, position, or relationship — things that can't be expressed with static HTML attributes. All carry class-level specificity **(0-1-0)** unless they contain a selector argument.

USER INTERACTION STATE

`:hover`

Mouse pointer is over the element. **Does not fire on touch devices** unless tapped.

`:focus`

Element has keyboard or programmatic focus. **Never remove the outline without providing an alternative.**

`:focus-visible`

Focus ring only shows for keyboard users — not after mouse clicks. The correct approach to custom focus styles.

`:focus-within`

Element or any descendant has focus. Useful for styling a form container when any field inside it is active.

`:active`

Element is being activated (mouse button down). Use for press animations — scale down on :active for tactile feel.

`:visited`

Link has been visited. **Privacy-restricted** — only color, background-color, border-color, outline-color can be changed.

FORM / INPUT STATE

`:checked`

Checkbox or radio is checked. Combine with ~ to style adjacent labels — CSS-only toggle patterns.

`:indeterminate`

Checkbox is in the indeterminate state (neither checked nor unchecked — set via JS).

`:disabled`

Form element has the disabled attribute. Prefer this over [disabled] — it's more specific to form elements.

`:enabled`

Form element is not disabled. Rarely needed — useful when you need to explicitly re-style enabled elements in a context where many are disabled.

`:placeholder-shown`

Input has a visible placeholder (i.e. it's empty). Use to detect whether a field is empty — CSS-only float label pattern.

`:required / :optional`

Input has or lacks the required attribute. Style required fields to communicate expectation before submission.

:valid / :invalid

Input passes or fails HTML constraint validation. **Fires immediately on page load** — combine with :user-invalid to delay.

:user-valid / :user-invalid

Like :valid/:invalid but only fires after the user has interacted with the field. The correct approach for showing validation errors.

:in-range / :out-of-range

Number/date input value is inside or outside the min/max range.

:read-only / :read-write

Element has or lacks the readonly attribute. :read-write also matches contenteditable elements.

STRUCTURAL — TREE POSITION

:first-child

Element is the first child of its parent (regardless of type).

:last-child

Element is the last child of its parent.

:only-child

Element is the only child of its parent. Useful for hiding UI that only makes sense with siblings (prev/next buttons).

:nth-child(n)

Matches by position. Accepts a number, keyword (odd/even), or formula (An+B). Counts **all siblings**, then filters by type.

:nth-last-child(n)

Same as nth-child but counted from the end. :nth-last-child(1) = :last-child.

:first-of-type

First sibling of that **element type** — ignores siblings of other types. Different from :first-child.

:last-of-type

Last sibling of that element type.

:only-of-type

Only sibling of that element type.

:nth-of-type(n)

nth sibling of that element type. Counts only siblings matching the element type.

:empty

Element has no children and no text (whitespace counts as content in some browsers — use carefully). Good for hiding empty containers.

:root

The root element (html in HTML documents). Higher specificity than the html type selector. Where CSS custom properties are declared.

:target

Element whose ID matches the URL fragment (#section). Used for CSS-only tab and accordion patterns, and scroll-linked highlighting.

DOCUMENT / PAGE

:any-link

Matches any anchor with an href — combines :link and :visited without writing both.

:link

An anchor with href that has not been visited.

:local-link

Link pointing to the current page. Limited support — use with care.

:scope

In stylesheets, same as :root. In JS querySelector(), represents the

:fullscreen

Element is being displayed in fullscreen mode. Style fullscreen

:picture-in-picture

Video element is currently in picture-in-picture mode.

element the query is called on. Useful with @scope.

video players or presentations differently.

:nth-child() — the An+B formula

```
/* An+B formula — A = step, B = offset, n starts at 0 */

li:nth-child(odd)    { /* 1, 3, 5, 7 … = 2n+1 */ }
li:nth-child(even)   { /* 2, 4, 6, 8 … = 2n+2 or 2n */ }
li:nth-child(3)      { /* exactly the 3rd item */ }
li:nth-child(3n)     { /* every 3rd: 3, 6, 9, 12 … */ }
li:nth-child(3n+1)   { /* 1, 4, 7, 10 … (first item in each group of 3) */ }
li:nth-child(-n+3)   { /* first 3 items only: 3, 2, 1 */ }
li:nth-child(n+4)    { /* all items FROM the 4th onward */ }

/* Modern: :nth-child() now accepts a selector argument */
li:nth-child(2 of .featured) {
    /* The 2nd li that has class .featured */
    /* Counts only .featured items, skips the rest */
}
```

> **:nth-child vs :nth-of-type — the key difference.** `:nth-child(2)` selects an element if it is the 2nd child of its parent, regardless of type. `:nth-of-type(2)` selects the 2nd sibling of the same element type. If a `<div>` is the only div but the 4th child, `div:nth-child(2)` won't match it, but `div:nth-of-type(1)` will.

5. Functional Pseudo-Classes

:is() — forgiving selector list

```
/* Without :is() — verbose */
header h1, header h2, header h3,
main h1, main h2, main h3,
footer h1, footer h2, footer h3 {
    line-height: 1.2;
}

/* With :is() — compact */
```

```css
:is(header, main, footer) :is(h1, h2, h3) {
    line-height: 1.2;
}

/* :is() is "forgiving" — an invalid selector in the list is ignored */
:is(h1, h2, :unsupported-selector) {
    /* the :unsupported-selector is skipped, h1 and h2 still match */
    /* Traditional comma lists would discard the WHOLE rule if any selector is i
}

/* Specificity: takes the highest specificity of any argument */
:is(#id, .class) p   { /* specificity: (1-0-1) — #id raises the whole :is() */ }
:is(.a, .b, .c) p    { /* specificity: (0-1-1) — highest in list is a class */ }
```

:where() — zero specificity

```css
/* :where() is identical to :is() but ALWAYS contributes zero specificity */

:where(h1, h2, h3) {
    margin-top: 0;
    /* Specificity: (0-0-0) — any single class will override this */
}

/* Perfect for base/reset styles that should be easy to override */
:where(ul, ol) {
    padding-left: 0;
    list-style:   none;
}

/* User code can override with a single class — no specificity battle */
.nav-list {
    padding-left: 1rem;   /* (0-1-0) — wins over :where()'s (0-0-0) */
}
```

:not() — exclusion

```css
/* :not() excludes matching elements */

p:not(.intro) {
    color: #8b949e;   /* all p except .intro */
```

```
}

/* Modern :not() accepts a full selector list */
a:not(:hover, :focus, .active) {
    text-decoration: none;
}

/* Specificity: takes the highest of its argument */
p:not(#special)  { /* (1-0-1) — #special raises it to ID level */ }
p:not(.active)   { /* (0-1-1) — class level + element */ }
p:not(span)      { /* (0-0-2) — element level + element */ }

/* Practical: style all buttons except disabled ones */
button:not(:disabled):hover {
    transform: translateY(-1px);
}
```

:has() — the parent selector (and more)

`:has()` is the most powerful selector added to CSS in years. It selects an element based on what it *contains* — effectively a parent selector, but more accurately a "relational" selector. Supported in all modern browsers since 2023.

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

/* Style a form when it contains an invalid field */
form:has(:user-invalid) {
    border-color: #ff7b72;
}

/* Style a list item when its checkbox is checked */
li:has(input:checked) {
    text-decoration: line-through;
    opacity:         0.5;
}

/* Style the body when a modal is open */
body:has(dialog[open]) {
```

```
    overflow: hidden;    /* prevent scroll while modal is open */
}

/* :has() as previous sibling selector — targets element BEFORE the match */
h2:has(+ p.warning) {
    color: #d29922;  /* style h2 when immediately followed by a warning paragrap
}

/* Quantity queries — style items when there are exactly 3 */
li:has(~ li:nth-child(3)):nth-child(1) {
    /* complex but possible without JS */
}

/* :has() specificity takes the highest in its argument */
.card:has(img)     { /* (0-1-1) — .card (class) + img (element) */ }
.card:has(.hero)   { /* (0-2-0) — .card (class) + .hero (class) */ }
```

> **:has() does not accept pseudo-elements** as its argument. You can't write `:has(::before)`. It also cannot be nested — `:has(:has(...))` is invalid. But you can combine it with other pseudo-classes: `:has(:not(.excluded))` and `:has(:is(img, video))` both work.

6. Pseudo-Elements

Pseudo-elements create virtual elements in the DOM. They use double-colon syntax (`::`) to distinguish from pseudo-classes, and carry element-level specificity **(0-0-1)**.

Pseudo-element	What it creates / selects
`::before`	Inserts a generated element as the first child. Requires `content` property (can be empty string). Cannot be added to void elements (img, input, br).
`::after`	Inserts a generated element as the last child. Same rules as ::before. Together they give every non-void element two free extra elements for decoration.
`::first-line`	The first rendered line of a block element. Only a subset of properties apply (color, font, text-decoration, background, word-spacing, letter-spacing).
`::first-letter`	The first character of a block element. Used for drop caps. Supports more properties than ::first-line including float, width, height.

Pseudo-element	What it creates / selects
`::selection`	Text selected by the user. Only color, background-color, and text-shadow can be changed. Used for brand-coloured text selection.
`::placeholder`	The placeholder text in an input. Style sparingly — placeholder text should have lower contrast than real input, not styled to look like a label.
`::marker`	The bullet or number of a list item. Supports color, font, content. Replace list-style-type with content in ::marker for full Unicode/emoji bullets.
`::backdrop`	The full-viewport overlay behind a <dialog> or fullscreen element. Style it to darken the background: `background: rgb(0 0 0 / 0.5)`.
`::cue`	WebVTT cue boxes (subtitles/captions) on video elements. Style subtitle appearance without JavaScript.
`::file-selector-button`	The button part of `<input type="file">`. Finally lets you style file inputs without hiding them.

```
/* ::before / ::after practical patterns */

/* Decorative quote mark */
blockquote::before {
    content:     '\201C';    /* " opening double quote */
    font-size:   4rem;
    line-height: 0;
    color:       #58a6ff;
}

/* Counter via ::before */
ol   { counter-reset: step; list-style: none; }
li   { counter-increment: step; }
li::before {
    content:     counter(step, decimal-leading-zero);
    font-variant-numeric: tabular-nums;
    color:       #58a6ff;
    margin-right: 0.75rem;
}

/* ::marker — custom bullet */
li::marker {
    content: '→  ';
    color:   #3fb950;
}
```

```css
/* ::selection */
::selection {
    background-color: #1a3a5a;
    color:            #ffffff;
}

/* ::backdrop — style dialog overlay */
dialog::backdrop {
    background:       rgb(0 0 0 / 0.6);
    backdrop-filter:  blur(4px);
}
```

Chapter Summary

Concept	Key point
`Combinators`	Space = any descendant. > = direct child only. + = immediately following sibling. ~ = all following siblings. None add their own specificity.
`Attribute selectors`	^= starts with, $= ends with, *= contains, ~= word in list, |= equals or starts with hyphen. All (0-1-0) specificity. Append i for case-insensitive.
`:nth-child An+B`	n starts at 0. -n+3 = first 3. n+4 = from 4th onward. odd = 2n+1, even = 2n. Modern browsers support "of .class" argument.
`:nth-child vs :nth-of-type`	nth-child counts all siblings then checks type. nth-of-type counts only siblings of the same type. A common source of confusion.
`:is()`	Forgiving selector list — invalid selectors are skipped. Takes specificity of most specific argument. Reduces repetition dramatically.
`:where()`	Identical to :is() but always (0-0-0) specificity. Use for base/reset styles that should be trivially overridable.
`:has()`	Selects an element based on its contents or following siblings. Parent selector, quantity queries, conditional styling without JavaScript. Supported in all modern browsers.
`::before / ::after`	Require content property. Cannot be used on void elements. Two free generated elements per non-void element for decoration.

Concept	Key point
`::marker`	Style list bullets with color and content — use content: to replace the bullet entirely with any character or emoji.
`:focus-visible`	Shows focus ring for keyboard navigation only — not after mouse clicks. The correct way to style focus without harming accessibility.
`:user-valid / :user-invalid`	Only fires after the user has interacted with an input. Prevents showing error states immediately on page load.

EXERCISES

1. **Attribute selectors:** Style all external links (href starting with http) with a small ↗ icon appended via `::after`. Then style all PDF links (href ending in .pdf) in red. Use only attribute selectors — no classes.
2. **:nth-child patterns:** Create a 6-item grid. Using only `:nth-child()`, make item 1 span 2 columns, items 2–3 normal, item 4 have a different background, and items 5–6 appear at 50% opacity. No classes allowed.
3. **:has() todo list:** Build an unordered list of tasks each containing a checkbox. When a checkbox is checked, style the parent `<li>` with a strikethrough and reduced opacity using only `:has(input:checked)` — no JavaScript.
4. **:has() modal lock:** Add a `<dialog>` element to a page with an open/close button. Using `body:has(dialog[open])`, prevent body scrolling and dim the background — no JavaScript for the CSS part.
5. **::marker customisation:** Create an ordered list where each marker shows "Step N →" using CSS counters and `::marker { content: ... }`. The counter should be styled in blue and larger than the list text.

Next: Chapter 5 — Deep Dive: Flexbox. The complete flexbox reference — every container and item property, alignment axes, flex-grow / flex-shrink / flex-basis demystified, wrapping, ordering, and real-world layout patterns including navigation bars, card grids, and centring.

CHAPTER 5

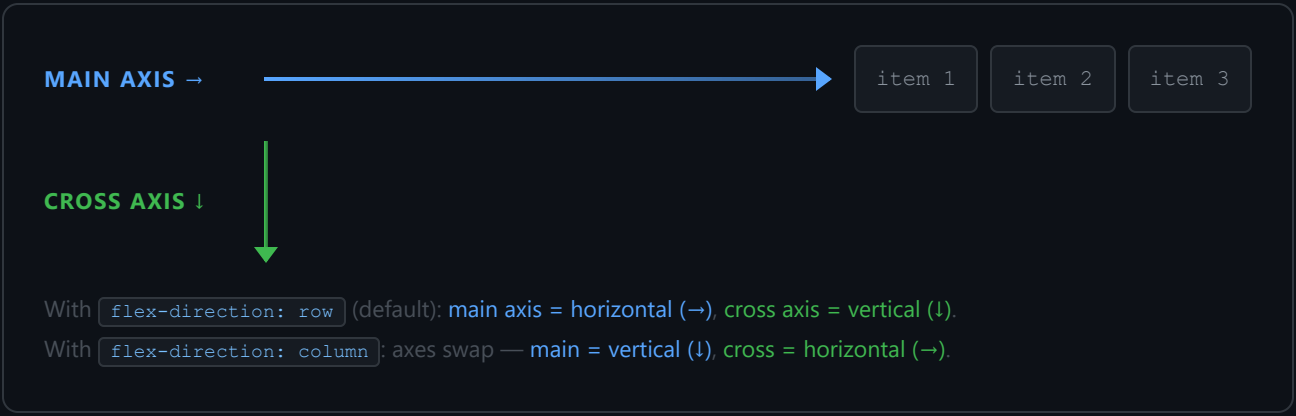
Deep Dive: Flexbox

Chapter 5 — Deep Dive: Flexbox

Flexbox is a one-dimensional layout system — it distributes items along a single axis. It excels at navigation bars, toolbars, card rows, form rows, and any situation where you need items to share space intelligently. This chapter covers every container and item property, demystifies flex-grow / flex-shrink / flex-basis, and ends with a library of production-ready patterns.

1. The Two Axes

Every flexbox layout has two perpendicular axes. Which one is which depends entirely on `flex-direction`. Getting this mental model right makes the alignment properties click immediately.



> **This is the single most important flexbox concept.** `justify-content` always controls the *main* axis. `align-items` always controls the *cross* axis. If you get confused about which property does what, ask: "which direction is my main axis right now?" — then the property names make sense.

2. Container Properties

Set these on the flex container (the parent with `display: flex`).

DIRECTION AND WRAPPING

flex-direction

`row · row-reverse · column · column-reverse`

Sets the main axis. **row** (default) = left→right. **column** = top→bottom. The `-reverse` variants flip item order visually but do not change DOM order — keyboard navigation still follows DOM order.

flex-wrap

`nowrap · wrap · wrap-reverse`

nowrap (default) — all items stay on one line, may overflow or shrink. **wrap** — items that don't fit start a new line. **wrap-reverse** — new lines are added above rather than below.

flex-flow

`[ direction ] [ wrap ]`

Shorthand for flex-direction + flex-wrap. Example: `flex-flow: row wrap`. Write them separately for clarity in most codebases.

MAIN AXIS ALIGNMENT (JUSTIFY-*)

justify-content

`flex-start · flex-end · center · space-between · space-around · space-evenly · start · end`

Distributes **free space along the main axis** after items are sized. Has no effect when items overflow or when a flex item has `flex-grow` consuming all free space.

justify-items

`normal (ignored in flex)`

Has no effect in flexbox — it only applies to Grid. Don't confuse with justify-content. A common mistake.

CROSS AXIS ALIGNMENT (ALIGN-*)

align-items

`stretch · flex-start · flex-end · center · baseline · first baseline · last baseline`

Aligns items along the **cross axis within their line**. **stretch** (default) makes all items the same cross-axis size. **baseline** aligns the text baselines — great for mixed font-size nav items.

align-content

`stretch · flex-start · flex-end · center · space-between · space-around · space-evenly`

Distributes **space between wrapped lines** on the cross axis. **Has no effect when there is only one line** (no wrapping, or all items fit). Requires flex-wrap: wrap.

GAPS

gap

`[ row-gap ] [ column-gap ]`

Sets space between flex items. **Does not add space at the edges** — only between items. Replaced the old margin-based hack. One value sets both; two values set row-gap then column-gap.

row-gap / column-gap

`length · percentage`

The longhand components of gap. In row flex-direction, column-gap is the between-item space and row-gap controls space between wrapped lines.

3. Item Properties

Set these on the flex items (the direct children of the flex container).

THE FLEX SIZING TRIAD

flex-basis

`auto · content · length · percentage`

The **starting size** of the item along the main axis before free space is distributed. **auto** = use the item's width/height. **content** = size to content. Takes precedence over width/height when both are set.

flex-grow

`0 (default) · positive number`

How much of the **remaining free space** the item absorbs. The value is a ratio — flex-grow: 2 on one item and 1 on another means the first gets twice the free space. **Zero means the item does not grow.**

flex-shrink

`1 (default) · 0 · positive number`

How much the item shrinks when there is **not enough space**. Default 1 = all items shrink proportionally. **0 means the item refuses to shrink** — it will overflow if needed. Useful for fixed-width sidebars.

flex

`[ grow ] [ shrink ] [ basis ]`

The shorthand. **Always use the shorthand** — it sets sensible defaults for omitted values. `flex: 1` = grow: 1, shrink: 1, basis: 0% (not auto). `flex: auto` = 1 1 auto. `flex: none` = 0 0 auto (rigid).

flex-grow / flex-shrink — how the maths works

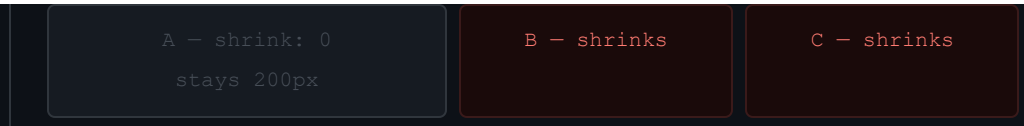
CONTAINER: 600PX WIDE. ITEMS A, B, C EACH FLEX-BASIS: 100PX. FREE SPACE = 300PX.

A — flex-grow: 1	B — flex-grow: 1	C — flex-grow: 1
+100px	+100px	+100px

ITEM A HAS FLEX-GROW: 2, B AND C HAVE FLEX-GROW: 1. TOTAL GROWTH UNITS = 4.

A — flex-grow: 2	B — flex-grow: 1	C — flex-grow: 1
+150px → 250px	+75px → 175px	+75px → 175px

FLEX-SHRINK: A=0 (WON'T SHRINK), B AND C SHRINK NORMALLY WHEN CONTAINER NARROWS.




