CSS COURSE SERIES — COURSE 3 OF 4

CSS Advanced

12 chapters — Modern CSS Techniques for Production

A twelve-chapter deep dive into the cutting edge of CSS: cascade layers, container queries, subgrid, scroll-driven animations, logical properties, the oklch colour model, CSS Houdini, rendering performance, framework integration, and design token systems.

01 CSS Layers and Modern Cascade Control

02 Container Queries

03 Subgrid

04 The :has() Selector and Advanced Selector Patterns

05 Scroll-Driven Animations

06 Logical Properties and Writing Modes

07 Advanced Color — oklch and color-mix()

08 CSS Houdini — Painting and Typed OM

09 Performance — Repaints, Compositing, will-change

10 CSS in Component Frameworks

11 Building a Design System / Token-Based Theming

12 Project: Component Library with Modern CSS

@layer • Container Queries • Subgrid • :has() • Scroll-Driven • oklch • Houdini • Tokens

CHAPTER 1

CSS Layers and Modern Cascade Control

Chapter 1 — @layer and Modern Cascade Control

Cascade layers are the most significant addition to the CSS cascade since specificity was defined. They add an explicit priority tier *above* specificity — one that you control by declaration order rather than by selector weight. This chapter goes beyond the introduction in the Overview course and covers the full algorithm, every edge case, the `revert-layer` keyword, anonymous layers, nested layer hierarchies, `@import layer()` for vendor sandboxing, and how to debug layers in Chrome and Firefox DevTools.

> **Prerequisite.** This chapter assumes familiarity with @layer basics from the Overview course (Ch 10–11). Here we go to the algorithm level — reading browser source-style precision into every rule.

1. The Full Cascade Algorithm

The cascade is not a single check — it is a sequence of five filters applied in order. The browser stops at the first filter that produces a single winner. Only if a filter results in a tie does it proceed to the next one. Understanding the exact order is what makes cascade layers predictable.

1	**Origin + importance**	User-agent < author < user stylesheet. !important reverses this order completely.	Always first
2	**Cascade layer**	Within the same origin, the layer declared last wins. Unlayered styles beat all layers.	@layer — new
3	**@scope proximity**	The @scope root closest to the element wins when two scopes compete.	@scope — new
4	**Specificity**	A-B-C scoring: IDs > classes/attributes/pseudos > elements. Only reached after layers and scope tie.	Classic
5	**Source order**	Last declaration wins when all previous filters tie. The original tiebreaker.	Classic

The critical insight: **cascade layer (step 2) is resolved before specificity (step 4)**. A zero-specificity utility class in a higher-priority layer permanently beats a 3-ID chain in a lower-priority layer. Specificity is irrelevant across layer boundaries — it only matters within the same layer.

```
/* Proof: layer beats specificity across boundaries */
@layer base, components, utils;

@layer components {
    /* Very high specificity: 0-3-0 */
    .sidebar .card .card__title { color: #58a6ff; }
}

@layer utils {
    /* Low specificity: 0-1-0 — but utils is higher priority than components */
    .text-red { color: #ff7b72; }
}

/* <h2 class="card__title text-red">  →  red wins */
/* No !important, no higher specificity needed — layer order decides it */
```

2. Layer Ordering — Every Rule

The ordering statement

```
/* A @layer statement with no block declares the order */
@layer vendor, base, components, utils;
/* "utils" wins any conflict — it's declared last */

/* Layers can be filled in any order after the statement */
@layer utils       { .mt-4 { margin-top: 1rem; } }
@layer vendor      { .btn { padding: 8px; } }   /* still lowest priority */
@layer components  { .btn { padding: 10px; } }

/* Order is set by the STATEMENT, not by where the @layer blocks appear */

/* First-seen rule — if there is no statement, order is determined by */
/* the first time each layer name appears in the stylesheet */
@layer A { .x { color: red; } }   /* A is first-seen → lowest */
@layer B { .x { color: blue; } }  /* B is second → higher → blue wins */
/* Avoid relying on first-seen order — always use an explicit statement */
```

Unlayered styles always win

Priority order (highest → lowest)			specificity shown for comparison only
1	**unlayered**	Any style NOT inside an @layer block. Always wins regardless of specificity.	0-1-0

2	`@layer utils`	Declared last in the order statement — highest priority among layers.	0-3-0
3	`@layer components`	Middle priority.	0-2-0
4	`@layer base`	Low priority — even with high specificity, loses to higher layers.	0-3-0
5	`@layer vendor`	Declared first — lowest priority of all. Third-party CSS sandboxed here.	1-0-0

> **Unlayered styles as a safe default.** Because unlayered styles beat all layers, any existing CSS that you haven't migrated to a layer automatically takes priority. This makes incremental adoption safe — you can layer new code while old code continues to work. But it also means you must be deliberate: accidental unlayered styles will silently override everything inside your layers.

```
/* Gotcha: unlayered styles in an imported file */
@import url('legacy.css');          /* NOT layered — beats all your @layers */
@import url('legacy.css') layer(vendor);  /* sandboxed — safe */

/* Wrapping an existing stylesheet in a layer after the fact */
@layer vendor {
    @import url('bootstrap.css');   /* invalid — @import inside @layer block */
}
/* Correct syntax: @import url() layer() — the layer() goes on the import */
@import url('bootstrap.css') layer(vendor);  /* correct */
```

3. !important — The Reversal

Inside `!important` declarations the layer priority order is completely reversed. This mirrors how the origin stack works (user-agent !important beats author !important) and exists for the same reason — it lets lower-priority layers enforce overrides that cannot be beaten by higher-priority ones.

NORMAL DECLARATIONS — LAST LAYER WINS

```
4 @layer vendor                lowest
3 @layer base
2 @layer components
1 @layer utils                 wins

unlayered                      always wins
```

!IMPORTANT DECLARATIONS — REVERSED

```
1 @layer vendor !imp           wins
2 @layer base !imp
3 @layer components !imp
4 @layer utils !imp

unlayered !imp                 lowest
```

```
/* Practical consequence: !important in vendor layer beats your !important */
@layer vendor, base, components, utils;

@layer vendor {
    .btn { display: inline-block !important; }
}

@layer utils {
    .hidden { display: none !important; }
    /* utils is highest priority for NORMAL declarations */
    /* but for !important, vendor is highest — vendor wins here */
}

/* This is why you should NEVER put !important inside @layer blocks. */
/* It creates a hidden inverse priority system that is very hard to debug. */
/* @layer exists specifically so you don't need !important at all. */
```

The rule. Never use `!important` inside a layer. If you need a declaration to always win, put it in unlayered styles or in the highest-priority layer as a normal declaration. If a third-party stylesheet uses `!important`, wrap it in `layer(vendor)` — that `!important` then becomes the *lowest*-priority important and your own normal declarations will beat it.

4. Anonymous Layers

```
/* An @layer block with no name creates an anonymous layer */
@layer {
    .tooltip { position: absolute; }
}

/* Anonymous layers: */
/* - Cannot be referenced again — each anonymous block is its own layer */
/* - Cannot be reordered via a statement — they take priority by source order */
/* - Useful for one-off styles you never want overridden by named layers */
/* - Useful for quick sandboxing without naming the layer */

@layer named;           /* low priority */

@layer        { .a { color: red; }  }    /* anonymous-1: position sets priority */
@layer named { .a { color: blue; } }    /* named layer — declared before anonymous-1 → lo
@layer        { .a { color: green; } } /* anonymous-2: appears last → highest of all lay
/* Result: green wins — anonymous-2 is last, so highest priority */

/* Practical use: quick vendor wrap without inventing a name */
@import url('plugin.css') layer();    /* anonymous layer via @import */
```

5. Nested Layers

Layers can be nested inside other layers, creating a sub-priority hierarchy within a parent layer. Nested layers are only visible within their parent — they cannot affect the priority of anything outside it.

```
@layer components
   @layer base, variants, states;
   @layer base     { .btn { ... } }    lowest within components
   @layer variants { .btn--ghost { ... } }
   @layer states   { .btn:hover { ... } }    highest within components

@layer utils
      still beats ALL of components, regardless of component nesting
```

```css
/* Nested layer syntax */
@layer components {
    /* Declare sub-layer order first */
    @layer base, variants, states;

    @layer base {
        .btn {
            display:         inline-flex;
            padding:         0.5rem 1rem;
            border-radius:   6px;
            background:      var(--color-primary);
            transition:      transform 0.15s ease;
        }
    }

    @layer variants {
        .btn--ghost {
            background: transparent;
            border:     1px solid currentColor;
        }
    }

    @layer states {
        .btn:hover   { transform: translateY(-2px); }
        .btn:active  { transform: scale(0.97); }
        .btn:disabled{ opacity: 0.4; pointer-events: none; }
    }
}

/* Reference nested layers with dot notation */
/* components.base, components.variants, components.states */
/* Useful when debugging in DevTools — full dotted path shown */

/* Appending to a nested layer from outside — works, but consider readability */
@layer components.states {
    .btn.is-loading { cursor: wait; }
}
```

6. The revert-layer Keyword

`revert-layer` is a CSS-wide keyword (like `initial` or `inherit`) that rolls a property back to the value it would have had in the *next lower-priority layer*. It is the `@layer`-aware equivalent of `revert`, which rolls back to the browser stylesheet.

```css
/* revert vs revert-layer */
@layer base       { .link { color: #58a6ff; } }
@layer components { .nav .link { color: #fff;    } }
@layer utils      {
    .reset-color {
        color: revert-layer;
        /* rolls back to what components layer says: #fff */
        /* NOT back to base, NOT back to initial */
    }
}

/* revert-layer on an element in the components layer */
@layer components {
    .card--plain .card__title {
        font-size: revert-layer;
        /* rolls back to the value from the NEXT LOWER layer (base) */
        /* if base has no value, rolls back to the browser stylesheet */
        /* if no browser stylesheet value, falls back to initial */
    }
}

/* Classic use case: opt a specific element OUT of a component layer rule */
@layer base, components;

@layer base       { button { padding: 0.5rem 1rem; } }
@layer components {
    button         { padding: 0.75rem 1.5rem; }
    button.compact { padding: revert-layer;  }
    /* .compact button gets the base layer padding: 0.5rem 1rem */
}
```

7. @import + layer() — Vendor Sandboxing

```
/* Basic syntax */
@import url('normalize.css')  layer(base);
@import url('bootstrap.css')  layer(vendor);
@import url('animation-lib.css') layer();  /* anonymous layer */

/* Combine with media/supports conditions */
@import url('print.css')
    layer(print)
    supports(display: grid)
    print;

/* The layer() must appear BEFORE supports() and media conditions */
/* Order: url() → layer() → supports() → media */

/* What sandboxing achieves: */
/* Before: Bootstrap's .container has high specificity → hard to override */
/* After: Bootstrap is in vendor layer → a zero-specificity utility beats it */

@layer vendor, base, components, utils;
@import url('bootstrap.css') layer(vendor);

@layer utils {
    /* This beats Bootstrap's .container-fluid.px-4 (however specific) */
    .p-0 { padding: 0; }
}

/* @import must be at the top of the file — before any rule blocks */
/* The layer order statement must also come before the @imports that fill layers */

/* Correct file structure: */
/* 1. @layer order statement */
/* 2. @import url() layer() statements */
/* 3. @layer blocks with your own rules */
```

8. Layers with @media, @supports, and @scope

```
/* @layer inside @media — valid and useful */
@media (min-width: 768px) {
    @layer layout {
        .sidebar { display: block; }
    }
}
/* This appends to the 'layout' layer, but only on screens >= 768px */
/* Layer priority is unchanged — it's still in the 'layout' layer */

/* @layer inside @supports — feature-gate whole layer blocks */
@supports (container-type: inline-size) {
    @layer components {
        .card { container-type: inline-size; }
    }
}

/* @scope inside @layer — scope IS layer-aware */
@layer components {
    @scope (.card) {
        .title { font-size: 1.25rem; }
    }
}
/* Scope proximity resolves WITHIN the same layer */
/* Layer priority still applies first */

/* CSS nesting and @layer — fully compatible */
.card {
    background: var(--surface);

    /* @layer nested inside a rule — adds to that layer globally */
    @layer state {
        &.is-selected { outline: 2px solid var(--accent); }
    }
}
/* Equivalent to: @layer state { .card.is-selected { outline: ... } } */
```

9. Debugging Layers in DevTools

01 **Open the Styles panel** in Chrome DevTools (F12 → Elements → Styles). When @layer is in use, a **layer badge** appears next to each rule showing its layer name (e.g. `components.states`). Nested layers show the full dotted path.

02 **Crossed-out rules** in the Styles panel indicate a rule that lost the cascade. Hover the crossed-out rule to see which rule beat it and why. If a rule you expect to win is crossed out, check whether the winning rule is in a higher-priority layer — specificity alone won't tell you this.

03 **Layers panel** (Chrome 102+): in the Elements → Styles sidebar, there is a **Layers** pane that lists all layers on the page in their priority order (highest to lowest). Click a layer to filter the Styles panel to show only rules from that layer. Unlayered styles appear at the top as `(unlayered)`.

04 **Firefox DevTools**: the Inspector → Rules panel shows a `layer:` annotation next to layered rules. The cascade icon (⊕) next to a property shows the full cascade calculation — click it to see which layer and which rule won.

05 **Console inspection**: `document.adoptedStyleSheets` lists programmatically constructed stylesheets. For file-based layers, `document.styleSheets` exposes each imported file's rules — you can read `cssRule.layerName` on a `CSSLayerBlockRule` or `CSSLayerStatementRule` to inspect layer metadata from JavaScript.

```
/* Inspecting layer structure from JavaScript */
const sheets = [...document.styleSheets];

for (const sheet of sheets) {
    for (const rule of sheet.cssRules) {
        if (rule instanceof CSSLayerBlockRule) {
            console.log('Layer block:', rule.name);        // 'components'
        }
        if (rule instanceof CSSLayerStatementRule) {
            console.log('Layer order:', rule.nameList);  // ['vendor', 'base', ...]
        }
    }
}
```

10. Real-World Layer Strategy Patterns

```
/* ── Pattern A: Design system with consumer overrides ──────────── */
/* The design system ships with lower-priority layers so consumers can override */

/* design-system/index.css (consumed as a package) */
@layer ds.reset, ds.tokens, ds.base, ds.components;

@layer ds.components {
    .button { background: var(--ds-color-primary); }
}

/* consumer/app.css */
@layer ds.reset, ds.tokens, ds.base, ds.components, overrides;
@import url('design-system/index.css') layer();
/* anonymous layer wraps the entire design system at low priority */
/* consumer's own rules outside any layer beat all of it */

/* ── Pattern B: Multi-team monorepo ──────────────────────────────── */
/* Each team owns a named sub-layer, ordered by the platform team */
@layer platform, team-a, team-b, team-c, overrides;

@import url('platform/styles.css') layer(platform);
@import url('team-a/styles.css')   layer(team-a);
@import url('team-b/styles.css')   layer(team-b);
/* Conflicts between teams are resolved by the priority order */
/* The platform team can set the order and each team's work stays sandboxed */

/* ── Pattern C: Progressive enhancement ──────────────────────────── */
/* Older browsers don't support @layer — ensure they still work */
@supports not (selector(@layer test { })) {
    /* Styles for browsers without @layer support */
    .btn { background: blue; }
}
/* In practice: @layer is Baseline 2022 — all evergreen browsers support it */
/* Only IE and very old Edge (pre-Chromium) do not. For those, unlayered fallback. */
```

Chapter Summary

Concept	Key point
`Cascade position`	Layer is step 2 of 5 in the cascade — resolved before specificity. A layer win cannot be overridden by specificity in a lower-priority layer.
`Order rule`	The layer declared last in the order statement wins. Unlayered styles always beat every named layer, regardless of specificity.
`Order statement`	Always declare order with @layer a, b, c; at the top of main.css. Never rely on first-seen order across multiple files.
`!important reversal`	Inside !important, layer priority reverses — the lowest-priority layer's !important beats the highest-priority layer's !important. Never use !important inside a layer.
`Anonymous layers`	@layer { } with no name creates a one-off anonymous layer ordered by source position. Cannot be appended to. Useful for quick vendor wrapping.
`Nested layers`	Layers can nest — the dotted path is components.states. Nesting creates a sub-hierarchy within the parent layer. External layers still beat the entire parent.
`revert-layer`	Rolls a property back to the value from the next lower layer. Used to opt one element out of a rule without !important or specificity escalation.
`@import layer()`	@import url() layer(name) assigns the imported stylesheet to a named layer. Must appear before any rule blocks. @import inside a @layer block is invalid.
`DevTools`	Chrome shows layer badges in the Styles panel and a Layers pane for priority overview. Firefox shows layer: annotations. CSSLayerBlockRule / CSSLayerStatementRule available in JS.
`Vendor sandboxing`	@import url('library.css') layer(vendor) makes the library's !important declarations the lowest priority, and any zero-specificity rule in a higher layer beats all its selectors.

EXERCISES

1. **Layer the cascade proof:** Create a stylesheet with `@layer base, components, utils`. In `base` write `h1 { color: red }`, in `components` write `.hero h1 { color: blue }` (higher specificity), and in `utils` write `.text-green { color: green }` (lower specificity). Apply all three classes to one element. Confirm green wins. Then remove the layer order statement and observe how priority collapses to specificity-based resolution.
2. **Vendor sandboxing:** Find or create a small CSS library file with at least one `!important` declaration. Import it with `@import url() layer(vendor)`. Write a normal utility class in an unlayered stylesheet that targets the same property. Confirm your utility wins despite the library's `!important` — and explain in a comment why this works using the reversal rule.
3. **Nested layer component:** Build a button component using three nested layers inside `@layer components`: `base` (structure and colour), `variants` (.btn--ghost, .btn--danger), and `states` (:hover, :focus-visible, :disabled). Write a rule in the `base` sub-layer that conflicts with one in `states`. Verify the correct sub-layer wins by source position.
4. **revert-layer in practice:** Create a `@layer base` that sets `a { color: #58a6ff; text-decoration: underline }`. Create a `@layer components` that sets `.nav a { color: #fff; text-decoration: none }`. Add a `.nav__link--plain` rule inside `components` that uses `revert-layer` on both properties. Confirm the link inside .nav with the plain class picks up the base layer values.
5. **DevTools layer audit:** Load any page that uses @layer (or add layers to one of your own). Open Chrome DevTools, inspect an element, and use the Layers pane to list all layers in priority order. Find one rule that is crossed out due to a layer conflict (not a specificity conflict) — identify the winning layer and rule. Document the full dotted layer path of both the winner and the loser.

Next: Chapter 2 — Container Queries. Size-based and style-based container queries in depth — container-type, container-name, @container syntax, cqw/cqh/cqi/cqb units, the containment model, style queries with custom properties, and patterns for truly component-driven responsive design.

CHAPTER 2

Container Queries

Chapter 2 — Container Queries

Media queries let components react to the *viewport*. Container queries let them react to their own *parent's size*. The distinction sounds small — it changes everything about how you write component CSS. A card that adapts based on whether it's in a narrow sidebar or a wide main column no longer needs two separate media query breakpoints; it adapts based on the space it actually has. This chapter covers the full containment model, size queries, style queries, container query units, nested queries, and every edge case you'll encounter building real components.

> **Baseline 2023.** Container queries (`@container`), size containment (`container-type: inline-size`), and container query units (`cqi`, `cqb`, etc.) are supported in all evergreen browsers. Style queries are also supported in all evergreens as of 2024, but only for custom properties — not yet for standard properties like `color`.

1. The Containment Model

Container queries build on CSS **containment** — the `contain` property. Containment tells the browser that an element's rendering is isolated from the rest of the page, enabling optimisations. `container-type` applies a specific flavour of containment automatically.

SIZE CONTAINMENT

`contain: size`

Element's size is independent of its content. The element behaves as if its children don't exist for sizing purposes. Applied by `container-type: size`. Rarely used alone — mostly in `container-type: size` where you need both axes queryable.

INLINE-SIZE CONTAINMENT

`contain: inline-size`

Same as size containment but only on the inline axis (width in horizontal writing modes). Content can still affect block size. Applied by `container-type: inline-size`. The most useful type — use this 90% of the time.

LAYOUT CONTAINMENT

`contain: layout`

The element's internal layout is isolated — absolutely-positioned children don't escape it, counters are isolated, and the element creates a new stacking context. Applied implicitly by all container-type values.

STYLE CONTAINMENT

`contain: style`

CSS counters and quotes are scoped to the element. Separate from the `container-type: style` that enables style queries — these are different

concepts sharing confusingly similar names.

```css
/* What container-type actually does under the hood */
container-type: inline-size;
/* Is roughly equivalent to: */
contain: layout inline-size style;

container-type: size;
/* Is roughly equivalent to: */
contain: layout size style;

container-type: normal;   /* default — no containment, no queryability */
container-type: scroll-state;  /* draft spec — enables scroll-state queries */
```

container-type values

Value	Containment applied	Width queryable	Height queryable	Use case
`inline-size`	layout, inline-size, style	Yes	No	Cards, sidebars, any width-responsive component. Default choice.
`size`	layout, size, style	Yes	Yes	Elements with explicit heights where you also need height queries. Rare.
`normal`	none	No	No	Default. Can still receive style queries if a custom property is set.
`style`	style only	No	No	Style queries only — no size queries. See Section 5.

2. Syntax — container-type and container-name

```
/* Longhand */
.sidebar {
    container-type: inline-size;
    container-name: sidebar;
}

/* Shorthand: container: <name> / <type> */
.sidebar {
    container: sidebar / inline-size;
}

/* Multiple names on one container */
.card-wrapper {
    container: card layout-region / inline-size;
    /* can be queried as 'card' or as 'layout-region' */
}

/* @container syntax */
@container          (min-width: 400px) {    /* unnamed — matches nearest named ancestor *
    .card { flex-direction: row; }
}

@container sidebar (min-width: 300px) {  /* named — only matches container named 'sideba
    .nav-item { display: flex; }
}

/* Logical operators */
@container card (min-width: 400px) and (max-width: 700px) {
    .card__media { width: 40%; }
}

@container (min-width: 400px) or (min-height: 600px) {
    .card { padding: 2rem; }
}

@container not (min-width: 400px) {
    .card { flex-direction: column; }
}
```

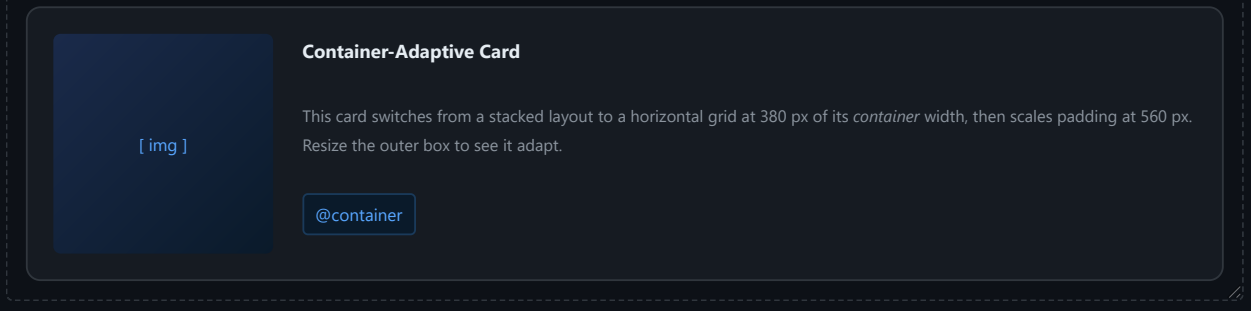
Size features available in @container

```
/* All queryable size features */
/* (container-type: inline-size only unlocks width/inline-size) */

@container (width:        400px) {}      /* exact width */
@container (min-width:    400px) {}      /* width >= 400px */
@container (max-width:    700px) {}      /* width <= 700px */
@container (inline-size:  400px) {}      /* logical equivalent of width */

/* Range syntax (same as modern media queries) */
@container (400px <= width < 800px) {}   /* modern range */
@container (width >= 500px) {}           /* alternative range syntax */

/* Height queries — only if container-type: size (not inline-size) */
@container (height:       300px) {}
@container (block-size:   300px) {}      /* logical block */
@container (aspect-ratio: 16 / 9) {}     /* ratio — requires container-type: size */
@container (orientation:  landscape) {} /* derived from aspect-ratio */
```

3. Live Demo — Resizable Card

Drag the right edge of the box below. The card adapts its layout at 380 px and 560 px of its container's width — not the viewport.

LIVE — RESIZE ME



Resize handle appears at the right edge of the dashed box.

```css
/* The CSS behind the demo above */
.card-wrapper {
    container: card / inline-size;
}

.card {
    display:     grid;
    grid-template-columns: 1fr;   /* default: stacked */
    gap:         10px;
    padding:     14px;
}

@container card (min-width: 380px) {
    .card { grid-template-columns: 120px 1fr; }
}

@container card (min-width: 560px) {
    .card { padding: 20px; gap: 16px; }
}
```

4. Container Query Units

Container query units (cq units) are length values relative to the queried container — not the viewport. They let you size things proportionally to the container without needing a query breakpoint at all.

```css
/* cqw / cqi — use inside any descendant of a sized container */

/* Fluid type scale locked to the card container, not the viewport */
.card-wrapper {
    container-type: inline-size;
}

.card__title {
    font-size: clamp(1rem, 4cqi + 0.5rem, 2rem);
    /* grows from 1rem to 2rem as the container widens */
}

.card__image {
    width: 30cqi;  /* always 30% of the card's width */
}

/* cq units in calc() */
.card__gap {
    gap: calc(2cqi + 8px);
}

/* cq units vs vw units — the key difference */
.hero-text {
    font-size: 5vw;   /* relative to viewport — breaks in narrow sidebars */
    font-size: 5cqi;  /* relative to container — correct in any context */
}

/* cqh / cqb — require container-type: size (both axes contained) */
.modal-content {
    container: modal / size;   /* must have explicit height on .modal-content */
}

.modal-body {
    max-height: 80cqb;  /* 80% of the modal's block (height) */
}

/* Nearest container resolution — cq units always resolve against */
/* the nearest ANCESTOR with appropriate containment, not the root */
/* This makes them genuinely component-scoped, not global */
```

cqh and cqb require an explicit height on the container. With `container-type: inline-size`, the block axis is not contained — the element still grows with its content and has no queryable block size. You need `container-type: size` and an explicit `height` (or `block-size`) on the container element for these units to resolve.

5. Style Queries

Style queries let you query the *computed value of a CSS custom property* on an ancestor container, rather than its size. They are available in all evergreen browsers as of 2024 — but only for custom properties, not for standard properties like `color` or `display` (that part of the spec is not yet implemented).

```
/* Style query syntax — query a custom property on the container */
.theme-wrapper {
    container-type: style;   /* or 'inline-size' — style queries work on any container-t
    container-name: theme;
}

/* Apply a theme variant via a custom property on the container */
.theme-wrapper[data-theme="dark"] {
    --theme: dark;
}
.theme-wrapper[data-theme="light"] {
    --theme: light;
}

/* Query that custom property from any descendant */
@container theme style(--theme: dark) {
    .card {
        background: #161b22;
        color:      #c9d1d9;
        border:     1px solid #30363d;
    }
}

@container theme style(--theme: light) {
    .card {
        background: #fff;
        color:      #24292f;
        border:     1px solid #d0d7de;
    }
}
```

Why style queries matter

Before style queries	With style queries

A card component that needs dark-theme styles inside a dark wrapper requires:

```
.dark-parent .card { ... }
```

This is a context-coupled selector — the card component must know about its parent.

The card queries its container for --theme: dark itself.

```
@container style(--theme: dark)
{ .card { ... } }
```

The card is self-contained — it adapts without needing knowledge of the parent class.

```css
/* Advanced: combining style + size queries */
@container card style(--variant: featured) {
    .card { border: 2px solid var(--accent); }
}

@container card (min-width: 400px) and style(--variant: featured) {
    .card { grid-template-columns: 1fr 1fr; }
    /* only wide AND featured cards get the two-column layout */
}

/* Checking for a property BEING SET (any value) */
/* Not yet standardised — avoid for now */
/* The safe pattern: use a boolean-like custom property */
.wrapper { --is-featured: true; }
@container style(--is-featured: true) {
    .card { border-color: gold; }
}
```

6. The Containment Trap

A container cannot query itself

When you set `container-type: inline-size` on an element, that element is the container. The `@container` rules inside it only affect its **descendants** — they cannot change properties on the container element itself. Trying to style the container element from within its own @container query has no effect.

The fix: wrap it.

Add a wrapper element one level up. Apply `container-type` to the wrapper, then write container queries that target the inner element you actually want to style.

```
/* Wrong — the container cannot style itself */
.card {
    container: card / inline-size;
    padding:   1rem;
}

@container card (min-width: 400px) {
    .card { padding: 2rem; }   /* has no effect */
}

/* ———————————————————————————————————————————————— */
/* Correct — the WRAPPER is the container, the card is the target */
.card-wrapper {
    container: card / inline-size;
}

.card {
    padding: 1rem;
}

@container card (min-width: 400px) {
    .card { padding: 2rem; }   /* works */
}

/* In practice, many grid and flex wrappers are already structural */
/* elements — the wrapper rarely needs to be a purely cosmetic div */
```

7. Nested Container Queries

```
/* Multiple nested containers — each @container resolves against */
/* its nearest NAMED ancestor matching the name, or the nearest */
/* unnamed ancestor with containment if no name specified */

.layout {
    container: layout / inline-size;
}
.sidebar {
    container: sidebar / inline-size;
}
.card-wrapper {
    container: card / inline-size;
}

/* This query matches the 'card' container directly */
@container card (min-width: 300px) {
    .card { flex-direction: row; }
}

/* This one matches the 'sidebar' ancestor */
@container sidebar (max-width: 200px) {
    .card .card__title { font-size: 0.875rem; }
}

/* This one matches the outermost 'layout' container */
@container layout (min-width: 1200px) {
    .sidebar { display: block; }
}

/* Nesting @container rules */
@container card (min-width: 300px) {
    .card { display: grid; }

    /* Inner @container query — queries the named 'layout' container */
    @container layout (min-width: 900px) {
        .card { grid-template-columns: 1fr 1fr; }
    }
}
```

8. Container Queries with CSS Grid and Flexbox

```
/* Grid auto-fit — still viewport-based */
.card-grid {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(280px, 1fr));
    /* this uses the grid container's width — already component-relative */
    /* auto-fill tracks the card-grid element's own width */
}

/* BUT each card inside the grid needs its OWN container query */
/* because each card is sized by the grid, not by the viewport */
.card-grid__item {
    container: card / inline-size;
}

@container card (min-width: 360px) {
    .card { display: flex; }
    .card__image { width: 120px; flex-shrink: 0; }
}

/* ── Sidebar + main layout — classic container query use case ─── */
.page-layout {
    container: page / inline-size;
    display:   grid;
    grid-template-columns: 1fr;
}

@container page (min-width: 768px) {
    .page-layout {
        grid-template-columns: 240px 1fr;
    }
}

/* Wait — .page-layout is both the container AND the element being styled! */
/* This hits the containment trap. Solution: apply container-type to a
   grandparent, or use a media query here since this IS a layout-level concern */

/* Better pattern: media query for page-level layout decisions */
@media (min-width: 768px) {
    .page-layout { grid-template-columns: 240px 1fr; }
}
/* container queries for components WITHIN the layout */
.card-wrapper {
```

```
    container: card / inline-size;
}
```

9. Container Queries vs Media Queries

	Media query (@media)	Container query (@container)
Query target	The viewport (browser window)	The nearest qualifying ancestor element
Use for	Page-level layout, typography scale, breakpoints for the overall page structure	Component-level layout — a card, sidebar widget, nav, or any repeated UI element
Reusability	Component must be rebuilt at each breakpoint if context changes	Component adapts wherever it's placed — true portability
Performance	Browser matches immediately — very fast	Requires layout pass to know container size — slightly more work but negligible in practice
Self-styling	Can style any element	Cannot style the container element itself (containment trap)
Height queries	Can query viewport height freely	Height queries require `container-type: size` + explicit height
Print/other media	Can target print, screen, speech, etc.	No media-type targeting — size only (or style)

> **The rule of thumb.** Use media queries for the page skeleton — the overall layout grid, the sidebar appearing, the nav collapsing. Use container queries for every repeated component inside that skeleton: cards, widgets, article previews, form sections. If the thing being styled could appear in multiple contexts at different widths, it's a container query candidate.

10. Container Queries and @layer

```
/* @container inside @layer — the query lives in that layer */
@layer components {
    .card-wrapper {
        container: card / inline-size;
    }

    @container card (min-width: 400px) {
        .card { display: flex; }
    }
}

/* Overriding a container query from a higher-priority layer */
@layer utils {
    /* This wins over the components container query — */
    /* layer priority trumps everything before specificity */
    .card { display: block; }
}

/* Container queries INSIDE a layer are still subject to that layer's priority */
/* A container query in utils will beat one in components (same layer rules apply) */

/* The interplay: specificity of @container rules */
/* @container conditions don't add specificity to the contained selectors */
/* The selectors inside @container have exactly their own specificity */
@container card (min-width: 400px) {
    .card { color: red; }   /* specificity: 0-1-0 (just .card) */
}
/* The @container condition is NOT part of the specificity calculation */
/* This means two rules inside different @container blocks may tie on specificity */
/* and source order then decides — so always write container queries after base rules */
```

Chapter Summary

Concept	Key point
`container-type`	inline-size — width queryable, height not. size — both axes queryable, requires explicit height. normal — default, no queries. style — custom property queries only.
`container-name`	Required for named queries. One element can have multiple names. Use the shorthand: container: name / type.
`@container syntax`	Unnamed queries match nearest qualifying ancestor. Named queries match the nearest ancestor with that name. Supports and, or, not, and range syntax.
`Size features`	width, min-width, max-width, inline-size (logical). height and block-size need container-type: size. aspect-ratio and orientation derived from both axes.
`CQ units`	cqw/cqi — inline axis. cqh/cqb — block axis (needs size). cqmin/cqmax — smaller/larger axis. Resolve against nearest ancestor with appropriate containment.
`Style queries`	Query computed value of a custom property on an ancestor container. Currently works for custom properties only — standard property queries not yet implemented in browsers.
`Containment trap`	A container cannot style itself via its own @container query. Wrap it: apply container-type to the parent, query from descendants.
`Nested containers`	Each @container resolves against the nearest qualifying ancestor with the matching name. Inner containers shadow outer ones of the same name.
`CQ vs MQ`	Media queries for page-level layout. Container queries for components. Mixing both is correct — use each at the right level of abstraction.
`@container specificity`	The @container condition does not add specificity. Selectors inside have their own specificity only. Write container queries after base rules to ensure source-order wins correctly.

EXERCISES

1. **Three-state card component:** Build a card that switches layout at two container breakpoints — stacked below 320 px, horizontal between 320 px and 600 px, and a featured two-column layout above 600 px. Place the card in a CSS grid with auto-fill and watch it switch layouts as the grid resizes it. Confirm that a media query alone could not achieve this correctly in multiple grid-column contexts.
2. **Container query fluid type:** Create a component with a heading and body text. Using only `cqi` units and `clamp()`, make the heading scale fluidly between 1 rem and 2.5 rem as the container grows from 200 px to 800 px — with no breakpoints. Confirm the type scale is locked to the container, not the viewport, by placing two instances at different column widths side by side.
3. **Style query theme system:** Build a theme wrapper component with a `data-theme` attribute and a `--theme` custom property. Inside it, place a button and a card. Write style queries so that both the button and card automatically adapt their colour scheme (background, border, text) based on the `--theme` custom property — without the button or card knowing their parent class name. Demonstrate with both `data-theme="dark"` and `data-theme="light"` wrappers on the same page.
4. **Named container hierarchy:** Create a three-level hierarchy: `page` (outermost) → `sidebar` → `widget`. Write separate @container queries for each level that change the widget's appearance. Verify that each query correctly targets its named container and not the others, then add an unnamed query and confirm it resolves against the nearest named ancestor.
5. **CQ + @layer interaction:** Set up a layer order: base, components, utils. In the components layer, create a card container query that changes the card's flex-direction at 400 px. In the utils layer, add a rule that forces `flex-direction: column` on all cards. Confirm the utils layer wins regardless of the container condition being true. Then move the utils rule inside a lower-priority layer and confirm the container query wins. Document the specificity vs layer priority interaction in a comment.

Next: Chapter 3 — Subgrid. How `grid-template-rows: subgrid` and `grid-template-columns: subgrid` let nested elements participate in their ancestor's grid tracks — solving the alignment problems that were previously impossible without JavaScript. Includes cross-nested-element alignment, subgrid and named lines, and real card-grid use cases.

CHAPTER 3

Subgrid

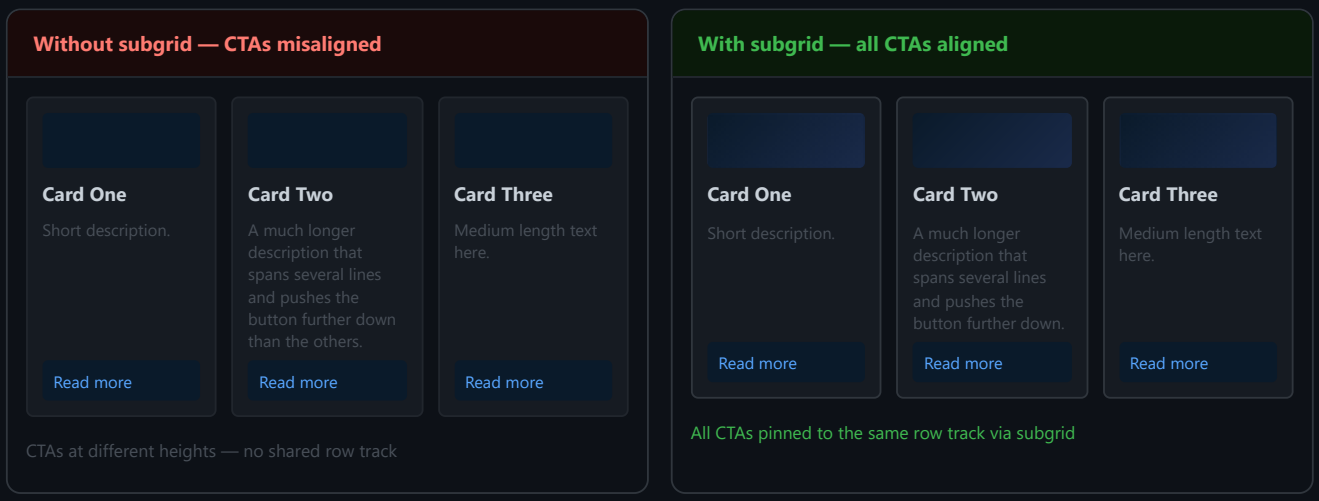
Chapter 3 — Subgrid

For years, CSS Grid was excellent at aligning items within a single grid container, but completely powerless to align content *across* sibling grid items. Cards in a grid could not share internal row alignment — each card was its own island. You ended up hacking around this with `display: contents`, JavaScript height matching, or just accepting misaligned CTAs. Subgrid solves this properly: a nested element can opt into its ancestor's grid tracks and participate in the same line system.

> **Baseline 2023.** `grid-template-rows: subgrid` and `grid-template-columns: subgrid` are supported in all evergreen browsers since late 2023. Safari shipped it first (2022), Firefox in 2019, Chrome in 2023. There are no polyfills — it requires a real subgrid implementation.

1. The Problem Subgrid Solves

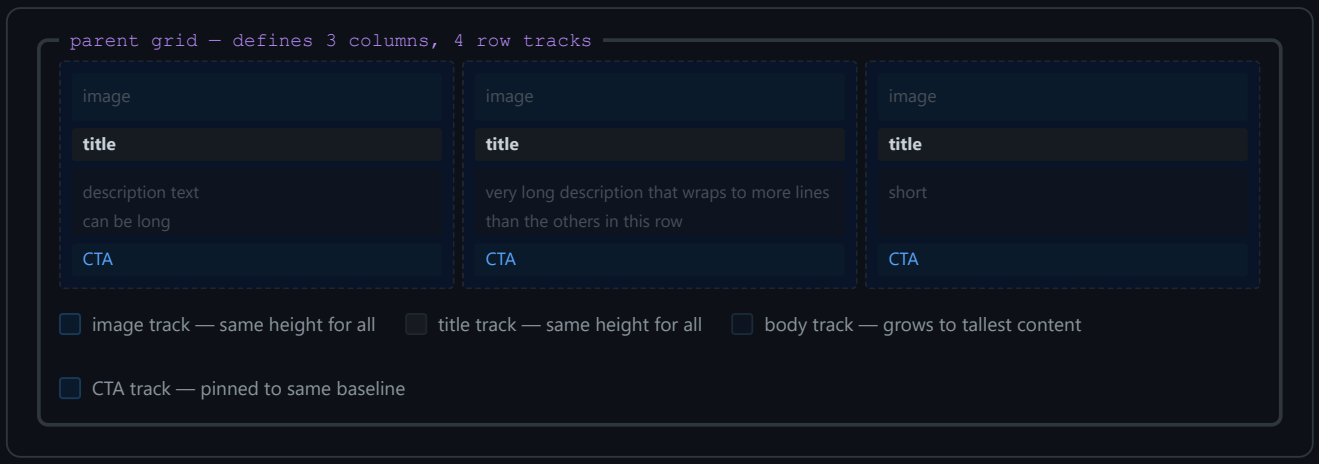
A card grid with varying content lengths produces misaligned internal sections when each card manages its own layout. The card with the longest description pushes its CTA button lower, breaking visual alignment across the row.



The right panel above is live — the cards are using `grid-template-rows: subgrid` to participate in the outer grid's row tracks. All CTAs sit on the same track regardless of how long the description text is.

2. How Subgrid Works

When a grid item has `grid-template-rows: subgrid` (or `grid-template-columns: subgrid`), it becomes a **subgrid container**. Its children are placed into the parent grid's tracks for that axis — they don't create new independent tracks. The parent grid determines track sizes; the subgrid just borrows them.



```css
/* ── The canonical card-grid subgrid pattern ──────────────────── */
.card-grid {
    display:               grid;
    grid-template-columns: repeat(3, 1fr);

    /* DO NOT define row tracks here — let the cards define them via content */
    /* The parent just needs auto rows; subgrid inherits whatever size they become */
    grid-auto-rows:        auto;
    gap:                   1rem;
}

.card {
    /* Span 4 rows of the parent grid (one per internal section) */
    grid-row:              span 4;

    /* Become a subgrid on the row axis */
    display:              grid;
    grid-template-rows:   subgrid;

    /* gap on a subgrid uses the PARENT's row gap — override if needed */
    gap:                   0.5rem;
    padding:               1rem;
    background:            var(--surface);
    border-radius:         8px;
}

/* No special CSS needed on the card children — they just flow into the 4 subgrid rows */
/* Each child occupies one row track automatically */
.card__image       { /* row 1 — fixed image height via min-height or aspect-ratio */
    aspect-ratio: 16 / 9;
}
.card__title       { align-self: start; }  /* row 2 */
.card__description { }                     /* row 3 — grows to tallest in group */
.card__cta         { align-self: end;   }  /* row 4 — pinned to bottom of track */
```

3. Axis Independence

Subgrid can be applied to rows, columns, or both axes independently. Mixing a subgrid axis with an explicit-tracks axis on the other axis is fully valid.

```css
/* Subgrid on rows only — most common */
.card {
    display:              grid;
    grid-row:             span 4;
    grid-template-rows:   subgrid;           /* rows come from parent */
    grid-template-columns: auto 1fr;          /* columns are independent */
}

/* Subgrid on columns only — useful for nested navigation lists */
.nav-group {
    display:              grid;
    grid-column:          span 3;
    grid-template-columns: subgrid;           /* columns come from parent */
}

/* Subgrid on BOTH axes — card participates in parent rows and columns */
.feature-block {
    display:              grid;
    grid-column:          span 2;
    grid-row:             span 3;
    grid-template-columns: subgrid;
    grid-template-rows:   subgrid;
}

/* Mixing explicit tracks on the subgrid axis is NOT valid */
/* grid-template-rows: subgrid 100px → invalid — it's subgrid or explicit, not both */
/* The number of subgrid tracks is determined by the span, not by additional values */
```

4. Gap Behaviour in Subgrid

```
/* By default, a subgrid inherits the parent's gap for its subgrid axis */

.card-grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    row-gap:    2rem;    /* gap between cards — subgrid children inherit this */
    column-gap: 1rem;
}

.card {
    display:            grid;
    grid-row:           span 4;
    grid-template-rows: subgrid;
    row-gap:            0.5rem;  /* OVERRIDES parent gap within this subgrid */
    /* Without this, the parent's 2rem row-gap would appear between card sections */
}

/* Summary of gap rules: */
/* 1. A subgrid axis inherits the parent's gap for that axis by default */
/* 2. Setting gap on the subgrid element overrides for the subgrid's children */
/* 3. The NON-subgrid axis uses the subgrid element's own gap setting */

/* Practical consequence: if your card's internal spacing looks too large, */
/* set a smaller row-gap on the card element to override the parent's row-gap */
```

5. Named Grid Lines in Subgrid

The parent grid's named lines are inherited by the subgrid. Any names you define on the parent's tracks are available inside the subgrid to position children. You can also define additional names on the subgrid element itself — they are layered on top of the inherited names.

```css
/* Parent defines named row tracks */
.card-grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);

    /* Named lines on the implicit row axis */
    /* No explicit row template — rows are implicit (auto) */
}

/* Alternatively: explicit named rows on the parent */
.card-grid--named {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    grid-template-rows:
        [card-image-start]    auto
        [card-image-end card-title-start]  auto
        [card-title-end card-body-start]   auto
        [card-body-end  card-cta-start]    auto
        [card-cta-end];
}

.card {
    display:            grid;
    grid-row:           card-image-start / card-cta-end;
    grid-template-rows: subgrid;
}

/* Subgrid children can now use the inherited named lines */
.card__image  { grid-row: card-image-start / card-image-end; }
.card__title  { grid-row: card-title-start / card-title-end; }
.card__body   { grid-row: card-body-start  / card-body-end;  }
.card__cta    { grid-row: card-cta-start   / card-cta-end;   }

/* Adding names to the subgrid element itself */
.card {
    display:            grid;
    grid-row:           span 4;
    grid-template-rows: subgrid [inner-start] [img-end] [body-start] [inner-end];
    /* The bracket names here apply to the tracks within the subgrid */
    /* They are layered on top of any inherited names from the parent */
}
```

6. Subgrid with auto-fill and auto-fit

```
/* Subgrid works with auto-fill column grids — cards wrap to new rows */
/* Each row group of cards shares row track sizes */
.card-grid {
    display:               grid;
    grid-template-columns: repeat(auto-fill, minmax(280px, 1fr));
    gap:                   1rem;
}

.card {
    display:              grid;
    grid-row:             span 4;
    grid-template-rows:   subgrid;
    row-gap:              0.75rem;
}

/* Important: with auto-fill, the number of columns per row is dynamic */
/* At 600px wide: 2 cards per row → they share 4 row tracks in that group
   At 1200px wide: 4 cards per row → different group composition
   Row tracks are sized per row — cards in the same row share, not cards across rows
   This is correct behaviour: you want alignment within a visual row, not the whole grid

/* Limitation: you cannot guarantee that specific named cards appear in the same
   row — the auto-placement algorithm controls that. For curated layouts, use
   explicit grid-row placement alongside subgrid. */
```

7. Nested Subgrid

```
/* A subgrid element's children can themselves be subgrids */
/* Each level inherits from its parent's tracks */

.page {
    display:               grid;
    grid-template-columns: repeat(12, 1fr);  /* 12-column layout grid */
    column-gap:            1rem;
}

.article {
    grid-column:           1 / -1;    /* spans all 12 columns */
    display:               grid;
    grid-template-columns: subgrid;  /* inherits the 12-column grid */
}

.article__sidebar {
    grid-column:           1 / 4;   /* columns 1-3 */
    display:               grid;
    grid-template-columns: subgrid;  /* inherits 3 of the 12 columns */
}

.widget {
    grid-column:           1 / 3;   /* within the sidebar's 3 columns */
    display:               grid;
    grid-template-columns: subgrid;  /* inherits 2 of the sidebar's 3 columns */
    /* which are also 2 of the root 12 — all are the same tracks */
}

/* The entire nested tree participates in the root 12-column system */
/* Deeply nested content can be aligned to the root grid without any custom widths */
```

8. Implicit Tracks and Subgrid

```
/* Subgrid only covers EXPLICIT parent tracks that the item spans */
/* It cannot extend into implicit tracks created by auto-placement */

.card-grid {
    display:              grid;
    grid-template-columns: repeat(3, 1fr);
    grid-auto-rows:       auto;   /* implicit rows — subgrid can use these */
}

.card {
    display:            grid;
    grid-row:           span 4;   /* spans 4 implicit rows */
    grid-template-rows: subgrid;  /* inherits those 4 implicit rows */
    /* This works even though the rows are implicit — they're just auto rows */
}

/* What does NOT work: trying to subgrid more tracks than you span */
.card {
    grid-row:           span 4;
    grid-template-rows: subgrid;  /* inherits exactly 4 rows — the span */
    /* A 5th child would overflow into the card's block axis, not get a subgrid track */
}

/* Rule: span = number of subgrid tracks available */
/* Children beyond that number are placed in implicit tracks ON THE SUBGRID ELEMENT */
/* (not on the parent grid) — they break alignment */
/* Always match your span to the number of children you have */
```

9. Alignment within Subgrid

```
/* align-items / justify-items on the PARENT apply to all items including subgrid childr
.card-grid {
    align-items: start;  /* affects the card elements (grid items) */
}

/* align-items on the SUBGRID element applies to ITS children (subgrid items) */
.card {
    display:            grid;
    grid-row:           span 4;
    grid-template-rows: subgrid;
    align-items:        stretch;  /* all card sections stretch to fill their track */
}

/* Override per section */
.card__image {
    align-self:  stretch;
    object-fit:  cover;
}
.card__title {
    align-self:  end;     /* title sits at the bottom of its track */
}
.card__body {
    align-self:  start;   /* body text starts at top of its (expandable) track */
}
.card__cta {
    align-self:  end;     /* CTA button anchored to bottom of its track */
}

/* Writing the CTA pattern without subgrid for comparison */
/* Old pattern: display:flex flex-direction:column on the card + margin-top:auto on CTA
.card--flex {
    display:        flex;
    flex-direction: column;
}
.card--flex .card__cta {
    margin-top: auto;
    /* pushes CTA to bottom within its card — but still misaligned ACROSS cards */
}
/* Subgrid is the only pure-CSS way to align CTA buttons ACROSS sibling cards */
```

10. Real-World Patterns

Form layout with label–input pairs

```
/* Labels and inputs aligned across a two-column parent */
.form {
    display:               grid;
    grid-template-columns: repeat(2, 1fr);
    gap:                   1rem;
}

.form__group {
    display:               grid;
    grid-row:              span 2;    /* label + input = 2 tracks */
    grid-template-rows:    subgrid;
    row-gap:               4px;       /* tight gap between label and input */
}

/* All labels align to one row track, all inputs to the next */
/* regardless of label text length — no flexbox hacks needed */

.form__group label { align-self: end;     }   /* label sits at bottom of first track */
.form__group input { align-self: stretch; }   /* input fills second track */

/* Full-width field that spans both columns */
.form__group--full {
    grid-column:           1 / -1;
    display:               grid;
    grid-column-span:      subgrid;   /* spans both columns — subgrids them too */
    grid-row:              span 2;
    grid-template-rows:    subgrid;
}
```

12-column macro layout with nested sections

```
/* Classic editorial layout: sidebar + content sharing a column grid */
.page {
    display:               grid;
    grid-template-columns:
        [full-start]
        minmax(1rem, 1fr)
        [content-start]
        repeat(10, 1fr)
        [content-end]
        minmax(1rem, 1fr)
        [full-end];
}

.article {
    grid-column:           content-start / content-end;
    display:               grid;
    grid-template-columns: subgrid;
}

.article__body {
    grid-column:           1 / 8;   /* 7 of the 10 content columns */
}
.article__aside {
    grid-column:           8 / -1;  /* remaining 3 columns */
}

/* A pull-quote that bleeds into the margin */
.pull-quote {
    grid-column:           full-start / 5;  /* bleeds into the margin column */
}
```

Navigation with icon + label pairs

```
/* Sidebar nav: icons aligned in one column, labels in another */
.nav {
    display:               grid;
    grid-template-columns: 40px 1fr;  /* icon width | label */
    row-gap:               4px;
}

.nav__item {
    display:               grid;
    grid-column:           1 / -1;     /* spans both nav columns */
    grid-template-columns: subgrid;     /* icon | label from parent */
    align-items:           center;
    padding:               8px 12px;
    border-radius:         6px;
}

/* All icons align to the exact same 40px column regardless of icon size variation */
/* All labels start at the exact same column line */
/* Previously required explicit width:40px on every icon element */
```

11. Property Reference

Property / value	Where it goes	What it does
`grid-template-rows: subgrid`	On the grid item (subgrid container)	Row tracks are inherited from the parent grid. Number of tracks = grid-row span.
`grid-template-columns: subgrid`	On the grid item (subgrid container)	Column tracks are inherited from the parent grid. Number of tracks = grid-column span.
`grid-row: span N`	On the grid item	Defines how many parent row tracks this item occupies — and therefore how many tracks the subgrid inherits.
`row-gap / column-gap`	On the subgrid container	Overrides the parent's gap for that axis within this subgrid. Default is to inherit the parent's gap.
`align-items`	On the subgrid container	Controls block-axis alignment of the subgrid's own children within their inherited tracks.
`align-self`	On a subgrid child	Overrides alignment for one child within its inherited track — typically start, end, center, or stretch.
`display: contents`	On a grid item	Older workaround — collapses the element's box so its children become direct grid items. Has accessibility issues and doesn't support gap. Use

Property / value	Where it goes	What it does
		subgrid instead.