```css
/* Common flex shorthand values — memorise these */

.item {
    flex: 1;        /* grow:1 shrink:1 basis:0% — equal share of ALL space */
    flex: auto;     /* grow:1 shrink:1 basis:auto — grow/shrink from content size
    flex: none;     /* grow:0 shrink:0 basis:auto — rigid, sized to content */
    flex: 0 0 200px; /* fixed 200px — won't grow OR shrink */
    flex: 1 0 200px; /* starts at 200px, can grow, won't shrink */
}

/* flex: 1 vs flex: auto — the key difference */
/* flex: 1   → basis: 0%  → items share the TOTAL container width equally */
/* flex: auto → basis: auto → items share only the FREE space; content size matt
```

INDIVIDUAL ITEM OVERRIDES

align-self

`auto` · `stretch` · `flex-start` · `flex-end` · `center` · `baseline`

Overrides `align-items` for a single item. **auto** inherits the container's align-items value. Use to push one item to the opposite end of the cross axis.

order

`integer (default: 0)`

Changes visual order without altering DOM order. Lower numbers appear first. **Accessibility warning:** keyboard navigation and screen readers follow DOM order, not visual order. Only use order for purely visual reordering where DOM order makes semantic sense.

4. Alignment Values in Full

justify-content — main axis

`flex-start`
items packed to start

`flex-end`
items packed to end

`center`
items centred

`space-between`
first/last flush, gaps between

align-items — cross axis (per line)

`stretch`
default — fill cross-axis height

`flex-start`
align to cross-axis start

`flex-end`
align to cross-axis end

`center`
centre on cross axis

space-around

equal space around each item (half at edges)

space-evenly

equal space everywhere including edges

baseline

align text baselines

first baseline

align to first line's baseline

align-content — cross axis (between lines)

stretch

default — lines fill cross axis

flex-start

lines packed to start

flex-end

lines packed to end

center

lines centred

space-between

first/last flush, space between

space-evenly

equal space everywhere

The place-* shorthands

place-content

align-content + justify-content

place-items

align-items + justify-items

place-self

align-self + justify-self

One value sets both axes equally. Two values: first = block (cross) axis, second = inline (main) axis.

5. Wrapping in Depth

```
/* flex-wrap: wrap — items flow onto new lines when they don't fit */
.card-row {
    display:   flex;
    flex-wrap: wrap;
    gap:       16px;
}

.card {
    flex: 1 1 280px;  /* min 280px, grow to fill, shrink if needed */
    /* On a 900px container: 3 cards per row (3×280 + 2×16 = 872px) */
    /* On a 600px container: 2 cards per row */
    /* On a 320px container: 1 card per row */
}

/* The "stretching last row" problem with flex: 1 */
/* If 5 items wrap to 3+2, the last 2 each take 50% width */
/* Fix: add invisible spacer items, or switch to Grid auto-fit */
.card-row::after {
    content: '';
```

```
    flex:    1 1 280px;  /* phantom item prevents last-row stretching */
}
```

> **align-content has no effect with a single line.** If your flex container doesn't wrap (or all items fit on one line), `align-content` does nothing — only `align-items` applies. This is a common source of confusion when switching between wrapping and non-wrapping containers.

6. Production Patterns

Perfect centring

```
display: flex;
justify-content: center;
align-items: center;
/* Works for both row and column direction
```

Nav bar — logo left, links right

```
/* container */
display: flex;
align-items: center;

/* push links to the right */
.nav-links { margin-left: auto; }
```

Sticky footer

```
body {
  display: flex;
  flex-direction: column;
  min-height: 100dvh;
}
main { flex: 1; } /* grows to fill */
```

Icon + text alignment

```
display: inline-flex;
align-items: center;
gap: 0.5em;
/* icon and text share baseline */
```

Equal-height cards

```
/* container */
display: flex;
align-items: stretch; /* default */
/* all cards naturally equal height */
/* push CTA to card bottom: */
.card { display: flex;
        flex-direction: column; }
.card__cta { margin-top: auto; }
```

Holy grail layout (flex)

```
body { display: flex;
       flex-direction: column; }
.content-row { display: flex;
               flex: 1; }
nav    { flex: 0 0 200px; }
main   { flex: 1; }
aside  { flex: 0 0 160px; }
```

Responsive nav collapse

Auto margin trick

```
nav { display: flex;
      flex-wrap: wrap;
      gap: 8px; }
nav a { flex: 1 1 120px; }
/* links wrap naturally — no media query nee
```

```
/* margin: auto on a flex item absorbs */
/* all remaining free space */
.item { margin-left: auto; }
/* pushes this item to the far right */

.item { margin: auto; }
/* centres this item in all directions */
```

7. Common Mistakes

```
/* MISTAKE 1: Setting width on flex items instead of flex-basis */
.item {
    width: 200px;  /* overridden by flex-basis if both exist */
}
/* Fix: use flex: 0 0 200px instead */

/* MISTAKE 2: Forgetting min-width: 0 on flex items with overflow content */
.item {
    flex: 1;
    /* flex items have min-width: auto by default */
    /* this prevents shrinking below content size */
    /* long text or images may overflow the container */
}
.item {
    flex:      1;
    min-width: 0;  /* allows flex item to shrink below content size */
}

/* MISTAKE 3: Using justify-items in flexbox (it's a Grid property) */
.flex-container {
    justify-items: center;  /* does nothing in flex */
}
/* Fix: use justify-content for main axis, align-items for cross axis */

/* MISTAKE 4: Expecting align-content to work without wrapping */
.flex-container {
    flex-wrap:     nowrap;     /* default */
    align-content: space-between;  /* ignored — only one line */
}

/* MISTAKE 5: flex: 1 on items inside a container without a height */
```

```
.container {
    display:        flex;
    flex-direction: column;
    /* no height set — flex: 1 on children can't distribute space they don't hav
}
/* Fix: give the container a height (height: 100dvh, min-height, etc.) */
```

Chapter Summary

Concept	Key point
`Main vs cross axis`	justify-* = main axis. align-* = cross axis. Which is which depends on flex-direction. In row: main = horizontal, cross = vertical.
`flex-grow`	Distributes free space as a ratio. 0 = don't grow (default). flex-grow: 2 gets twice the free space of flex-grow: 1.
`flex-shrink`	Controls shrinking when content overflows. 1 = shrink (default). 0 = refuse to shrink (rigid). Weighted by flex-basis.
`flex-basis`	The starting size before grow/shrink apply. auto = use width/height. 0% = ignore content size (used by flex: 1).
`flex shorthand`	flex: 1 → (1 1 0%). flex: auto → (1 1 auto). flex: none → (0 0 auto). Always use shorthand — it sets correct defaults for omitted values.
`min-width: 0`	Flex items default to min-width: auto, preventing shrink below content size. Add min-width: 0 to items containing text or images that need to shrink.
`margin: auto`	In flexbox, auto margins absorb all remaining free space in that direction. The cleanest way to push one item to the opposite end.
`align-content`	Only works with flex-wrap: wrap and multiple lines. Has zero effect on single-line flex containers.
`order`	Changes visual order only. Keyboard nav and screen readers follow DOM order. Use sparingly and only for non-semantic reordering.
`justify-items`	Does not exist in flexbox — Grid only. Don't confuse with justify-content.

EXERCISES

1. **Flex grow maths:** Create a flex container 800px wide with three items: flex-basis 100px each, flex-grow 1, 2, 3. Calculate the final width of each item before opening DevTools to verify.
2. **Navigation bar:** Build a nav bar with a logo on the left, three navigation links in the centre, and a "Sign in" button on the right — using only flexbox and `margin-left: auto` (no absolute positioning).
3. **Sticky footer:** Create a page where the footer always sits at the bottom of the viewport even when content is short, using `display: flex; flex-direction: column` on the body and `flex: 1` on main.
4. **Equal-height cards with bottom CTA:** Build a row of three cards with varying content lengths. Make all cards equal height and ensure each card's "Read more" button is pinned to the bottom of the card regardless of content length.
5. **min-width: 0 debugging:** Create a flex container with two items: one containing a very long unbroken word, one normal. Observe the overflow. Fix it with min-width: 0 and overflow-wrap: break-word. Explain why the default min-width: auto caused the issue.

Next: Chapter 6 — Deep Dive: Grid. Two-dimensional layouts, named template areas, auto-fit vs auto-fill, subgrid, implicit vs explicit grids, and the patterns that make Grid indispensable for page-level layout.

CHAPTER 6

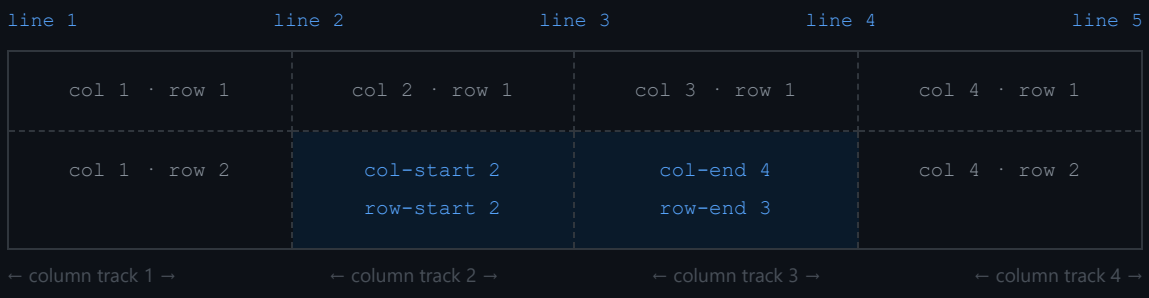
Deep Dive: Grid

Chapter 6 — Deep Dive: Grid

CSS Grid is a two-dimensional layout system — it places items on both rows and columns simultaneously. While Flexbox distributes items along a single axis, Grid lets you design the layout first and place content into it. It is the right tool for page-level structure, dashboard layouts, card grids, and anything that benefits from alignment across two dimensions at once.

1. Grid Lines and Tracks

A grid is defined by its *lines*. Lines are numbered starting at 1 from the start edge. Between two adjacent lines is a *track* (column track or row track). The area bounded by four lines is a *cell*. An item can span multiple cells to form a *grid area*.



Blue item: `grid-column: 2 / 4` · `grid-row: 2 / 3` — spans from line 2 to line 4 horizontally, line 2 to line 3 vertically. Negative line numbers count from the end: `grid-column: 1 / -1` spans the full width regardless of column count.

2. Defining the Grid — Container Properties

Property	Values	Notes
TRACK DEFINITION		

Property	Values	Notes
`grid-template-columns`	`length` · `fr` · `auto` · `min-content` · `max-content` · `repeat()` · `minmax()`	Defines the explicit column tracks. Space-separated list. The most important grid property.
`grid-template-rows`	`same as columns`	Defines the explicit row tracks. Often left unset — rows grow to content height by default.
`grid-template-areas`	`quoted strings of area names`	Names regions of the grid visually. Each string is a row; each word is a cell. Use `.` for empty cells. Makes layouts readable at a glance.
`grid-template`	`rows / columns`	Shorthand for template-rows + template-columns + template-areas. Avoid — overly terse for most team codebases.
IMPLICIT GRID (AUTO-GENERATED TRACKS)		
`grid-auto-columns`	`length` · `fr` · `auto` · `minmax()`	Size of implicitly created column tracks (when items are placed outside the explicit grid).
`grid-auto-rows`	`length` · `fr` · `auto` · `minmax()`	Size of implicitly created row tracks. `grid-auto-rows: minmax(100px, auto)` is a common pattern for variable-height rows with a minimum.
`grid-auto-flow`	`row` · `column` · `dense` · `row dense` · `column dense`	**row** (default) — items fill rows left to right. **column** — fills columns top to bottom. **dense** — backfills gaps with smaller items (changes visual order — accessibility concern).
GAPS AND ALIGNMENT		
`gap` / `row-gap` / `column-gap`	`length` · `percentage`	Same as flexbox. gap: 16px 24px sets row-gap 16px, column-gap 24px.
`justify-items`	`stretch` · `start` · `end` · `center`	Aligns items **inline (horizontally)** within their grid cell. Default stretch. Unlike flexbox, this property is meaningful in Grid.
`align-items`	`stretch` · `start` · `end` · `center` · `baseline`	Aligns items **block (vertically)** within their grid cell.

Property	Values	Notes
`justify-content`	`start` · `end` · `center` · `stretch` · `space-between` · `space-around` · `space-evenly`	Distributes the tracks themselves within the grid container when tracks are smaller than the container. Rarely needed with fr units (fr absorbs all space).
`align-content`	same as `justify-content`	Distributes row tracks vertically. Only applies when the total track height is less than the container height.

3. Track Sizing — `fr`, `minmax()`, `repeat()`

The fr unit

```
/* fr = fraction of remaining space after fixed tracks are placed */

.grid {
    grid-template-columns: 200px 1fr;
    /* sidebar: fixed 200px. main: gets ALL remaining space */
}

.grid {
    grid-template-columns: 1fr 2fr 1fr;
    /* total 4 parts: left=25%, centre=50%, right=25% */
}

.grid {
    grid-template-columns: 200px 1fr 160px;
    /* left sidebar 200px, right sidebar 160px, middle gets everything left ove
}

/* fr vs % — fr is better because it accounts for gap */
/* 1fr 1fr with gap: 20px → each column = (container - 20px) / 2 */
/* 50% 50% with gap: 20px → total = container + 20px → overflow */
```

minmax()

```css
/* minmax(minimum, maximum) — a track can't be smaller than min or larger than m

.grid {
    grid-template-columns: minmax(200px, 1fr) minmax(200px, 1fr);
    /* each column: at least 200px, share remaining space equally */
}

.grid {
    grid-auto-rows: minmax(100px, auto);
    /* rows: at least 100px tall, grow to fit content */
}

/* min-content and max-content as minmax arguments */
.grid {
    grid-template-columns: minmax(min-content, 200px) 1fr;
    /* first column shrinks to the smallest content unit but never exceeds 200px
}
```

repeat()

```css
/* repeat(count, track-definition) */

.grid {
    grid-template-columns: repeat(3, 1fr);          /* 3 equal columns */
    grid-template-columns: repeat(4, 200px 1fr);    /* 8 columns: 200px 1fr 200px
    grid-template-columns: repeat(12, 1fr);          /* classic 12-column grid */
}

/* auto-fill and auto-fit — responsive without media queries */
.grid {
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    grid-template-columns: repeat(auto-fit,  minmax(200px, 1fr));
}
```

auto-fit vs auto-fill — the critical difference

auto-fit

auto-fill

Creates as many tracks as fit. **Collapses empty tracks to zero width.** If you have 3 items and space for 5 columns, the 3 items grow to fill the full container width. Items stretch to use all available space.

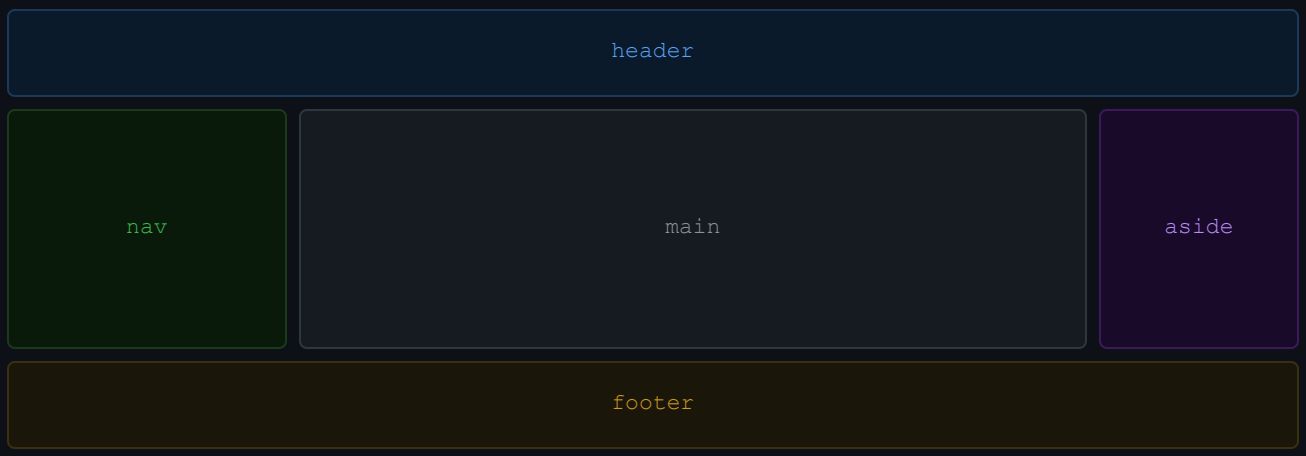
```
repeat(auto-fit, minmax(200px, 1fr))
```

Creates as many tracks as fit. **Keeps empty tracks at their minimum size.** If you have 3 items and space for 5 columns, the 3 items stay at minimum size and the 2 empty columns reserve space at the right edge.

```
repeat(auto-fill, minmax(200px, 1fr))
```

> **Use auto-fit for most card grids.** You almost always want items to grow and fill available space when there are fewer items than columns. Use auto-fill when you want to preserve the column positions for future items — like a calendar grid or a fixed-column dashboard that must stay aligned even when some cells are empty.

4. Named Template Areas




```
/* The CSS that produces the layout above */
.layout {
    display:               grid;
    grid-template-columns: 140px 1fr 100px;
    grid-template-rows:    44px 1fr 44px;
    grid-template-areas:
        "header header header"
        "nav    main   aside"
        "footer footer footer";
    gap:                   8px;
    min-height:            100dvh;
}

.header { grid-area: header; }
```

```
.nav    { grid-area: nav;    }
.main   { grid-area: main;   }
.aside  { grid-area: aside;  }
.footer { grid-area: footer; }

/* Responsive: stack on mobile by redefining the template areas */
@media (max-width: 600px) {
    .layout {
        grid-template-columns: 1fr;
        grid-template-areas:
            "header"
            "nav"
            "main"
            "aside"
            "footer";
    }
}
```

> **Named areas implicitly create named lines.** Declaring `grid-area: header` also creates lines named `header-start` and `header-end` (both column and row). You can reference these in `grid-column: header-start / header-end` — useful for overlapping items onto named regions without repeating line numbers.

5. Item Placement Properties

```
/* Placing items by line number */
.item {
    grid-column-start: 2;
    grid-column-end:   4;   /* spans from line 2 to line 4 = 2 column tracks */
    grid-row-start:    1;
    grid-row-end:      3;   /* spans 2 row tracks */
}

/* Shorthand: grid-column: start / end */
.item {
    grid-column: 2 / 4;   /* line 2 to line 4 */
    grid-column: 1 / -1;  /* line 1 to last line — full width */
    grid-column: 2 / span 3; /* start at line 2, span 3 tracks */
    grid-column: span 2;   /* auto-placed but spans 2 columns */
```

```
}

/* grid-area shorthand: row-start / col-start / row-end / col-end */
.item {
    grid-area: 1 / 2 / 3 / 4;  /* row 1→3, col 2→4 */
}

/* Individual item alignment override */
.item {
    justify-self: center;  /* inline axis — overrides justify-items */
    align-self:   end;     /* block axis — overrides align-items */
    place-self:   center;  /* shorthand: both axes */
}
```

6. Explicit vs Implicit Grid

```
/* Explicit grid — you define the tracks */
.grid {
    grid-template-columns: repeat(3, 1fr);
    grid-template-rows:    200px 200px;
    /* Defines a 3×2 explicit grid */
}

/* If a 7th item is placed, it creates an IMPLICIT row */
/* By default implicit rows are sized to auto (content height) */

.grid {
    grid-auto-rows: minmax(200px, auto);
    /* All implicit rows: minimum 200px, grow to content */
}

/* grid-auto-flow: dense — backfills holes with smaller items */
.masonry-ish {
    display:         grid;
    grid-template-columns: repeat(3, 1fr);
    grid-auto-flow:  dense;
    /* Items that span 2 columns leave a gap; dense fills it with the next smal
```

```
    /* WARNING: reorders items visually — bad for keyboard/screen reader users */
}
```

7. Named Lines

```
/* Lines can be named in square brackets */
.grid {
    grid-template-columns:
        [full-start] 1fr
        [content-start] minmax(0, 800px) [content-end]
        1fr [full-end];
}

/* Items then reference lines by name */
.full-bleed {
    grid-column: full-start / full-end;
}
.content {
    grid-column: content-start / content-end;
}

/* Great for article layouts where most content is centred */
/* but hero images, code blocks, etc. break out to full width */
```

8. Subgrid

Subgrid allows a nested grid to inherit and participate in its parent's track definitions. Without subgrid, nested grids have completely independent track sizes — columns can't align across cards. Supported in all modern browsers since late 2023.