> **display: contents is not the same as subgrid.** `display: contents` removes the intermediate element from the box tree entirely — its children become grid items of the parent, but the element itself disappears, losing its background, padding, and border. It also breaks accessibility in some browsers by stripping the element from the accessibility tree. Subgrid keeps the intermediate element as a real box while letting its children align to the parent grid.

Chapter Summary

Concept	Key point
`What subgrid does`	A grid item with grid-template-rows/columns: subgrid lets its children participate in the parent grid's tracks, sharing row or column sizing with siblings.
`span = track count`	The number of subgrid tracks available equals the element's grid-row (or grid-column) span. Set the span before adding subgrid.
`Axis independence`	Subgrid can apply to rows, columns, or both. One axis can be subgrid while the other uses explicit tracks.
`Gap inheritance`	A subgrid inherits the parent's gap for its subgrid axis by default. Override with row-gap / column-gap on the subgrid element.
`Named lines`	Parent named lines are inherited by the subgrid. Additional names can be declared on the subgrid element using bracket notation alongside the subgrid keyword.
`auto-fill grids`	Subgrid works with auto-fill. Each row group shares track sizes across its members. Alignment is within a visual row, not across all rows.
`Nested subgrid`	A subgrid's children can themselves be subgrids. Each level inherits from its parent, which may itself be inheriting from a higher ancestor.
`Implicit tracks`	Subgrid can inherit implicit tracks (created by grid-auto-rows) as well as explicit ones. Children beyond the span don't get inherited tracks.
`vs display: contents`	display: contents was the old workaround — it collapses the element's box, removes it from accessibility tree, and loses padding/background. Subgrid is the correct solution.

EXERCISES

1. **Card grid with four tracks:** Create a three-column card grid where each card has exactly four sections: image, title, description, and a "Read more" link. Use `grid-template-rows: subgrid` and `grid-row: span 4` on each card. Populate the cards with different-length descriptions and confirm that all "Read more" links sit on the same horizontal baseline. Then remove subgrid and use `margin-top: auto` on the CTA instead — observe the difference when multiple cards appear in the same row.
2. **Gap override:** Set a `row-gap: 2rem` on the parent grid for large row spacing between card groups. Override the gap to `0.5rem` on the subgrid card element for tighter internal spacing. Confirm that the large gap appears between card rows but not between sections within a card. Then remove the override and describe what happens to the card's internal layout.
3. **12-column macro layout:** Build a page wrapper with a named 12-column grid (with named margin columns using `minmax`). Create an `.article` that spans the central columns using `grid-template-columns: subgrid`. Inside the article, place a body column (7 columns wide), an aside (3 columns), and a pull-quote that bleeds one column into the margin. Use only named grid lines for placement — no numeric column positions on the article's children.
4. **Form label alignment:** Build a two-column form where each field group contains a label and an input. Use `grid-row: span 2` and `grid-template-rows: subgrid` on each group so that all labels share one row track and all inputs share the next. Add a full-width textarea group that spans both columns. Confirm that the label baseline aligns horizontally across both columns.
5. **Nested subgrid:** Create a three-level subgrid: a 12-column root page grid, a section that subgrids it for 8 columns, and a nested component inside the section that further subgrids 4 of those 8 columns. Place content inside the innermost subgrid and confirm it aligns to the original root 12-column lines. Inspect with DevTools grid overlay to verify all three levels share the same track lines.

Next: Chapter 4 — The :has() Selector and Advanced Selector Patterns. The parent selector in depth — how :has() interacts with specificity, performance, pseudo-classes, and combinators. Advanced selector patterns: :is(), :where(), :not() with complex arguments, :nth-child(An+B of selector), and building quantity queries without JavaScript.

CHAPTER 4

The :has() Selector and Advanced Selector Patterns

Chapter 4 — :has() and Advanced Selector Patterns

The selector specification has gained substantial power in recent years. `:has()` — the so-called "parent selector" — arrived in all evergreen browsers in 2023 and removes an entire class of layout problems that previously required JavaScript or extra markup. Combined with forgiving selector lists (`:is()`, `:where()`), improved `:not()`, and filtered `:nth-child(An+B of selector)`, modern CSS can express relationships between elements that were previously impossible in a stylesheet.

> **Baseline 2023.** `:has()` is supported in all evergreen browsers. `:is()`, `:where()`, and `:not()` with complex arguments are Baseline 2021. `:nth-child(An+B of S)` is Baseline 2024. All are safe to use without fallbacks for modern-browser targets.

1. :has() — The Relational Pseudo-Class

`:has()` matches an element if any of the selectors in its argument match *relative to that element*. It does not select what is inside the parentheses — it selects the **anchor element** when the argument is satisfied.

What :has() selects

```
.card:has(.card__img)

Selected element → .card
Condition        → .card__img exists inside it

✓ <div class="card">
  ✓ <img class="card__img">
</div>
```

Relative selector argument

```
The argument is a relative selector list.
It's evaluated from the anchor's scope.

> .child   → direct child
+ .next    → adjacent sibling
~ .later   → subsequent sibling
.desc      → any descendant

A bare selector like .card__img
implies descendant (space combinator).
```

```css
/* ── :has() with every combinator ──────────────────────────── */

/* Descendant (default) — card contains an image anywhere inside */
.card:has(.card__image)      { grid-template-columns: 120px 1fr; }

/* Direct child — card's DIRECT child is a figure */
.card:has(> figure)          { padding-top: 0; }

/* Adjacent sibling — element immediately after an h2 */
h2:has(+ p)                  { margin-bottom: 0.25rem; }

/* Subsequent sibling — label that has a required input AFTER it */
label:has(~ input:required) { font-weight: 700; }

/* Multiple arguments (OR logic) */
.card:has(.image, video, canvas) {
    border-top: none;
}

/* Negating :has() — card WITHOUT an image */
.card:not(:has(.card__image)) {
    grid-template-columns: 1fr;
}

/* Chaining :has() — both conditions must be true (AND) */
.form:has(input:invalid):has(button[type="submit"]:hover) {
    outline: 2px solid #ff7b72;
}

/* :has() with pseudo-classes */
section:has(:focus-visible)  { outline: 2px solid var(--focus-ring); }
nav:has(a:hover)             { background: var(--nav-active-bg); }
.modal:has(input:checked)    { display: flex; }   /* checkbox-driven modal */
```

2. :has() Specificity

The specificity of a `:has()` selector is the specificity of the anchor element plus the specificity of the **most specific argument** in its list. The `:has()` pseudo-class itself contributes no specificity — only its contents do.

Selector	Specificity	How it's calculated
`.card:has(.image)`	`0-2-0`	0-1-0 (.card) + 0-1-0 (.image) — :has adds nothing

Selector	Specificity	How it's calculated
`.card:has(img)`	`0-1-1`	0-1-0 (.card) + 0-0-1 (img element)
`.card:has(#hero)`	`1-1-0`	0-1-0 (.card) + 1-0-0 (#hero ID)
`.card:has(.a, #b)`	`1-1-0`	Takes the highest specificity argument — #b wins at 1-0-0
`:has(.a, #b)`	`1-0-0`	No anchor specificity + highest argument: 1-0-0 (#b)

> **:has() with an ID argument is dangerous.** Because the argument's specificity contributes to the whole selector, putting an ID inside `:has()` gives the entire rule a 1-0-0 base specificity — the same penalty as writing an ID selector directly. Avoid ID selectors inside `:has()` arguments. Use custom attributes or data attributes instead.

```
/* :has() specificity in practice */

/* This rule has 0-2-0 specificity */
.card:has(.card__image) { display: grid; }

/* A plain .card rule with 0-1-0 can be overridden — the :has() rule wins */
.card { display: flex; }  /* loses to the :has() rule */

/* Equalise with :where() to zero-out the argument contribution */
.card:has(:where(.card__image)) {
    display: grid;
}
/* Now specificity is 0-1-0 — same as .card */
/* :where() inside :has() zeros out the argument's contribution */
```

3. Live :has() Demos

Layout switch based on content

CARD ADAPTS ITS LAYOUT BASED ON WHETHER AN IMAGE IS PRESENT

with image → two-column

Project Alpha

A card with an image present — :has(.card__img) switches to a two-column grid.

without image → single column

Project Beta

No image present — the card remains a single-column layout. No extra class needed.

Form field state without JavaScript

LABEL AND INPUT COLOUR DRIVEN BY :HAS() + :VALID/:INVALID

Email address

you@example.com

Username (letters only)

username

```
/* The CSS behind the form demo */

/* Parent (.hd-field) reacts to child state */
.field:has(input:invalid:not(:placeholder-shown)) {
    /* :not(:placeholder-shown) = user has typed something */
}

.field:has(input:invalid:not(:placeholder-shown)) label {
    color: #ff7b72;
}

.field:has(input:invalid:not(:placeholder-shown)) input {
    border-color: #ff7b72;
    outline:      2px solid #ff7b2240;
}

.field:has(input:invalid:not(:placeholder-shown)) .hint {
    display: block;   /* reveal hint message */
}

.field:has(input:valid) label {
    color: #3fb950;
}

/* No JavaScript. No extra classes. No event listeners. */
```

4. :has() Performance

CSS selectors are matched right-to-left. `:has()` breaks this model slightly — the browser must examine the anchor element and then look forward into its subtree to evaluate the argument. This is why `:has()` was historically considered "too expensive" to standardise. Modern implementations use invalidation tracking — when a descendant changes, the browser re-evaluates only the `:has()` rules that could be affected.

```
/* Performance guidance */

/* Fast — specific anchor, shallow argument */
.card:has(> .card__image) {}        /* direct child check — O(1) */
.card:has(.card__cta:hover) {}       /* bounded to .card's subtree */

/* Slower — broad anchor, deep/expensive argument */
body:has(input:focus) {}            /* any input on the page — full subtree scan */
html:has(.modal--open) {}           /* full document — commonly used, acceptable */

/* html:has() is the standard pattern for page-level state */
/* (scroll lock, dark mode toggle, modal open) */
/* Browsers optimise this case specifically — it's fine in practice */
html:has(.modal[open]) {
    overflow: hidden;    /* prevent body scroll while modal is open */
}

/* Prefer class-bounded anchors over document-level ones */
/* Keep :has() arguments shallow — direct child (>) when possible */
/* Avoid :has() on very high-churn elements (items in a list that updates often) */
```

5. :is() — Forgiving Selector Lists

Without `:is()`, a single invalid selector in a comma-separated list invalidates the entire rule. `:is()` uses a *forgiving* list — invalid selectors are silently ignored, and valid ones still apply. Its specificity equals the most specific selector in the list that actually matches.

```css
/* Without :is() — one bad selector kills the whole rule */
h1, h2, h3, h4:invalid-pseudo, h5, h6 {
    font-family: var(--font-heading);
    /* h4:invalid-pseudo causes the ENTIRE rule to be discarded */
}

/* With :is() — bad selector is ignored, rest apply */
:is(h1, h2, h3, h4:invalid-pseudo, h5, h6) {
    font-family: var(--font-heading);
    /* h4:invalid-pseudo is dropped; h1–h3, h5, h6 all apply */
}

/* :is() for DRY selector lists */
/* Before: */
article h2, article h3, article h4,
section h2, section h3, section h4 { color: var(--heading); }

/* After: */
:is(article, section) :is(h2, h3, h4) { color: var(--heading); }

/* :is() specificity — takes the highest matching argument */
:is(h1, .title, #hero-title) {}
/* On a plain h1:   specificity 0-0-1 (h1 wins as the matching argument) */
/* On .title:       specificity 0-1-0 (.title wins) */
/* On #hero-title:  specificity 1-0-0 (#hero-title wins) */
/* This can cause surprises — :where() is the zero-specificity alternative */

/* :is() in complex selectors */
.card :is(h2, h3)           { margin-top: 0; }
.nav :is(a, button):hover   { background: var(--hover-bg); }
:is(.dark, [data-theme="dark"]) .text { color: #c9d1d9; }
```

6. :where() — Zero-Specificity Forgiving Lists

`:where()` is identical to `:is()` in every way — same forgiving selector list, same matching behaviour — except that its specificity is always **zero**. Use it wherever you want the benefit of a grouped selector but need to ensure easy overrideability.

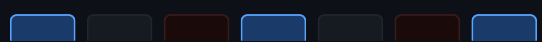
```css
/* :where() always contributes 0-0-0 specificity */
:where(h1, h2, h3, h4, h5, h6) {
    font-family: var(--font-heading);
    line-height: 1.2;
    /* Specificity: 0-0-0 — even a plain element selector can override this */
}

/* The primary use case: resets and base layers */
@layer reset {
    /* :where() in a reset means application code never needs !important to override */
    :where(button, input, select, textarea) {
        font-family: inherit;
        font-size:   inherit;
    }

    :where(ul, ol) {
        list-style: none;
        padding:    0;
    }
}

/* :where() for default component styles that consumers can easily override */
@layer base {
    /* Components use :where() so a plain class override always wins */
    :where(.card) {
        padding:       1rem;
        border-radius: 8px;
        background:    var(--surface);
    }
}

@layer overrides {
    /* 0-1-0 — easily overrides the 0-0-0 :where() base style */
    .card { border-radius: 0; }
}

/* :where() inside :has() to zero out argument contribution */
.card:has(:where(.card__image)) {
    /* specificity: 0-1-0 — same as plain .card */
    display: grid;
}
```

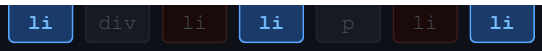
7. :not() with Complex Arguments

```
/* Old :not() — only one simple selector allowed */
a:not(.external)              {}   /* pre-Selectors Level 4 — supported everywhere */

/* Modern :not() — full forgiving selector list */
a:not(.external, .button, [download]) {}  /* multiple exclusions */
p:not(:first-child)           {}   /* pseudo-class argument */
li:not(:last-child)           { border-bottom: 1px solid var(--border); }

/* :not() specificity — same rule as :is() */
/* Takes the highest specificity of all arguments, even if they don't match */
a:not(.external, #home) {}
/* Specificity: 0-0-1 (a) + 1-0-0 (#home — highest argument) = 1-0-1 */
/* Even if #home never matches, it still inflates specificity */
/* This is the same gotcha as :is() — avoid IDs in :not() arguments */

/* :not() with compound selectors */
.nav__item:not(.nav__item--active, .nav__item--disabled) {
    opacity: 0.75;
}

/* :not() excluding multiple element types */
:not(h1, h2, h3, h4, h5, h6, p, li, td) {
    display: block;
}

/* Practical — style all links that are not in nav or footer */
a:not(:is(nav *, footer *)) {
    text-decoration: underline;
    color:           var(--link);
}

/* :not() cannot take a combinator argument (no :not(a > b)) */
/* and cannot be nested (:not(:not(...)) is meaningless) */
```

8. :nth-child(An+B of selector)

Classic `:nth-child()` counts all siblings of any type. The `of S` extension lets you count only elements that match a selector — it's a filtered nth-child. Baseline 2024.





:NTH-CHILD(ODD OF LI) — COUNTS ONLY LI SIBLINGS

Blue = matched. The `of li` version only counts li elements in the sequence — the div and p are invisible to the counter.

```
/* Classic — counts EVERY child at that position */
li:nth-child(odd)   { background: var(--stripe); }
/* Problem: if there's a heading or div mixed in, counting goes wrong */

/* Filtered — counts only li children */
li:nth-child(odd of li) { background: var(--stripe); }
/* The counter increments only when the sibling matches 'li' */

/* :nth-child(of) with class selectors */
:nth-child(1 of .featured)  { grid-column: 1 / -1; }   /* first featured item only */
:nth-child(-n+3 of .card)   { border-top: 3px solid var(--accent); }

/* Quantity queries — style when AT LEAST N items are present */

/* Select all items when there are at least 4 */
.item:nth-child(n+4):nth-child(n+4) ~ .item,
.item:nth-child(4) {
    /* :nth-child(n+4) selects item 4+ */
    /* combined with ~ selects later siblings */
}

/* Modern quantity query: "4+ items → grid-template-columns changes" */
/* Use :has() instead — much cleaner */
.grid:has(.item:nth-child(4))  { grid-template-columns: repeat(4, 1fr); }
.grid:has(.item:nth-child(3))  { grid-template-columns: repeat(3, 1fr); }
.grid:has(.item:nth-child(2))  { grid-template-columns: repeat(2, 1fr); }

/* The 4-column rule must come LAST (source order) so it wins when 4+ items exist */
/* because .grid:has(.item:nth-child(3)) is also true when there are 4+ items */
```

9. Quantity Queries — Layout by Item Count

```css
/* ── Count-based grid using :has() ──────────────────────────── */

/* Grid switches layout based on how many .item children it has */
/* Must be written in ascending order (smallest last because specificity ties → source o

.grid {
    display: grid;
    grid-template-columns: 1fr;       /* default: 1 column */
}

.grid:has(.item:nth-child(2))  { grid-template-columns: repeat(2, 1fr); }
.grid:has(.item:nth-child(3))  { grid-template-columns: repeat(3, 1fr); }
.grid:has(.item:nth-child(4))  { grid-template-columns: repeat(4, 1fr); }

/* Why the order matters:
   When 4 items exist, :has(.item:nth-child(2)) is also true (there IS a 2nd item).
   So all three rules match. They have identical specificity (0-3-0 each).
   Source order decides → the last rule wins → 4 columns. Correct. */

/* ── "Exactly N items" query ────────────────────────────────── */

/* Exactly 3 items: has a 3rd child but NOT a 4th */
.grid:has(.item:nth-child(3)):not(:has(.item:nth-child(4))) {
    grid-template-columns: repeat(3, 1fr);
}

/* ── Previous sibling selection ─────────────────────────────── */

/* CSS has no "previous sibling" combinator, but :has() creates one */
/* Style an h2 that is IMMEDIATELY BEFORE a figure */
h2:has(+ figure) { margin-bottom: 0; }

/* Style a label that comes BEFORE an invalid input (ancestor + sibling trick) */
.field:has(input:invalid) label { color: var(--error); }
/* The label can be BEFORE the input in the DOM — :has() checks the field container */

/* Count-based feature: spotlight first article in a group of 1 */
article:only-of-type                    { grid-column: 1 / -1; }
.grid:not(:has(article:nth-of-type(2))) article {
    grid-column: 1 / -1;    /* single article spans full width */
}
```

10. Combining the Selectors — Real Patterns

```
/* ── Pattern: page-level state via html:has() ──────────────────── */

/* Disable scroll when a dialog is open */
html:has(dialog[open])          { overflow: hidden; }

/* CSS-only sidebar toggle via checkbox */
html:has(#sidebar-toggle:checked) .sidebar {
    transform: translateX(0);
}

/* Dark mode via a toggle button */
html:has(#theme-toggle:checked) {
    color-scheme: dark;
    --bg:          #0d1117;
    --text:        #c9d1d9;
}

/* ── Pattern: type-adaptive prose ──────────────────────────────── */

/* Give a heading a reduced top margin only when preceded by another heading */
:is(h2, h3, h4):has(+ :is(h2, h3, h4)) {
    margin-bottom: 0.25rem;
}

/* Remove top margin from the first child of an article */
article > :first-child:is(h1, h2, p) {
    margin-top: 0;
}

/* ── Pattern: grid counts with :nth-child(of) ──────────────────── */

/* Highlight every other featured card (ignoring non-featured siblings) */
.card:nth-child(even of .card--featured) {
    background: var(--surface-alt);
}

/* First three of a specific type */
.card:nth-child(-n+3 of .card--new) {
    border: 2px solid var(--accent);
}

/* ── Pattern: card layout driven entirely by content ──────────── */

.card                           { display: flex; flex-direction: column;  }
```

```css
.card:has(.card__image)       { display: grid; grid-template-columns: 120px 1fr; }
.card:has(video)              { display: grid; grid-template-rows:    auto 1fr auto; }
.card:has(.card__image):has(video) {
    grid-template-columns: 1fr;
    grid-template-rows:    auto auto 1fr auto;
}

/* No conditional classes. No JavaScript. The CSS reads the content itself. */
```

Chapter Summary

Selector	Key behaviour
`:has(arg)`	Matches the anchor element when the argument matches something relative to it. Supports all combinators in the argument. Specificity = anchor + highest argument. Forgiving selector list.
`:has() chaining`	Multiple :has() on one selector = AND logic. Multiple arguments inside one :has() = OR logic. Use :not(:has()) to negate.
`html:has()`	Standard pattern for page-level state: modal open, scroll lock, sidebar toggle, theme. Browsers optimise this case. Keep the argument specific.
`:is(args)`	Forgiving selector list — invalid selectors silently dropped. Specificity = highest matching argument. Used for DRY selector groups.
`:where(args)`	Identical to :is() but specificity is always 0-0-0. Use in resets, base layers, and default component styles to guarantee easy overrideability.
`:not(args)`	Now accepts a full forgiving selector list. Specificity = highest argument (even non-matching ones). Avoid IDs in :not() arguments.
`:nth-child(of S)`	Counts only siblings matching S — ignores unmatched siblings. Use for accurate striping in mixed-content lists. Baseline 2024.
`Quantity queries`	:has(.item:nth-child(N)) on the parent changes layout based on how many children exist. Write rules smallest-first so source order breaks ties correctly.
`Previous sibling`	CSS has no "previous sibling" combinator, but el:has(+ .next) selects el when its adjacent sibling is .next — effectively a previous-sibling selector.

EXERCISES

1. **Content-adaptive card:** Build a single card component with no JavaScript. Add or remove a child `.card__image` element and have the card switch between a stacked single-column layout and a two-column image+content layout automatically using `:has()`. Then add a `video` element and make the card switch to a third layout. Use `:not(:has())` to ensure the text-only variant still has explicit styles. Calculate the specificity of each rule and verify no rule unintentionally overrides another.
2. **Form field states:** Create a form with at least four fields of different types (email, text with pattern, number with min/max, required checkbox). Using only `:has()` and form pseudo-classes (`:valid`, `:invalid`, `:checked`, `:placeholder-shown`), make each field's label, input border, and a hint message below each field all respond to the field's validity state. No JavaScript, no classes added by JS. As a bonus, make the submit button's background change only when all required fields are valid.
3. **Quantity-responsive grid:** Create a `.gallery` container with `.photo` children. Write `:has()` quantity queries so the grid uses 1 column for 1 photo, 2 columns for 2 photos, 3 columns for 3 photos, and a masonry-style layout (using `grid-template-rows`) for 4 or more. Write the rules in the correct source order to ensure specificity ties resolve correctly. Add a 5th photo and verify the grid changes.
4. **:is() and :where() architecture:** Write a stylesheet reset using only `:where()` so that every rule has zero specificity. Then write component styles using plain class selectors and verify they always override the reset without needing `!important`. Next, rewrite the same reset using `:is()` instead and document (with comments) which classes now unexpectedly fail to override the reset, and why.
5. **:nth-child(of) striping:** Create a `<ul>` containing a mix of `<li>` items and `<li class="divider">` separator items. Using `:nth-child(odd)`, apply alternating row shading. Observe that the dividers break the count. Then switch to `:nth-child(odd of li:not(.divider))` and confirm the striping is correct regardless of how many dividers are inserted, and that dividers themselves are never shaded.

Next: Chapter 5 — Scroll-Driven Animations. The animation-timeline property in depth — scroll() timelines anchored to a scroller, view() timelines anchored to element visibility, animation-range, named timelines, subject vs scroller distinction, and compositing scroll-driven animations safely without jank.

CHAPTER 5

Scroll-Driven Animations

Chapter 5 — Scroll-Driven Animations

Scroll-driven animations let you tie CSS animation playback to scroll position rather than time. As the user scrolls, the animation advances — no JavaScript event listeners, no `requestAnimationFrame`, no scroll handler debouncing. Because the browser handles playback on the compositor thread, these animations can run at 60/120 fps without touching the main thread at all, provided you animate only composited properties (`transform` and `opacity`).

The blue–purple progress bar at the top of this page is a live scroll-driven animation — it is driven by `animation-timeline: scroll(root block)` and has exactly zero lines of JavaScript.

> **Browser support.** Scroll-driven animations (the `animation-timeline` / `scroll()` / `view()` API) shipped in Chrome/Edge 115 (July 2023) and Safari 18 (September 2024). Firefox support landed in Firefox 132 (October 2024). All evergreen browsers now support the full API. Use `@supports (animation-timeline: scroll())` to gate enhanced experiences for older browsers.

1. The Two Timeline Types

scroll() — scroll progress timeline

What it tracks

The scroll position of a scroll container — how far from the top (or left) to the bottom (or right) the scroller has travelled.

Progress: 0% → 100%

0% = scroller at start position. 100% = scroller at end position.

Typical use

Reading progress bars, sticky header effects, parallax that tracks the whole page scroll, any effect that should be proportional to total scroll distance.

```
animation-timeline: scroll()
```

view() — view progress timeline

What it tracks

The visibility of an element (the *subject*) within a scroll container — how far the subject has entered, traversed, and left the visible area.

Progress: 0% → 100%

0% = subject just about to enter. 100% = subject just finished exiting. Controlled by `animation-range`.

Typical use

Reveal-on-scroll, entrance animations, exit animations, elements that animate while in view.

```
animation-timeline: view()
```

2. scroll() — Syntax and Arguments

```
scroll(nearest | root | self | element-name  block | inline | x | y)
```

scroller argument — which scroll container to track axis argument — which scroll axis (default: block)

```
/* scroll() scroller values */
animation-timeline: scroll();          /* nearest — closest scrollable ancestor */
animation-timeline: scroll(nearest);   /* same as above — explicit */
animation-timeline: scroll(root);      /* the document's root scroller (the page) */
animation-timeline: scroll(self);      /* the element itself must be a scroller */
animation-timeline: scroll(--my-scroller);  /* named scroll timeline (see Section 5) */

/* scroll() axis values */
animation-timeline: scroll(root block);   /* vertical scroll (default axis) */
animation-timeline: scroll(root inline);  /* horizontal scroll */
animation-timeline: scroll(root x);       /* physical horizontal */
animation-timeline: scroll(root y);       /* physical vertical */

/* The reading progress bar pattern */
.progress-bar {
    position:         fixed;
    top:              0;
    left:             0;
    width:            100%;
    height:           3px;
    background:       var(--accent);
    transform-origin: left;         /* scale from the left edge */
    animation:        grow-bar linear both;
    animation-timeline: scroll(root block);
}

@keyframes grow-bar {
    from { transform: scaleX(0); }
    to   { transform: scaleX(1); }
}

/* animation-fill-mode: both — ensures the bar starts at 0 and stays at 100% */
/* animation-duration is NOT needed for scroll-driven — the timeline replaces time */
/* But you CAN still set a duration if you want to limit the range (advanced) */
```

3. view() — Syntax and the Subject/Scroller Distinction

```
view(block | inline | x | y    inset)
```

- axis — which axis to track visibility on (default: block)
- inset — shrink or expand the "visible" zone (like margin on the viewport)

```
/* The subject is the element the animation is applied TO */
/* The scroller is the nearest scrollable ancestor of the subject */
/* view() creates a timeline based on the subject's visibility in the scroller */

.card {
    animation:          fade-in linear both;
    animation-timeline: view();  /* subject = .card, scroller = nearest ancestor */
}

/* view() axis — which axis of visibility to track */
animation-timeline: view(block);   /* vertical (default) */
animation-timeline: view(inline);  /* horizontal */

/* view() inset — shrinks the "visible" zone */
/* Positive inset = zone is smaller than the scroller (triggers earlier) */
/* Negative inset = zone is larger (triggers later) */
animation-timeline: view(block 100px);           /* 100px inset all sides */
animation-timeline: view(block 100px 0px);       /* 100px start inset, 0 end inset */
animation-timeline: view(block auto);             /* auto = use scroll-padding */

/* IMPORTANT: the element must NOT have overflow:hidden on its scroller ancestor */
/* because overflow:hidden blocks the scroll container detection */
/* Use overflow:clip instead if you need to clip without a scroll container */
```

Subject vs Scroller — the critical distinction

```
/* The subject is the animated element itself */
/* The scroller is found automatically (nearest scrollable ancestor) */

/* Scenario: animating a card in a scrollable list */
.scroller {
    height:      400px;
    overflow-y:  scroll;  /* this is now the view() scroller */
}

.card {
    animation:          reveal linear both;
    animation-timeline: view();
    /* subject = .card, scroller = .scroller (nearest scrollable ancestor) */
    /* animation tracks .card's visibility WITHIN .scroller, not within the page */
}

/* Scenario: page-level reveal (subject is the card, scroller is the page) */
.card {
    animation:          reveal linear both;
    animation-timeline: view();
    /* If no ancestor has overflow:scroll/auto, uses the page scroller */
    /* Most reveal-on-scroll patterns fall into this case */
}

/* What view() CANNOT do: animate based on a DIFFERENT element's visibility */
/* To do that, use a named view timeline (Section 5) */
```

4. animation-range and Range Keywords

`animation-range` controls which portion of a view timeline maps to the animation's 0%–100%. Without it, the full view timeline (cover) is used — the animation runs across the entire time the element is in any part of the scroll viewport.

exit — leaving the viewport

```css
/* animation-range keywords */

/* cover: full timeline — entering to exiting (default) */
animation-range: cover;
animation-range: cover 0% cover 100%;   /* same, explicit */

/* entry: element entering the viewport (bottom edge to fully visible) */
animation-range: entry;                  /* entire entry phase */
animation-range: entry 0% entry 50%;    /* first half of entry */

/* exit: element leaving the viewport */
animation-range: exit;
animation-range: exit 0% exit 100%;

/* contain: element fully inside the viewport */
animation-range: contain;

/* Mixing keywords and percentages */
animation-range: entry 0% entry 40%;
/* Animation runs during the first 40% of the entry phase */
/* After that, animation-fill-mode: both keeps it at the 'to' state */

/* Longhand versions */
animation-range-start: entry 0%;
animation-range-end:   entry 40%;

/* Percentage-only range (relative to the full cover timeline) */
animation-range: 20% 80%;

/* The reveal-on-scroll pattern */
.reveal {
    animation:          fade-up linear both;
    animation-timeline: view();
    animation-range:    entry 0% entry 35%;
    /* completes the animation while the element is entering — stays visible after */
}

@keyframes fade-up {
    from { opacity: 0; transform: translateY(30px); }
    to   { opacity: 1; transform: none;              }
}
```

5. Named Timelines

Named timelines let one element's scroll or view progress drive another element's animation — breaking the default rule that the animated element must be a descendant of the scroller or subject.

```
/* ── Named scroll timeline ──────────────────────────────── */

/* Create a named scroll timeline on the scroller */
.my-scroller {
    overflow-y:           scroll;
    scroll-timeline-name: --my-list;     /* custom property name */
    scroll-timeline-axis: block;         /* which axis */
    /* shorthand: scroll-timeline: --my-list block; */
}

/* Reference it from any element in the same subtree or a descendant */
.sidebar-indicator {
    animation:          track-progress linear both;
    animation-timeline: --my-list;  /* no scroll() wrapper — just the name */
}

/* ── Named view timeline ────────────────────────────────── */

/* Create a named view timeline on the SUBJECT element */
.hero-section {
    view-timeline-name: --hero;
    view-timeline-axis: block;
    /* shorthand: view-timeline: --hero block; */
}

/* Animate a SIBLING element based on the hero section's visibility */
.hero-caption {
    animation:          caption-fade linear both;
    animation-timeline: --hero;
    animation-range:    contain 0% contain 100%;
    /* .hero-caption animates based on .hero-section's visibility, not its own */
}

/* Named timelines are scoped to the timeline-scope property */
/* By default a named timeline is visible to all descendants */
/* timeline-scope: --hero makes it available to elements OUTSIDE the subject */
.page-layout {
    timeline-scope: --hero;   /* allows siblings of .hero-section to use --hero */
}
```

6. Live Demos

Reveal on scroll — view() with animation-range: entry

CARDS ANIMATE IN AS THEY ENTER THE VIEWPORT — SCROLL TO TRIGGER

Parallax — view() with animation-range: cover

BACKGROUND LAYER MOVES SLOWER THAN THE SCROLL — CLASSIC PARALLAX

Parallax background — view() cover range — translateY(-20px) → translateY(20px)

Staggered reveal — view() with animation-delay

LIST ITEMS REVEAL IN SEQUENCE USING CSS CUSTOM-PROPERTY STAGGER

```css
/* ── Stagger with CSS custom property index ──────────────────── */
.list-item {
    animation:          slide-in linear both;
    animation-timeline: view();
    animation-range:    entry 0% entry 60%;
    animation-delay:    calc(var(--index, 0) * 60ms);
}

/* Set --index in HTML or with :nth-child */
.list-item:nth-child(1) { --index: 0; }
.list-item:nth-child(2) { --index: 1; }
.list-item:nth-child(3) { --index: 2; }
/* Or set style="--index: N" in your template */
```

7. Compositing — Which Properties Are Safe

Scroll-driven animations run on the compositor thread only if the animated properties are **composited**. Composited properties skip layout and paint entirely — the GPU handles them. Any non-composited property forces the animation back to the main thread, where it can stutter under scroll load.

Property	Composited	Why / consequence
`transform`	**Yes**	GPU layer — translate, scale, rotate, skew all safe. The primary composited property.
`opacity`	**Yes**	GPU layer — completely safe. Combine with transform for most reveal patterns.
`filter`	**Mostly**	blur(), brightness() etc. are composited in Chrome/Edge. May paint in Firefox.
`clip-path`	**Mostly**	Simple shapes composited in modern browsers. Complex paths may repaint.
`width / height`	**No**	Triggers layout → paint → composite. Will jank under scroll. Use transform: scaleX/scaleY instead.
`background-color`	**No**	Triggers repaint every frame. Use opacity on a pseudo-element overlay instead.
`color / font-size`	**No**	Layout and paint on every frame. Only use if the effect requires it and performance is acceptable.
`left / top / margin`	**No**	Triggers layout. Use transform: translate() for positioning effects.

```css
/* ── will-change with scroll-driven animations ──────────────────── */

/* For elements that use view() and enter/exit frequently */
.reveal-card {
    will-change:        transform, opacity;
    /* Pre-promotes the element to its own GPU layer */
    /* Only add when you have measured a benefit — excess GPU layers waste VRAM */
}

/* For the progress bar (always animating while page scrolls) */
.progress-bar {
    will-change:        transform;   /* reasonable — it's always in motion */
}

/* DO NOT do this — it defeats compositing for the scroll-driven animation */
.card {
    animation:          reveal linear both;
    animation-timeline: view();
}
@keyframes reveal {
    from { height: 0; opacity: 0; }   /* height is non-composited → main thread stutter
    to   { height: auto; opacity: 1; }
}

/* The composited equivalent */
@keyframes reveal {
    from { transform: scaleY(0); opacity: 0; }
    to   { transform: none;         opacity: 1; }
}
```

8. Production Patterns

```css
/* ── Sticky header that fades in a background on scroll ──────── */
.site-header {
    position: sticky;
    top:      0;
    animation:            header-bg linear both;
    animation-timeline:   scroll(root block);
    animation-range:      0px 120px;  /* first 120px of scroll */
}

@keyframes header-bg {
    from { background: transparent;         backdrop-filter: blur(0px);  }
    to   { background: rgba(13,17,23,0.85); backdrop-filter: blur(12px); }
}

/* ── Horizontal scroll gallery ──────────────────────────────── */
.gallery {
    display:    flex;
    overflow-x: scroll;
    scroll-snap-type: x mandatory;
    scroll-timeline:  --gallery inline;
}

.gallery__indicator {
    /* Dot that fills as the gallery scrolls right */
    animation:          fill-dot linear both;
    animation-timeline: --gallery;
}

/* ── Parallax hero ──────────────────────────────────────────── */
.hero {
    view-timeline:  --hero block;
    overflow:       hidden;
}

.hero__bg {
    position:           absolute;
    inset:             -15% 0;   /* extra height for parallax room */
    animation:          parallax linear both;
    animation-timeline: --hero;
    animation-range:    cover;
}

@keyframes parallax {
    from { transform: translateY(-15%); }
```

```
    to   { transform: translateY(15%);  }
}

/* ── prefers-reduced-motion reset ──────────────────────────── */
@media (prefers-reduced-motion: reduce) {
    /* Remove all scroll-driven animations cleanly */
    *, *::before, *::after {
        animation-timeline:      auto !important;   /* fall back to time-based */
        animation-duration:       0.01ms !important; /* instant — preserves fill-mode */
        animation-iteration-count: 1 !important;
    }
}

/* ── Feature detection ─────────────────────────────────────── */
@supports (animation-timeline: scroll()) {
    .progress-bar {
        animation-timeline: scroll(root block);
    }
}
/* Without @supports, provide a static fallback (no bar, or JS-based bar) */
```

9. animation-timeline Interaction with Other Animation Properties

```
/* animation-duration: does it matter? */
/* For scroll-driven animations, duration is IGNORED for playback rate */
/* The timeline (scroll position) drives playback, not time */
/* But duration IS used if you also set animation-delay in time units */
/* Best practice: omit or set to 'auto' for scroll-driven animations */
animation-duration: auto;   /* explicit no-time — clearer than omitting */

/* animation-fill-mode: both is almost always what you want */
/* 'both' = apply 'from' state before the range starts, 'to' state after it ends */
/* Without 'both', elements snap to unstyled state before/after their range */
animation-fill-mode: both;   /* nearly always required */

/* animation-play-state — works with scroll-driven */
/* pause it programmatically: element.style.animationPlayState = 'paused' */
/* Note: pausing freezes the animation at current scroll position */

/* animation-iteration-count: only '1' makes sense for scroll-driven */
/* 'infinite' would restart the animation each time progress hits 100% */
/* which never happens during scroll — so infinite loops don't apply here */

/* animation-direction: alternate DOES work interestingly with scroll: */
/* as the user scrolls down, animation plays forward;
   as they scroll up, it reverses — naturally undoing the effect */
/* This is the default behaviour of all scroll-driven animations
   (scroll position drives progress in both directions) */

/* Combining time-based and scroll-driven on the same element */
.card {
    /* Time-based entrance (runs once on page load) */
    animation: fade-in 0.4s ease both,
               reveal 1s linear both;

    /* Only the second animation uses a scroll timeline */
    animation-timeline: auto, view();
    animation-range:    normal, entry 0% entry 40%;
}
```

Chapter Summary

Concept	Key point
`scroll()`	Tracks a scroll container's position. Arguments: scroller (nearest/root/self/name) and axis (block/inline/x/y). 0% = start, 100% = end of scroll range.
`view()`	Tracks subject visibility in a scroller. Arguments: axis and inset. Subject = the animated element. Scroller = nearest scrollable ancestor. Named timelines break this constraint.
`animation-range`	Maps a portion of the timeline to the animation's 0–100%. Keywords: cover (default), entry, exit, contain. Mix with percentages: entry 0% entry 40%.
`Named timelines`	scroll-timeline-name / view-timeline-name create named timelines on scroller/subject. Reference by name on any element. timeline-scope expands visibility to non-descendants.
`animation-fill-mode`	Always set both for scroll-driven animations. Without it, elements snap to unstyled state before/after their animation range.
`Compositing`	Only transform and opacity are guaranteed composited (GPU thread). Animating width, height, color, or background on a scroll timeline causes main-thread jank.
`Reverse on scroll-up`	Scroll-driven animations automatically reverse when the user scrolls back. This is often desirable — plan animation-range so the reversed state looks correct.
`prefers-reduced-motion`	Use @media (prefers-reduced-motion: reduce) to reset animation-timeline: auto and collapse duration to 0.01ms. This jumps to the final keyframe state rather than hiding content.
`Feature detection`	@supports (animation-timeline: scroll()) gates the enhanced experience. Provide a static fallback for browsers without support.

EXERCISES

1. **Reading progress bar:** Add a fixed reading progress bar to a long-form page using `scroll(root block)`. Use `transform: scaleX()` (not `width`) animated from 0 to 1 with `transform-origin: left`. Add a gradient fill and verify the bar correctly reaches 100% only at the very bottom of the page content, not the viewport bottom. Then add a `prefers-reduced-motion` block that hides the bar entirely (opacity: 0 on the bar rather than removing the animation).
2. **Reveal-on-scroll grid:** Create a grid of at least 6 cards. Apply a `view()` timeline with `animation-range: entry 0% entry 35%` to each card for a fade-up entrance. Add a stagger using `:nth-child()` selectors to set `--index` on each card and use `calc(var(--index) * 80ms)` for `animation-delay`. Verify that scrolling back up plays the reveal in reverse. Then add a `prefers-reduced-motion` reset and confirm cards are visible without any animation.
3. **Named view timeline — cross-element animation:** Create a `.hero` section that sets `view-timeline: --hero block`. Create a separate `.hero-badge` element that is a sibling (not a descendant) of `.hero`. Apply `timeline-scope: --hero` on their common ancestor. Animate the badge's opacity and transform based on `--hero`'s visibility — it should fade out as the hero leaves the viewport. Confirm it does not use `view()` directly (which would track the badge's own visibility).
4. **Horizontal scroll indicator:** Build a horizontally scrolling card carousel with `scroll-snap-type: x mandatory`. Create a named scroll timeline on the carousel with `scroll-timeline: --carousel inline`. Drive a set of indicator dots below the carousel: animate each dot's scale and opacity using `--carousel` at different `animation-range` percentage slices so each dot is active when its corresponding card is visible. No JavaScript.
5. **Performance audit:** Build a scroll-driven animation that deliberately animates a non-composited property (e.g. `background-color` or `font-size`). Open Chrome DevTools → Performance panel, record while scrolling, and identify the paint events triggered by each scroll frame. Then refactor to use only `transform` and `opacity` — record again and confirm paint events disappear. Document the before/after frame time difference.

Next: Chapter 6 — Logical Properties and Writing Modes. Physical vs logical property model — how inline-size, block-size, margin-inline, padding-block, border-start, and inset replace their physical counterparts. Writing modes (horizontal-tb, vertical-rl, vertical-lr, sideways-rl), text directionality, and building genuinely internationalised layouts without per-language CSS overrides.

CHAPTER 6

Logical Properties and Writing Modes

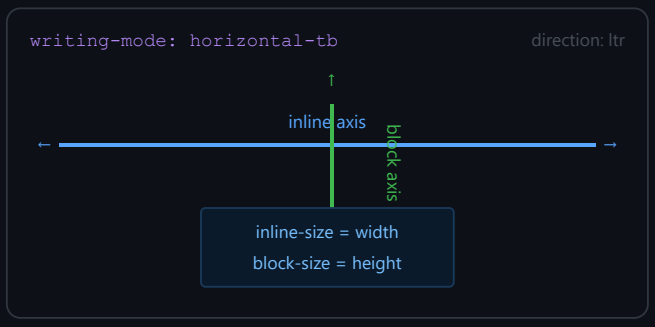
Chapter 6 — Logical Properties and Writing Modes

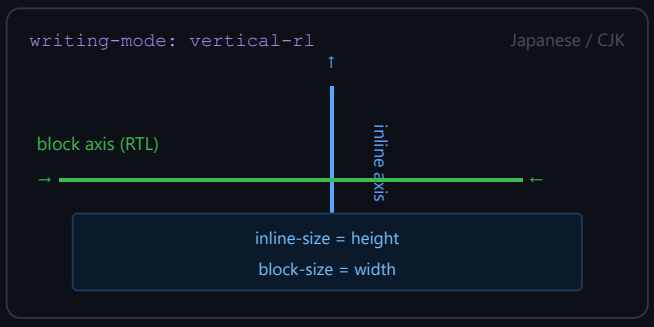
Physical CSS properties — `width`, `margin-left`, `border-top`, `top` — assume horizontal text flowing left to right. They bake a specific writing system into the stylesheet. Logical properties replace physical directions with **flow-relative** references: *inline* (the axis text runs along) and *block* (the axis paragraphs stack along). A layout written with logical properties works correctly in Arabic (right to left), Japanese vertical text (top to bottom), and English — without a single overriding rule.

> **Baseline 2023.** All logical properties and values covered in this chapter are supported in every evergreen browser. Writing modes have been supported since 2016. `sideways-rl` and `sideways-lr` are supported everywhere except Safari 17 and below — use `vertical-rl` with `rotate` as a fallback where needed.

1. The Inline and Block Axes

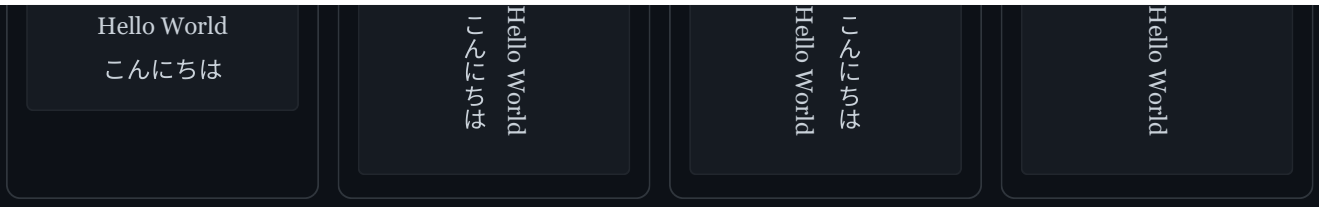
Every writing mode defines two axes. **Inline** is the direction text flows within a line. **Block** is the direction lines stack. The mapping of these axes to physical directions depends on the active writing mode and text direction.





The key insight: `inline-size` means "the size along the inline axis." In horizontal writing that is the width. In vertical writing that is the height. Write `inline-size: 300px` once and it works correctly in both.