```
/* The card alignment problem subgrid solves */
.card-grid {
    display:               grid;
    grid-template-columns: repeat(3, 1fr);
    grid-template-rows:    auto auto auto 1fr;
    /* row 1: image · row 2: title · row 3: body · row 4: CTA */
```

```
}

.card {
    display:        grid;
    grid-row:       span 4;     /* each card spans all 4 row tracks */
    grid-template-rows: subgrid; /* inherit parent's row track sizes */
}

/* Now all card images, titles, and body text align horizontally */
/* across cards even when content lengths differ */

.card__image { grid-row: 1; }
.card__title { grid-row: 2; }
.card__body  { grid-row: 3; }
.card__cta   { grid-row: 4; }
```

9. Production Patterns

Responsive card grid (no media queries)

```
display: grid;
grid-template-columns:
  repeat(auto-fit, minmax(260px, 1fr));
gap: 24px;
```

Full-page layout

```
display: grid;
grid-template-rows: auto 1fr auto;
min-height: 100dvh;
/* header · main · footer */
```

Sidebar layout

```
display: grid;
grid-template-columns:
  minmax(200px, 25%) 1fr;
/* sidebar never below 200px */
```

Perfect centring

```
display: grid;
place-items: center;
/* single-line centering
   works in any grid cell */
```

Overlapping items

```
/* multiple items, same area */
.grid { display: grid; }
.image { grid-area: 1/1/3/3; }
.caption { grid-area: 1/1/3/3;
           z-index: 1;
           align-self: end; }
```

Article with breakout

```
grid-template-columns:
  [full-s] 1fr
  [content-s] min(65ch,100%) [content-e]
  1fr [full-e];
.content { grid-column: content; }
.full    { grid-column: full; }
```

Dashboard grid

```
grid-template-columns: repeat(12, 1fr);
grid-auto-rows: minmax(80px, auto);

.widget-sm { grid-column: span 3; }
.widget-md { grid-column: span 6; }
.widget-lg { grid-column: span 12; }
```

Masonry-ish (dense)

```
grid-template-columns:
  repeat(auto-fill, minmax(200px, 1fr));
grid-auto-rows: 8px;
grid-auto-flow: dense;
/* items set grid-row: span N
   JS measures height → sets N */
```

10. Grid vs Flexbox — When to Use Which

```
/* Use FLEXBOX when: */
/* - One-dimensional: a row of buttons, a horizontal nav, a stack of cards */
/* - Items dictate their own size and you distribute remaining space */
/* - Content-out: "fit as many as will go, then wrap" */
/* - Examples: nav bar, button group, tag list, media object (image + text) */

/* Use GRID when: */
/* - Two-dimensional: rows AND columns matter simultaneously */
/* - Layout-in: you define the layout structure first, then place content into i
/* - Items need to align across rows (equal-height columns, card grids) */
/* - You need named areas, overlapping, or breakout elements */
/* - Examples: page layout, dashboard, data table, photo gallery, card grid */

/* They compose freely — a grid item can be a flex container and vice versa */
```

Chapter Summary

Concept	Key point
`fr unit`	Fraction of remaining space after fixed tracks are placed. Better than % because it accounts for gap. 1fr 2fr = 33% / 67% of available space.
`minmax()`	Sets a min and max for a track. minmax(200px, 1fr) = at least 200px, grow to fill. minmax(min-content, auto) = shrink to content, grow freely.
`auto-fit vs auto-fill`	Both fill as many tracks as fit. auto-fit collapses empty tracks so items grow. auto-fill preserves empty track space. Use auto-fit for most card grids.

Concept	Key point
`grid-template-areas`	Names regions visually in CSS. Makes page layouts readable at a glance. Implicitly creates named lines (area-start / area-end). Responsive via media query redefinition.
`grid-column: span N`	Place an item without specifying start line — auto-placed but spans N tracks. Combine with auto-fit for responsive span layouts.
`-1 line number`	grid-column: 1 / -1 spans the full explicit grid width. Negative numbers count from the end edge of the explicit grid.
`Implicit grid`	Tracks created automatically for items that overflow the explicit grid. Size them with grid-auto-rows / grid-auto-columns.
`grid-auto-flow: dense`	Backfills gaps with smaller items — changes visual order. Avoid for content with meaningful reading order.
`subgrid`	Child grid inherits parent track definitions. Solves cross-card alignment for titles, bodies, and CTAs. Supported in all modern browsers.
`place-items: center`	The single cleanest way to centre content inside a grid cell. Shorthand for align-items + justify-items.

EXERCISES

1. **Named area layout:** Build a full-page layout with header, sidebar nav, main content, and footer using `grid-template-areas`. Add a single media query that stacks everything into one column on mobile by redefining the template areas string.
2. **auto-fit card grid:** Create a card grid using `repeat(auto-fit, minmax(240px, 1fr))`. Add 3 cards, then 6, then 2. Observe how cards grow when there are fewer than would fill a row. Compare with auto-fill.
3. **Spanning items:** Build a photo gallery grid where the first photo spans 2 columns and 2 rows, and every other photo occupies a single cell. Use `grid-column: span 2; grid-row: span 2` on the featured photo.
4. **Article breakout:** Create an article layout using named lines — `[full-start]` and `[content-start / content-end]` and `[full-end]`. Make body text constrained to the content column but hero images and pull-quotes break out to full width.
5. **Subgrid cards:** Build a 3-column card grid where each card has an image, title, body text, and CTA button. Without subgrid, observe how mismatched content lengths misalign the

CTAs. Then apply `grid-template-rows: subgrid` to the cards and fix the alignment.

Next: Chapter 7 — Deep Dive: Responsive Design. Media queries in depth, container queries, the viewport meta tag, fluid sizing with clamp() and viewport units, intrinsic layouts that need no breakpoints, and a modern responsive workflow.

CHAPTER 7

Deep Dive: Responsive Design

Chapter 7 — Deep Dive: Responsive Design

Responsive design is the practice of making a layout adapt to the space available to it — whether that's a 320px phone, a 1440px desktop, or a 280px sidebar widget. Modern CSS has three distinct tools for this: media queries (respond to the viewport), container queries (respond to the component's own container), and intrinsic layouts (respond to content without any query at all). Understanding when to use each is the mark of a mature CSS practitioner.

1. The Viewport Meta Tag

Before any CSS responsive work can function on a mobile device, the correct viewport meta tag must be in the `<head>`. Without it, mobile browsers render the page at a virtual ~980px width and scale it down — your media queries fire at the wrong sizes and everything looks zoomed out.

```
<!-- Required in every HTML page -- goes in <head> -->
<meta name="viewport" content="width=device-width, initial-scale=1">

/* width=device-width  — use the actual device pixel width, not the virtual 980p
/* initial-scale=1     — don't scale the page on first load */

/* Do NOT add: */
/* user-scalable=no    — prevents pinch-zoom, accessibility violation */
/* maximum-scale=1     — same problem — also blocks browser text zoom */
```

> **Never disable user scaling.** `user-scalable=no` and `maximum-scale=1` prevent users from zooming in — breaking WCAG 1.4.4 (Resize Text). Users with low vision depend on this. Both values are ignored by iOS Safari since version 10, but they still harm accessibility on Android and desktop. Never use them.

2. Media Queries in Depth

Syntax — old and new

```
/* Classic syntax (still fully valid) */
@media screen and (min-width: 768px) { }
@media screen and (max-width: 1023px) { }
@media screen and (min-width: 600px) and (max-width: 900px) { }
@media print { }
@media not screen { }

/* Modern range syntax (Chrome 104+, Firefox 63+, Safari 16.4+) */
@media (width >= 768px) { }
@media (width <= 1023px) { }
@media (600px <= width <= 900px) { }

/* Boolean logic */
@media (min-width: 600px) and (orientation: landscape) { }
@media (max-width: 400px), (min-width: 900px) { /* comma = OR */ }
@media not (hover: hover) { }

/* In <link> tags — loads CSS but always applies, just with different specificit
/* <link rel="stylesheet" media="(min-width: 768px)" href="wide.css"> */
/* The file still downloads — only use for print stylesheets to avoid blocking
```

Mobile-first vs desktop-first

```
/* MOBILE-FIRST: write base styles for small screens, add complexity up */
.nav {
    display: flex;
    flex-direction: column;    /* mobile: stacked */
}

@media (min-width: 768px) {
    .nav {
        flex-direction: row;       /* tablet+: horizontal */
    }
}

/* DESKTOP-FIRST: write for large screens, subtract down */
```

```css
.nav {
    display: flex;
    flex-direction: row;
}

@media (max-width: 767px) {
    .nav {
        flex-direction: column;
    }
}

/* Mobile-first is recommended: simpler base styles, progressively enhanced. */
/* Network: simple CSS loads first, complex overrides only if needed. */
```

Common breakpoints — a practical reference

Name	min-width	Targets	Notes
xs / base	—	Small phones (320–479px)	The default — no query needed with mobile-first approach.
sm	480px	Large phones	Side-by-side form fields, slightly wider cards.
md	768px	Tablets / large phones landscape	Horizontal nav, two-column layouts. Most significant breakpoint.
lg	1024px	Small laptops	Three-column layouts, sidebars appear.
xl	1280px	Desktops	Full layouts, wider content areas.
2xl	1536px	Large monitors	Max-width containers become relevant here — constrain prose width.

> **Design to content, not to device.** The exact breakpoint values above are conventions from frameworks like Tailwind — not laws. The right breakpoints are wherever your specific content breaks. Resize your browser slowly and add a breakpoint wherever the layout looks uncomfortable, not at arbitrary pixel targets.

Every media feature — the full set

width / height

`min-width` · `max-width` · `exact`

Viewport dimensions. The most-used feature. Use range syntax for clarity: `(width >= 768px)`.

orientation

`portrait` · `landscape`

Portrait = height > width. **Unreliable alone** — a tall desktop window is "portrait". Combine with width for meaningful rules.

hover

`hover` · `none`

hover: hover = primary pointing device can hover (mouse). **hover: none** = touch-only. Use to conditionally show hover styles and avoid sticky hover states on mobile.

pointer

`fine` · `coarse` · `none`

fine = mouse (precise). **coarse** = touchscreen or game controller (imprecise). Use to enlarge tap targets: `(pointer: coarse)`.

any-hover / any-pointer

`same as hover / pointer`

Considers **any** connected input device, not just the primary one. A device with both a touchscreen and a Bluetooth mouse has any-hover: hover.

prefers-color-scheme

`light` · `dark`

Respects the OS-level dark/light mode setting. Use to provide a dark theme without JavaScript. Combine with CSS custom properties for clean theme switching.

prefers-reduced-motion

`reduce` · `no-preference`

User has requested reduced motion in OS accessibility settings. **Must be respected** — disable or reduce all non-essential animations and transitions.

prefers-contrast

`more` · `less` · `no-preference` · `forced`

User has increased contrast in OS settings. Use to boost border visibility, text contrast, or switch to a high-contrast palette.

forced-colors

`active` · `none`

Windows High Contrast Mode is active. The OS overrides colours. Use to ensure your UI remains functional — borders become critical as backgrounds are replaced.

resolution

`dpi` · `dpcm` · `dppx`

Screen pixel density. `min-resolution: 2dppx` targets retina/HiDPI screens. Use to serve higher-resolution images — though `srcset` on img is better for this.

display-mode

`browser` · `standalone` · `fullscreen` · `minimal-ui`

How the web app is displayed. `standalone` = installed as PWA. Use to show/hide browser-specific UI elements when running as an installed app.

prefers-reduced-data

`reduce` · `no-preference`

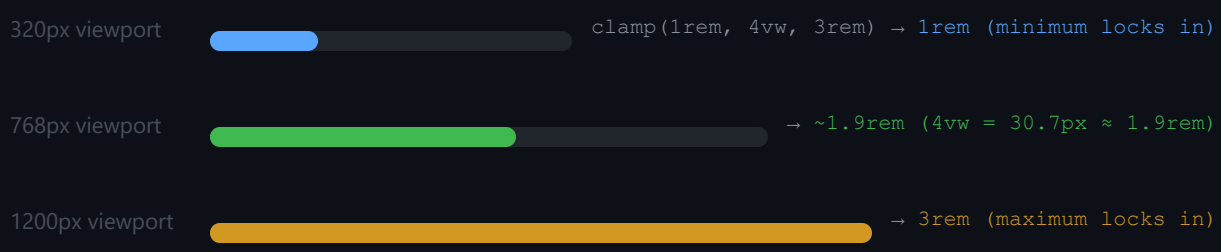
User is on a metered/slow connection. Use to skip loading decorative images, web fonts, or video backgrounds. Limited browser support currently.

3. Responsive Units

Unit	Definition	Best used for
`%`	Percentage of the parent element's corresponding dimension	Widths inside a container. Not reliable for heights (parent must have explicit height).
`vw`	1% of the viewport width	Full-width sections, fluid font sizes. Causes layout shift on mobile when scrollbar appears — use svw or dvw instead.
`vh`	1% of the viewport height	Avoid for mobile layouts — the address bar changes viewport height dynamically, causing content to jump.
`svh`	1% of the **small** viewport height (browser chrome visible)	Safe hero sections that fit when the address bar is shown. Most conservative — always fits.
`lvh`	1% of the **large** viewport height (browser chrome hidden)	Full-screen elements when you want to use maximum space after scrolling hides the address bar.
`dvh`	1% of the **dynamic** viewport height (updates as chrome shows/hides)	Always-accurate full-height sections — but causes reflow as address bar animates. Use deliberately.
`cqw / cqh`	1% of the query container's width / height	Inside container queries — size elements relative to their container, not the viewport. The container-equivalent of vw/vh.
`ch`	Width of the "0" character in the current font	Prose line length: `max-width: 65ch`. Scales with font size automatically.
`rem`	Root element font size (usually 16px)	Spacing, font sizes. Consistent across the component tree. Respects browser font size preference.
`em`	Current element's font size	Padding/margin that should scale with an element's own text size (e.g. button padding). Compounding risk in nested elements.

clamp() — fluid sizing without breakpoints

`clamp(minimum, preferred, maximum)` — the value is the preferred expression, clamped between min and max.



```css
/* clamp() patterns */

/* Fluid heading */
h1 {
    font-size: clamp(1.75rem, 4vw + 1rem, 3.5rem);
    /* The "4vw + 1rem" preferred value grows with viewport width */
    /* Adding 1rem prevents it being tiny on very small screens */
}

/* Fluid spacing */
.section {
    padding: clamp(2rem, 5vw, 6rem);
}

/* Fluid container width */
.container {
    width:     min(100% - 2rem, 1200px);
    margin:    0 auto;
    /* min() picks the smaller: full width minus padding, or 1200px max */
    /* No media query needed — automatically adapts */
}

/* Fluid type scale — all headings scale together */
:root {
    --step-0: clamp(1rem,    0.91rem + 0.46vw, 1.25rem);
    --step-1: clamp(1.25rem, 1.15rem + 0.54vw, 1.56rem);
    --step-2: clamp(1.56rem, 1.41rem + 0.76vw, 1.95rem);
    --step-3: clamp(1.95rem, 1.73rem + 1.11vw, 2.44rem);
    /* Each step is ~1.25× the previous — a fluid modular scale */
}
```

4. Container Queries

Container queries let a component respond to the size of its *own container* rather than the viewport. This solves the fundamental problem with media queries for component-based design: a card in a sidebar and the same card in the main content area are at the same viewport width, but need different layouts.

Media query problem	Container query solution

A card component lives in a 3-column grid. It responds to `@media (min-width: 768px)`. You move the same card into a narrow sidebar. **The viewport is still wide, so the media query fires — the card renders in its wide layout inside a narrow column.** You must override with more CSS.

The card responds to `@container (min-width: 400px)`. In the 3-column grid its container is wide — wide layout. In the sidebar its container is narrow — compact layout. **The same component works correctly in both places automatically.**

```
/* Step 1: establish a containment context on the parent */
.card-wrapper {
    container-type: inline-size;  /* respond to width changes */
    container-name: card;         /* optional name for targeting */
}

/* Shorthand */
.card-wrapper {
    container: card / inline-size;
}

/* Step 2: write a container query inside the component */
@container card (min-width: 400px) {
    .card {
        display:               grid;
        grid-template-columns: 120px 1fr;
    }
}

/* Without a name, @container matches the nearest container ancestor */
@container (min-width: 400px) {
    .card { }
}

/* container-type values: */
/* inline-size — respond to width (most common) */
/* size        — respond to both width and height */
/* normal      — default, establishes style query context only */

/* Container query units */
@container (min-width: 400px) {
    .card__image {
        width: 30cqw;   /* 30% of the container's width */
    }
}
```

Style queries — the next frontier

```
/* Style queries: respond to a container's CSS custom property value */
/* Still limited browser support (Chrome 111+) — use as progressive enhancement

.card-wrapper {
    container-type: style;
    --variant:       featured;
}

@container style(--variant: featured) {
    .card {
        background: linear-gradient(135deg, #1a2a4a, #0d1117);
        border-color: #58a6ff;
    }
}
/* Enables data-driven styling without class toggling */
```

> **The container cannot query itself.** The containment context must be set on a *parent* of the element being restyled. If you set `container-type` on the card itself and write `@container` rules for `.card`, it won't work — the query must travel up to the nearest ancestor with containment. Wrap components in a neutral div or use the parent element.

5. Intrinsic Layouts — No Queries Needed

The best responsive CSS often needs no query at all. Several modern CSS features produce naturally adaptive layouts by responding to available space intrinsically.

Responsive grid (0 queries)

```
display: grid;
grid-template-columns:
  repeat(auto-fit, minmax(240px, 1fr));
gap: 24px;
/* wraps automatically at 240px */
```

Fluid container (0 queries)

```
width: min(100% - 2rem, 1200px);
margin-inline: auto;
/* full width on small, max
   1200px centred on large */
```

Flex wrapping nav (0 queries)

Fluid typography (0 queries)

```
display: flex;
flex-wrap: wrap;
gap: 8px;
nav a { flex: 1 1 120px; }
/* links wrap at 120px min */
```

```
font-size:
  clamp(1rem, 2.5vw + 0.5rem, 1.5rem);
/* grows smoothly with viewport,
   min/max clamps prevent extremes */
```

Responsive sidebar (RAM pattern)

```
/* Responsive, Asymmetric, Modular */
display: flex;
flex-wrap: wrap;
.sidebar { flex: 1 1 200px; }
.main    { flex: 1 1 500px; }
/* stack when combined < 700px */
```

Aspect ratio boxes

```
aspect-ratio: 16 / 9;
width: 100%;
/* height is always 56.25%
   of width — responsive video */
```

6. Responsive Images

```
/* CSS baseline — always set this */
img {
    max-width:  100%;   /* never overflow container */
    height:     auto;   /* preserve aspect ratio */
    display:    block;  /* remove inline whitespace gap */
}

/* object-fit — control how img fills its box (like background-size) */
.card__image {
    width:      100%;
    height:     200px;
    object-fit: cover;     /* fill box, crop to maintain ratio */
    object-position: center top;  /* anchor to top (for portraits) */
}

/* HTML for art-directed responsive images */
<picture>
  <source media="(min-width: 1024px)" srcset="hero-wide.webp">
  <source media="(min-width: 640px)"  srcset="hero-mid.webp">
  <img src="hero-mobile.jpg" alt="Hero image" loading="lazy" decoding="async">
</picture>

/* srcset + sizes for resolution-switching (not art direction) */
<img
```

```
  src="photo-800.jpg"
  srcset="photo-400.jpg 400w, photo-800.jpg 800w, photo-1600.jpg 1600w"
  sizes="(min-width: 1024px) 800px, (min-width: 640px) 50vw, 100vw"
  alt="Photo">
```

7. Dark Mode with prefers-color-scheme

```
/* CSS custom property approach — cleanest architecture */
:root {
    --color-bg:   #ffffff;
    --color-text: #1a1a1a;
    --color-card: #f5f5f5;
    --color-border: #e0e0e0;
}

@media (prefers-color-scheme: dark) {
    :root {
        --color-bg:   #0d1117;
        --color-text: #c9d1d9;
        --color-card: #161b22;
        --color-border: #30363d;
    }
}