2. Writing Modes




```
/* writing-mode values */

writing-mode: horizontal-tb;
/* Default. Text flows left→right (or right→left), lines stack top→bottom. */
/* Used by: Latin, Arabic, Hebrew, Greek, Cyrillic, Thai, Devanagari, etc. */

writing-mode: vertical-rl;
/* Text flows top→bottom. Lines (columns) stack right→left. */
/* Used by: Traditional Chinese, Japanese, Korean (tategumi / vertical layout). */
/* The 'rl' means new columns appear to the LEFT of earlier columns. */

writing-mode: vertical-lr;
/* Text flows top→bottom. Lines stack left→right. */
/* Used by: Mongolian script (traditional). */

writing-mode: sideways-rl;
/* Entire glyphs are rotated 90° clockwise. Used for rotated Latin labels. */
/* Unlike vertical-rl, letters themselves are rotated — not individual glyphs. */

writing-mode: sideways-lr;
/* Entire glyphs rotated 90° counter-clockwise. */

/* Axis mapping per writing mode */
/*                 inline-axis      block-axis */
/* horizontal-tb:  horizontal  →    vertical ↓  */
/* vertical-rl:    vertical    ↓    horizontal ← */
/* vertical-lr:    vertical    ↓    horizontal → */
/* sideways-rl:    vertical    ↓    horizontal ← */
```

text-orientation

```
/* text-orientation — applies only in vertical writing modes */
/* Controls how individual glyphs are oriented within a vertical line */

text-orientation: mixed;
/* Default. CJK characters stand upright; Latin/digits are rotated 90° */

text-orientation: upright;
/* All characters (including Latin) stand upright */

text-orientation: sideways;
/* All characters are rotated 90° as a group (same as sideways-rl) */

/* Practical use: chapter numbers in a Japanese sidebar */
.chapter-number {
    writing-mode:     vertical-rl;
    text-orientation: upright;   /* digits stand upright in the vertical line */
}
```

3. Physical → Logical Property Mapping

Physical property	Logical equivalent	Notes
SIZING		
`width`	`inline-size`	Size along the inline axis
`height`	`block-size`	Size along the block axis
`min-width / max-width`	`min-inline-size / max-inline-size`	
`min-height / max-height`	`min-block-size / max-block-size`	
MARGIN		
`margin-top`	`margin-block-start`	
`margin-bottom`	`margin-block-end`	
`margin-left`	`margin-inline-start`	RTL: maps to right
`margin-right`	`margin-inline-end`	RTL: maps to left
`margin: top right bottom left`	`margin-block: start end; margin-inline: start end`	Two separate shorthands
PADDING		

Physical property	Logical equivalent	Notes
padding-top	padding-block-start	
padding-bottom	padding-block-end	
padding-left	padding-inline-start	
padding-right	padding-inline-end	
BORDER		
border-top	border-block-start	Shorthand: width style color
border-bottom	border-block-end	
border-left	border-inline-start	The accent border pattern
border-right	border-inline-end	
border-top-width	border-block-start-width	Longhand for width/style/color
POSITION / INSET		
top	inset-block-start	
bottom	inset-block-end	
left	inset-inline-start	
right	inset-inline-end	
top:0; right:0; bottom:0; left:0	inset: 0	inset is the shorthand for all four
BORDER RADIUS		
border-top-left-radius	border-start-start-radius	block-start + inline-start corner
border-top-right-radius	border-start-end-radius	block-start + inline-end corner
border-bottom-left-radius	border-end-start-radius	block-end + inline-start corner
border-bottom-right-radius	border-end-end-radius	block-end + inline-end corner
TEXT AND FLOAT		
text-align: left	text-align: start	start/end are logical values for text-align
text-align: right	text-align: end	
float: left	float: inline-start	

Physical property	Logical equivalent	Notes
`float: right`	`float: inline-end`	
`resize: horizontal`	`resize: inline`	
`resize: vertical`	`resize: block`	
`overflow-x / overflow-y`	`overflow-inline / overflow-block`	

4. Shorthands and Two-Value Syntax

```
/* Axis shorthands — apply to start and end together */

margin-block:  1rem;        /* both block-start and block-end: 1rem */
margin-block:  1rem 2rem;   /* block-start: 1rem; block-end: 2rem */

margin-inline: auto;        /* centre horizontally (like margin: 0 auto but logical) */
margin-inline: 1rem 3rem;   /* start: 1rem; end: 3rem */

padding-block:  0.5rem 1rem;
padding-inline: 1.5rem;

border-block:         1px solid var(--border);    /* top and bottom borders */
border-block-start:   3px solid var(--accent);    /* top border (or left in vertical) */
border-inline-start:  3px solid var(--accent);    /* left border (or top in vertical) */

inset:         0;             /* top:0; right:0; bottom:0; left:0 */
inset:         10px 20px;     /* block:10px; inline:20px */
inset-block:   0;             /* top:0; bottom:0 */
inset-inline:  0;             /* left:0; right:0 — stretch horizontally */

/* Common pattern: stretch an element to fill its container */
.overlay {
    position: absolute;
    inset:    0;         /* replaces top:0;right:0;bottom:0;left:0 */
}

/* Centre with margin-inline: auto */
.container {
    max-inline-size: 1200px;
    margin-inline:   auto;     /* perfectly centred in any writing mode */
}
```

5. direction and RTL Layouts

```css
/* direction controls text direction within horizontal-tb writing mode */

direction: ltr;  /* default — inline-start = left, inline-end = right */
direction: rtl;  /* Arabic/Hebrew — inline-start = right, inline-end = left */

/* In practice: set direction via HTML lang attribute and the browser user-agent */
/* Or use dir="rtl" on the HTML element or a section */
/* CSS direction should match the HTML dir attribute — don't set them independently */

/* The html element approach */
/* <html lang="ar" dir="rtl"> */

/* Logical properties respond automatically to dir="rtl" */
.card {
    border-inline-start: 3px solid var(--accent);
    padding-inline-start: 1rem;
    /* LTR: left border and left padding */
    /* RTL: right border and right padding — no override needed */
}

/* Physical properties require explicit RTL overrides */
.card--physical {
    border-left:  3px solid var(--accent);
    padding-left: 1rem;
}
[dir="rtl"] .card--physical {
    border-left:  none;
    border-right: 3px solid var(--accent);
    padding-left: 0;
    padding-right:1rem;
    /* Four extra declarations for what logical did in zero */
}
```

LTR — dir="ltr"

Article Title

border-inline-start creates a left accent border in LTR layout. padding-inline-start adds left padding.

RTL — dir="rtl"

عنوان المقال

نفس خصائص CSS تماماً — ينعكس تلقائياً إلى اليمين في وضع RTL.

Both panels above use exactly the same CSS — `border-inline-start` and `padding-inline-start`. In LTR the accent appears on the left; in RTL it appears on the right. Zero extra CSS was written for the RTL version.

6. Logical Border Radius

```
/* Logical border-radius — format: border-{block-position}-{inline-position}-radius */

/* In horizontal-tb LTR: */
border-start-start-radius: 8px;   /* top-left     */
border-start-end-radius:   8px;   /* top-right    */
border-end-start-radius:   8px;   /* bottom-left  */
border-end-end-radius:     8px;   /* bottom-right */

/* In horizontal-tb RTL: */
border-start-start-radius: 8px;   /* top-right    (inline-start = right in RTL) */
border-start-end-radius:   8px;   /* top-left     */

/* In vertical-rl: */
border-start-start-radius: 8px;   /* top-right    (block-start = top, inline-start = rig

/* Practical use: tab component that rounds the correct corners regardless of mode */
.tab {
    border-start-start-radius: 6px;
    border-start-end-radius:   6px;
    border-end-start-radius:   0;
    border-end-end-radius:     0;
    /* LTR: rounded top-left and top-right → correct horizontal tab shape */
    /* vertical-rl: rounded top-right and bottom-right → correct vertical tab shape */
}
```

7. Flexbox, Grid, and Logical Properties

```
/* Flexbox already uses logical concepts internally */
/* flex-direction: row aligns items along the inline axis */
/* flex-direction: column aligns items along the block axis */
/* This means flex row items reverse order correctly when dir="rtl" */

/* However: explicit margins and padding still need logical properties */
.nav-item {
    margin-inline-end: 1rem;    /* gap between items: right in LTR, left in RTL */
}

/* CSS Grid: column and row always mean physical axes */
/* But gap, padding, and margin on grid children should use logical properties */
.grid-item {
    padding-inline:  1rem;
    padding-block:   0.75rem;
}

/* Writing mode changes in flexbox — the axis swaps */
.vertical-nav {
    writing-mode:   vertical-rl;
    display:        flex;
    /* flex-direction: row still means "along the inline axis" */
    /* but in vertical-rl the inline axis is vertical */
    /* so items stack vertically by default — no flex-direction change needed */
}

/* Scroll bar position with logical overflow */
.scrollable {
    overflow-block:  auto;    /* vertical scrollbar in horizontal writing */
    overflow-inline: hidden;  /* no horizontal overflow */
}

/* text-align: start / end — logical values */
.caption {
    text-align: start;    /* left in LTR, right in RTL */
}
.label {
    text-align: end;      /* right in LTR, left in RTL */
}
```

8. Vertical Text Patterns

```css
/* ── Vertical navigation labels (sideways) ──────────────────── */
.sidebar-label {
    writing-mode:  sideways-rl;
    white-space:   nowrap;
    text-align:    center;
}

/* Fallback for Safari < 18 using rotate */
.sidebar-label {
    transform:    rotate(90deg);
    white-space:  nowrap;
}
@supports (writing-mode: sideways-rl) {
    .sidebar-label {
        transform:    none;
        writing-mode: sideways-rl;
    }
}

/* ── Japanese magazine-style layout ─────────────────────────── */
.article--tategumi {
    writing-mode:    vertical-rl;
    block-size:      80vh;          /* height of the reading column */
    overflow-inline: auto;            /* horizontal scroll (now the "block" axis) */
    text-align:      justify;
    hanging-punctuation: force-end;  /* Japanese punctuation rules */
}

/* ── Inline-size and block-size swap in vertical mode ────────── */
.vertical-card {
    writing-mode:  vertical-rl;
    inline-size:   200px;   /* this is the HEIGHT in vertical mode */
    block-size:    auto;     /* this is the WIDTH — expands with content */
}

/* ── Chapter headings with combined writing-mode + rotation ─── */
.chapter-heading {
    writing-mode:    vertical-lr;
    text-orientation:upright;
    font-variant-numeric: tabular-nums;
    /* Digits stand upright in vertical text */
}
```

9. When to Use Logical vs Physical Properties

Default to logical. For any new CSS you write — sizing, spacing, borders, positioning — use logical properties. They work correctly across all writing modes and directions without any extra overrides. The cognitive overhead of mapping start/end to left/right is minimal and the payoff in internationalisation correctness is large.

```
/* When physical properties ARE correct */

/* 1. Truly physical UI that must not flip */
/*    A progress bar that fills left-to-right regardless of document direction */
.progress-fill {
    width: 60%;           /* must be physical — filling left-to-right is the metaphor */
    left:  0;             /* must anchor to the physical left edge */
}

/* 2. Geometric/decorative elements that are direction-agnostic */
.avatar {
    width:  48px;          /* a circle is a circle — physical or logical makes no differ
    height: 48px;
}

/* 3. Fixed-position elements anchored to physical screen edges */
.corner-badge {
    position: fixed;
    top:      1rem;         /* physical screen corner — intentionally not logical */
    right:    1rem;
}

/* 4. Images and media with inherent physical direction */
.hero-image {
    width:    100%;           /* images are physical — width/height are correct */
    height:   auto;
}

/* The rule of thumb: */
/* If the property describes a relationship to the READING DIRECTION → logical */
/* If the property describes a geometric fact about PHYSICAL SPACE → physical */
```

Mixing logical and physical on the same property causes conflicts. `margin-left` and `margin-inline-start` both set the same computed value in LTR horizontal writing. If you set both, the one that

appears later in the stylesheet wins (source order). Avoid mixing them — pick one model per component and be consistent.

Chapter Summary

Concept	Key point
`Inline axis`	The direction text flows within a line. Horizontal in horizontal-tb (LTR/RTL), vertical in vertical-rl/vertical-lr. inline-size, margin-inline, padding-inline, border-inline, inset-inline.
`Block axis`	The direction lines stack. Vertical in horizontal-tb, horizontal in vertical writing modes. block-size, margin-block, padding-block, border-block, inset-block.
`start / end`	Flow-relative start and end of an axis. In LTR: inline-start = left, inline-end = right. In RTL: reversed. In vertical-rl: block-start = right.
`writing-mode`	horizontal-tb (default), vertical-rl (Japanese), vertical-lr (Mongolian), sideways-rl/lr (rotated Latin labels). Changes which axis is inline and which is block.
`text-orientation`	Applies in vertical writing modes. mixed (default: CJK upright, Latin rotated), upright (all upright), sideways (all rotated). Controls individual glyph orientation.
`direction`	ltr (default) or rtl. Controls inline-start/end mapping within horizontal-tb. Set via HTML dir attribute; match with CSS direction. Never set direction independently of dir.
`inset`	Shorthand for all four position offsets (top/right/bottom/left). inset: 0 replaces the four-property stretch pattern. inset-block and inset-inline for axis shorthands.
`Logical border-radius`	border-start-start-radius etc. First word = block position (start/end), second word = inline position. Useful for tabs and dialog corners that must work in all writing modes.
`text-align: start/end`	Logical values for text-align. start = left in LTR, right in RTL. Prefer over text-align: left for internationalised content.
`When to use physical`	Truly physical UI (progress bars, fixed corners, geometric shapes, images). Use physical when the meaning is about screen geometry, not reading direction.

EXERCISES

1. **Full logical property audit:** Take a component you've built previously (card, nav, or form). Replace every physical property with its logical equivalent: `width → inline-size`, `margin-left → margin-inline-start`, `border-top → border-block-start`, `top/left → inset-block-start/inset-inline-start`, `text-align: left → text-align: start`. Wrap the component in a `div dir="rtl"` and confirm the layout mirrors correctly without any additional CSS.
2. **Writing mode sidebar label:** Create a fixed-width sidebar. Add section labels that display as vertical rotated text using `writing-mode: sideways-rl`. Implement the `@supports` fallback using `transform: rotate(90deg)` for browsers without support. Confirm that `inline-size` on the label element controls its height (because the writing mode is vertical) and `block-size` controls its width.
3. **RTL-safe card component:** Build a card with an icon on the start side, a title and body in the remaining space, and a "learn more" arrow link on the end side. Use only logical properties. Toggle between `dir="ltr"` and `dir="rtl"` on the card's container and verify: icon moves to the right, text aligns correctly, and the arrow points the right way (use `inline-end` margin on the arrow). Document in a comment which property would break if you had used its physical equivalent.
4. **Vertical tab navigation:** Create a tab component where the tab list runs vertically along one side. Use `writing-mode: vertical-rl` on the tab labels so they read top-to-bottom. Apply `border-start-start-radius` and `border-end-start-radius` to the active tab to give it the correct rounding for a vertical tab shape. Then switch the whole component to `dir="rtl"` and observe which border-radius corners flip automatically.
5. **Container query + logical:** Combine what you've learned in Chapters 2 and 6: create a card with `container-type: inline-size`. Write container queries that change the card's layout at a certain `inline-size` threshold. Inside the card, use only logical properties for all spacing and borders. Confirm the card works correctly when placed inside a `writing-mode: vertical-rl` container — the container query threshold should now be relative to the vertical axis since that is the inline axis in vertical writing.

Next: Chapter 7 — Advanced Color. The oklch() color space — why it produces perceptually uniform color scales, how to build a complete design token palette with it, the `color-mix()` function, wide-gamut colors with `display-p3`, `light-dark()`, relative color syntax, and `@media (color-gamut)` for progressive enhancement.

CHAPTER 7

Advanced Color — oklch and color-mix()

Chapter 7 — Advanced Color: oklch, color-mix(), and Color Spaces

CSS has supported color since 1996 but for most of that time browsers worked in a single color space — sRGB — with two authoring models: hex/rgb and hsl. Modern monitors can display colors that sRGB cannot represent, and HSL has a fundamental flaw: two colors with the same *L* value look dramatically different in brightness to the human eye. The CSS Color Level 4 and Level 5 specifications fix both problems.

> **Baseline 2023.** oklch(), color-mix(), display-p3, and relative color syntax are supported in all evergreen browsers (Chrome 111+, Firefox 113+, Safari 15.4+). light-dark() landed in Baseline 2024. Use @supports for graceful fallbacks where needed — older browsers still receive your fallback color, not a broken layout.

1. Why HSL Fails as a Design Tool

HSL (Hue, Saturation, Lightness) was designed for programmer convenience, not perceptual accuracy. Its Lightness axis is *mathematically* uniform but *perceptually* wrong: yellow at 50% L appears far brighter than blue at 50% L. This breaks color scales, makes accessible contrast hard to predict, and produces jarring hover states when you only change the L value.

HSL — PROGRAMMER MODEL

- H — **Hue (0–360°)** Position on the color wheel. Not uniform — hue steps look different sizes in different regions.
- S — **Saturation (0–100%)** How vivid vs grey. Interacts with L non-linearly.
- L — **Lightness (0–100%)** Mathematical midpoint. Yellow at 50% L looks almost white; blue at 50% L looks dark.

Problem: hsl(60 100% 50%) yellow vs hsl(240 100% 50%) blue — same L, wildly different perceived brightness.

OKLCH — PERCEPTUAL MODEL

- L — **Lightness (0–1)** Perceptually uniform. Colors with the same L value look equally bright to the human eye.
- C — **Chroma (0–0.37+)** How vivid the color is. 0 = grey. Max varies by hue and the gamut of the display.
- H — **Hue (0–360°)** Position on the color wheel. Perceptually uniform — equal hue steps look equal in size.

Result: changing only L gives truly lighter/darker variants. Building accessible palettes becomes predictable.

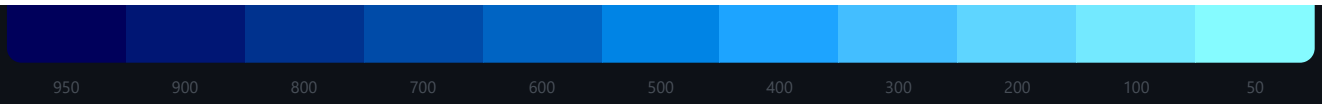
2. oklch() Syntax

```
/* oklch(L C H) */
/* oklch(L C H / alpha) */

oklch(0.7 0.15 250)
/*     L    C    H  */
/*     |    |    └─ hue in degrees (0–360, wraps) */
/*     |    └────── chroma: 0 = grey, ~0.37 = maximum sRGB-safe for most hues */
/*     └─────────── lightness: 0 = black, 1 = white (or 0%–100%) */

/* Percentage syntax for L */
oklch(70% 0.15 250)   /* same as 0.7 */

/* With alpha */
oklch(0.7 0.15 250 / 50%)   /* 50% opacity */
oklch(0.7 0.15 250 / 0.5)   /* same */

/* none — the missing-component value */
oklch(0.7 none 250)    /* chroma = 0, treated as grey for interpolation */

/* Key hue landmarks */
/*   0°  / 360° = red                        */
/*  30°         = orange                     */
/*  60°         = yellow                     */
/*  120°        = green                      */
/*  180°        = cyan                       */
/*  220–260°    = blue (most UI blues here)  */
/*  300°        = magenta/pink               */

/* Practical named references */
--color-blue:   oklch(0.62 0.18 248);   /* GitHub-ish blue */
--color-green:  oklch(0.68 0.19 145);
--color-red:    oklch(0.63 0.22 25);
--color-amber:  oklch(0.72 0.17 65);
--color-purple: oklch(0.65 0.20 300);
```

3. Building a Perceptually Uniform Color Scale

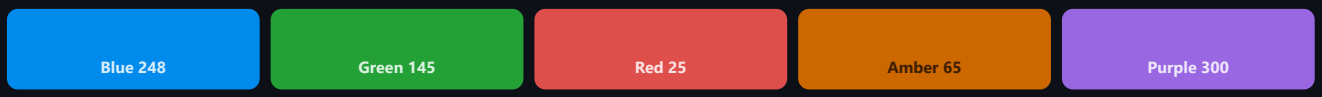
The value of oklch for design systems: you can generate a full 50–950 scale by varying only L, keeping C and H constant. Each step looks the same perceptual distance from the one below it.

Blue scale — H=248, C=0.18, L varies from 0.18 to 0.96



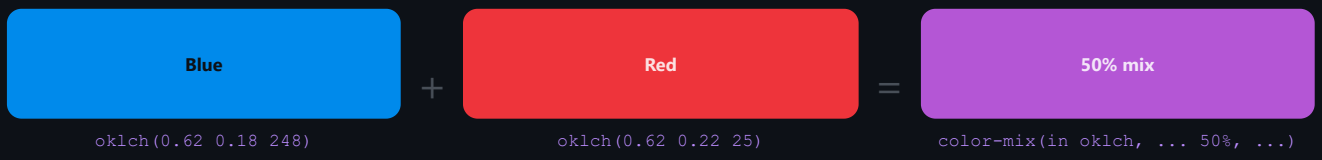

```css
/* A complete scale using CSS custom properties */
:root {
    /* Blue scale — H fixed at 248, C fixed at 0.18 */
    --blue-950: oklch(0.18 0.18 248);
    --blue-900: oklch(0.26 0.18 248);
    --blue-800: oklch(0.34 0.18 248);
    --blue-700: oklch(0.42 0.18 248);
    --blue-600: oklch(0.50 0.18 248);
    --blue-500: oklch(0.60 0.18 248);   /* base */
    --blue-400: oklch(0.70 0.18 248);
    --blue-300: oklch(0.78 0.18 248);
    --blue-200: oklch(0.85 0.18 248);
    --blue-100: oklch(0.91 0.18 248);
    --blue-50:  oklch(0.96 0.18 248);

    /* Multi-hue palette — same L and C, only H changes */
    /* All five of these have the same perceived lightness */
    --brand-blue:   oklch(0.62 0.18 248);
    --brand-green:  oklch(0.62 0.18 145);
    --brand-red:    oklch(0.62 0.18 25);
    --brand-amber:  oklch(0.62 0.18 65);
    --brand-purple: oklch(0.62 0.18 300);
}
```

Same L=0.62, C=0.18 — only H changes. Equal perceived brightness.



4. color-mix()

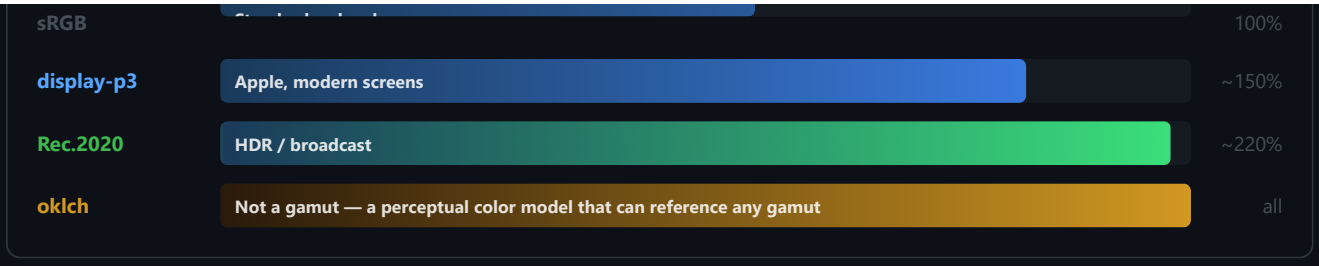
`color-mix()` blends two colors together in a specified color space. The color space choice matters enormously — the same mix in oklch vs sRGB produces noticeably different results, especially for complementary colors.



```css
/* color-mix(in <color-space>, <color1> [pct], <color2> [pct]) */

/* Basic mix — equal halves */
color-mix(in oklch, blue 50%, red)

/* Asymmetric mix — 30% of first color, 70% of second */
color-mix(in oklch, blue 30%, red 70%)

/* Single percentage — the remainder is taken from the second color */
color-mix(in oklch, blue 30%, red)    /* same as above */

/* Color spaces: oklch, oklab, srgb, hsl, hwb, lab, lch, display-p3 */
color-mix(in srgb,  blue, red)    /* produces muddy purple */
color-mix(in oklch, blue, red)    /* travels the hue wheel — cleaner midpoint */

/* Hue interpolation methods for oklch and lch */
color-mix(in oklch shorter hue, blue, red)    /* default — shortest path on hue wheel */
color-mix(in oklch longer hue,  blue, red)    /* longest path — goes through more hues */
color-mix(in oklch increasing hue, blue, red)  /* always increasing H value */

/* Practical: tinting towards white (mix with a very-high-L neutral) */
--btn-hover-bg: color-mix(in oklch, var(--brand-blue) 80%, white);
--btn-active-bg:color-mix(in oklch, var(--brand-blue) 80%, black);

/* Creating opacity-equivalent using color-mix */
/* color-mix(in srgb, blue 20%, transparent) ≈ blue at 20% opacity */
--overlay: color-mix(in srgb, black 60%, transparent);

/* Generating a tonal surface from a brand color */
:root {
    --brand:           oklch(0.55 0.22 248);
    --surface-brand:  color-mix(in oklch, var(--brand) 12%, oklch(0.1 0 0));
    --border-brand:   color-mix(in oklch, var(--brand) 40%, oklch(0.1 0 0));
    /* Both surface and border derive from the same brand token */
}
```

5. Color Spaces, Gamuts, and display-p3

A **color space** is a mathematical model for representing colors. A **gamut** is the range of colors a color space (or device) can express. Modern displays can show colors that sRGB cannot — the P3 gamut used by Apple and most high-end screens covers roughly 50% more colors than sRGB.




```
/* oklch can reference colors outside sRGB gamut */
/* C (chroma) > ~0.37 may be out of sRGB gamut for some hues */

oklch(0.65 0.30 145)    /* vivid green — outside sRGB, inside P3 */
oklch(0.65 0.37 145)    /* more vivid — may exceed even P3 on some hues */

/* display-p3 — explicit P3 color space syntax */
color(display-p3 0.2 0.6 1)    /* P3 triplet — brighter blue than sRGB allows */

/* ── Progressive enhancement with @media (color-gamut) ──────── */
:root {
    /* Fallback for standard displays */
    --accent: oklch(0.62 0.18 248);
}

@media (color-gamut: p3) {
    :root {
        /* Richer blue only on P3-capable displays */
        --accent: oklch(0.62 0.26 248);    /* higher C, outside sRGB */
    }
}

@media (color-gamut: rec2020) {
    :root {
        /* Maximum vividness for HDR displays */
        --accent: oklch(0.62 0.32 248);
    }
}

/* color-gamut values: srgb | p3 | rec2020 (progressive: p3 matches p3 AND rec2020) */

/* Gamut mapping — browser behaviour when a color is out of gamut */
/* Browsers automatically gamut-map out-of-gamut oklch values to the nearest */
/* in-gamut color on screens that can't display the specified color. */
/* You get the best available approximation — nothing breaks. */
```

6. Relative Color Syntax

Relative color syntax lets you derive a new color from an existing one by reading out its channel values and modifying them with CSS math functions. This is the feature that unlocks truly dynamic, token-driven theming without JavaScript.

```
/* Syntax: color-function(from <origin-color> channel1 channel2 channel3) */

/* Read the origin color's channels into named variables l, c, h */
oklch(from var(--brand)   l c h)            /* exact copy */
oklch(from var(--brand)   calc(l + 0.15) c h)   /* lighter variant */
oklch(from var(--brand)   calc(l - 0.15) c h)   /* darker variant */
oklch(from var(--brand)   l calc(c * 0.5) h)   /* less saturated */
oklch(from var(--brand)   l c calc(h + 30))   /* hue-shifted 30° */

/* Channel names vary by color space: */
/*   oklch:       l, c, h             */
/*   oklab:       l, a, b             */
/*   hsl:         h, s, l             */
/*   rgb/srgb:    r, g, b             */
/*   display-p3:  r, g, b             */

/* Practical: derive all interactive states from one token */
:root {
    --btn-bg:      oklch(0.55 0.22 248);
    --btn-hover:  oklch(from var(--btn-bg) calc(l + 0.08) c h);
    --btn-active: oklch(from var(--btn-bg) calc(l - 0.08) c h);
    --btn-text:   oklch(from var(--btn-bg) max(0.95, calc(l + 0.4)) 0.02 h);
    /* --btn-text ensures contrast by pushing L near 0.95 — near-white */
}

/* Cross-space derivation — read from one space, output to another */
oklch(from rebeccapurple   l c h)         /* named color → oklch */
oklch(from #58a6ff         l c h)         /* hex → oklch */
oklch(from hsl(220 80% 60%)  l c h)      /* hsl → oklch */

/* alpha channel — available as 'alpha' in all color spaces */
oklch(from var(--brand)   l c h / 0.5)    /* 50% transparent */
oklch(from var(--brand)   l c h / alpha)  /* preserve original alpha */
```

Five states derived from one base token: `oklch(0.55 0.22 248)`

7. light-dark()

`light-dark()` is a CSS function that selects between two values based on whether the user prefers a light or dark color scheme. It requires `color-scheme` to be declared on the element or its ancestors — typically set on `:root`.