/* Components use variables — no per-component dark mode overrides needed */
body  { background: var(--color-bg);   color: var(--color-text); }
.card { background: var(--color-card); border: 1px solid var(--color-border); }

/* Allow manual override via data attribute (user toggle button) */
[data-theme="dark"]  { --color-bg: #0d1117; /* … */ }
[data-theme="light"] { --color-bg: #ffffff; /* … */ }
/* JS sets document.documentElement.dataset.theme = 'dark' */
```

Respecting reduced motion

```
/* Always wrap non-essential animations in this query */
@media (prefers-reduced-motion: no-preference) {
    .card {
        transition: transform 0.3s ease;
```

```
    }
    .card:hover {
        transform: translateY(-4px);
    }
}

/* Alternative: define all animations, then strip in reduced-motion */
.animated {
    animation: slide-in 0.4s ease;
}

@media (prefers-reduced-motion: reduce) {
    *, *::before, *::after {
        animation-duration:   0.01ms !important;
        animation-iteration-count: 1 !important;
        transition-duration:  0.01ms !important;
        scroll-behavior:      auto !important;
    }
}
```

Chapter Summary

Concept	Key point
`Viewport meta`	width=device-width, initial-scale=1 is required on every page. Never disable user-scalable — it's an accessibility violation.
`Mobile-first`	Write base styles for small screens, use min-width queries to add complexity for larger screens. Simpler base CSS, progressive enhancement.
`Range syntax`	Modern @media (width >= 768px) is cleaner than min-width. Supported in all modern browsers since 2023.
`hover / pointer`	Use (hover: hover) to gate hover effects — prevents sticky hover states on touch devices. Use (pointer: coarse) to enlarge touch targets.
`svh / lvh / dvh`	Prefer svh for safe full-height elements on mobile (avoids address bar jump). dvh is accurate but causes reflow as bar shows/hides.
`clamp()`	clamp(min, preferred, max) for fluid sizing without breakpoints. The preferred value uses vw to grow with viewport. Add a rem offset to prevent tiny sizes.

Concept	Key point
`Container queries`	container-type: inline-size on the parent, then @container (min-width: X) for the child. The component responds to its container, not the viewport — works correctly in any context.
`cqw / cqh`	Container query units — 1% of the container's width/height. Use inside @container rules to size elements relative to the container.
`Intrinsic layouts`	auto-fit with minmax(), flex-wrap, clamp(), min(), aspect-ratio — these adapt without any query. Prefer them over breakpoints where possible.
`prefers-reduced-motion`	Always wrap non-essential animation in @media (prefers-reduced-motion: no-preference). Or strip all durations to 0.01ms in the reduce branch.
`Dark mode`	Define all colours as CSS custom properties in :root. Override them in @media (prefers-color-scheme: dark). Allow data-theme override for user toggle.

EXERCISES

1. **Mobile-first nav:** Build a navigation bar that stacks vertically on mobile (base styles), switches to a horizontal row at 768px, and adds a right-aligned CTA button at 1024px — using only mobile-first min-width media queries.
2. **Fluid type scale:** Create a complete typographic scale (body, h3, h2, h1) using `clamp()` for each. The body should range from 1rem to 1.125rem, and each heading level should be approximately 1.25× larger. No breakpoints allowed.
3. **Container query card:** Build a product card that shows image above content in a narrow container (<360px), and image beside content in a wider container. Place the card in both a full-width area and a narrow sidebar — verify both layouts work without changing the card's CSS.
4. **Dark mode toggle:** Implement a complete dark/light mode system using CSS custom properties. The page should respect `prefers-color-scheme` by default, but a toggle button sets `data-theme` on the root element to override the OS preference. Store the choice in localStorage.

5. **Reduced motion:** Add a subtle card hover animation (lift + shadow). Wrap it in `@media (prefers-reduced-motion: no-preference)`. Test by enabling "Reduce motion" in your OS settings — the animation should disappear entirely, not just slow down.

Next: Chapter 8 — Deep Dive: Transitions and Animations. The transition shorthand in full, cubic-bezier easing, all animation properties, keyframe techniques, scroll-driven animations, performance (composited vs non-composited properties), and accessible motion patterns.

CHAPTER 8

Deep Dive: Transitions and Animations

Chapter 8 — Deep Dive: Transitions and Animations

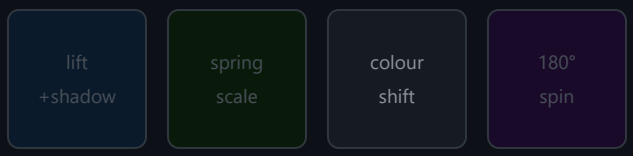
Motion in UI serves communication — it confirms actions, guides attention, and signals state changes. CSS provides two motion systems: **transitions**, which animate between two CSS states triggered by an event, and **animations**, which run on a timeline defined by keyframes, independent of state changes. A third system — **scroll-driven animations** — ties animation progress to scroll position without JavaScript.

1. Transitions

The transition shorthand in full

```
/* transition: property duration easing delay */

.btn {
    transition: background-color 0.2s ease;
    transition: transform 0.2s ease, box-shadow 0.2s ease;   /* multiple */
    transition: all 0.2s ease;        /* all properties — avoid, causes perf iss
    transition: color 0.2s ease 0.1s;  /* 0.1s delay before starting */
}

/* Longhand — when you need different settings per property */
.btn {
    transition-property:        transform, opacity;
    transition-duration:        0.3s, 0.2s;      /* one per property */
    transition-timing-function: ease-out, linear;
    transition-delay:           0s, 0.1s;
}

/* Best practice: put transition on the BASE state, not the :hover state */
.btn       { background: #58a6ff; transition: background 0.2s ease; }
.btn:hover { background: #79c0ff; }
/* Putting it on :hover only would mean no transition OUT of hover state */
```

Hover to demonstrate — live transition demos



transition-behavior — discrete properties

```
/* Most properties transition continuously (opacity, transform, color) */
/* Discrete properties (display, visibility) snap — they can't interpolate */

/* Old problem: animating display: none → display: block */
.modal {
    display:    none;
    opacity:   0;
    transition: opacity 0.3s;
}
.modal.open {
    display:  block;  /* snaps immediately — opacity transition never runs */
    opacity:  1;
}

/* Modern fix: allow-discrete (Chrome 117+, Firefox 129+) */
.modal {
    display:    none;
    opacity:   0;
    transition: opacity 0.3s, display 0.3s allow-discrete;
}
.modal.open {
    display: block;
    opacity: 1;
    /* display switches at END of transition (fade out then hide) */
    /* or START (show then fade in) depending on direction */
}

/* Also enables animating the Popover API and <dialog> in/out */
@starting-style {
    .modal.open {
        opacity: 0;  /* defines the "before entering" state */
    }
}
```

2. Easing Functions

The easing function controls the *rate of change* through a transition or animation — how fast it accelerates and decelerates. Choosing the right easing is as important as the duration. Natural motion rarely moves at a constant speed.

ease

`cubic-bezier(0.25,0.1,0.25,1)`

Starts fast, decelerates. **The default.** Good general purpose.

ease-in

`cubic-bezier(0.42,0,1,1)`

Starts slow, ends fast. **Use for exits** — elements leaving the screen accelerate away naturally.

ease-out

`cubic-bezier(0,0,0.58,1)`

Starts fast, ends slow. **Use for entrances** — elements arriving decelerate to a stop.

ease-in-out

`cubic-bezier(0.42,0,0.58,1)`

Slow start and end, fast middle. **Use for looping animations** and transitions where the element stays on screen.

linear

`cubic-bezier(0,0,1,1)`

Constant speed. Feels mechanical. **Use for colour/opacity transitions** and spinner animations.

spring (custom)

`cubic-bezier(0.34,1.56,0.64,1)`

Overshoots then settles. **Use for scale/position feedback** — gives UI a physical, springy feel.

steps()

`steps(4, end)`

Jumps through N discrete frames. **Use for sprite animations**, typewriter effects, and pixel art.

linear() new

`linear(0,0.5 25%,1)`

Modern multi-point linear easing. Can approximate any curve including springs and bounces with full CSS — no cubic-bezier needed.

```
/* cubic-bezier(x1, y1, x2, y2) — define your own curve */
/* P0 = (0,0) start. P3 = (1,1) end. P1 and P2 are control handles. */
/* y values can exceed 0-1 range to create overshoot (spring effect) */

.spring   { transition-timing-function: cubic-bezier(0.34, 1.56, 0.64, 1); }
.snappy   { transition-timing-function: cubic-bezier(0.2, 0, 0, 1);       }
.material { transition-timing-function: cubic-bezier(0.4, 0, 0.2, 1);     }

/* steps() — discrete jumps */
.typewriter { transition-timing-function: steps(20, end); }
/* step-start = steps(1, start), step-end = steps(1, end) */
```

```
/* Modern linear() — approximate any curve with control points */
.bounce {
    transition-timing-function: linear(
        0, 0.009, 0.035 2.1%, 0.141, 0.281 6.7%, 0.723 12.9%, 0.938 16.7%,
        1.017, 1.077, 1.121, 1.149 24.3%, 1.159, 1.163 26.3%, 1.154 28.3%,
        1.129 30.7%, 1.051 35.7%, 1.017 38.2%, 0.991, 0.977 42.7%, 0.974 44.1%,
        0.975 45.7%, 1.001 55%, 1.004 60%, 1
    );
}
```

3. Keyframe Animations

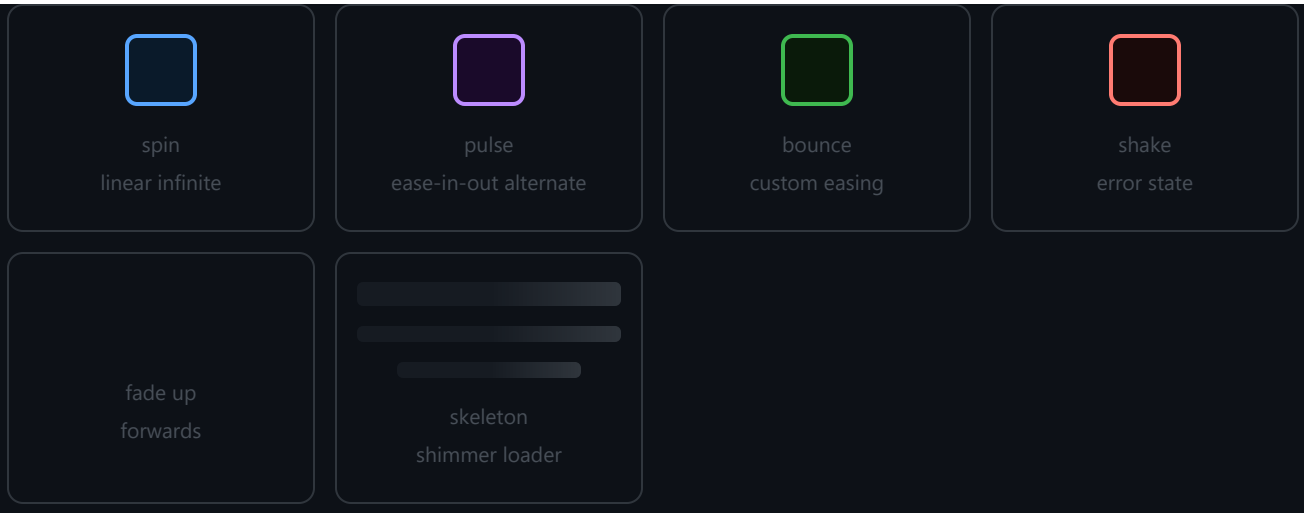
@keyframes syntax

```
/* Named keyframe block */
@keyframes slide-in {
    from {                      /* from = 0% */
        opacity:   0;
        transform: translateY(20px);
    }
    to {                        /* to = 100% */
        opacity:   1;
        transform: translateY(0);
    }
}

/* Multiple stops */
@keyframes progress-pulse {
    0%   { transform: scaleX(0);    opacity: 1;   }
    60%  { transform: scaleX(0.8);  opacity: 1;   }
    100% { transform: scaleX(1);    opacity: 0;   }
}

/* Group stops that share styles */
@keyframes blink {
    0%, 100% { opacity: 1; }
    50%      { opacity: 0; }
}
```

All animation properties

Property	Values	Notes
`animation-name`	`@keyframes name` · `none`	References the @keyframes block. Multiple names comma-separated to run several animations.
`animation-duration`	`time (s or ms)`	How long one cycle takes. 0s is valid — the animation runs but is invisible (useful for staggered delays).
`animation-timing-function`	`ease` · `linear` · `cubic-bezier()` · `steps()` · `linear()`	Applied per keyframe interval, not across the whole animation — important for multi-stop keyframes.
`animation-delay`	`time (s or ms)` · `negative values`	Negative delay starts the animation mid-way through. `animation-delay: -0.5s` with a 1s duration starts at the 50% keyframe — great for staggering.
`animation-iteration-count`	`number` · `infinite`	How many times the animation runs. `infinite` for spinners and loaders. Fractional values are valid (0.5 plays half a cycle).
`animation-direction`	`normal` · `reverse` · `alternate` · `alternate-reverse`	**alternate** plays forward then backward — seamless loops without needing mirrored keyframes. Use for pulsing effects.
`animation-fill-mode`	`none` · `forwards` · `backwards` · `both`	**forwards**: element keeps the final keyframe state after the animation ends. **backwards**: applies 0% keyframe during the delay. **both**: does both. Use `forwards` for one-shot entrance animations.
`animation-play-state`	`running` · `paused`	Pause/resume via CSS or JS. A paused animation stays at its current frame. Toggle with `:hover` to pause spinners on hover.
`animation-composition`	`replace` · `add` · `accumulate`	Controls how multiple animations combine when they target the same property. **add** stacks transforms additively.
`animation`	`shorthand`	Order: name duration easing delay iterations direction fill-mode play-state. The fill-mode and play-state are often omitted. Multiple animations: comma-separated.

Live animation demos



Staggered animations with negative delay

```
/* Stagger items without JavaScript using negative delay */
.list-item {
    animation: slide-in 0.4s ease-out both;
}

.list-item:nth-child(1) { animation-delay: 0ms;   }
.list-item:nth-child(2) { animation-delay: 60ms;  }
.list-item:nth-child(3) { animation-delay: 120ms; }
.list-item:nth-child(4) { animation-delay: 180ms; }

/* Or with a CSS custom property set per element via JS or inline style */
.list-item {
    animation-delay: calc(var(--index) * 60ms);
}
/* <li class="list-item" style="--index: 0"> */
/* <li class="list-item" style="--index: 1"> */
```

4. Performance — Composited Properties

The browser rendering pipeline has several stages: Style → Layout → Paint → Composite. Properties that trigger Layout (reflow) force the browser to recalculate the geometry of every affected element — expensive. Properties that only trigger Paint or Composite are cheap. Stick to composited properties for smooth 60fps animations.

Property	Triggers Layout?	Triggers Paint?	Composited?	Verdict
`transform`	No	No	Yes	Use freely — GPU handled, never jank.
`opacity`	No	No	Yes	Use freely — composited. Combine with transform for entrance effects.
`filter`	No	Sometimes	Often	Usually fine — test on low-end hardware with complex blur values.
`background-color` · `color` · `box-shadow`	No	Yes	No	Paint-only — acceptable for short transitions, avoid in 60fps loops.
`width` · `height` · `margin` · `padding` · `top` · `left`	Yes	Yes	No	Avoid animating. Causes full reflow. Use transform: scale/translate instead.
`font-size` · `line-height`	Yes	Yes	No	Never animate — extremely expensive. Use transform: scale on a wrapper.

```
/* Always use transform/opacity equivalents */

/* SLOW: animating layout properties */
.bad { transition: left 0.3s, width 0.3s; }

/* FAST: transform equivalents */
.good { transition: transform 0.3s;    }

/* Move: left/top → translateX/translateY */
/* Resize: width/height → scaleX/scaleY (size the element in layout, animate sca
/* Show/hide: display → opacity + visibility, or opacity + pointer-events */

/* will-change — promote element to its own layer ahead of time */
.animated-card {
    will-change: transform;
    /* Use sparingly — each layer consumes GPU memory */
    /* Add on hover/focus, remove after animation with JS when possible */
```

```
    /* Never apply to hundreds of elements simultaneously */
}
```

5. Scroll-Driven Animations

Scroll-driven animations link animation progress to scroll position — no JavaScript required. Supported in Chrome 115+ and Firefox 110+ (behind a flag). Two timeline types: **scroll()** links to the scroll position of a scroll container, and **view()** links to how much of an element is visible in the viewport.

```
/* scroll() — animate based on how far a container has been scrolled */
@keyframes widen-bar {
    from { transform: scaleX(0); }
    to   { transform: scaleX(1); }
}

.progress-bar {
    position:       fixed;
    top:            0;
    left:           0;
    width:          100%;
    height:         4px;
    transform-origin: left;
    animation:      widen-bar linear;
    animation-timeline: scroll(root block);
    /* root = the root scroll container, block = vertical axis */
}

/* view() — animate based on element's visibility in the viewport */
@keyframes reveal {
    from { opacity: 0; transform: translateY(40px); }
    to   { opacity: 1; transform: translateY(0);    }
}

.reveal-on-scroll {
    animation:          reveal linear both;
    animation-timeline: view();
    animation-range:    entry 0% entry 40%;
    /* plays while element enters viewport — 0% to 40% of entry */
```

```
}

/* Named timelines for cross-element coordination */
.scroll-container {
    scroll-timeline-name: --my-scroll;
    scroll-timeline-axis: block;
    overflow-y:           scroll;
}

.driven-element {
    animation-timeline: --my-scroll;
    /* can be in a different part of the DOM */
}
```

6. Production Patterns

Loading spinner

```
/* element is a ring, border-top is coloured
width: 24px; height: 24px;
border: 3px solid #30363d;
border-top-color: #58a6ff;
border-radius: 50%;
animation: spin 0.8s linear infinite;
/* @keyframes spin { to { transform: rotate
```

Skeleton shimmer

```
background: linear-gradient(
  90deg,
  #161b22 25%,
  #30363d 50%,
  #161b22 75%
);
background-size: 200% 100%;
animation: shimmer 1.5s linear infinite;
/* from{bg-pos:-200%} to{bg-pos:200%} */
```

Button press feedback

```
.btn {
  transition: transform 0.1s ease-out;
}
.btn:active {
  transform: scale(0.96);
}
/* scale(0.96) on :active = tactile press */
```

Entrance animation

```
@keyframes fade-up {
  from { opacity:0; transform:translateY(16p
  to   { opacity:1; transform:translateY(0);
}
.card {
  animation: fade-up 0.4s ease-out both;
}
```

Typewriter effect

Error shake

```
@keyframes typing {
  from { width: 0; }
  to   { width: 100%; }
}
.typewriter {
  overflow: hidden;
  white-space: nowrap;
  animation: typing 2s steps(30) forwards,
             blink 0.5s step-end infinite;
  border-right: 2px solid #58a6ff;
}
```

```
@keyframes shake {
  0%,100%{ transform:translateX(0); }
  20%    { transform:translateX(-6px); }
  40%    { transform:translateX(6px); }
  60%    { transform:translateX(-4px); }
  80%    { transform:translateX(4px); }
}
.invalid {
  animation: shake 0.4s ease both;
}
```

Read-progress bar

```
.progress-bar {
  position: fixed; top: 0;
  width: 100%; height: 3px;
  transform-origin: left;
  animation: widen-bar linear;
  animation-timeline: scroll();
}
```

Reduced motion reset