Light color-scheme

Component card

Automatically uses the light values from light-dark() calls without any @media override.

Dark color-scheme

Component card

Same CSS — automatically uses the dark values. No @media (prefers-color-scheme) block needed.

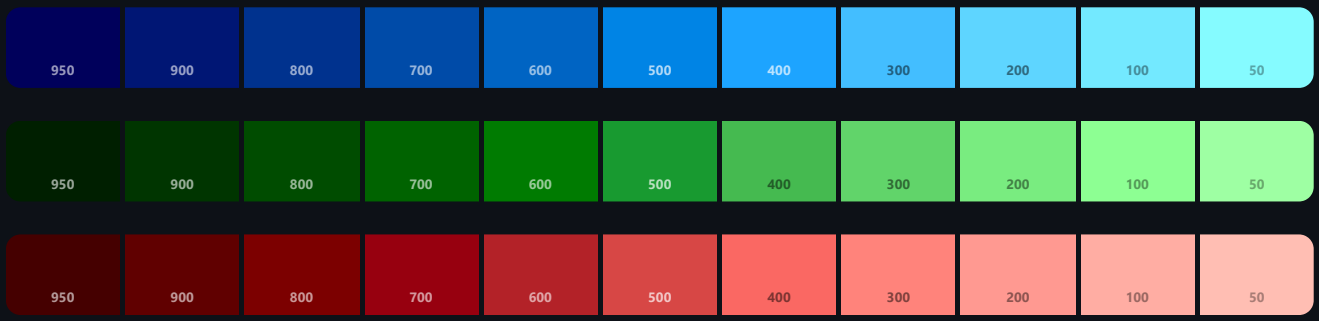
```
/* light-dark(light-value, dark-value) */
/* color-scheme must be set on :root (or the element) */

:root {
    color-scheme: light dark;   /* tell the browser both schemes are supported */
}

:root {
    --bg:      light-dark(oklch(0.97 0.01 248), oklch(0.10 0.01 248));
    --surface: light-dark(oklch(1.00 0.00 0),   oklch(0.14 0.01 248));
    --text:    light-dark(oklch(0.15 0.01 248), oklch(0.92 0.01 248));
    --border:  light-dark(oklch(0.85 0.02 248), oklch(0.22 0.02 248));
    --accent:  light-dark(oklch(0.50 0.22 248), oklch(0.70 0.18 248));
    /* Note: lighter accent in dark mode for contrast against dark surface */
}

/* Switching scheme for a specific section */
.dark-panel {
    color-scheme: dark;   /* forces dark values for light-dark() in this subtree */
}

/* vs the older approach — more verbose, duplicate selectors */
:root                                { --bg: oklch(0.97 0.01 248); }
@media (prefers-color-scheme: dark) {
    :root                            { --bg: oklch(0.10 0.01 248); }
}
/* light-dark() collapses this to one declaration */
```

8. Color Interpolation in Gradients and Animations

```
/* Gradients can specify which color space to interpolate in */
/* This dramatically affects midpoint quality */

/* Default sRGB interpolation — can produce muddy grey midpoints */
linear-gradient(to right, blue, yellow)

/* oklch interpolation — travels the hue wheel, no grey midpoint */
linear-gradient(in oklch, blue, yellow)

/* Controlling hue travel direction */
linear-gradient(in oklch shorter hue, oklch(0.6 0.2 0), oklch(0.6 0.2 200))
linear-gradient(in oklch longer hue,  oklch(0.6 0.2 0), oklch(0.6 0.2 200))

/* Rainbow gradient using hue wheel */
.rainbow {
    background: linear-gradient(
        in oklch longer hue,
        oklch(0.65 0.22 0),
        oklch(0.65 0.22 360)
    );
}

/* Animations: @keyframes interpolate in sRGB by default */
/* Use color-interpolation-method to change this */
@keyframes pulse {
    from { color: oklch(0.6 0.22 248); }
    to   { color: oklch(0.6 0.22 300); }
}
/* Both keyframe values are in oklch — the browser interpolates correctly */
/* CSS Animations Level 2 will add color-interpolation-method to @keyframes */
```

9. A Complete oklch Token System

Brand palette: H=248 (blue), H=145 (green), H=25 (red), H=65 (amber) — C=0.18 throughout



```css
/* Complete design token system using oklch + relative color + light-dark() */

:root {
    color-scheme: light dark;

    /* ── Primitive tokens (the palette) ──────────────────────── */
    --blue-500:   oklch(0.60 0.18 248);
    --blue-400:   oklch(0.70 0.18 248);
    --blue-200:   oklch(0.85 0.18 248);
    --green-500:  oklch(0.60 0.18 145);
    --red-500:    oklch(0.60 0.18 25);
    --amber-400:  oklch(0.78 0.18 65);
    --grey-950:   oklch(0.12 0.01 248);
    --grey-100:   oklch(0.94 0.01 248);
    --grey-50:    oklch(0.98 0.01 248);

    /* ── Semantic tokens (light / dark) ──────────────────────── */
    --color-bg:       light-dark(var(--grey-50),  var(--grey-950));
    --color-surface:  light-dark(oklch(1 0 0),      oklch(0.17 0.01 248));
    --color-text:     light-dark(oklch(0.15 0 0),    oklch(0.90 0.01 248));
    --color-text-sub: light-dark(oklch(0.45 0 0),    oklch(0.60 0.01 248));
    --color-border:   light-dark(oklch(0.85 0.02 248),oklch(0.24 0.02 248));
    --color-accent:   light-dark(var(--blue-500),var(--blue-400));
    --color-success:  light-dark(var(--green-500),oklch(0.72 0.18 145));
    --color-danger:   light-dark(var(--red-500),  oklch(0.72 0.18 25));

    /* ── Component tokens derived from semantic ──────────────── */
    --btn-bg:     var(--color-accent);
    --btn-hover:  oklch(from var(--color-accent) calc(l + 0.07) c h);
    --btn-active: oklch(from var(--color-accent) calc(l - 0.07) c h);
    --btn-ring:   oklch(from var(--color-accent) l c h / 0.4);
}
```

Chapter Summary

Concept	Key point
`oklch(L C H)`	Perceptually uniform color space. L: 0–1 lightness; C: 0–0.37+ chroma; H: 0–360° hue. Equal L values look equally bright across all hues — HSL does not guarantee this.
`Color scales`	Build a full 50–950 scale by varying only L while holding C and H constant. Each step is a perceptually equal distance from the adjacent one. Ideal for design tokens.
`color-mix()`	color-mix(in oklch, color1 pct, color2) blends two colors in a chosen color space. The color space changes the midpoint dramatically — oklch avoids muddy grey midpoints. Supports hue interpolation keywords: shorter/longer/increasing/decreasing hue.
`display-p3`	Color space covering ~50% more colors than sRGB. Supported on most modern screens. Use color(display-p3 r g b) syntax or high-C oklch values. Browsers gamut-map out-of-gamut colors automatically — nothing breaks on older screens.
`@media (color-gamut)`	Values: srgb, p3, rec2020. Progressive enhancement: set sRGB-safe colors as default, override with richer P3 colors inside the media query. p3 matches both p3 and rec2020 displays.
`Relative color syntax`	oklch(from <origin> l c h) reads channel values from an existing color and lets you manipulate them with calc(). Derive hover/active/disabled states from one base token. Works across color spaces.
`light-dark()`	light-dark(light-val, dark-val) selects based on prefers-color-scheme. Requires color-scheme to be declared on :root. Eliminates separate @media (prefers-color-scheme: dark) blocks for color values. Use color-scheme: dark on a subtree to force dark values in that section.
`Gradient interpolation`	linear-gradient(in oklch, ...) interpolates in oklch space. Cleaner midpoints than sRGB default. Add shorter/longer hue keywords to control which direction the hue wheel travels.
`Token architecture`	Three tiers: primitive (the full palette), semantic (light-dark mapped tokens), component (derived from semantic via relative color). Change one primitive and all derived values update.

EXERCISES

1. **Build a full oklch palette:** Choose a brand hue (any H value that isn't already a web standard color). Generate 11 steps (50–950) by varying L from 0.96 down to 0.18 in even increments, keeping C at 0.18. Render all 11 swatches. Then repeat with C=0.10 for a muted tonal version. Compare the two scales side by side and note where sRGB gamut clipping occurs (if at all) at your chosen hue.
2. **color-mix() hover states:** Create a button component that uses only a single `--btn-color` custom property. Derive the hover background using `color-mix(in oklch, var(--btn-color) 80%, white)`, the active background using `color-mix(in oklch, var(--btn-color) 80%, black)`, and a focus ring using `color-mix(in srgb, var(--btn-color) 40%, transparent)`. Verify you can change the button's color by updating just one token.
3. **Relative color interaction states:** Rewrite the exercise above using relative color syntax instead of color-mix. Use `oklch(from var(--btn-color) calc(l + 0.1) c h)` for hover and `calc(l - 0.1)` for active. Compare the visual results: when do relative color and color-mix produce different results, and which looks better?
4. **light-dark() complete theme:** Implement a card component that uses only `light-dark()` calls for every color property: background, text, subtext, border, and a coloured tag badge. Set `color-scheme: light dark` on `:root`. Then create a `.theme-override` class that sets `color-scheme: dark` on a container, forcing dark values on a card inside it even when the OS is in light mode. Verify no `@media` blocks were needed.
5. **Wide-gamut progressive enhancement:** Define a set of semantic color tokens at three levels: sRGB-safe (C ≤ 0.18), P3-enhanced (C = 0.28), and a descriptive comment for Rec.2020 (C = 0.35). Implement the sRGB base in `:root`, override with P3 values inside `@media (color-gamut: p3)`. Open the result in a browser on an Apple Display and confirm the P3 overrides take effect using DevTools.

Next: Chapter 8 — CSS Houdini. The CSS Houdini APIs: `CSS.paintWorklet` and the Paint API (custom backgrounds, borders, and highlights without SVG), the Typed Object Model (`element.attributeStyleMap`), `CSS.registerProperty()` for animatable custom properties, the Properties and Values API, and a practical custom checkered paint worklet.

CHAPTER 8

CSS Houdini — Painting and Typed OM

Chapter 8 — CSS Houdini: Painting and Typed OM

CSS Houdini is a collection of low-level browser APIs that expose parts of the CSS rendering engine to JavaScript. Before Houdini, browser vendors implemented new CSS features and developers waited years for support. Houdini lets you implement new CSS capabilities yourself — custom paint operations, animatable custom properties with full type information, and a typed JavaScript interface to the CSS Object Model — today, in any supporting browser.

> **Support is uneven across APIs.** The Paint API and `@property` / `CSS.registerProperty()` are the most widely supported and production-ready. The Layout Worklet and Animation Worklet are Chrome-only and not covered in detail here. Always check support before shipping Houdini features — and always include a CSS fallback.

Paint API

Register a JavaScript worklet that draws into a canvas-like context. The output can be used anywhere CSS expects an image — background-image, border-image, mask-image. Inputs come from CSS custom properties.

Chrome ✓ Edge ✓ Firefox — Safari —

Properties & Values API

Register a custom property with a type, initial value, and inheritance flag — via @property in CSS or CSS.registerProperty() in JS. Enables transitions and animations on custom properties used in gradients, transforms, etc.

Chrome ✓ Edge ✓ Firefox ✓ Safari ✓

Typed Object Model

A typed JavaScript API for reading and writing CSS values — attributeStyleMap and computedStyleMap() instead of element.style. Values are objects (CSSUnitValue, CSSMathSum) rather than raw strings, making arithmetic safe.

Chrome ✓ Edge ✓ Firefox — Safari —

Layout API

Write custom layout algorithms as a worklet — similar to how browsers implement flexbox and grid internally. Very powerful but highly experimental. Not production-ready.

Chrome flag Firefox — Safari —

Animation Worklet

Run animation logic off the main thread in a worklet. Use for scroll-linked animations and time-based animations that need to be jank-free. Partially superseded by the CSS Scroll-Driven Animations spec.

Chrome only Firefox — Safari —

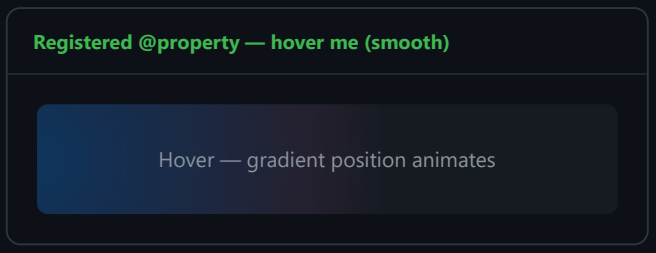
Parser API

Parse arbitrary CSS strings from JavaScript, getting back a typed AST. Useful for tooling and polyfills but not yet widely useful for production code. Very early stage.

Not yet shipped

1. @property and CSS.registerProperty()

Unregistered CSS custom properties are treated as opaque strings by the browser. The browser cannot interpolate between two string values — so transitions and animations on custom properties used inside gradients or transforms simply jump rather than tween. `@property` fixes this by giving the browser the type information it needs to interpolate correctly.



```css
/* ── @property — CSS syntax (Baseline 2024) ──────────────────── */
@property --highlight-pos {
    syntax:        '<percentage>';
    initial-value: 0%;
    inherits:      false;
}

.card {
    --highlight-pos: 0%;
    background: radial-gradient(
        circle at var(--highlight-pos) 50%,
        oklch(0.55 0.22 248 / 0.3),
        transparent 60%
    ), #161b22;
    transition: --highlight-pos 0.4s ease;
    /* Without @property: transition has no effect — browser sees a string jump */
    /* With @property: browser knows it's a <percentage> — interpolates correctly */
}
.card:hover { --highlight-pos: 100%; }

/* ── JavaScript equivalent ──────────────────────────────────── */
CSS.registerProperty({
    name:         '--highlight-pos',
    syntax:       '<percentage>',
    initialValue: '0%',
    inherits:     false,
});
/* Same effect as @property. Use JS form when registering dynamically. */
/* Use @property in CSS when you want the registration collocated with the rule. */

/* ── Supported syntax strings ──────────────────────────────── */
/* '<length>'           px, em, rem, vw, etc.          */
/* '<number>'           unitless numbers               */
/* '<percentage>'       0%-100%                        */
/* '<length-percentage>' calc(50px + 10%)              */
/* '<color>'            any CSS color                  */
/* '<image>'            url(), gradient functions      */
/* '<angle>'            deg, rad, turn                 */
/* '<time>'             s, ms                          */
/* '<resolution>'       dpi, dpcm, dppx                */
/* '<transform-list>'   translate(), rotate(), etc.    */
/* '<custom-ident>'     unquoted identifier            */
/* '*'                  any value — same as unregistered */
/* 'red | blue | green' explicit keyword list          */
/* '<color>+'           space-separated list of colors */
/* '<color>#'           comma-separated list of colors */
```

Animating custom properties — practical patterns

```
/* ── Animating a gradient angle ──────────────────────────── */
@property --gradient-angle {
    syntax:        '<angle>';
    initial-value: 0deg;
    inherits:      false;
}
@keyframes spin-gradient {
    to { --gradient-angle: 360deg; }
}
.spinning-card {
    background: conic-gradient(from var(--gradient-angle),
        oklch(0.55 0.22 248), oklch(0.55 0.22 300), oklch(0.55 0.22 248));
    animation: spin-gradient 4s linear infinite;
}

/* ── Animating a color token ─────────────────────────────── */
@property --accent-color {
    syntax:        '<color>';
    initial-value: oklch(0.60 0.18 248);
    inherits:      true;
}
.theme-toggle {
    transition: --accent-color 0.6s ease;
}
.theme-toggle.warm {
    --accent-color: oklch(0.65 0.20 40);  /* smooth transition to warm amber */
}

/* ── Animating a transform component ─────────────────────── */
@property --rotate-y {
    syntax:        '<angle>';
    initial-value: 0deg;
    inherits:      false;
}
.flip-card {
    transform:  rotateY(var(--rotate-y));
    transition: --rotate-y 0.5s ease;
}
.flip-card:hover { --rotate-y: 180deg; }
```

2. The Paint API

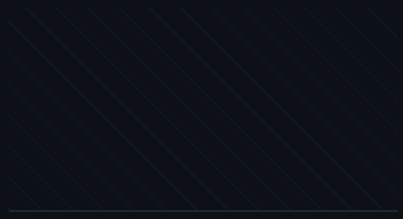
The Paint API lets you write a JavaScript class that draws to a `PaintRenderingContext2D` — a subset of the Canvas 2D API — and use the output wherever CSS expects an `<image>`. The worklet runs in a separate thread with no DOM access, keeping the main thread free.

checkered()

CSS-controllable checkerboard. Cell size and colors come from custom properties.

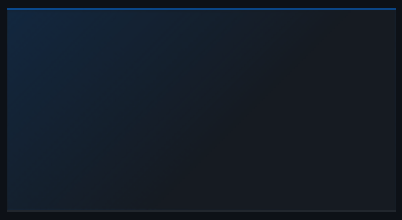
polka-dots()

Dot grid with variable spacing, size, and color. Adapts to element dimensions.



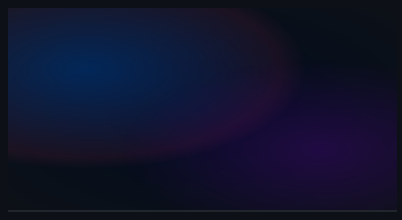
diagonal-lines()

Diagonal stripe pattern — angle, spacing, and opacity from custom properties.



glow-border()

Top-edge glow that responds to --glow-color. No pseudo-elements needed.



noise-bg()

Perlin-noise texture that would be impossible with pure CSS gradients.

circular-progress()

Progress arc drawn via paint worklet. --progress controls fill amount.

```
/* ── Registering and using a paint worklet ──────────────────────── */

/* Step 1: Write the worklet in a separate JS file (checkerboard.js) */
/* The file must be served with a JS MIME type, same origin, HTTPS */

/* checkerboard.js */
class CheckerboardPainter {
    static get inputProperties() {
        return ['--cb-size', '--cb-color-a', '--cb-color-b'];
    }
    paint(ctx, size, properties) {
        const cellSize = parseInt(properties.get('--cb-size')) || 20;
        const colorA   = properties.get('--cb-color-a').toString().trim() || '#fff';
        const colorB   = properties.get('--cb-color-b').toString().trim() || '#000';

        const cols = Math.ceil(size.width  / cellSize);
        const rows = Math.ceil(size.height / cellSize);

        for (let r = 0; r < rows; r++) {
            for (let c = 0; c < cols; c++) {
                ctx.fillStyle = (r + c) % 2 === 0 ? colorA : colorB;
                ctx.fillRect(c * cellSize, r * cellSize, cellSize, cellSize);
            }
        }
    }
}
registerPaint('checkerboard', CheckerboardPainter);

/* ──────────────────────────────────────────────────────────────────── */
/* Step 2: Register the worklet from the main JS file */

if ('paintWorklet' in CSS) {
    CSS.paintWorklet.addModule('/worklets/checkerboard.js');
}

/* ──────────────────────────────────────────────────────────────────── */
/* Step 3: Use it in CSS with paint() */

.checker-bg {
    --cb-size:    24;                    /* unitless int for the worklet */
    --cb-color-a: oklch(0.20 0.02 248);
    --cb-color-b: oklch(0.15 0.02 248);
    background:   paint(checkerboard);
}

/* paint() also works with border-image and mask-image */
.fancy-border {
```

```
    border:       12px solid transparent;
    border-image: paint(gradient-border) 12;
}
```

inputArguments — passing extra values directly

```
/* inputArguments lets you pass values directly in paint() rather than via custom props
/* Useful for one-off parameterisation without polluting the custom property namespace *

class CircleProgressPainter {
    static get inputProperties() { return ['--progress-color']; }
    static get inputArguments()  { return ['<number>']; }

    paint(ctx, size, properties, args) {
        const progress = args[0].value / 100;  /* 0-100 → 0-1 */
        const cx       = size.width  / 2;
        const cy       = size.height / 2;
        const r        = Math.min(cx, cy) - 8;

        ctx.strokeStyle = '#21262d';
        ctx.lineWidth   = 8;
        ctx.beginPath();
        ctx.arc(cx, cy, r, 0, Math.PI * 2);
        ctx.stroke();

        ctx.strokeStyle = properties.get('--progress-color').toString();
        ctx.lineCap     = 'round';
        ctx.beginPath();
        ctx.arc(cx, cy, r, -Math.PI / 2, (-Math.PI / 2) + (progress * Math.PI * 2));
        ctx.stroke();
    }
}
registerPaint('circle-progress', CircleProgressPainter);

/* Usage — the argument is passed directly in paint() */
.progress {
    --progress-color: oklch(0.55 0.22 248);
    background:       paint(circle-progress, 75);  /* 75% progress */
}
```

Paint worklet limitations. The `PaintRenderingContext2D` has no access to DOM, cookies, the network, or any global state. It cannot use `requestAnimationFrame` — animations must be driven by `@keyframes` or

custom property transitions on the host element. The worklet may be called on any thread at any time. Write it as a pure function of its inputs.

Fallback pattern

```
/* Always provide a plain CSS fallback before the paint() value */
/* Browsers that don't support the Paint API ignore paint() and use the first value */

.card {
    background: oklch(0.15 0.02 248);          /* fallback */
    background: paint(noise-background);       /* progressive enhancement */
}

/* Or use @supports */
@supports (background: paint(x)) {
    .card { background: paint(noise-background); }
}

/* Feature detection in JavaScript */
if ('paintWorklet' in CSS) {
    CSS.paintWorklet.addModule('/worklets/noise.js');
    document.documentElement.classList.add('paint-api');
}

.paint-api .card { background: paint(noise-background); }
```

3. The CSS Typed Object Model (Typed OM)

The traditional `element.style` API deals in raw strings. Typed OM replaces this with typed JavaScript objects — `CSSUnitValue`, `CSSMathSum`, `CSSKeywordValue` — that let you read, modify, and do arithmetic on CSS values without string parsing.