```
@media (prefers-reduced-motion: reduce) {
  *, *::before, *::after {
    animation-duration:
      0.01ms !important;
    animation-iteration-count:
      1 !important;
    transition-duration:
      0.01ms !important;
  }
}
```

Chapter Summary

Concept	Key point
`transition shorthand`	property duration easing delay. Put it on the base state so both the forward and reverse transitions run. Never use transition: all — it hits every property including ones you don't want animated.
`ease-in vs ease-out`	ease-in for exits (accelerates away). ease-out for entrances (decelerates to rest). ease-in-out for looping. linear for colour/opacity and spinners.
`cubic-bezier overshoot`	Y values outside 0–1 create overshoot (spring effect). cubic-bezier(0.34, 1.56, 0.64, 1) is a good starting spring.

Concept	Key point
`animation-fill-mode`	forwards keeps the end state after the animation completes. both applies 0% during delay and forwards at end. Essential for one-shot entrance animations.
`Negative delay stagger`	animation-delay: calc(var(--index) * 60ms) staggers items. Set --index via inline style from HTML or JS.
`Composited properties`	Only animate transform and opacity for guaranteed 60fps. Colour, shadow, size changes trigger paint or layout — acceptable for short transitions, not for continuous loops.
`will-change`	Promotes an element to its own GPU layer before animation starts. Use sparingly — each layer consumes memory. Add before animation, remove after.
`scroll() timeline`	animation-timeline: scroll() links animation progress to the scroll position of the nearest scroll container. No JS needed.
`view() timeline`	animation-timeline: view() links to element visibility. animation-range: entry 0% entry 40% plays the animation as the element enters the viewport.
`transition-behavior: allow-discrete`	Enables transitioning discrete properties like display. Use with @starting-style to define the pre-entry state for enter animations.

EXERCISES

1. **Easing comparison:** Create five identical buttons, each with a different easing function (ease, ease-in, ease-out, ease-in-out, and a custom spring cubic-bezier). Give them all `transform: translateX(100px)` on hover with a 0.4s duration. Compare how each feels — identify which suits an entrance vs an exit vs a toggle.
2. **Staggered list entrance:** Create an unordered list of 6 items. When the page loads, each item should fade up with a 60ms stagger between them using `animation-delay: calc(var(--i) * 60ms)` and inline `style="--i: N"`. Use `animation-fill-mode: both` to keep items hidden before their animation starts.
3. **Performance audit:** Build an animation that moves a card across the screen using `left`. Open DevTools Performance panel, record it, and observe the layout recalculations. Rewrite it using `transform: translateX()`. Record again and compare the frame times.

4. **Skeleton loader:** Build a card skeleton with three shimmer bars (title, subtitle, body) using a single `@keyframes shimmer` animation and staggered `animation-delay` on each bar so the shimmer sweeps across sequentially.

5. **Scroll-driven reveal:** Create a long page with six sections. Each section heading should fade in and rise as it enters the viewport using `animation-timeline: view()` and `animation-range: entry 0% entry 30%`. Wrap the whole animation in `@media (prefers-reduced-motion: no-preference)`.

Next: Chapter 9 — CSS Variables and Functions. Custom properties in depth — inheritance, fallbacks, JS interop, and theming patterns. The full CSS function library: calc(), clamp(), min(), max(), color-mix(), env(), attr(), and the emerging typed custom properties with @property.

CHAPTER 9

CSS Variables and Functions

Chapter 9 — CSS Variables and Functions

CSS custom properties (commonly called CSS variables) are one of the most transformative features in modern CSS. Unlike preprocessor variables (Sass `$var`, Less `@var`), they exist at runtime in the browser, inherit through the DOM, can be read and written by JavaScript, and respond to media queries. Combined with the growing CSS function library — `calc()`, `clamp()`, `min()`, `max()`, `color-mix()` and more — they enable design systems that adapt without any tooling.

1. Custom Properties — Fundamentals

Declaration and use

```
/* Declare with -- prefix. Convention: :root for globals */
:root {
    --color-primary:    #58a6ff;
    --color-surface:    #161b22;
    --spacing-base:     1rem;
    --border-radius:    6px;
    --font-heading:     'Georgia', serif;
}

/* Use with var() */
.card {
    background:    var(--color-surface);
    border-radius: var(--border-radius);
    padding:       var(--spacing-base);
}

/* Fallback — second argument to var() */
.widget {
    color: var(--color-accent, #58a6ff);        /* single fallback */
    color: var(--color-accent, var(--color-primary, blue));  /* chained */
}
```

```css
/* Custom properties can hold ANY valid CSS value fragment */
:root {
    --shadow:    0 4px 12px rgb(0 0 0 / 0.3);
    --gradient:  linear-gradient(to right, #58a6ff, #bc8cff);
    --transition: 0.2s ease;
}
.btn {
    box-shadow:  var(--shadow);
    background:  var(--gradient);
    transition:  transform var(--transition);
}

/* Numbers without units — combine with calc() */
:root {
    --scale: 1.25;  /* unitless — can't use directly in font-size */
}
h1 {
    font-size: calc(var(--scale) * 1rem);  /* multiply by unit to get length */
}
```

> **Gotcha — invalid values are not errors.** If a custom property is declared but holds a value that doesn't make sense for the property it's used in (e.g. `--color: hello` used in `color: var(--color)`), the browser doesn't use the fallback — it uses the **inherited value**, or `initial` if there is none. This is different from a missing variable (which does use the fallback).

Inheritance — how custom properties flow down the DOM

Custom properties inherit like any other CSS property — a child element gets the value declared on its nearest ancestor. Redeclaring on a child *overrides* it for that subtree only.

```
:root ← --color-brand: #58a6ff

  body ← inherits --color-brand: #58a6ff

    .card ← inherits --color-brand: #58a6ff

      .card__title ← inherits --color-brand: #58a6ff   used via var()

    .card--alt ← --color-brand: #bc8cff (redeclared!)   overrides for this subtree
```

```
.card--alt .title ← sees --color-brand: #bc8cff   inherits override
```

```css
/* Component-level override — no new class needed on children */
:root       { --color-brand: #58a6ff; }
.card--alt { --color-brand: #bc8cff; }

/* Any descendant of .card--alt now sees the purple value */
.card__title { color: var(--color-brand); }

/* Prevent inheritance — set to initial on a boundary element */
.isolated { --color-brand: initial; }
```

2. Scoping Patterns

GLOBAL TOKEN SCALE

```css
/* Design tokens on :root */
:root {
  --space-1: 4px;
  --space-2: 8px;
  --space-3: 16px;
  --space-4: 24px;
  --space-5: 40px;
  --radius-sm: 4px;
  --radius-md: 8px;
  --radius-full: 9999px;
}
```

COMPONENT-SCOPED

```css
/* Expose knobs, use tokens internally */
.btn {
  --btn-bg: var(--color-primary);
  --btn-color: #fff;
  --btn-radius: var(--radius-md);

  background: var(--btn-bg);
  color:      var(--btn-color);
  border-radius: var(--btn-radius);
}
.btn--ghost {
  --btn-bg: transparent;
  --btn-color: var(--color-primary);
}
```

MEDIA-QUERY OVERRIDE

```css
/* Responsive spacing via variable swap */
:root {
  --page-padding: 1rem;
  --heading-size: 1.5rem;
}
@media (min-width: 768px) {
  :root {
    --page-padding: 3rem;
```

DARK MODE THEMING

```css
:root {
  --bg:   #ffffff;
  --text: #1a1a1a;
  --surface: #f5f5f5;
}
@media (prefers-color-scheme: dark) {
  :root {
    --bg:      #0d1117;
```

```
    --heading-size: 2.5rem;
  }
}
/* All elements using --page-padding
   now update at the breakpoint */
```

```
    --text:    #c9d1d9;
    --surface: #161b22;
  }
}
/* Manual toggle overrides this */
[data-theme="dark"] {
  --bg:      #0d1117;
  --text:    #c9d1d9;
}
```

Dark mode in practice — two themes, one set of rules

Light theme

Enter email

Sign up

Dark theme

Enter email

Sign up

3. JavaScript Interop

```
/* Read a custom property value */
const root = document.documentElement;
const styles = getComputedStyle(root);
const primary = styles.getPropertyValue('--color-primary').trim();
// '--color-primary' → '#58a6ff'

/* Write / override a custom property */
root.style.setProperty('--color-primary', '#ff7b72');

/* Remove an override (reverts to stylesheet value) */
root.style.removeProperty('--color-primary');

/* Theme toggle pattern */
const toggle = document.querySelector('#theme-btn');
toggle.addEventListener('click', () => {
    const current = root.dataset.theme;
    root.dataset.theme = current === 'dark' ? 'light' : 'dark';
    localStorage.setItem('theme', root.dataset.theme);
});
```

```
/* Animate a CSS variable with JS (pointer-tracked glow effect) */
document.addEventListener('mousemove', (e) => {
    root.style.setProperty('--mouse-x', e.clientX + 'px');
    root.style.setProperty('--mouse-y', e.clientY + 'px');
});
/* CSS picks it up: background: radial-gradient(at var(--mouse-x) var(--mouse-y)
```

Performance tip. Setting a custom property on a DOM element triggers a style recalculation for that element and its descendants. Setting it on `:root` via `document.documentElement.style.setProperty()` triggers a global recalc. For high-frequency updates (mouse tracking, scroll), scope the variable to a small subtree or use `requestAnimationFrame` to batch updates.

4. @property — Typed Custom Properties

By default, CSS custom properties are untyped strings. The browser doesn't know if `--angle` is a number, an angle, or a colour — it just substitutes the text. This means you **cannot transition or animate them**. `@property` registers a custom property with a type, initial value, and inheritance flag — unlocking animation, validation, and IDE autocomplete.

```
/* @property syntax */
@property --hue {
    syntax:        '<number>';    /* the CSS type */
    inherits:     false;          /* does not flow to children */
    initial-value: 220;           /* required when inherits: false */
}

/* Now --hue can be animated! */
.rainbow-text {
    --hue: 0;
    color: hsl(var(--hue) 80% 60%);
    animation: cycle-hue 3s linear infinite;
}
@keyframes cycle-hue {
    to { --hue: 360; }
```

```
}
/* Without @property this would snap — the browser wouldn't know --hue is a numb
```

Available syntax types

`<number>`

Any real number. Can be animated between two values. Use for **counters, ratios, multipliers**.

```
--scale: 1.25; --opacity-level: 0.8;
```

`<integer>`

Whole numbers only. Snaps during animation — no interpolation between steps.

```
--z-index-level: 10; --columns: 3;
```

`<length>`

Any CSS length value (px, rem, em, vw...). Can be animated. Use for **spacing, offsets, sizes**.

```
--offset: 24px; --hero-height: 60vh;
```

`<percentage>`

A % value. Animatable. Often combined with `<length-percentage>` to accept both.

```
--fill: 75%; --progress: 0%;
```

`<color>`

Any valid colour. Can be animated — the browser interpolates between colours. Use for **theme tokens you want to transition**.

```
--brand: oklch(70% 0.2 250); --
surface: #161b22;
```

`<angle>`

deg, rad, turn, grad. Can be animated. Use for **gradient rotation, conic effects, transform rotation via custom property**.

```
--rotation: 0deg; --gradient-angle:
135deg;
```

`<image>`

url(), gradient functions. Cannot be animated but can be swapped. Use for **swappable background tokens**.

```
--hero-bg: url('/hero.webp'); --
pattern: repeating-linear-
gradient(...);
```

`*` (universal)

Accepts any value — behaves like an unregistered custom property. Still prevents animation but adds initial-value and inheritance control.

```
/* Same as unregistered but with
defaults */
```

5. The CSS Function Library

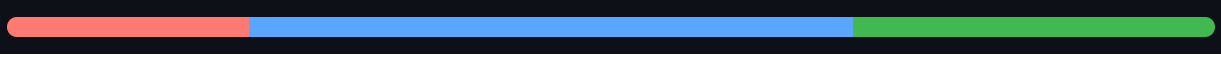
Math functions — calc(), min(), max(), clamp()

```
width: calc(100% - 2rem)          →  full width minus 2rem padding
width: min(100%, 600px)           →  smaller of 100% or 600px
width: max(200px, 50%)            →  larger of 200px or 50%
font-size: clamp(1rem, 4vw, 2rem) →  min 1rem, preferred 4vw, max 2rem
```

```css
/* calc() — mix any units, all four operators */
.sidebar {
    width:  calc(300px - 2 * var(--gap));  /* variables inside calc */
    margin: calc(var(--spacing) * 1.5);  /* multiply unitless var by unit */
}
/* Note: + and - operators MUST have spaces around them */
/* calc(100% -1rem) is invalid. calc(100% - 1rem) is valid. */

/* min() — use for intrinsic fluid container widths */
.container {
    width: min(100% - 2rem, 1200px);  /* fluid up to 1200px, always 1rem margin
}

/* max() — enforce a minimum */
.touch-target {
    min-height: max(44px, 10vh);  /* at least 44px (WCAG), or 10vh if larger */
}

/* clamp(MIN, PREFERRED, MAX) */
:root {
    --step-0: clamp(1rem,    calc(0.875rem + 0.5vw),  1.125rem);
    --step-1: clamp(1.25rem, calc(1rem + 1vw),       1.75rem);
    --step-2: clamp(1.5rem,  calc(1rem + 2vw),       2.5rem);
    --step-3: clamp(2rem,    calc(1rem + 3.5vw),     3.5rem);
}
/* Fluid type scale — no media queries at all */
```

clamp() visualised

`clamp(1rem, 4vw, 2.5rem)` — how font-size responds to viewport width



Narrow viewport	Mid viewport	Wide viewport
locked at 1rem (MIN)	scales with 4vw (PREFERRED)	locked at 2.5rem (MAX)

Full function reference

Function	Category	Description + Example
`calc()`	Math	Arithmetic with mixed units. Supports +, -, *, /. Spaces required around + and -. `calc(100% - var(--sidebar-w) - 2rem)`
`min()`	Math	Returns the smallest argument. Accepts any number of comma-separated values. `width: min(100% - 2rem, 960px)`
`max()`	Math	Returns the largest argument. Useful for enforcing minimum values. `padding: max(1rem, 5vw)`
`clamp()`	Math	Equivalent to max(MIN, min(PREFERRED, MAX)). Clamps a fluid value between two bounds. `font-size: clamp(1rem, 2.5vw, 1.75rem)`
`round()`	Math	Rounds a value to the nearest multiple. Args: strategy (nearest/up/down/to-zero), value, interval. `width: round(nearest, 53.7px, 5px) → 55px`
`mod() / rem()`	Math	Modulo operations. mod() matches divisor sign; rem() matches dividend sign. `mod(18, 5) → 3`
`abs() / sign()`	Math	abs() returns magnitude; sign() returns -1, 0, or 1. Useful with custom properties for mirroring. `margin-left: abs(var(--offset))`
`sin() cos() tan() asin() acos() atan() atan2()`	Trig	Trigonometric functions. Input in radians or angle values. Used for circular layouts and wave effects.

Function	Category	Description + Example
		`--x: calc(cos(var(--angle)) * var(--radius))`
`pow() sqrt() hypot() log() exp()`	Math	Exponential functions. pow(base, exponent), sqrt(), hypot() = $\sqrt{(a^2+b^2)}$. Useful for modular scale generation. `--step-2: calc(pow(1.25, 2) * 1rem)`
`var()`	Variable	Reads a custom property. Optional fallback as second argument. `color: var(--text-color, #c9d1d9)`
`env()`	Environment	Reads environment variables set by the browser. Most commonly used for iOS safe area insets. `padding-bottom: env(safe-area-inset-bottom)`
`attr()`	DOM	Reads an HTML attribute value for use in CSS content. Currently limited to content: in pseudo-elements; typed attr() with units is an emerging spec. `content: attr(data-label)`
`color-mix()`	Color	Mixes two colours in a specified colour space. Percentage controls the mix ratio. `color-mix(in oklch, #58a6ff 70%, #bc8cff)`
`color-contrast()`	Color	Selects from a list the colour with highest contrast against a base. (Emerging — limited support.) `color: color-contrast(#161b22 vs white, black)`
`light-dark()`	Color	Returns the first value in light mode, second in dark mode. Requires color-scheme to be set. `color: light-dark(#1a1a1a, #c9d1d9)`

Function	Category	Description + Example
`oklch() oklab() lch() lab()`	Color	Perceptual colour space functions. oklch(lightness chroma hue) for design-system tokens. `--brand: oklch(60% 0.22 250)`
`hsl() hwb() rgb()`	Color	Classic colour functions. Modern syntax drops commas and uses / for alpha. `hsl(220 80% 60% / 0.8)`
`url()`	Reference	References an external resource — image, font, SVG. `background: url('/images/bg.webp')`
`format() local()`	Font	Used inside @font-face src. format() hints the file type; local() checks for installed fonts. `src: local('Inter'), url('inter.woff2') format('woff2')`
`linear() cubic-bezier() steps()`	Easing	Custom easing functions for transitions and animations. Covered in Chapter 8. `transition-timing-function: cubic-bezier(0.34,1.56,0.64,1)`

6. Full Design Token System

```
/* ── Layer 1: Primitive tokens (raw values, no semantic meaning) */
:root {
    /* Color palette */
    --blue-50:  oklch(96% 0.03 250);
    --blue-500: oklch(60% 0.22 250);
    --blue-900: oklch(20% 0.10 250);

    /* Spacing scale */
    --space-1: 0.25rem;
    --space-2: 0.5rem;
    --space-3: 0.75rem;
    --space-4: 1rem;
    --space-6: 1.5rem;
```

```
    --space-8: 2rem;

    /* Type scale */
    --text-sm:   clamp(0.8rem,  1.5vw, 0.875rem);
    --text-base: clamp(1rem,    2vw,   1.125rem);
    --text-lg:   clamp(1.25rem, 2.5vw, 1.5rem);
    --text-xl:   clamp(1.5rem,  3.5vw, 2.25rem);
    --text-2xl:  clamp(2rem,    5vw,   3.5rem);
}

/* ── Layer 2: Semantic tokens (reference primitives, carry meaning) */
:root {
    --color-bg:       var(--blue-50);
    --color-text:     var(--blue-900);
    --color-accent:   var(--blue-500);
    --gap-sm:         var(--space-2);
    --gap-md:         var(--space-4);
    --gap-lg:         var(--space-8);
}

/* ── Layer 3: Dark mode swaps semantic tokens only */
@media (prefers-color-scheme: dark) {
    :root {
        --color-bg:   var(--blue-900);
        --color-text: var(--blue-50);
        /* --color-accent stays the same */
    }
}