Old CSSOM — strings, fragile

Typed OM — objects, safe

```
/* Read */
const w = el.style.width;
// "120px" — a string

/* Arithmetic — must parse manually */
const num = parseFloat(w);
el.style.width = (num + 10) + 'px';

/* Computed style — also strings */
const cs = getComputedStyle(el);
cs.getPropertyValue('margin-top');
// "16px" — parse required
```

```
/* Read — get a typed value */
const w = el.attributeStyleMap.get('width');
// CSSUnitValue {value: 120, unit: 'px'}

/* Arithmetic — direct, typed */
w.value += 10;
el.attributeStyleMap.set('width', w);

/* Computed style — typed */
const cs = el.computedStyleMap();
cs.get('margin-top');
// CSSUnitValue {value: 16, unit: 'px'}
```

```
/* ── CSS factory functions ──────────────────────────────────────── */

/* Create typed values from JavaScript */
CSS.px(42)          // CSSUnitValue {value: 42,  unit: 'px'}
CSS.em(1.5)         // CSSUnitValue {value: 1.5, unit: 'em'}
CSS.rem(2)          // CSSUnitValue {value: 2,   unit: 'rem'}
CSS.percent(50)     // CSSUnitValue {value: 50,  unit: 'percent'}
CSS.vw(100)         // CSSUnitValue {value: 100, unit: 'vw'}
CSS.deg(45)         // CSSUnitValue {value: 45,  unit: 'deg'}
CSS.s(0.3)           // CSSUnitValue {value: 0.3, unit: 's'}
CSS.number(1.5)     // CSSUnitValue {value: 1.5, unit: 'number'}

/* Math types */
new CSSMathSum(CSS.px(100), CSS.percent(50))    // calc(100px + 50%)
new CSSMathProduct(CSS.px(10), CSS.number(3))   // calc(10px * 3)
new CSSMathMax(CSS.px(48), CSS.rem(3))        // max(48px, 3rem)

/* ── Writing values ─────────────────────────────────────────────── */
el.attributeStyleMap.set('width',     CSS.px(200));
el.attributeStyleMap.set('opacity',  CSS.number(0.5));
el.attributeStyleMap.set('display',  new CSSKeywordValue('flex'));
el.attributeStyleMap.delete('margin-top');

/* ── Reading computed values ────────────────────────────────────── */
const map = el.computedStyleMap();

const mt = map.get('margin-top');
mt.value;    // 16
mt.unit;     // 'px'

/* Reading shorthand — returns CSSStyleValue array */
const padding = map.getAll('padding');    // [top, right, bottom, left]

/* Unit conversion */
const inPx = CSS.em(2).to('px');
// resolves relative em to absolute px value in the document context

/* ── Practical: animate element width with typed arithmetic ─── */
function expandBy(el, amount) {
    const current = el.computedStyleMap().get('width');
    el.attributeStyleMap.set('width', CSS.px(current.value + amount));
    /* No string parsing, no unit stripping, no NaN risk */
}
```

4. Typed OM inside Worklets

```
/* Inside a paint worklet, properties.get() returns Typed OM values */
/* This is safer than parsing strings manually */

class RipplePainter {
    static get inputProperties() {
        return ['--ripple-color', '--ripple-radius'];
    }
    paint(ctx, size, props) {
        /* --ripple-color is a registered <color> — comes back as CSSStyleValue */
        const color  = props.get('--ripple-color').toString();

        /* --ripple-radius is a registered <number> — comes back as CSSUnitValue */
        const radius = props.get('--ripple-radius').value;  /* numeric, no parsing */

        ctx.fillStyle = color;
        ctx.beginPath();
        ctx.arc(size.width / 2, size.height / 2, radius, 0, Math.PI * 2);
        ctx.fill();
    }
}
registerPaint('ripple', RipplePainter);

/* The CSS for the worklet */
@property --ripple-radius {
    syntax:        '<number>';
    initial-value: 0;
    inherits:     false;
}
.btn {
    --ripple-radius: 0;
    --ripple-color:  oklch(1 0 0 / 0.2);
    background:      paint(ripple);
    transition:      --ripple-radius 0.6s ease-out;
}
.btn:active { --ripple-radius: 200; }
/* @property makes the radius animatable — the ripple grows smoothly */
/* Paint API redraws on every animation frame as --ripple-radius changes */
```

5. Browser Support Reference

API / Feature	Chrome	Edge	Firefox	Safari	Notes
`@property`	105+	105+	128+	16.4+	Baseline 2024 — use in production
`CSS.registerProperty()`	78+	79+	128+	16.4+	Prefer @property — declarative and collocated
`CSS.paintWorklet / Paint API`	65+	79+	No	No	Chrome/Edge only. Always provide a CSS fallback
`element.attributeStyleMap`	66+	79+	No	No	Chrome/Edge only. Graceful degrade to element.style
`element.computedStyleMap()`	66+	79+	No	No	Chrome/Edge only. Degrade to getComputedStyle()
`CSS.px(), CSS.em() etc.`	66+	79+	No	No	Factory functions — Chrome/Edge only
`Layout Worklet`	Flag	No	No	No	Experimental only — do not ship
`Animation Worklet`	Flag	No	No	No	Partially superseded by Scroll-Driven Animations

Chapter Summary

Concept	Key point
`@property`	Registers a custom property with a type (syntax), initial value, and inheritance flag. Enables CSS transitions and animations on custom properties used inside gradients, transforms, and clip-paths — which the browser cannot interpolate without type information. Baseline 2024 — use in production today.
`syntax values`	Common types: <length>, <number>, <percentage>, <color>, <angle>, <transform-list>. Combinators: + for space-separated list, # for comma-separated, | for keyword union. * means "any" — same as unregistered.
`Paint API`	Write a JavaScript class extending PaintRenderingContext2D, register it with CSS.paintWorklet.addModule(), then use paint(name) anywhere CSS expects an image. inputProperties declares which custom properties the worklet reads. inputArguments lets you pass values directly in paint(). Chrome/Edge only — always provide a CSS fallback.
`Worklet isolation`	Paint worklets have no DOM, network, or global state access. They may run on any thread. The worklet receives: ctx (canvas-like context), size (width/height of the element), properties (Typed OM values for inputProperties). Write as a pure function of its inputs.
`Typed OM`	element.attributeStyleMap and element.computedStyleMap() replace element.style and getComputedStyle() with typed objects. CSS.px(42) returns CSSUnitValue {value: 42, unit: 'px'} — arithmetic without parsing. Chrome/Edge only — degrade gracefully with feature detection.
`@property + Paint API`	The two Houdini APIs work together: register a custom property with @property so it can be animated, then read it in a paint worklet. The worklet redraws on every animation frame as the property value changes. This pattern enables CSS-driven animated custom backgrounds.
`Support strategy`	@property — ship everywhere, Baseline 2024. Paint API — Chrome/Edge progressive enhancement, always provide a plain CSS fallback. Typed OM — Chrome/Edge tooling enhancement, degrade to string-based CSSOM. Feature-detect with 'paintWorklet' in CSS or 'attributeStyleMap' in element.

EXERCISES

1. **Animated gradient angle:** Register a `--gradient-angle` custom property with `syntax: '<angle>'`. Apply it to a `conic-gradient()` background. Write a `@keyframes` rule that animates `--gradient-angle` from `0deg` to `360deg` over 6 seconds. Confirm the gradient spins smoothly. Then remove the `@property` block and observe that the gradient jumps instead of spinning — this demonstrates exactly why type registration is necessary.

2. **Spotlight button:** Register `--spotlight-x` and `--spotlight-y` as `<percentage>` properties. Apply a `radial-gradient(circle at var(--spotlight-x) var(--spotlight-y), ...)` to a button's background. Use a `mousemove` listener and `attributeStyleMap.set()` (with a string fallback for Firefox/Safari) to update both properties to the cursor's position within the button. The result: a spotlight that follows the cursor, with smooth interpolation on quick moves.

3. **Checkerboard paint worklet:** Write a complete paint worklet file called `checkerboard.js`. The worklet should read `--cb-cell-size` (an integer, default 20) and `--cb-color-a`, `--cb-color-b` from inputProperties. Draw the checkerboard pattern using `ctx.fillRect`. Register it from a script, provide a solid-color fallback in CSS, and use `@supports (background: paint(x))` to enable the worklet background. Apply it to a full-page `div` and verify it tiles correctly when you resize the browser.

4. **Progress ring with @property + Paint API:** Register `--ring-progress` as a `<number>` with initial value `0`. Write a `circle-progress` paint worklet that draws an arc from -90° based on the progress value (0–100). Wire up a button that transitions `--ring-progress` from 0 to 75. Because the property is registered, the arc should animate smoothly through each intermediate value. Add a second button that resets it back to 0.

5. **Typed OM arithmetic helper:** Write a utility function `addToProperty(el, prop, delta)` that: (1) reads the current value of `prop` from `el.computedStyleMap()` if available, else parses `getComputedStyle(el).getPropertyValue(prop)`; (2) adds `delta` (in the same unit) to the value; (3) writes it back via `el.attributeStyleMap.set()` if available, else `el.style.setProperty()`. Test it by repeatedly calling `addToProperty(box, 'width', 10)` on click — the box should grow by 10px each click in all browsers, using the typed path in Chrome and the fallback path in Firefox/Safari.

Next: Chapter 9 — CSS Performance. How browsers paint and composite layers; what triggers repaints vs reflows; the compositor thread and why `transform` and `opacity` are fast; `will-change` and when it helps vs hurts; `contain` for layout and paint isolation; `content-visibility: auto` for offscreen content; `@layer` and the cascade engine; and DevTools workflows for diagnosing and eliminating jank.

CHAPTER 9

Performance — Repaints, Compositing, will-change

Chapter 9 — CSS Performance: Repaints, Compositing, and will-change

Smooth animation means 60 frames per second — the browser has 16.7 ms to produce each frame. Every frame that takes longer than that produces visible jank. Understanding what CSS properties trigger which parts of the browser's rendering pipeline is the foundation of performance work: once you know why something is slow, the fix is usually a one-line change.

1. The Rendering Pipeline

Every visual change goes through some or all of these stages. The further left a property change triggers, the more work the browser must repeat and the more expensive it is.

A property change can enter the pipeline at different stages:

- **Layout (reflow):** changing geometry properties — `width`, `height`, `margin`, `padding`, `font-size`, `display`. Forces everything below it: layout → paint → composite.
- **Paint (repaint):** changing visual-only properties — `color`, `background-color`, `border-color`, `box-shadow`. Skips layout: paint → composite.
- **Composite only:** changing `transform` and `opacity` on composited layers. Skips layout and paint entirely. Runs on the GPU compositor thread.

2. Property Rendering Cost Reference

Property / Change	Pipeline entry	Cost	Notes
GEOMETRY — TRIGGERS LAYOUT			

Property / Change	Pipeline entry	Cost	Notes
`width, height, min/max-*`	Layout	High	Full reflow from the changed element outward
`margin, padding, border-width`	Layout	High	Affects box model — sibling and parent geometry can shift
`top, left, right, bottom (position)`	Layout	High	On static/relative/absolute — triggers full reflow
`font-size, line-height, font-family`	Layout	High	Changes text metrics — everything reflows
`display, position, float`	Layout	High	Changes the formatting context — major reflow
`overflow (changing to/from hidden)`	Layout	High	Can change scrollable area and clipping
VISUAL — TRIGGERS PAINT ONLY			
`color, background-color`	Paint	Medium	Repaints the element; no geometry change
`background-image (non-gradient)`	Paint	Medium	Image decode may add cost
`border-color, outline, text-decoration`	Paint	Medium	Visual only — no layout impact
`box-shadow, text-shadow`	Paint	Medium	Expensive to paint — blur is rasterised each frame
`border-radius (on non-composited element)`	Paint	Medium	Clips during paint — use on composited layers to push to GPU
COMPOSITE ONLY — GPU, FAST			
`transform (translate, rotate, scale, skew)`	Composite	Low ✓	GPU compositing — no layout or paint. Use for ALL motion.
`opacity`	Composite	Low ✓	GPU compositing. Composited layer alpha fade.
`filter (on composited element)`	Composite	Low ✓	GPU filter. blur() is still costly — use sparingly.
`clip-path (on composited element)`	Composite	Low ✓	With will-change: transform — stays on compositor thread

3. Main Thread vs Compositor Thread

The browser has two threads relevant to rendering. The **main thread** runs JavaScript, style calculation, layout, and paint. The **compositor thread** runs on the GPU and handles compositing. When an animation only needs compositing, the GPU thread handles every frame independently — even if the main thread is busy running JavaScript.

Main Thread (CPU)	Compositor Thread (GPU)

- JavaScript execution
- Style recalculation
- Layout (reflow)
- Paint (rasterise) — can be threaded

Long JS tasks block layout and paint — causes jank even for CSS animations that trigger layout

- transform animations — never blocks
- opacity animations — never blocks
- Scroll compositing — fast scroll
- Layer compositing — combine layers

Runs independently — JS-heavy main thread cannot block a compositor-only animation

```
/* ── The golden rule: animate only transform and opacity ──────── */

/* ✗ Triggers layout → paint → composite on every frame */
@keyframes slide-bad {
    from { left: 0; }
    to   { left: 200px; }
}

/* ✓ Composite only — GPU, cannot jank even under main-thread load */
@keyframes slide-good {
    from { transform: translateX(0); }
    to   { transform: translateX(200px); }
}

/* ✗ Triggers repaint on every frame */
@keyframes fade-bad {
    from { background-color: oklch(0.55 0.22 248); }
    to   { background-color: oklch(0.55 0.22 300); }
}

/* ✓ Fade in/out with opacity — compositor only */
@keyframes fade-good {
    from { opacity: 0; }
    to   { opacity: 1; }
}

/* ✓ Use translateY instead of top/margin-top for vertical movement */
.toast {
    transform:  translateY(-100%);
    transition: transform 0.3s ease, opacity 0.3s ease;
    opacity:    0;
}
.toast.visible {
    transform: translateY(0);
    opacity:   1;
}
```

4. will-change

`will-change` hints to the browser that an element is about to be animated, allowing it to promote the element to its own compositor layer in advance. This eliminates the one-frame cost of the first animation frame where the layer would otherwise need to be created mid-animation.

Is this element animated with transform or opacity?

→ Yes: Is the animation triggered immediately on load (not on hover/scroll)?

→ Yes, starts immediately:

> Add `will-change: transform` or `will-change: opacity`. The browser pre-promotes the layer before the first frame.

→ No, triggered later (hover, click, scroll):

> Apply `will-change` just before the animation starts (add class in JS) and remove it afterwards. Or accept the negligible first-frame cost — often not worth the complexity.

→ No: Does animating it cause visible jank even with transform/opacity?

> Profile first with DevTools. If a layer boundary issue is confirmed, use `will-change: transform` as a last resort — but this has a memory cost and can backfire.

When NOT to use will-change:

> On every element pre-emptively. Each promoted layer uses GPU memory. Overuse can exhaust GPU memory, causing worse performance than no optimisation at all.

> As a fix for layout or paint jank. `will-change` only helps compositor-stage work. Fix the root cause (use transform instead of left/top) rather than promoting layers.

```css
/* ── will-change values ──────────────────────────────────── */

will-change: transform;
/* Promotes to GPU layer. Use for translateX/Y/Z, rotate, scale animations. */

will-change: opacity;
/* Promotes to GPU layer for opacity fade animations. */

will-change: transform, opacity;
/* Comma-separated for multiple properties. */

will-change: scroll-position;
/* Hint that the element will be scrolled — browser optimises scroll compositing. */

will-change: contents;
/* Hint that the element's children will change. Lets the browser optimise */
/* subtree rendering but does NOT promote to its own compositor layer. */

will-change: auto;
/* Default. Remove will-change after the animation to release the GPU memory: */

/* ── Pattern: apply and remove around animations ──────────── */
.card {
    transition: transform 0.25s ease;
}
.card:hover {
    will-change: transform;
    transform:   translateY(-4px) scale(1.02);
}
/* :hover is a natural scope — GPU layer is promoted on hover-start, */
/* released on hover-end. No JavaScript needed. */

/* ── JavaScript: apply/remove pattern for click animations ─── */
btn.addEventListener('pointerdown', () => {
    btn.style.willChange = 'transform';
});
btn.addEventListener('transitionend', () => {
    btn.style.willChange = 'auto';   /* release GPU memory after animation */
});

/* ── The 3D hack — old pattern, mostly unnecessary now ──────── */
/* You may see: transform: translateZ(0) or translate3d(0,0,0)  */
/* These force GPU promotion without will-change.               */
/* Use will-change: transform instead — it's explicit and clear. */
/* The translateZ(0) hack is only needed for very old browsers.  */
```

5. CSS Containment — contain

The `contain` property tells the browser that an element and its subtree are independent from the rest of the page. This lets the browser skip large portions of the rendering pipeline when only the contained element changes — a huge win for complex, dynamic UIs.

📐

`contain: layout`

The element's children cannot affect the layout of elements outside it. Internal reflows stay contained. Use on widgets that resize their own contents dynamically.

🎨

`contain: paint`

The element's children are clipped to its border box and cannot visually overflow. Tells the browser it never needs to repaint outside the box when internal content changes.

🔡

`contain: style`

Counters and quote scopes are contained. Rarely needed directly — mostly a concern for CSS custom properties used with @property.

📦

`contain: size`

The element's size does not depend on its children. The browser can skip children when computing the element's size. Almost always paired with a fixed size.

↔️

`contain: inline-size`

Like size but inline-axis only. The element's inline size does not depend on children. This is what container-type: inline-size applies under the hood.

⚡

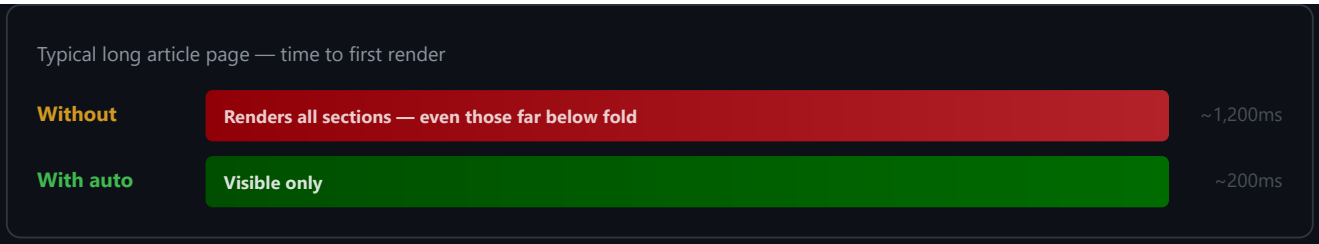
`contain: content`

Shorthand for layout + paint + style. The most common production value. Does not contain size, so the element can still grow with its children.

```css
/* ── contain syntax ──────────────────────────────────────── */

contain: none;            /* default — no containment */
contain: layout;          /* layout isolation only */
contain: paint;           /* paint isolation only — clips overflow */
contain: layout paint;    /* combine any subset */
contain: content;         /* layout + paint + style — most useful */
contain: strict;          /* layout + paint + style + size — most restrictive */

/* ── Practical use on a card grid ────────────────────────── */
.card-grid {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(280px, 1fr));
    gap: 1rem;
}

.card {
    contain: content;
    /* Each card is now isolated: */
    /* - Updating one card's content triggers layout/paint for that card only */
    /* - Other cards are not re-laid-out */
    /* - Browser can skip off-screen cards more aggressively */
}

/* ── contain: layout enables absolute positioning scoping ─── */
/* An absolutely-positioned child is contained within the element */
/* just like position: relative — because contain: layout creates a */
/* containing block for absolute positioning */
.widget {
    contain: layout;   /* absolute children are scoped to this element */
}

/* ── contain vs overflow: hidden ─────────────────────────── */
/* overflow: hidden clips visually but does NOT tell the browser   */
/* anything about layout independence.                             */
/* contain: paint clips visually AND declares layout independence. */
/* Prefer contain: content over overflow: hidden for performance.  */
```

6. content-visibility and contain-intrinsic-size

`content-visibility: auto` is the most impactful single-line performance optimisation for long pages. It tells the browser to skip rendering — layout, paint, and compositing — for any element that is not currently visible in the viewport.




```
/* ── content-visibility values ──────────────────────────────── */

content-visibility: visible;   /* default — no skipping */
content-visibility: hidden;    /* always skip rendering — like display:none but keeps la
content-visibility: auto;      /* skip when off-screen, render when approaching viewport

/* ── content-visibility: auto on article sections ──────────── */
.article-section {
    content-visibility:    auto;
    contain-intrinsic-size: auto 600px;
    /* contain-intrinsic-size tells the browser how tall to treat the section */
    /* while it is not rendered. Without it, the scrollbar would jump as */
    /* sections get rendered/unrendered, because their height is unknown. */
    /* 'auto' remembers the last-known size; '600px' is the initial estimate. */
}

/* ── contain-intrinsic-size syntax ─────────────────────────── */
contain-intrinsic-size: 800px;          /* width and height both 800px */
contain-intrinsic-size: 100% 400px;     /* width: 100%, height: 400px */
contain-intrinsic-size: auto 400px;     /* auto: use remembered size if available */
contain-intrinsic-block-size: auto 400px; /* logical property form */

/* ── content-visibility: hidden for off-screen panels ──────── */
/* Like display:none but preserved layout — ideal for tab panels, drawers */
.tab-panel          { content-visibility: hidden; }
.tab-panel.active   { content-visibility: visible; }
/* The panel's layout box still exists — no layout shift when shown */
/* The panel's contents are not rendered until made visible */

/* ── When NOT to use content-visibility: auto ──────────────── */
/* - Elements that contain sticky or fixed positioned children (breaks stacking) */
/* - Elements that must be included in fragment navigation (#anchor links) */
/* - The above-the-fold content — always rendered, no benefit */
/* - Elements that respond to IntersectionObserver (observer fires on skip) */
```

7. Avoiding Layout Thrash from JavaScript

```
/* Layout thrash: reading a layout property forces the browser to flush */
/* pending style changes and run layout immediately — interleaving reads */
/* and writes causes repeated reflows in the same frame */

/* ✗ Layout thrash — read/write alternation forces 3 separate reflows */
elements.forEach(el => {
    const height = el.offsetHeight;      /* READ — forces layout flush */
    el.style.height = height + 10 + 'px';  /* WRITE — invalidates layout */
});
/* Each iteration: write from previous iteration → read forces flush → layout runs */

/* ✓ Batch reads, then batch writes */
const heights = elements.map(el => el.offsetHeight);  /* all READs first */
elements.forEach((el, i) => {
    el.style.height = heights[i] + 10 + 'px';        /* all WRITEs after */
});
/* One layout flush, not N */

/* Properties that force layout when READ: */
/* offsetWidth, offsetHeight, offsetTop, offsetLeft                */
/* scrollWidth, scrollHeight, scrollTop, scrollLeft                */
/* clientWidth, clientHeight, clientTop, clientLeft                */
/* getBoundingClientRect(), getComputedStyle()                     */
/* innerText (triggers layout for text measurement)                */

/* ✓ Use requestAnimationFrame to batch DOM work in animation loops */
function animate() {
    const reads = [];
    elements.forEach(el => reads.push(el.getBoundingClientRect()));
    elements.forEach((el, i) => {
        el.style.transform = `translateX(${reads[i].width}px)`;
    });
    requestAnimationFrame(animate);
}
```

8. DevTools Performance Workflow

Tool / Panel	How to open	What to look for
Performance panel	`F12 → Performance → Record`	Purple "Layout" bars and green "Paint" bars in the main thread flamechart. Any bar taking more than 16ms in a frame is a jank

Tool / Panel	How to open	What to look for
		source. Look for repeated purple+green pairs triggered by JS.
Rendering → Paint Flashing	`F12 → ⋮ → Rendering → Paint flashing`	Green overlay shows any area that was repainted. Should only flash when content genuinely changes. Constant flashing on scroll or hover means expensive repaints on a non-composited layer.
Rendering → Layer Borders	`F12 → ⋮ → Rendering → Layer borders`	Orange borders = composited layer boundaries. Each orange border is a GPU-promoted layer. Check that will-change and 3D transforms are creating the expected layers — and not too many.
Layers panel	`F12 → ⋮ → More tools → Layers`	3D visualisation of compositor layers. Each layer shows its memory usage. Too many layers or unexpectedly large layers indicate over-promotion. Hover a layer to see why it was promoted.
Rendering → FPS meter	`F12 → ⋮ → Rendering → Frame Rendering Stats`	Live FPS counter and GPU memory overlay. Aim for 60fps during animation. Drops below 30fps are visibly jerky.
Performance Insights	`F12 → Performance Insights`	Guided analysis — flags Long Animation Frames, layout shifts, and render-blocking resources with specific recommendations. Faster than reading raw flamecharts.
coverage tab	`F12 → ⋮ → More tools → Coverage`	Shows what percentage of loaded CSS is actually used. Unused CSS still gets parsed, increasing style recalculation cost. High unused CSS % suggests code-splitting opportunity.

> **Profile on a slow device.** Animations that look smooth on a development machine can jank badly on a mid-range phone. Use DevTools CPU throttling (6× slowdown) to simulate realistic conditions, or connect a real Android device via USB for remote debugging. Most animation jank is invisible on fast hardware.