/* ── Component uses semantic tokens — works in both modes */
body  { background: var(--color-bg);   color: var(--color-text); }
.link { color: var(--color-accent); }
```

> **Three-layer token architecture:** Primitives (the raw palette) → Semantic tokens (what things mean — background, text, accent) → Component tokens (local knobs that reference semantics). Only swap semantic tokens for theming. Components never reference primitives directly — that decoupling is what makes theme switching trivial.

Chapter Summary

Concept	Key point
`var() fallback`	Second argument is the fallback — used only when the property is not set, not when it holds an invalid value. Invalid values fall back to inherited or initial.
`Inheritance scope`	Custom properties inherit like any CSS property. Redeclaring on a child overrides for that entire subtree — no new selectors needed on descendants.
`Unitless numbers`	Store unitless numbers (--scale: 1.25) and multiply by a unit inside calc(). Allows JS to update the number cleanly without knowing the unit.
`@property`	Registers a custom property with a type, initial-value, and inherits flag. Required to animate a custom property. Must declare syntax and initial-value when inherits: false.
`calc() gotcha`	Spaces are mandatory around + and -. calc(100%+2rem) fails silently; calc(100% + 2rem) works. Multiplication and division don't need spaces.
`min() for containers`	min(100% - 2rem, 1200px) is the cleanest fluid container — no media query, always has 1rem margin on small screens, caps at 1200px on wide.
`clamp() for type`	clamp(MIN, preferred vw expression, MAX) creates fluid typography. The preferred value should use viewport units so it scales; min and max cap the extremes.
`color-mix()`	color-mix(in oklch, colorA 70%, colorB) mixes two colours. oklch gives perceptually uniform mixing — in hsl midpoints often go grey or muddy.
`light-dark()`	light-dark(light-value, dark-value) is a shorthand for two-value colour-scheme switching. Requires color-scheme: light dark on :root.
`env()`	Reads browser environment variables. safe-area-inset-* is essential for PWAs and mobile web apps on notched devices.
`Three-layer tokens`	Primitives → Semantic → Component. Only swap semantics for theming. Components reference semantics, never primitives directly.

EXERCISES

1. **Fluid type scale:** Define five type scale steps on `:root` using `clamp()`. Apply them to h1–h4 and body. Open DevTools device toolbar and drag the viewport width between 320px and 1400px — confirm the text scales smoothly with no jumps.

2. **Component token pattern:** Build a `.badge` component that exposes `--badge-bg`, `--badge-color`, and `--badge-radius`. Create three variants (`.badge--success`, `.badge--warning`, `.badge--error`) that only override the component-level tokens — no duplicating the structural rules.

3. **Dark mode with data-theme:** Set up a light/dark theme using semantic tokens (`--bg`, `--text`, `--surface`, `--accent`). Implement both `@media (prefers-color-scheme: dark)` and a manual `[data-theme="dark"]` override. Add a button that toggles the attribute and persists the choice to `localStorage`.

4. **@property animation:** Register `--progress` as a `<percentage>` with `initial-value: 0%`. Build a progress bar that uses `width: var(--progress)`. Animate it from 0% to 100% over 3 seconds. Confirm it interpolates smoothly (it will fail without `@property` — test both ways).

5. **color-mix() palette generator:** Define a single brand colour as an oklch value. Generate tints and shades using `color-mix(in oklch, var(--brand) N%, white)` and `color-mix(in oklch, var(--brand) N%, black)`. Build a swatch row showing the full scale from 10% brand to 90% brand.

Next: Chapter 10 — Modern CSS Features. Cascade layers (`@layer`), the `@scope` rule, nesting, `:has()` in layout contexts, the new logical properties system, `content-visibility`, and what's shipping in 2025–2026.

CHAPTER 10

Modern CSS Features

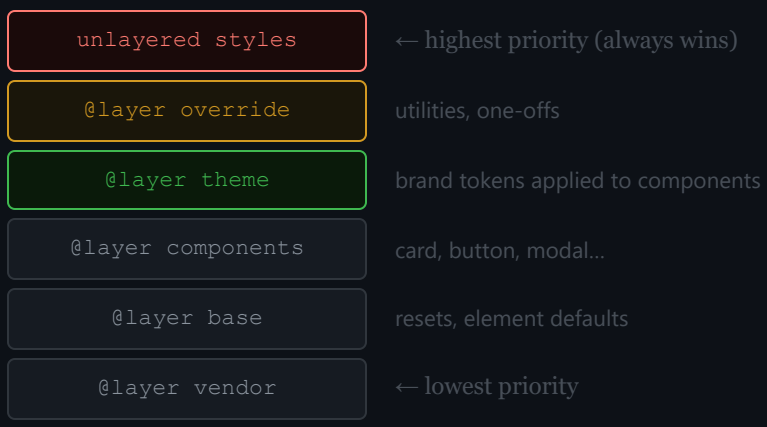
Chapter 10 — Modern CSS Features

The CSS that shipped between 2022 and 2025 is the most significant upgrade to the language since Flexbox. Cascade layers, native nesting, `@scope`, logical properties, and `content-visibility` each solve problems that previously required either preprocessors, naming conventions, or JavaScript. This chapter covers the features that are either baseline-available today or shipping in all evergreen browsers right now.

1. Cascade Layers — @layer

The cascade has always resolved conflicts by specificity and source order. That works fine until a third-party library uses high-specificity selectors that override your styles, or until your utility classes need to beat your component styles without resorting to `!important`. **Cascade layers** add an explicit priority tier *above* specificity — styles in a higher-priority layer always win, regardless of selector weight.

Layer priority order (highest wins)



> **Key insight.** The layer declared *last* in the order statement wins. Unlayered styles (no @layer at all) always beat all layered styles — so existing code without layers automatically takes priority over anything you put in a layer. This makes layered adoption safe: old code isn't broken by new layers.

```css
/* Step 1: Declare the order. Layer declared LAST wins. */
@layer vendor, base, components, theme, override;

/* Step 2: Assign styles to layers */
@layer vendor {
    /* Third-party CSS — sandboxed, can't leak out */
    .btn { background: blue; padding: 8px 16px; }
}

@layer base {
    /* Reset + element defaults */
    *, *::before, *::after { box-sizing: border-box; }
    body { margin: 0; font-family: system-ui, sans-serif; }
}

@layer components {
    /* High-specificity selectors here DON'T beat 'override' layer */
    .card .card__header .card__title {
        font-size: 1.25rem;
        color: inherit;
    }
}

@layer override {
    /* Low-specificity utilities — but in the highest layer, so they win */
    .text-sm  { font-size: 0.875rem; }
    .text-red { color: #ff7b72; }
    .mt-4     { margin-top: 1rem; }
}

/* Layers can be spread across multiple files and still honour the order */
@import url('reset.css') layer(base);
@import url('bootstrap.min.css') layer(vendor);

/* Nested layers — creates a sub-hierarchy */
@layer components {
    @layer base, state;
    @layer base  { .btn { background: var(--color-primary); } }
    @layer state { .btn:hover { background: var(--color-primary-hover); } }
}
/* Reference as components.base and components.state */
```

!important flips the layer order. Inside `!important` declarations, the cascade layer priority is *reversed* — the lowest-priority layer's `!important` beats all higher layers. This mirrors how browser user stylesheets work. In practice: avoid `!important` inside layers entirely — that's what layers are for.

2. CSS Nesting

Native CSS nesting landed in all evergreen browsers in 2023. It eliminates the main reason developers reached for Sass or Less — the ability to write parent-child selectors without repeating the parent selector name.

BEFORE — FLAT CSS (OR SASS)

```
.card {
  background: #161b22;
  padding: 1rem;
}
.card:hover {
  box-shadow: 0 4px 12px ...;
}
.card .card__title {
  font-size: 1.25rem;
}
.card .card__title a {
  color: inherit;
}
.card--featured {
  border: 1px solid #58a6ff;
}
/* Repetition. Scroll to find related rules.
```

AFTER — NATIVE NESTING

```
.card {
  background: #161b22;
  padding: 1rem;

  &:hover {
    box-shadow: 0 4px 12px ...;
  }

  .card__title {
    font-size: 1.25rem;

    a { color: inherit; }
  }

  &.card--featured {
    border: 1px solid #58a6ff;
  }
}
```

```
/* The & selector — explicit parent reference */
.btn {
    background: var(--color-primary);

    /* & = .btn — for pseudo-classes, pseudo-elements, modifiers */
    &:hover   { background: var(--color-primary-dark); }
    &:focus-visible { outline: 2px solid currentColor; }
    &::before { content: ''; }
```

```
    &--large  { padding: 0.75rem 1.5rem; }    /* .btn--large */
    &[disabled] { opacity: 0.4; }

    /* Without & — the nested selector is a descendant */
    .icon { width: 1em; }              /* .btn .icon */

    /* & can appear anywhere — even at the end */
    .dark-mode & { background: #0d1117; }  /* .dark-mode .btn */

    /* @rules nest too */
    @media (max-width: 768px) {
        width: 100%;
    }

    @layer state {
        &.is-loading { cursor: wait; }
    }
}

/* :is() for complex nesting without specificity accumulation */
.prose {
    :is(h1, h2, h3, h4) {
        margin-top: 1.5em;
        line-height: 1.3;
    }
}
```

> **Nesting and element selectors.** A bare element selector inside a nest (e.g. `p { color: red; }`) is treated as a descendant, not a direct child. Element selectors in nests were initially problematic in early implementations — all current browsers handle them correctly, but if you need to be safe, wrap in `:is(p) { }` or use `& p { }`.

3. @scope — Scoped Styles

`@scope` solves the same problem as BEM naming, CSS Modules, and Shadow DOM style encapsulation — preventing styles from leaking out of a component. Unlike those solutions, `@scope` is pure CSS with no tooling or build step, and it uses *proximity* rather than specificity to resolve conflicts.

```
/* @scope(root) to(limit) — styles apply only inside root, not inside limit */
@scope (.card) to (.card__body) {
    /* applies inside .card, but NOT inside .card__body */
    .title { font-size: 1.25rem; }
}

/* Proximity wins over specificity for competing @scope rules */
@scope (.light-theme) {
    p { color: #1a1a1a; }
}
@scope (.dark-theme) {
    p { color: #c9d1d9; }
}
/* <div class="dark-theme"><p> → dark wins because it's closer */
/* <div class="dark-theme"><div class="light-theme"><p> → light wins */

/* Inside <style scoped> — an inline scope targeting its own element */
/* <style> in a component template */
@scope {
    /* :scope = the element that owns this style block */
    :scope { border: 1px solid #30363d; }
    :scope .title { font-weight: bold; }
}

/* Donut scope — apply between root and limit */
@scope (.media-widget) to (.ad-slot, .third-party) {
    /* styles apply in .media-widget but stop at .ad-slot */
    /* third-party content inside .ad-slot is safely excluded */
    img { max-width: 100%; }
}
```

4. :has() in Layout Contexts

`:has()` was introduced in Chapter 4 as a selector feature. Here we look at its role in larger layout patterns — replacing JavaScript that previously watched the DOM for conditional layout changes.

```css
/* Layout switch based on content type */
.card:has(img) {
    display: grid;
    grid-template-columns: 180px 1fr;
}
.card:not(:has(img)) {
    display: block;
}

/* Body layout based on open sidebar */
body:has(.sidebar[aria-expanded="true"]) {
    grid-template-columns: 260px 1fr;
}
body:has(.sidebar[aria-expanded="false"]) {
    grid-template-columns: 60px 1fr;
}

/* Dialog/modal open — page scroll lock without JS */
html:has(dialog[open]) {
    overflow: hidden;
}

/* Form validation state — highlight the label when input is invalid */
.form-group:has(input:invalid:not(:placeholder-shown)) {
    label  { color: #ff7b72; }
    input  { border-color: #ff7b72; }
    .hint { display: block; }
}

/* Previous sibling — what :has() makes possible */
.item:has(+ .item--highlight) {
    opacity: 0.5;  /* dim the item BEFORE a highlighted item */
}

/* Count-based layout — no JavaScript */
.grid:has(.item:nth-child(4))  { grid-template-columns: repeat(2, 1fr); }
.grid:has(.item:nth-child(7))  { grid-template-columns: repeat(3, 1fr); }
.grid:has(.item:nth-child(10)) { grid-template-columns: repeat(4, 1fr); }
```

5. Logical Properties

Physical properties (`margin-left`, `padding-top`) are tied to screen directions. Logical properties (`margin-inline-start`, `padding-block-start`) map to the *writing mode* — so your layout automatically mirrors for right-to-left languages (Arabic, Hebrew) and vertical writing systems (Japanese, Mongolian) without any direction-specific overrides.

Physical → Logical mapping

~~margin-top~~ **margin-block-start** Block axis start	~~margin-bottom~~ **margin-block-end** Block axis end	~~margin-left~~ **margin-inline-start** Inline axis start
~~margin-right~~ **margin-inline-end** Inline axis end	~~padding-top / bottom~~ **padding-block** Both block edges	~~padding-left / right~~ **padding-inline** Both inline edges
~~width~~ **inline-size** Size along inline axis	~~height~~ **block-size** Size along block axis	~~top / bottom / left / right~~ **inset-block / inset-inline** Inset shorthands
~~border-top~~ **border-block-start** Block start border	~~border-left~~ **border-inline-start** Inline start border	~~text-align: left~~ **text-align: start** Start of inline direction

```
/* Block = vertical direction (in horizontal writing) */
/* Inline = horizontal direction (in horizontal writing) */

/* Component that works in LTR, RTL, and vertical writing — same code */
.card {
    padding-block:  1.5rem;     /* top and bottom in LTR */
    padding-inline: 1.25rem;    /* left and right in LTR */
    border-inline-start: 3px solid var(--color-accent);
    /* left border in LTR, right border in RTL — automatically */
}

.icon-badge {
    margin-inline-end: 0.5rem;  /* right margin in LTR, left margin in RTL */
}

/* max-inline-size replaces max-width for flow-aware sizing */
.prose {
```

```
    max-inline-size: 65ch;
    margin-inline:  auto;       /* centres horizontally in LTR */
}

/* Vertical centring with logical properties */
.overlay {
    position: absolute;
    inset:    0;              /* shorthand for top/right/bottom/left: 0 */
    inset-block-start: 1rem;  /* override just the top */
}

/* Writing mode — vertical Japanese layout */
.japanese-novel {
    writing-mode: vertical-rl;
    /* block axis is now horizontal, inline is vertical */
    /* margin-block-start = margin-right, padding-inline = padding top/bottom */
}
```

> **When to use logical properties.** Use them by default in new code — the overhead is zero and the i18n benefit is free. The only exception is when you genuinely mean a physical direction (e.g. a fixed header at the top of the screen should use `top: 0`, not `inset-block-start: 0`, since it's physically anchored regardless of writing mode).

6. content-visibility

`content-visibility: auto` lets the browser skip rendering off-screen content entirely — style, layout, and paint are skipped until the element enters the viewport. On long pages (dashboards, feeds, long-form articles), this can cut initial render time by 50% or more with a single CSS line.

```
/* content-visibility: auto — skip rendering off-screen sections */
section {
    content-visibility: auto;
    contain-intrinsic-size: auto 800px;
    /* Browser reserves ~800px height for off-screen sections */
    /* Prevents scroll jumps as content renders in */
    /* 'auto' prefix means it uses the measured height after first render */
```

```
}

/* content-visibility: hidden — like display: none but preserves state */
.offscreen-panel {
    content-visibility: hidden;
    /* Not rendered, not in accessibility tree, but state preserved */
    /* Better than display: none for pre-rendered panels (dialogs, drawers) */
    /* Switch to content-visibility: visible to show */
}

/* contain — the underlying isolation primitive */
.widget {
    contain: layout style paint;
    /* layout: changes inside don't affect outside layout */
    /* style: counters and quotes don't escape this element */
    /* paint: children don't paint outside the border box */
    /* size: element size doesn't depend on children — needs explicit size */
    /* content = layout + style + paint (most useful shorthand) */
    /* strict = layout + style + paint + size */
}

/* Practical: long comment thread or infinite scroll feed */
.comment {
    content-visibility:    auto;
    contain-intrinsic-size: auto 120px;
}
/* A 1,000-comment page renders only the visible viewport.
   Off-screen comments are skipped until scroll brings them in. */
```

Gotcha — find-in-page and accessibility. With `content-visibility: auto`, off-screen elements are not in the accessibility tree and Ctrl+F may not find text inside them until they're rendered. This is acceptable for media feeds but inappropriate for legal text, articles, or anything users would reasonably search. Use `hidden` only for content that is intentionally not presented (pre-rendered panels).

7. Baseline 2024–2025 — What Just Shipped

Feature	Status	What it does
`CSS nesting`	**Baseline 2023**	Native & parent selector, nested @rules. All evergreen browsers.
`@layer`	**Baseline 2022**	Explicit cascade priority tiers. Full cross-browser support.
`:has()`	**Baseline 2023**	Parent selector. All evergreen browsers since late 2023.
`container queries`	**Baseline 2023**	@container, cqw/cqh units. Full support.
`color-mix()`	**Baseline 2023**	Mix two colours in any colour space. Full support.
`oklch() / oklab()`	**Baseline 2023**	Perceptual colour spaces. Full support.
`scroll-driven animations`	**Baseline 2024**	animation-timeline: scroll() / view(). Chrome + Firefox + Safari 18.
`@property`	**Baseline 2024**	Typed custom properties. All evergreen as of mid-2024.
`light-dark()`	**Baseline 2024**	Two-value colour-scheme helper. Full support 2024.
`transition-behavior: allow-discrete`	**Baseline 2024**	Animate display, visibility, overlay. Full support 2024.
`@starting-style`	**Baseline 2024**	Define enter-animation start state. Full support 2024.
`popover API + ::backdrop`	**Baseline 2024**	HTML popover attribute with CSS ::backdrop animations.
`@scope`	**Baseline 2024**	Scoped styles with proximity. Chrome, Firefox, Safari — all 2024.
`anchor positioning`	**Emerging 2024**	Position an element relative to another (tooltips, dropdowns) — anchor-name / anchor() / position-area. Chrome 125+; Safari and Firefox flags only.
`view transitions`	**Emerging 2024**	document.startViewTransition() + ::view-transition CSS. Cross-document transitions in Chrome 126+.
`if() in CSS`	**Proposed 2025**	Inline conditional: color: if(style(--variant: dark): white; else: black). No browser support yet.

Feature	Status	What it does
CSS mixins / @apply revival	Proposed 2025	Reusable rule blocks without duplication. Early spec work only.

Anchor positioning — the future of tooltips

```css
/* Anchor positioning — position an element relative to another DOM element */
/* No JS, no getBoundingClientRect, no Popper.js */

.trigger {
    anchor-name: --my-anchor;
}

.tooltip {
    position:        absolute;
    position-anchor: --my-anchor;
    position-area:   top center;     /* above the anchor, centred */
    margin-block-end: 8px;

    /* position-try-fallbacks — flip if no room */
    position-try-fallbacks: flip-block, flip-inline, flip-start;
}
```

View transitions — page-change animations

```
/* Same-page view transitions — animate between two states */
button.addEventListener('click', () => {
    document.startViewTransition(() => {
        updateTheDOMSomehow();
    });
});

/* CSS controls the animation */
::view-transition-old(root) {
    animation: fade-out 0.3s ease;
}
::view-transition-new(root) {
    animation: fade-in 0.3s ease;
}
```

```
/* Named transitions for individual elements */
.hero-image {
    view-transition-name: hero;  /* browser morphs this between pages */
}
::view-transition-group(hero) {
    animation-duration: 0.5s;
    animation-timing-function: ease-in-out;
}
```

Chapter Summary

Feature	Key point
`@layer order`	Layers declared last win. Declare the order once at the top: @layer vendor, base, components, theme, override. Unlayered styles always beat all layers.
`@layer + !important`	!important reverses the layer order — the lowest-priority layer's !important beats higher layers. Avoid !important inside layers entirely.
`Nesting &`	& is explicit parent reference. Without &, nested selectors are descendants. & can appear anywhere in the selector — .dark & targets .dark .btn. @media and @layer can be nested inside rules.
`@scope proximity`	When two @scope rules compete, the closer ancestor wins — regardless of specificity. Donut scope (root) to (limit) excludes content inside limit subtrees.
`:has() layout`	html:has(dialog[open]) { overflow: hidden } replaces a common JS pattern. body:has(.sidebar[open]) can switch the entire grid template. No JS needed.
`Logical properties`	Use block/inline instead of top/bottom and left/right. margin-inline: auto centres. border-inline-start is the "left" border that flips for RTL automatically.
`inset shorthand`	inset: 0 is shorthand for top: 0; right: 0; bottom: 0; left: 0. inset-block and inset-inline are the logical variants.
`content-visibility`	content-visibility: auto skips rendering off-screen elements. Always pair with contain-intrinsic-size: auto Npx to reserve space and prevent scroll jumps.
`Baseline 2024`	Fully baseline in all evergreens: @property, light-dark(), allow-discrete, @starting-style, @scope, scroll-driven animations, popover API.
`anchor positioning`	anchor-name on the trigger + position-anchor on the tooltip. position-area replaces manual offset calculations. position-try-fallbacks for auto-flip. Chrome

Feature	Key point
	125+ only as of 2025.

EXERCISES

1. **Layer your stylesheet:** Take an existing stylesheet (or create a small one) and split it into @layer base, components, and utilities. Declare the order at the top. Verify that a low-specificity utility class in the utilities layer can override a high-specificity rule in the components layer without !important.
2. **Nest a component:** Convert a BEM-style flat stylesheet for a `.nav` component (with `.nav__item`, `.nav__link`, `.nav__link--active`, `.nav__link:hover`) into native CSS nesting. Add a nested `@media` rule for mobile. Verify the compiled output in DevTools matches the flat version.
3. **:has() form validation:** Build a form with three inputs. Using only CSS and `:has()`, make the submit button go green when all three inputs are `:valid`, and display inline error messages (hidden by default) when any input is `:invalid:not(:placeholder-shown)`. No JavaScript.
4. **Logical properties RTL test:** Build a card with `padding-inline`, `margin-inline-end` on an icon, and a `border-inline-start` accent. Add `dir="rtl"` to the card's parent. Confirm the layout mirrors correctly — the border moves to the right, the icon margin flips.
5. **content-visibility performance:** Build a page with 100 identical article cards. Open DevTools Performance, record a page load without `content-visibility`. Add `content-visibility: auto; contain-intrinsic-size: auto 200px` to each card. Record again and compare the Rendering and Painting timing.

Next: Chapter 11 — CSS Architecture. Naming conventions (BEM, CUBE CSS, Utility-first), design token organisation, CSS custom property strategies at scale, layer strategy for large teams, and how to evaluate whether a methodology fits your project.

CHAPTER 11

CSS Architecture

Chapter 11 — CSS Architecture

Writing CSS that works is easy. Writing CSS that a team can maintain six months later is hard. Architecture is the set of decisions that keep a growing stylesheet predictable: how classes are named, how files are split, how specificity is kept low, and how the cascade is made explicit rather than accidental. This chapter covers the major methodologies, when to use each, and how modern CSS features (`@layer`, custom properties, nesting) change the architecture calculus.

1. The Core Problem CSS Architecture Solves

CSS has three properties that cause trouble at scale:

- **Global by default.** Every selector can affect any element anywhere in the document. There is no built-in encapsulation boundary.
- **Cascade order matters.** A rule declared later wins a specificity tie. Split files across multiple stylesheets or developers and source order becomes unpredictable.
- **Specificity wars.** Once a component uses `.card .title`, anything trying to override it must match or beat that specificity — leading to ever-escalating selectors and eventual `!important`.

Every CSS methodology is essentially an answer to one question: *how do we prevent styles from bleeding into things they weren't meant to affect?* The solutions differ in where they put the constraint — in naming conventions, in tooling, in selector rules, or in the cascade itself.

```
/* The classic specificity war — how it escalates */

/* Developer A writes: */
.sidebar .card .title { color: #58a6ff; }   /* specificity: 0-3-0 */

/* Developer B tries to override: */
```

```
.title { color: #c9d1d9; }                    /* 0-1-0 — loses */
.card .title { color: #c9d1d9; }              /* 0-2-0 — loses */
.sidebar .card .title.override { color: #c9d1d9 !important; }  /* gives up */

/* Root cause: specificity was never kept low in the first place */
/* Architecture fix: write .title { } everywhere and use the cascade to order */
```

2. The Major Methodologies

BEM

Block__Element--Modifier — naming convention

STRENGTHS

+ Forces flat, low-specificity selectors
+ Class name communicates component structure
+ No tooling needed — pure naming discipline
+ Universally understood across teams

WEAKNESSES

- Long, verbose class names in HTML
- No cascade control — order still matters
- Modifier explosion for state variants
- Doesn't solve file organisation

BEST FOR

Multi-developer teams who share templates

Projects without a build step

Utility-First (Tailwind)

Single-purpose classes composed in HTML

STRENGTHS

+ Virtually no specificity conflicts
+ Design constraints enforced by available classes
+ Co-location — styles live with the element
+ PurgeCSS removes unused classes automatically

WEAKNESSES

- HTML becomes verbose and hard to scan
- Requires build tooling and config
- Repeating identical class lists across similar elements
- Difficult to express truly custom one-off styles

BEST FOR

React/Vue/Svelte component-based projects

Teams that want design constraints enforced by the tool

CUBE CSS

Composition, Utility, Block, Exception

STRENGTHS

+ Works *with* the cascade rather than fighting it
+ Custom properties as the primary API
+ Minimal class names — leans on HTML semantics

CSS Modules / Scoped CSS

Tooling-enforced local scope

STRENGTHS

+ True encapsulation — no naming discipline required
+ Collisions are physically impossible
+ Works with any class naming style

+ Pairs naturally with @layer and modern CSS

WEAKNESSES

- Less prescriptive — requires more design decisions
- Newer; fewer examples and ecosystem resources
- Requires team to understand the cascade deeply

BEST FOR

Projects using a design token system

Content-heavy sites with variable markup

+ Dead code elimination via tree-shaking

WEAKNESSES

- Requires bundler integration (webpack, Vite)
- Global styles (resets, themes) need :global() escape hatch
- Hard to share styles across module boundaries
- Generated class names make DevTools harder to use

BEST FOR

Large React/Next.js or Vue/Nuxt projects

Teams where naming discipline is hard to enforce

3. BEM in Depth

```
/* BEM: Block__Element--Modifier */

/* Block — standalone component */
.card {
    background: var(--surface);
    border-radius: 8px;
    padding: 1.5rem;
}

/* Elements — parts of the block, separated by __ */
.card__header  { margin-bottom: 1rem; }
.card__title   { font-size: 1.25rem; font-weight: 700; }
.card__body    { color: var(--text-muted); }
.card__footer  { margin-top: 1rem; display: flex; gap: 0.5rem; }

/* Modifiers — variants, separated by -- */
.card--featured { border: 2px solid var(--color-accent); }
.card--compact  { padding: 0.75rem; }
.card--loading  { opacity: 0.6; pointer-events: none; }

/* HTML — block and modifier on the same element */
/* <article class="card card--featured"> */
/*   <header class="card__header"> */
/*     <h2 class="card__title">...</h2> */
/*   </header> */
```

```
/*   <div class="card__body">...</div> */
/* </article> */

/* BEM rules: */
/* 1. Elements are always children of their Block — never nest .card__title insi
/* 2. Never use element selectors like .card h2 — always .card__title */
/* 3. Modifiers never stand alone — always paired with the block: class="card ca
/* 4. Specificity stays flat: all selectors are one class = 0-1-0 */

/* BEM + nesting (modern hybrid) */
.card {
    background: var(--surface);

    &__title { font-size: 1.25rem; }   /* compiles to .card__title */
    &__body  { color: var(--text-muted); }
    &--featured { border: 2px solid var(--color-accent); }
}
/* Co-location of BEM rules without sacrificing flat specificity */
```

4. CUBE CSS

CUBE CSS (Composition, Utility, Block, Exception) was coined by Andy Bell. Its central premise is that the cascade is not the enemy — fighting it with ever-tighter selectors is. Instead, use the cascade deliberately: global composition styles flow from broad selectors, utilities make targeted overrides, blocks handle components, and exceptions handle the rare one-off.

```
/* C — Composition: layout primitives, flow, cluster, stack */
/* These use element and low-specificity selectors intentionally */

.flow > * + * {
    margin-block-start: var(--flow-space, 1em);
}

.cluster {
    display: flex;
    flex-wrap: wrap;
    gap: var(--cluster-gap, 1rem);
```

```
    align-items: var(--cluster-align, center);
}

.stack {
    display: flex;
    flex-direction: column;
    gap: var(--stack-gap, 1.5rem);
}

.sidebar-layout {
    display: flex;
    flex-wrap: wrap;
    gap: var(--sidebar-gap, 1rem);

    > :first-child { flex-basis: var(--sidebar-w, 20rem); }
    > :last-child  { flex-basis: 0; flex-grow: 999; min-inline-size: 50%; }
}

/* U — Utilities: single-purpose, very low specificity */
.text-center  { text-align: center; }
.text-muted   { color: var(--color-text-muted); }
.visually-hidden {
    position: absolute; width: 1px; height: 1px;
    clip: rect(0,0,0,0); overflow: hidden;
}

/* B — Block: component-level styles, higher specificity acceptable */
.card {
    background:    var(--card-bg, var(--color-surface));
    border-radius: var(--card-radius, 8px);
    padding:       var(--card-padding, 1.5rem);
}

/* E — Exception: data attributes for contextual overrides */
.card[data-variant="featured"] {
    --card-bg: var(--color-featured-surface);
}

/* HTML usage: composition + utility + block + exception */
/* <ul class="cluster" role="list"> */
/*   <li class="card" data-variant="featured"> */
/*     <div class="stack"> */
/*       <h2 class="text-muted">...</h2> */
```

5. Design Token Organisation at Scale

LAYER 1 — PRIMITIVE TOKENS

`--blue-100: oklch(95% 0.04 250) --blue-500: oklch(60% 0.22 250) --blue-900: oklch(20% 0.10 250) --space-1: 4px --space-2: 8px --space-4: 16px --space-8: 32px --radius-sm: 4px --radius-md: 8px --radius-full: 9999px --font-sans: system-ui, sans-serif --font-mono: 'Courier New', monospace`

Rule: Raw values only. No semantic meaning. Never used directly in components.

LAYER 2 — SEMANTIC TOKENS

`--color-bg: var(--blue-950) --color-text: var(--blue-50) --color-surface: var(--blue-900) --color-accent: var(--blue-500) --color-text-muted: var(--blue-300) --color-border: var(--blue-800) --gap-sm: var(--space-2) --gap-md: var(--space-4) --gap-lg: var(--space-8)`

Rule: References primitives. Carries intent. Only layer swapped for theming (dark/light/brand).

LAYER 3 — COMPONENT TOKENS

`--btn-bg: var(--color-accent) --btn-color: #fff --btn-radius: var(--radius-md) --btn-padding: var(--space-2) var(--space-4) --card-bg: var(--color-surface) --card-radius: var(--radius-md) --input-border: var(--color-border) --input-focus-ring: var(--color-accent)`

Rule: References semantics. Exposes component API. Overridden per variant, never per theme.

```
/* Token file organisation */
/* tokens/primitives.css — raw palette, never import directly in components */
:root {
    --blue-50:  oklch(97% 0.02 250);
    --blue-500: oklch(60% 0.22 250);
    --blue-950: oklch(12% 0.06 250);
}

/* tokens/semantic.css — the theming surface */
:root {
    --color-bg:      var(--blue-50);
    --color-text:    var(--blue-950);
    --color-accent:  var(--blue-500);
}
[data-theme="dark"] {
    --color-bg:      var(--blue-950);
    --color-text:    var(--blue-50);
```

```
    /* --color-accent stays the same */
}

/* Multiband theming — brand + dark + high contrast */
[data-brand="enterprise"] {
    --color-accent:  oklch(55% 0.18 150);  /* green brand */
}
@media (forced-colors: active) {
    :root {
        --color-bg:     Canvas;
        --color-text:   CanvasText;
        --color-accent: Highlight;
    }
}
```

6. @layer Strategy for Teams

HIGH	`@layer override`	Utility classes, debug helpers, forced states. Single-property rules only.	`utilities.css`
MID	`@layer theme`	Semantic token overrides for dark mode, brand switching, high-contrast.	`tokens/semantic.css`
MID	`@layer components`	Card, button, nav, modal — BEM or component-scoped rules. Uses token variables.	`components/*.css`
LOW	`@layer layout`	Page templates, grid systems, flow primitives, sidebar layouts.	`layout/*.css`
LOW	`@layer base`	Element defaults, typography scale, reset rules, :root token declarations.	`base.css · tokens/primitives.css`
LOW	`@layer vendor`	Third-party libraries imported with @import … layer(vendor). Sandboxed.	`@import … layer(vendor)`

```
/* main.css — single entry point declares layer order */
@layer vendor, base, layout, components, theme, override;

@import url('vendor/normalize.css') layer(vendor);
@import url('tokens/primitives.css') layer(base);
@import url('tokens/semantic.css')   layer(base);
```

```
@import url('base.css')                layer(base);

@import url('layout/grid.css')         layer(layout);
@import url('layout/flow.css')         layer(layout);

@import url('components/button.css') layer(components);
@import url('components/card.css')    layer(components);
@import url('components/nav.css')     layer(components);

@import url('themes/dark.css')         layer(theme);
@import url('utilities.css')           layer(override);

/* Result: any utility class beats any component rule,
   no matter how specific the component selector was. */
```

7. File Structure Patterns

```
styles/ ├── tokens/ │ ├── primitives.css — raw palette, spacing, radii, fonts │
└── semantic.css — intent-named tokens + theme overrides │ ├── base/ │ ├──
reset.css — modern CSS reset (box-sizing, margin 0) │ ├── typography.css —
element defaults: h1-h6, p, a, code │ └── root.css — :root token declarations,
@font-face │ ├── layout/ │ ├── grid.css — page grid, container, columns │ ├──
flow.css — .stack, .cluster, .sidebar-layout │ └── prose.css — long-form content
max-width, flow-space │ ├── components/ │ ├── button.css │ ├── card.css │ ├──
nav.css │ ├── modal.css │ └── form.css │ ├── themes/ │ ├── dark.css — [data-
theme="dark"] overrides │ └── brand-enterprise.css │ ├── utilities.css — single-
property utility classes └── main.css — @layer order declaration + all @imports
```

8. Choosing a Methodology

Situation	Recommended approach	Why
Small team, no build step, shared templates (PHP/Jinja)	`BEM + @layer`	*Flat specificity from naming, @layer handles cascade ordering*
React/Vue SPA with component files	`CSS Modules or scoped CSS`	*Tooling enforces encapsulation; no naming discipline needed*

Situation	Recommended approach	Why
Design system with token infrastructure	`CUBE CSS + @layer + tokens`	*Works with cascade, tokens are the API, @layer handles priority*
Rapid prototyping, design constraint enforcement	`Tailwind utility-first`	*Speed + built-in design constraints + excellent DX*
Third-party CSS integration (Bootstrap, etc.)	`@import layer(vendor) + override layer`	*Sandboxes vendor at lowest priority; override layer beats it cleanly*
Content-heavy site, varying markup (CMS-driven)	`CUBE + flow/cluster primitives`	*Composition primitives work on arbitrary markup; no class requirements on CMS content*
Multi-brand / multi-theme product	`Three-tier token system + @layer theme`	*Swap only semantic tokens per brand; components need no changes*

> **The golden rule.** Keep specificity flat. Every methodology that succeeds long-term does so because it makes high-specificity selectors the exception rather than the default. BEM does it with naming, utilities do it with single-class rules, CUBE does it by leaning on element selectors intentionally, and @layer does it by making layer position more important than selector weight. Pick the tool that fits your project — but don't skip the specificity discipline.

Chapter Summary

Concept	Key point
`The core problem`	CSS is global by default. Architecture is the set of conventions that prevent styles bleeding into unintended elements and prevent specificity wars.
`BEM`	Block__Element--Modifier. All selectors are a single class (0-1-0). Modifiers pair with the block class: class="card card--featured". Works without tooling.
`Utility-first`	Single-purpose classes composed in HTML. No cascade conflicts. Requires build tooling. Best for component-based JS frameworks.
`CUBE CSS`	Composition (layout primitives) → Utility (overrides) → Block (components) → Exception (data attributes). Works with the cascade; uses custom properties as the component API.
`CSS Modules`	Tooling scopes class names per file. True encapsulation without discipline. Global styles need :global() escape hatch. DevTools shows generated names.

Concept	Key point
Three-tier tokens	Primitives (raw palette) → Semantic (intent: --color-bg, --color-text) → Component (--btn-bg, --card-radius). Only swap semantic tokens for theming.
@layer for teams	Declare order once in main.css. Import each file into its layer. Utility layer always beats component layer regardless of specificity — no more !important.
File structure	tokens/ → base/ → layout/ → components/ → themes/ → utilities.css → main.css. main.css owns the @layer order and all @imports.
Vendor sandboxing	@import url('bootstrap.css') layer(vendor) puts third-party CSS in the lowest-priority layer. Your own unlayered styles always win.
Choosing a method	No single methodology wins universally. BEM for server-rendered teams, modules for JS SPAs, CUBE for design systems, Tailwind for rapid UI work. @layer + tokens applies to all of them.

EXERCISES

1. **BEM audit:** Take an existing component stylesheet (your own or a public one). List every selector and its specificity score. Identify any rule above 0-2-0. Refactor those rules to BEM so all selectors are a single class. Verify in DevTools that specificity is flat throughout.
2. **Build a CUBE flow primitive:** Implement `.flow`, `.cluster`, and `.stack` layout primitives. Apply them to a blog post layout: a `.stack` of sections, each section being a `.cluster` of cards, with `.flow` on prose content inside cards. Override spacing per-section with `--flow-space` and `--stack-gap` custom properties on individual elements.
3. **Three-tier token system:** Create `primitives.css`, `semantic.css`, and use both in a button component and a card component. Add a `[data-theme="dark"]` rule to `semantic.css` that swaps `--color-bg` and `--color-text`. Toggle the attribute in the browser and confirm both components update with zero component CSS changes.
4. **@layer import chain:** Create a `main.css` that declares `@layer vendor, base, components, override`. Import a small reset into `vendor`, write base typography into `base`, write a button component into `components`, and a `.mt-0` utility into `override`. Apply `.mt-0` to a button inside a deeply nested component and verify it wins without `!important`.
5. **Methodology comparison:** Implement the same three-component UI (nav, card, button) three times: once in BEM, once in utility-first (write the classes by hand, no Tailwind

needed), and once in CUBE CSS. Count the class names in the HTML, the lines in the CSS, and note which felt most natural. Write a short note on which you'd choose for a team project and why.

Next: Chapter 12 — Capstone Project. Build a complete dark-themed developer portfolio page from scratch — applying every concept from all twelve chapters: cascade layers, custom property token system, responsive fluid layout, accessible components, scroll-driven animations, and a production-ready file structure.

CHAPTER 12

Capstone Project: Developer Portfolio

Chapter 12 — Capstone Project: Developer Portfolio

This chapter brings all twelve chapters together in a single build: a dark-themed developer portfolio page with a three-tier token system, cascade layers, fluid responsive layout, accessible interactive components, and scroll-driven animation. Every technique is annotated with the chapter it comes from. By the end you will have a production-quality stylesheet and a mental model for how all the CSS concepts connect to each other in real work.

> **What you're building.** A personal portfolio with: sticky transparent nav with scroll-driven opacity, hero section with fluid type and radial glow, animated stat counters, filterable project grid using :has(), skills tag cloud, contact form with CSS-only validation feedback, dark/light theme toggle using data-theme, and a read-progress bar. No JavaScript for any of the CSS features — only the theme toggle and stat counter need JS.

Build phases

Phase 1

Token System + Layer Order

Ch 9 (tokens) + Ch 11 (@layer) + Ch 10 (@property)

Phase 2

Reset + Base Typography

Ch 1 (cascade) + Ch 3 (type) + Ch 10 (logical)

Phase 3

Layout Primitives

Ch 5 (flex) + Ch 6 (grid) + Ch 7 (responsive) + Ch 11 (CUBE)

Phase 4

Components

Ch 4 (selectors) + Ch 5 (flex) + Ch 11 (BEM) + Ch 10 (nesting)

Phase 5

Motion + Interaction

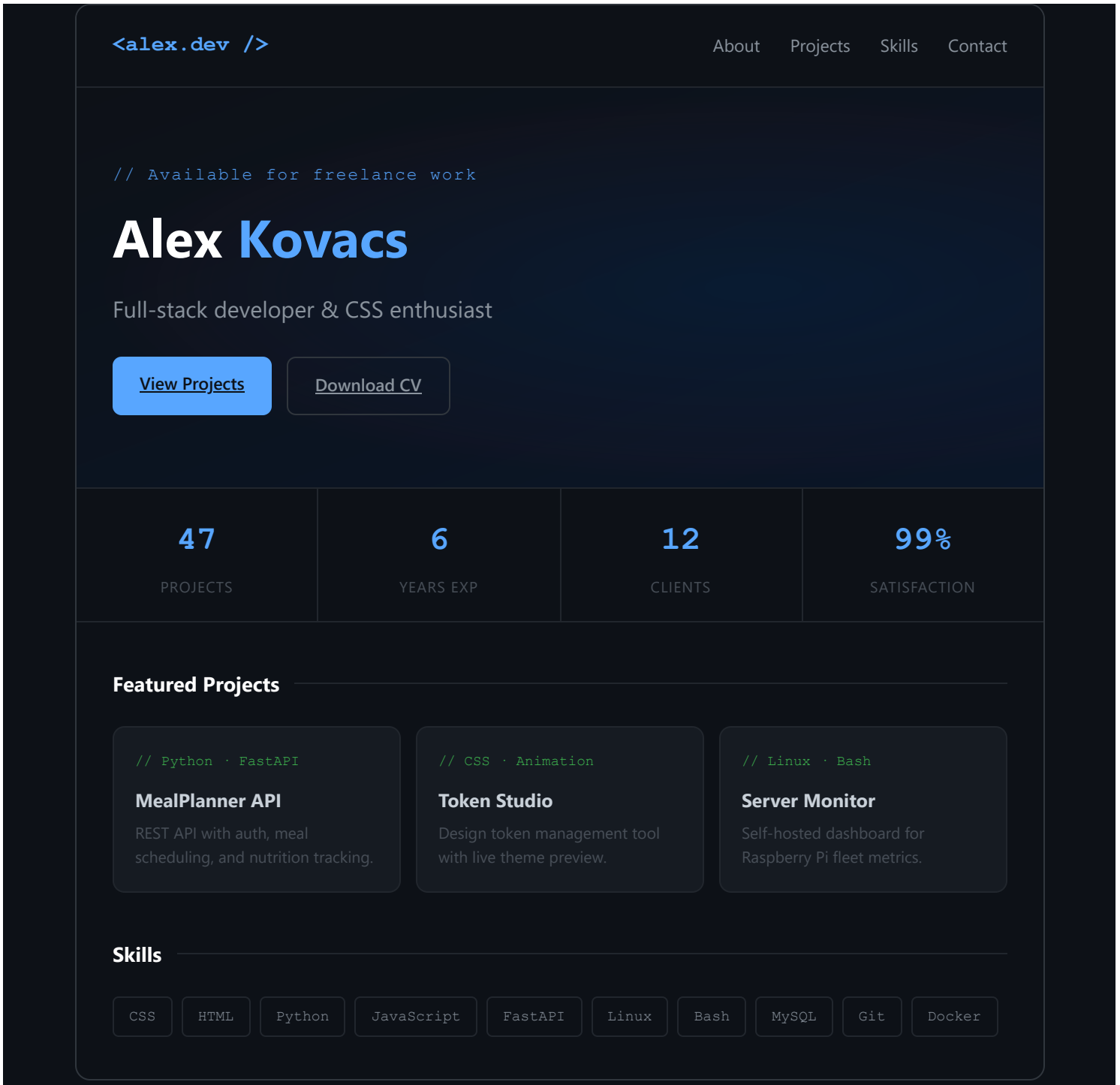
Ch 8 (transitions/animations) + Ch 10 (scroll-driven)

Phase 6

Theming + Accessibility

Ch 3 (color) + Ch 7 (reduced-motion) + Ch 9 (light-dark)

Live Preview



Phase 1 — Token System & Layer Order

```
/* ── main.css — entry point ──────────────────────────────────────── */
/* Ch 11: Declare layer order before any imports */
@layer reset, tokens, base, layout, components, theme, utils;

@import 'tokens/primitives.css'  layer(tokens);
@import 'tokens/semantic.css'    layer(tokens);
```

```
@import 'base/reset.css'         layer(reset);
@import 'base/typography.css'    layer(base);
@import 'layout/primitives.css'  layer(layout);
@import 'components/nav.css'     layer(components);
@import 'components/hero.css'    layer(components);
@import 'components/card.css'    layer(components);
@import 'components/form.css'    layer(components);
@import 'themes/dark.css'        layer(theme);
@import 'utils.css'             layer(utils);
```

```
/* ── tokens/primitives.css ──────────────────────────────── */
/* Ch 9: Raw palette — never used directly in components */
:root {
    /* Colour ramp in oklch for perceptual uniformity (Ch 3) */
    --blue-50:   oklch(96% 0.025 250);
    --blue-300:  oklch(75% 0.12  250);
    --blue-500:  oklch(60% 0.22  250);
    --blue-700:  oklch(45% 0.18  250);
    --blue-900:  oklch(20% 0.08  250);
    --blue-950:  oklch(12% 0.05  250);

    --green-400: oklch(72% 0.18  150);
    --amber-400: oklch(78% 0.16   80);
    --red-400:   oklch(68% 0.22   25);

    /* Spacing — 4px base */
    --space-1: 0.25rem;   --space-2: 0.5rem;
    --space-3: 0.75rem;  --space-4: 1rem;
    --space-6: 1.5rem;   --space-8: 2rem;
    --space-12: 3rem;   --space-16: 4rem;

    /* Radius */
    --radius-sm:   4px;
    --radius-md:   8px;
    --radius-lg:   12px;
    --radius-full: 9999px;

    /* Type scale using clamp() — Ch 7 + Ch 9 */
    --text-sm:   clamp(0.8rem,   1.5vw, 0.875rem);
    --text-base: clamp(1rem,     2vw,   1.125rem);
    --text-lg:   clamp(1.125rem, 2.5vw, 1.375rem);
    --text-xl:   clamp(1.5rem,   3.5vw, 2.25rem);
```

```
    --text-2xl:  clamp(2rem,     5vw,   3.5rem);
}

/* ── tokens/semantic.css ──────────────────────────────────── */
/* Light mode defaults (references primitives only) */
:root {
    --color-bg:         var(--blue-50);
    --color-surface:    #ffffff;
    --color-border:     oklch(85% 0.02 250);
    --color-text:       var(--blue-900);
    --color-text-muted: var(--blue-700);
    --color-accent:     var(--blue-500);
    --color-success:    var(--green-400);
    --color-error:      var(--red-400);
}
```

Phase 2 — Reset & Base Typography

```
/* ── base/reset.css ──────────────────────────────────────── */
/* Ch 2: universal box-sizing reset */
*, *::before, *::after {
    box-sizing: border-box;
    margin:     0;
    padding:    0;
}

html {
    color-scheme:         light dark;    /* enables light-dark() — Ch 9 */
    scroll-behavior:      smooth;
    text-size-adjust:     none;
}

body {
    font-family:  system-ui, -apple-system, sans-serif;
    font-size:    var(--text-base);      /* fluid — no media query needed */
    background:   var(--color-bg);
    color:        var(--color-text);
    line-height:  1.6;
    min-block-size: 100dvh;              /* dvh — Ch 7 */
}
```

```css
/* Ch 3: Images responsive by default */
img, video, svg {
    max-inline-size: 100%;            /* logical property — Ch 10 */
    block-size:      auto;
    display:         block;
}

/* Ch 7: Respect motion preferences — global kill-switch */
@media (prefers-reduced-motion: reduce) {
    *, *::before, *::after {
        animation-duration:        0.01ms !important;
        animation-iteration-count: 1 !important;
        transition-duration:       0.01ms !important;
        scroll-behavior:          auto !important;
    }
}

/* ── base/typography.css ──────────────────────────────────────── */
h1 { font-size: var(--text-2xl); line-height: 1.15; font-weight: 700; }
h2 { font-size: var(--text-xl);  line-height: 1.2;  font-weight: 700; }
h3 { font-size: var(--text-lg);  line-height: 1.3;  font-weight: 600; }
p  { max-inline-size: 65ch; }    /* Ch 3: prose line length */

a {
    color:           var(--color-accent);
    text-underline-offset: 3px;    /* Ch 3 */
    transition:      color 0.2s ease;
}

:focus-visible {
    outline:        2px solid var(--color-accent);
    outline-offset: 3px;
    border-radius:  var(--radius-sm);
}
```

Phase 3 — Layout Primitives

```css
/* ── layout/primitives.css — CUBE composition layer ──────────── */
/* Ch 11 (CUBE CSS) + Ch 5 (flex) + Ch 6 (grid) + Ch 10 (logical) */

/* Container — fluid centring without media query (Ch 9 min()) */
```

```css
.container {
    inline-size:   min(100% - 2rem, 1200px);
    margin-inline: auto;
}

/* Stack — vertical rhythm with custom gap */
.stack {
    display:        flex;
    flex-direction: column;
    gap:            var(--stack-gap, var(--space-6));
}

/* Cluster — wrapping flex row */
.cluster {
    display:     flex;
    flex-wrap:   wrap;
    gap:         var(--cluster-gap, var(--space-4));
    align-items: var(--cluster-align, center);
}

/* Auto grid — responsive without media queries (Ch 6 auto-fill) */
.auto-grid {
    display:               grid;
    grid-template-columns: repeat(auto-fill, minmax(var(--grid-min, 280px), 1fr)
    gap:                   var(--grid-gap, var(--space-6));
}

/* Sidebar layout — Ch 5 + 6 RAM pattern */
.with-sidebar {
    display:   flex;
    flex-wrap: wrap;
    gap:       var(--space-8);

    > :first-child { flex-basis: var(--sidebar-w, 240px); }
    > :last-child  { flex: 1; min-inline-size: 60%; }
}
```

Phase 4 — Components

```css
/* ── components/nav.css ──────────────────────────────────────── */
/* Ch 5 (flex), Ch 8 (transition), Ch 10 (nesting, scroll-driven) */

.nav {
    position:     sticky;
    top:          0;
    z-index:      100;                        /* Ch 2: stacking context */
    display:      flex;
    align-items:  center;
    padding-block:  var(--space-4);             /* Ch 10: logical */
    padding-inline: var(--space-6);
    backdrop-filter: blur(12px);
    background:   color-mix(in oklch, var(--color-bg) 80%, transparent);
    border-block-end: 1px solid var(--color-border);

    /* Ch 8 + Ch 10: scroll-driven opacity — no JS */
    animation: nav-fade linear both;
    animation-timeline: scroll(root);
    animation-range:    0px 120px;

    /* Ch 10: nesting */
    &__logo {
        font-family: 'Courier New', monospace;
        color:       var(--color-accent);
        font-weight: 700;
        font-size:   var(--text-lg);
    }

    &__links {
        display:     flex;
        gap:         var(--space-6);
        margin-inline-start: auto;      /* Ch 5: push to end */
        list-style:  none;

        /* Ch 7: hide links on small screens */
        @media (max-width: 640px) {
            display: none;
        }
    }

    &__link {
        color:       var(--color-text-muted);
        text-decoration: none;
```

```
            transition:  color 0.2s ease;

            &:hover, &[aria-current="page"] {
                color: var(--color-accent);
            }
        }
    }

    @keyframes nav-fade {
        from { background: transparent; border-block-end-color: transparent; }
        to   { background: color-mix(in oklch, var(--color-bg) 80%, transparent); }
    }

    /* ── components/card.css ──────────────────────────────────────── */
    /* Ch 4 (:has()), Ch 8 (transitions), Ch 11 (BEM component tokens) */

    .card {
        --card-bg:     var(--color-surface);
        --card-radius: var(--radius-md);
        --card-border: var(--color-border);

        background:   var(--card-bg);
        border-radius: var(--card-radius);
        border:       1px solid var(--card-border);
        overflow:     hidden;
        transition:   transform 0.2s ease, border-color 0.2s ease,
                      box-shadow 0.2s ease;

        &:hover {
            transform:    translateY(-4px);
            border-color: var(--color-accent);
            box-shadow:   0 8px 24px rgb(0 0 0 / 0.15);
        }

        /* Ch 4: :has() — layout changes when card has an image */
        &:has(.card__media) {
            display: grid;
            grid-template-rows: auto 1fr;
        }

        &__media {
            aspect-ratio: 16 / 9;
            object-fit:   cover;
            inline-size:  100%;
```

```
    }

    &__body   { padding: var(--space-6); }
    &__title  { font-size: var(--text-lg); margin-block-end: var(--space-2); }
    &__meta   { color: var(--color-text-muted); font-size: var(--text-sm); }
    &__footer { margin-block-start: auto; padding-block-start: var(--space-4); }

    /* Variant via CUBE exception pattern (Ch 11) */
    &[data-featured] {
        --card-border: var(--color-accent);
        --card-bg:     color-mix(in oklch, var(--color-accent) 5%, var(--color-s
    }
}

/* ── components/form.css ──────────────────────────────────────── */
/* Ch 4 (:has(), :invalid, :placeholder-shown), Ch 9 (@property animation) */

@property --border-hue {
    syntax:        '<number>';
    inherits:     false;
    initial-value: 250;
}

.form-group {
    display: grid;
    gap:     var(--space-1);

    label {
        font-size:   var(--text-sm);
        font-weight: 600;
        transition:  color 0.2s ease;
    }

    input, textarea {
        --border-hue: 250;
        background:    var(--color-surface);
        color:         var(--color-text);
        border:        1px solid oklch(50% 0.12 var(--border-hue));
        border-radius: var(--radius-sm);
        padding:       var(--space-3) var(--space-4);
        transition:    --border-hue 0.3s ease;  /* animates via @property */

        &:focus { --border-hue: 250; }
    }
```

```
    /* :has() — parent sees child state — Ch 4 + Ch 10 */
    &:has(input:invalid:not(:placeholder-shown)) {
        label  { color: var(--color-error); }
        input  { --border-hue: 25; }          /* red hue — animated by @property
        .hint { display: block; }
    }

    .hint {
        display:   none;
        font-size: var(--text-sm);
        color:     var(--color-error);
    }
}
```

Phase 5 — Motion & Interaction

```
/* ── Scroll-driven reveal — no IntersectionObserver needed */
/* Ch 8 (scroll-driven) + Ch 10 (modern CSS) */

@keyframes reveal-up {
    from { opacity: 0; translate: 0 30px; }
    to   { opacity: 1; translate: 0 0;    }
}

.reveal {
    animation:          reveal-up linear both;
    animation-timeline: view();
    animation-range:    entry 0% entry 35%;
}

/* Read-progress bar — no JS */
.progress-bar {
    position:        fixed;
    inset-block-start: 0;
    inset-inline:    0;
    block-size:      3px;
    background:      var(--color-accent);
    transform-origin: inline-start;
    z-index:         200;
```

```
    animation:       grow-bar linear;
    animation-timeline: scroll(root block);
}
@keyframes grow-bar {
    from { scale: 0 1; }
    to   { scale: 1 1; }
}

/* Staggered entrance for project cards */
.auto-grid .card {
    animation: reveal-up 0.4s ease-out both;
    animation-delay: calc(var(--index, 0) * 60ms);
}
/* Set --index on each card via style="--index: N" in HTML */
```

Phase 6 — Theming & Dark Mode

```
/* ── themes/dark.css ──────────────────────────────────────── */
/* Ch 3 (oklch) + Ch 9 (tokens) + Ch 10 (light-dark) */

/* System preference */
@media (prefers-color-scheme: dark) {
    :root {
        --color-bg:         var(--blue-950);
        --color-surface:    var(--blue-900);
        --color-border:     oklch(25% 0.04 250);
        --color-text:       var(--blue-50);
        --color-text-muted: var(--blue-300);
    }
}

/* Manual toggle — overrides system preference */
[data-theme="dark"] {
    --color-bg:         var(--blue-950);
    --color-surface:    var(--blue-900);
    --color-border:     oklch(25% 0.04 250);
    --color-text:       var(--blue-50);
    --color-text-muted: var(--blue-300);
}
[data-theme="light"] {
    --color-bg:         var(--blue-50);
```

```
    --color-surface:    #ffffff;
    --color-border:     oklch(85% 0.02 250);
    --color-text:       var(--blue-900);
    --color-text-muted: var(--blue-700);
}

/* Theme toggle JS — only 10 lines */
// const root = document.documentElement;
// const saved = localStorage.getItem('theme');
// if (saved) root.dataset.theme = saved;
// document.getElementById('theme-toggle').addEventListener('click', () => {
//   const next = root.dataset.theme === 'dark' ? 'light' : 'dark';
//   root.dataset.theme = next;
//   localStorage.setItem('theme', next);
// });
```

Chapter Reference Map

Chapter	Feature used in project	Where it appears
Ch 1	`@layer`, cascade, `:where()`	main.css layer order, reset using :where() for zero-specificity base styles
Ch 2	box-sizing, position sticky, z-index, stacking context	Universal reset, sticky nav with backdrop-filter, modal z-index management
Ch 3	`oklch()`, `@font-face`, `clamp()` type, gradient text, max-inline-size	Entire colour token system, fluid type scale, hero background glow, prose width
Ch 4	`:has()`, `:not()`, `:focus-visible`, `:invalid`, `:placeholder-shown`, `[attr]`	Form validation (no JS), card layout switching, nav active state, data-featured cards
Ch 5	flex, `margin-inline-start: auto`, align-items, flex-wrap	Nav layout, hero CTA row, cluster primitive, skills tag list
Ch 6	grid, auto-fill, `minmax()`, subgrid, `grid-template-rows: auto 1fr`	Project grid (auto-fill), card with image (auto 1fr rows), page template
Ch 7	`clamp()`, `min()`, dvh, container queries, prefers-reduced-motion	Fluid type tokens, fluid container, min-block-size: 100dvh, global motion kill-switch
Ch 8	transition, cubic-bezier, `@keyframes`, `animation-timeline: scroll()/view()`	Card hover lift, button spring, scroll-driven nav fade, read-progress bar, reveal-on-scroll

Chapter	Feature used in project	Where it appears
Ch 9	var(), @property, clamp(), color-mix(), three-tier tokens	Entire token architecture, animated border-hue in form, color-mix() for glassmorphism nav
Ch 10	@layer, nesting, :has() in layout, logical properties, @starting-style	Layer strategy, BEM components written with & nesting, all margin-block/padding-inline
Ch 11	BEM, CUBE CSS, component tokens, @layer file structure	Card/Nav/Form follow BEM naming, layout primitives follow CUBE, @layer owns cascade order
Ch 12	Everything integrated	This project

Final File Structure

```
portfolio/ ├── index.html ├── styles/ │ ├── main.css — @layer order + all @imports │ ├── tokens/ │ │ ├── primitives.css — oklch palette, spacing, radii, type scale │ │ └── semantic.css — --color-bg, --color-text, --color-accent… │ ├── base/ │ │ ├── reset.css — box-sizing, img, reduced-motion │ │ └── typography.css — h1–h3, a, p, :focus-visible │ ├── layout/ │ │ └── primitives.css — .container, .stack, .cluster, .auto-grid │ ├── components/ │ │ ├── nav.css — sticky nav + scroll-driven opacity │ │ ├── hero.css — radial glow, fluid heading, CTA buttons │ │ ├── card.css — BEM card + :has() image layout │ │ ├── button.css — primary + ghost variants via component tokens │ │ └── form.css — @property border animation + :has() validation │ ├── themes/ │ │ └── dark.css — prefers-color-scheme + [data-theme] overrides │ └── utils.css — .visually-hidden, .text-muted, .reveal └── scripts/ └── theme.js — 10-line theme toggle + localStorage
```

Chapter Summary

Concept	Decision made in this project
Layer order	reset → tokens → base → layout → components → theme → utils. Utils always beats components — no !important needed anywhere.
Token architecture	Primitives in oklch for perceptual uniformity. Semantic tokens are the only thing swapped for dark mode. Components reference semantics via component tokens.

Concept	Decision made in this project
`Fluid without queries`	clamp() for type, min(100% - 2rem, 1200px) for container, auto-fill minmax() for grid. Zero breakpoint media queries in the layout layer.
`Motion`	Scroll-driven animations for nav fade and progress bar. view() timeline for section reveals. Global prefers-reduced-motion reset in base layer.
`Form validation`	:has(input:invalid:not(:placeholder-shown)) makes label change colour and hint appear. @property animates the border colour between hues. Zero JavaScript.
`Component API`	BEM names with native nesting. Component-level custom properties (--card-bg, --btn-radius) are the knobs for variants. data-* exceptions for one-offs.
`Dark mode`	prefers-color-scheme handles automatic. [data-theme] handles manual toggle. Both swap only semantic tokens. Components need zero dark-mode-specific rules.

FINAL PROJECT EXERCISES

1. **Build the token layer:** Create `tokens/primitives.css` with a full oklch colour ramp (50–950), spacing scale (space-1 through space-16), and fluid type scale using clamp(). Import both files into `main.css` via `@import layer(tokens)`. Verify all values are accessible via DevTools custom properties panel on :root.
2. **Build the nav component:** Implement the sticky nav using BEM naming and native nesting. Add the scroll-driven opacity animation. Test that it starts transparent over the hero and gains the frosted glass background as you scroll. Verify it doesn't break on screens narrower than 640px (links should hide).
3. **Build the project grid:** Use the `.auto-grid` layout primitive with `--grid-min: 280px`. Add `.card` components with BEM structure. Test that the grid reflows at all viewport widths with no media query in the component CSS. Add `style="--index: N"` to each card and verify the staggered entrance animation.
4. **Build the contact form:** Implement a form with name, email, and message fields using the `.form-group` component. Use `@property --border-hue` and `:has(input:invalid:not(:placeholder-shown))` so the border animates from blue to red on invalid input without JavaScript. Test with a non-email string in the email field.

5. **Wire up the theme toggle:** Add a button with id `theme-toggle`. Write the 10-line JS from Phase 6 that reads/writes `data-theme` on `document.documentElement` and persists to localStorage. Reload the page and confirm the chosen theme persists. Test that it overrides both system light and system dark preference.

CSS Overview + Deep Dives — Complete

12 chapters covering the full modern CSS landscape: the cascade and specificity, the box model, typography and colour, selectors, Flexbox, Grid, responsive design, transitions and animations, custom properties and functions, modern CSS features, CSS architecture, and a full capstone project. You now have the theoretical foundation, the practical patterns, and the architectural thinking to write maintainable, performant, accessible CSS at any scale.