9. Practical Performance Patterns

```css
/* ── Pattern 1: GPU-safe modal animation ──────────────────────── */
.modal-backdrop {
    position: fixed;
    inset:    0;
    opacity:  0;
    transition: opacity 0.25s ease;
}
.modal-panel {
    transform:  translateY(24px) scale(0.97);
    opacity:    0;
    transition: transform 0.3s cubic-bezier(0.16, 1, 0.3, 1),
                opacity   0.2s ease;
}
.modal.open .modal-backdrop,
.modal.open .modal-panel {
    opacity:   1;
    transform: none;
}
/* Both animations run on the compositor thread — zero layout impact */

/* ── Pattern 2: Scroll-optimised infinite list ─────────────────── */
.list-item {
    content-visibility:    auto;
    contain-intrinsic-size: auto 80px;   /* estimated row height */
    contain:               content;
}
/* 1000 list items render only the ~10 visible ones + buffer */
/* No virtualisation library required */

/* ── Pattern 3: Safe hover card lift ──────────────────────────── */
.card {
    transition: transform 0.2s ease, box-shadow 0.2s ease;
}
.card:hover {
    transform:  translateY(-4px);
    box-shadow: 0 16px 40px oklch(0 0 0 / 0.25);
}
/* transform is compositor; box-shadow triggers repaint but only once per hover */
/* Acceptable: one-time repaint on hover is far cheaper than per-frame repaint */

/* ── Pattern 4: Contained sidebar widget ──────────────────────── */
.sidebar-widget {
    contain: content;
}
```

```css
/* A stock ticker updating every second triggers layout/paint only within */
/* the widget — never causes the rest of the page to re-layout */

/* ── Pattern 5: Carousel with GPU compositing ──────────────────── */
.carousel-track {
    display:  flex;
    transition:transform 0.5s cubic-bezier(0.77, 0, 0.175, 1);
    will-change:transform;  /* pre-promote — carousel always animating */
}
/* JS only ever sets transform: translateX(Npx) — never left/scrollLeft */
/* Smooth even with a heavy main-thread JS load */

/* ── Pattern 6: Reduce motion safely ──────────────────────────── */
@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
`Rendering pipeline`	Style → Layout → Paint → Composite. A property change enters the pipeline at a specific stage and forces everything after it. Geometry properties (width, top, margin) trigger layout. Visual properties (color, background-color, box-shadow) trigger paint. transform and opacity trigger only compositing.
`Compositor thread`	transform and opacity animations run on the GPU compositor thread independently of the main thread. Long JavaScript tasks cannot block them. All other properties animate on the main thread and can be blocked by JS. This is why transform/opacity are the animation properties of choice.
`will-change`	Hint to promote an element to its own compositor layer in advance of animation. Use will-change: transform for GPU animations that start immediately on load. Apply on :hover for triggered animations — the CSS scope naturally manages the layer lifetime. Remove after animation completes (style.willChange = 'auto') to release GPU memory. Never apply site-wide as a blanket optimisation.
`contain`	Isolates an element's subtree from the rest of the page's rendering. contain: content (= layout + paint + style) is the most useful production value. Limits the scope of reflow and repaint to within the element. Also creates a containing block for absolute positioning and a stacking context.
`content-visibility: auto`	Skips rendering (layout, paint, composite) for off-screen elements. Biggest single-line win for long-scrolling pages — can cut initial render time by 5–10×. Always pair with contain-intrinsic-size: auto [estimate] to prevent scrollbar jumping as elements are rendered/unrendered. Do not use on above-fold content or elements with sticky children.
`Layout thrash`	Interleaving DOM reads (offsetHeight, getBoundingClientRect) and DOM writes (style changes) in JavaScript forces the browser to flush and re-run layout repeatedly in a single frame. Fix by batching: all reads first, all writes after. In animation loops, use requestAnimationFrame to ensure writes happen before the next frame's reads.
`DevTools workflow`	Performance panel → look for purple Layout + green Paint bars. Rendering → Paint Flashing → anything that flashes constantly needs investigation. Rendering → Layer Borders → count compositor layers. Layers panel → check memory per layer. Profile at 6× CPU throttle to simulate real devices.

EXERCISES

1. **Pipeline audit — before and after:** Create a card that slides in on hover. Write version A using `margin-left` for motion and `background-color` for a highlight effect. Write version B using `translateX` for motion and `opacity` for fade. Open DevTools → Rendering → Paint Flashing. Hover over both versions and observe the difference in flash frequency. Record a Performance trace for each and compare how many purple Layout bars appear per hover interaction.
2. **will-change budget test:** Create a grid of 200 cards. Apply `will-change: transform` to all 200. Open DevTools → Layers panel and note the total GPU memory consumed. Then remove `will-change` from all cards and add it only on `:hover`. Reload the Layers panel and compare GPU memory at rest. Document the difference — this illustrates why blanket `will-change` is counterproductive.
3. **content-visibility benchmark:** Build a page with 500 article sections, each ~300px tall with a heading, two paragraphs, and an image. Measure Time to First Byte and the Rendering timeline in Performance (without `content-visibility`). Then add `content-visibility: auto; contain-intrinsic-size: auto 300px` to each section and re-measure. Note the render time difference and verify that scrolling remains smooth and the scrollbar height is stable.
4. **Layout thrash fix:** Write a function that reads each list item's `offsetTop` and then sets a `--stagger-delay` custom property based on that value (for an entrance animation). Write version A that interleaves reads and writes in a single forEach loop. Write version B that batches reads first, then writes. Record a Performance trace for each and show the reflow count difference in the Summary tab.
5. **contain: content in a live dashboard:** Build a mock dashboard with four widgets: a line chart that updates every 500ms, a scrolling news feed, a counter, and a static map placeholder. Without `contain`, observe (via Paint Flashing) how updating the counter repaints other widgets. Add `contain: content` to each widget and confirm that paint flashing is now contained to only the widget that actually changed its content.

Next: Chapter 10 — CSS in Component Frameworks. CSS Modules — scoped class names, composes, and local vs global scope; CSS-in-JS concepts — inline styles, style objects, and tagged template literals; utility-first CSS (Tailwind thinking) and when it wins vs loses; styling strategy trade-offs in React, Vue, and Svelte; and how modern bundlers handle CSS — tree-shaking, code splitting, and critical CSS extraction.

CHAPTER 10

CSS in Component Frameworks

Chapter 10 — CSS in Component Frameworks

Component-based frameworks — React, Vue, Svelte, and others — changed how we think about styling. Instead of one global stylesheet ruling the whole page, each component owns its styles. Four main approaches emerged: plain CSS with naming conventions (BEM), CSS Modules, CSS-in-JS, and utility-first CSS. Each solves the scoping and co-location problem differently, with different trade-offs for performance, DX, and long-term maintainability.

Global CSS + BEM

✓ No build step. Full CSS feature set. Easy to inspect.

✗ Manual scoping discipline. Global leaks possible. Naming overhead.

CSS Modules

✓ Automatic scoping. Real CSS. Works with any bundler.

✗ Co-location requires separate file. Dynamic styles need inline or data attributes.

CSS-in-JS

✓ JS-driven dynamic styles. Typed props. Zero dead CSS.

✗ Runtime cost. Larger bundle. Server rendering complexity.

Utility-first

✓ Tiny CSS bundle. No naming. Fast iteration. Design system built-in.

✗ Verbose markup. Hard to extract semantic meaning. Custom CSS still needed.

1. CSS Modules

CSS Modules processes a `.module.css` file at build time and generates unique class names scoped to that file. The component imports the generated name map as a JavaScript object and applies names via `styles.className`. The generated output is plain CSS — no runtime overhead.

```css
/* ── Card.module.css ──────────────────────────────────────── */

.card {
    background:    var(--color-surface);
    border:        1px solid var(--color-border);
    border-radius: 8px;
    padding:       1.25rem;
}

.title {
    font-size:   1.1rem;
    font-weight: 700;
    color:       var(--color-text);
    margin:      0 0 0.5rem;
}

/* composes — inherit styles from another class (same file or another module) */
.cardHighlighted {
    composes: card;                          /* same file */
    border-color: var(--color-accent);
}

.body {
    composes: prose from './typography.module.css';  /* another module */
    color:    var(--color-text-sub);
}

/* :global — escape scoping for third-party overrides */
.card :global(.react-tooltip) {
    font-size: 0.8rem;
}

/* :local — explicitly scoped (the default, rarely needed) */
:local(.icon) { width: 1.25rem; }
```

```jsx
/* ── Card.jsx — React component using CSS Modules ──────────── */

import styles from './Card.module.css';

export function Card({ title, body, highlighted }) {
    return (
        <div className={highlighted ? styles.cardHighlighted : styles.card}>
            <h2 className={styles.title}>{title}</h2>
            <p  className={styles.body}>{body}</p>
        </div>
    );
}

/* Build output — generated class names (hashes vary per bundler): */
/* .card      → ._card_x7k2p_1     */
/* .title     → ._title_x7k2p_12   */
/* No collision with any other component's .card or .title class   */

/* ── Composing classes conditionally (clsx / classnames helper) ── */
import clsx from 'clsx';

const className = clsx(
    styles.card,
    highlighted && styles.highlighted,
    size === 'lg' && styles.large,
);

/* ── Dynamic values: CSS custom properties are the clean solution — */
.progress {
    width: var(--progress-width);  /* set from JS via inline style */
}

/* In JSX — inline style sets the custom property, module CSS reads it */
<div className={styles.progress} style={{ '--progress-width': `${pct}%` }} />
```

> **CSS Modules + custom properties** is a powerful combination. The module handles scoping and static styles; custom properties bridge the gap for values that change based on JavaScript state. No runtime style injection needed.

2. CSS-in-JS

CSS-in-JS libraries write CSS directly inside JavaScript files, often as tagged template literals or style objects. They generate and inject styles at runtime (or at build time in newer zero-runtime approaches). The main appeal is that component props can drive styles with full JavaScript logic — no custom properties needed as an intermediary.

```
/* ── Styled-components (runtime CSS-in-JS) ──────────────────── */
import styled from 'styled-components';

const Button = styled.button`
    padding:          0.6rem 1.2rem;
    border-radius:    6px;
    font-weight:      700;
    background:       ${props => props.variant === 'primary'
        ? 'oklch(0.55 0.22 248)'
        : 'oklch(0.20 0.02 248)'};
    color:            white;
    border:           1px solid ${props => props.variant === 'primary'
        ? 'transparent'
        : 'oklch(0.30 0.05 248)'};

    &:hover {
        filter: brightness(1.1);
    }
`;

/* Usage */
<Button variant="primary">Save</Button>
<Button>Cancel</Button>

/* styled-components injects a <style> tag at runtime with a hashed class: */
/* .sc-abc123 { padding: 0.6rem 1.2rem; background: oklch(0.55 0.22 248); } */
```

```
/* ── Emotion — similar API, popular in design systems ──────────── */
import { css, cx } from '@emotion/css';

const cardStyle = css`
    background:    var(--color-surface);
    border:        1px solid var(--color-border);
    border-radius: 8px;
    padding:       1.25rem;
`;

const highlightedStyle = css`
    border-color: var(--color-accent);
`;

/* cx merges class names, deduplicating where possible */
<div className={cx(cardStyle, highlighted && highlightedStyle)}>

/* ── Zero-runtime CSS-in-JS (the modern shift) ─────────────────── */
/* Libraries: vanilla-extract, Linaria, Panda CSS, StyleX (Meta) */
/* These extract CSS at build time — no <style> injection at runtime */
/* Result: plain .css files, zero JS overhead, full SSR compatibility */

/* ── vanilla-extract example ──────────────────────────────────── */
/* card.css.ts — TypeScript file, processed at build time */
import { style, styleVariants } from '@vanilla-extract/css';

export const card = style({
    background:   'var(--color-surface)',
    borderRadius: 8,
    padding:      '1.25rem',
});

export const variants = styleVariants({
    default:     { border: '1px solid var(--color-border)' },
    highlighted: { border: '1px solid var(--color-accent)'  },
});

/* Build output: Card.css (static file) + hashed class names */
/* TypeScript knows the exact class names — autocomplete works */
```

3. Utility-First CSS

Utility-first CSS (popularised by Tailwind CSS) replaces custom class names with small, single-purpose classes. Instead of writing a `.card` class that sets background, padding, and border-radius, you apply `bg-surface p-5 rounded-lg` directly in the markup. The CSS bundle stays tiny because only classes actually used in the HTML are generated — unused utilities are tree-shaken away.

```
/* ── How Tailwind generates its CSS ──────────────────────────── */
/* tailwind.config.js — scans all template files for class names */

export default {
    content: ['./src/**/*.{html,js,jsx,ts,tsx,vue,svelte}'],
    theme: {
        extend: {
            colors: {
                accent: {
                    500: 'oklch(0.60 0.18 248)',
                    400: 'oklch(0.70 0.18 248)',
                },
            },
        },
    },
};

/* ── HTML / JSX using Tailwind classes ──────────────────────── */
/* A card with title and body — no .css file written at all */
<div className="bg-surface border border-border rounded-lg p-5">
    <h2 className="text-lg font-bold text-text mb-2">Title</h2>
    <p  className="text-sm text-text-sub">Body text</p>
</div>

/* ── Extracting a reusable component class with @apply ──────── */
/* Use sparingly — @apply is useful for prose and form elements  */
/* Not recommended for general component styling (defeats the point) */

.btn {
    @apply inline-flex items-center gap-2 px-4 py-2 rounded-md font-medium;
    @apply bg-accent-500 text-white transition-colors;
}
.btn:hover {
    @apply bg-accent-400;
}

/* ── Arbitrary values — escape hatch for one-off values ──────── */
/* Wrap in square brackets for values not in the design system */
<div className="w-[342px] top-[17px] bg-[oklch(0.55_0.22_248)]"></div>

/* ── Dynamic classes — the key Tailwind gotcha ──────────────── */
/* ✗ This does NOT work — Tailwind cannot see dynamically constructed strings */
const cls = `bg-${color}-500`;   /* 'bg-blue-500' — not scanned, not generated */

/* ✓ Use full class names in a lookup object */
const colorMap = {
    blue:  'bg-blue-500',
```

```
    green: 'bg-green-500',
    red:   'bg-red-500',
};
const cls = colorMap[color];   /* full strings — Tailwind sees all three at scan time */
```

4. Scoped Styles in Vue and Svelte

```
/* ── Vue Single File Component (SFC) ──────────────────────────── */
<!-- Card.vue -->

<template>
    <div :class="['card', highlighted && 'card--highlighted']">
        <h2 class="card__title">{{ title }}</h2>
    </div>
</template>

<style scoped>
/* scoped adds a unique data attribute to all selectors */
/* .card → .card[data-v-a3f2b1] — no global leak */

.card {
    background:    var(--color-surface);
    border-radius: 8px;
    padding:       1.25rem;
}
.card--highlighted {
    border: 1px solid var(--color-accent);
}

/* Penetrate scoping to reach child component's DOM elements */
.card :deep(.icon) {
    color: var(--color-accent);
}

/* Inject global rules from inside the SFC */
:global(.some-third-party) {
    font-size: 0.9rem;
}
</style>

/* ── Vue CSS v-bind — JS values directly in scoped CSS ──────── */
<script setup>
import { ref } from 'vue';
const accentColor = ref('oklch(0.55 0.22 248)');
</script>

<style scoped>
.card {
    border-color: v-bind(accentColor);   /* reactive JS ref → CSS value */
    /* Vue sets this as an inline custom property at runtime */
```

```
}
</style>
```

```
/* ── Svelte scoped styles ──────────────────────────────────── */
<!-- Card.svelte -->

<script>
    export let highlighted = false;
</script>

<div class="card" class:highlighted>
    <slot />
</div>

<style>
/* All styles in .svelte are automatically scoped — no keyword needed */
/* Svelte adds svelte-HASH to selectors at compile time */

.card {
    background:    var(--color-surface);
    border-radius: 8px;
    padding:       1.25rem;
}

/* class: directive adds 'highlighted' class when the prop is truthy */
.highlighted {
    border: 1px solid var(--color-accent);
}

/* :global — escape Svelte scoping */
:global(body) {
    font-family: system-ui, sans-serif;
}
</style>
```

5. How Bundlers Handle CSS

```
/* ── Vite CSS handling (the modern default) ──────────────────── */

/* *.module.css → CSS Modules (automatic) */
/* *.css        → global CSS injected via <style> tag in dev, extracted in build */
/* *.module.css with ?inline → returns CSS string, no injection */

/* vite.config.js — CSS options */
export default {
    css: {
        modules: {
            localsConvention: 'camelCase',    /* .my-class → styles.myClass */
            generateScopedName: '[name]__[local]__[hash:5]',
        },
        preprocessorOptions: {
            scss: { additionalData: '@use "./src/tokens" as *;' },
        },
    },
};

/* ── Code splitting — CSS follows JS chunks ──────────────────── */
/* When a route is lazy-loaded, its CSS is also in a separate chunk */
/* The browser fetches the CSS chunk only when the route is visited */
/* This keeps initial CSS payload small for large apps */

/* ── Critical CSS extraction ─────────────────────────────────── */
/* Tools: critters (used by Angular), penthouse, critical */
/* Extract above-fold CSS → inline in <head> → defer rest */
/* Result: page renders without waiting for the full CSS bundle */

/* Manual critical CSS pattern */
<!-- In <head> -->
<style> /* critical above-fold CSS here */ </style>
<link rel="preload" href="/styles.css" as="style" onload="this.rel='stylesheet'">
<noscript><link rel="stylesheet" href="/styles.css"></noscript>

/* ── CSS tree-shaking with PurgeCSS / Tailwind ──────────────── */
/* Tailwind's JIT engine generates only the classes actually used in HTML */
/* PurgeCSS does the same post-build for any CSS framework */

import purgecss from '@fullhuman/postcss-purgecss';

export default {
    plugins: [
        purgecss({
```

```
            content: ['./src/**/*.html', './src/**/*.jsx'],
            safelist: [/^js-/],   /* preserve classes added dynamically by JS */
        }),
    ],
};
```

6. Choosing an Approach

Approach	Runtime cost	Dynamic styles	Scoping	DX	Best for
Global CSS + BEM	Zero	Custom props	Manual	Familiar	Small sites, no framework
CSS Modules	Zero	Custom props	Automatic	Good	React apps, design systems
styled-components / Emotion	Runtime	Full JS	Automatic	Excellent	Highly dynamic UIs
vanilla-extract / Linaria	Zero	Custom props	Automatic	Typed	Large-scale typed codebases
Tailwind CSS	Zero	Lookup maps	No leaks	Fast iterations	Rapid prototyping, startups
Vue scoped / Svelte	Zero	v-bind / stores	Automatic	Native	Vue/Svelte projects

> **The modern consensus:** runtime CSS-in-JS (styled-components, Emotion) peaked around 2020 and is declining in new projects due to server rendering complexity and performance cost. Zero-runtime approaches (vanilla-extract, CSS Modules + custom properties, Tailwind) are the current direction for performance-sensitive applications.

Chapter Summary

Concept	Key point
CSS Modules	Build-time scoping via hashed class names. Import as a JS object (styles.card). Use composes: for inheritance. Bridge dynamic values with CSS custom properties set as inline styles. Zero runtime cost. Works with any bundler that supports CSS Modules.
CSS-in-JS (runtime)	styled-components and Emotion inject <style> tags at runtime driven by JS props. Full JavaScript logic in styles. Cost: runtime overhead, larger bundle, SSR complexity. Declining for new projects.
Zero-runtime CSS-in-JS	vanilla-extract, Linaria, StyleX extract CSS at build time. TypeScript types for class names. Same scoping as CSS Modules but with typed style objects. No runtime cost. The direction for typed codebases.
Utility-first (Tailwind)	Atomic utility classes in markup. Only used classes are generated — tiny bundles. Dynamic class names must be full strings in source (no string interpolation). Use @apply sparingly for semantic extraction. Arbitrary values via bracket syntax.
Vue scoped styles	Add scoped to the <style> block — Vue appends a data attribute to all selectors. Use :deep() to reach child component elements. v-bind() in CSS reads reactive JS values and injects them as inline custom properties.
Svelte styles	All <style> content is automatically scoped at compile time (no keyword needed). class: directive adds/removes a class based on a boolean prop. Use :global() to escape scoping.
Bundler CSS features	Vite handles CSS Modules, PostCSS, and preprocessors out of the box. CSS follows JS code-splitting — lazy routes get their own CSS chunk. Critical CSS extraction inlines above-fold CSS in <head> and defers the rest. PurgeCSS / Tailwind JIT remove unused CSS at build time.

EXERCISES

1. **CSS Modules card system:** Create a `Card.module.css` with a base `.card` class. Add a `.cardFeatured` class that uses `composes: card` and adds an accent border. Add a `.cardCompact` variant with reduced padding. In a React component, accept a `variant` prop (`'default' | 'featured' | 'compact'`) and map it to the correct module class. Use `clsx` to handle the mapping cleanly. Bridge the card's background colour to a JS prop using an inline `style` attribute setting `--card-bg` as a custom property and reading it in the CSS module.
2. **Dynamic Tailwind class lookup:** Build a badge component that accepts a `status` prop: `'success' | 'warning' | 'error' | 'info'`. Create a lookup object that maps each status to a complete set of Tailwind classes (background, text colour, border). Verify that each full class string appears in the source file so Tailwind's scanner can find it. Add a fifth status dynamically from a user input and observe that classes generated by string interpolation are absent from the build output.
3. **Vue CSS v-bind:** In a Vue SFC, create a colour-picker that sets a reactive `themeColor` ref. Use `v-bind(themeColor)` in the scoped styles to apply it as a border, background tint (`color-mix(in oklch, v-bind(themeColor) 15%, transparent)`), and text colour on a card component. Observe in DevTools how Vue sets the value as an inline CSS custom property on the element and updates it reactively.
4. **vanilla-extract typed tokens:** Create a `tokens.css.ts` file using `createGlobalTheme` from vanilla-extract to define colour, spacing, and typography tokens. Create a `button.css.ts` that imports those tokens and uses `styleVariants` to define `primary`, `secondary`, and `ghost` button styles. Inspect the build output — confirm a static `.css` file was generated and that the TypeScript types catch a typo in a token name at compile time.
5. **Critical CSS audit:** Take an existing multi-section page. Using Chrome DevTools → Coverage panel, record which CSS rules are used on first load vs after scrolling. Identify the above-fold critical CSS. Manually extract it into a `<style>` block in `<head>` and convert the main stylesheet link to use the `preload / onload` pattern. Measure the difference in Largest Contentful Paint (LCP) in the Performance panel before and after.

Next: Chapter 11 — Building a Design System / Token-Based Theming. Primitive, semantic, and component token tiers; CSS custom property architecture; multi-brand theming with a single token layer swap; dark mode with `light-dark()` and `color-scheme`; generating tokens from Figma; documenting and versioning a token system; and integrating tokens across CSS Modules, Tailwind, and vanilla-extract.

CHAPTER 11

Building a Design System / Token-Based Theming

Chapter 11 — Building a Design System: Token-Based Theming

A design system is a single source of truth for visual decisions — colours, spacing, typography, radius, shadow — shared between designers and engineers. Design tokens are the atomic unit of that system: named values that carry meaning rather than raw numbers. `--color-accent-500` is a token. `oklch(0.60 0.18 248)` is just a colour. Tokens are what let you swap a theme, add dark mode, or rebrand a product by changing one file rather than a thousand rules.

1. The Three Token Tiers

TIER 1 — PRIMITIVE TOKENS (THE PALETTE)

Raw values. No meaning. Never used directly in components. Changed only when the palette itself is updated. Designers define these first.

`--blue-500` `--blue-400` `--green-500` `--red-500` `--amber-400` `--purple-500` `--grey-950` `--space-4` `--space-8` `--radius-md` `--font-size-sm`

TIER 2 — SEMANTIC TOKENS (THE CONTRACT)

References to primitives, but named for their purpose. This is the layer that changes between themes and colour schemes. Components use these — never primitives directly.

`--color-bg` `--color-surface` `--color-text` `--color-accent` `--color-success` `--color-danger` `--color-border` `--space-component-gap` `--radius-card` `--shadow-card`

TIER 3 — COMPONENT TOKENS (THE DETAIL)

References to semantic tokens, scoped to a specific component. Allow fine-grained overrides without touching the semantic layer. Optional — not all systems need this tier.

`--btn-bg` `--btn-text` `--btn-radius` `--card-bg` `--card-border` `--input-bg` `--input-border-focus` `--nav-height`

> **The cardinal rule:** components must only consume Tier 2 (semantic) or Tier 3 (component) tokens. Never reference a primitive token (`--blue-500`) directly in a component. If you do, swapping the theme breaks the component — a blue background stays blue even when the theme is "green".

2. Token Naming Conventions

```
--color-accent-interactive-hover
```

color	accent	interactive	hover
Category	Role / family	Context	State / variant

```
/* ── Naming segments (use what you need, left to right) ──────── */

/* {category}-{role}-{variant}-{scale}-{state} */

/* Category: what kind of token */
/* color / space / size / radius / shadow / font / duration / easing */

/* Role: what it's for (semantic) */
/* bg / surface / text / border / accent / success / danger / warning / info */

/* Variant: sub-role or emphasis */
/* strong / subtle / muted / inverse / on-accent */

/* Scale: numeric step within a category */
/* 50 / 100 / 200 … 950 (primitives) or sm / md / lg (semantics) */

/* State: interaction or condition */
/* default / hover / active / focus / disabled / placeholder */

/* ── Practical name examples ──────────────────────────────────── */
/* Primitives */
--color-blue-500
--color-grey-100
--space-4           /* 4px (or 0.25rem on a 4pt grid) */
--radius-md
--font-size-sm

/* Semantic */
--color-bg
--color-bg-subtle
--color-surface
--color-text
--color-text-subtle
--color-text-on-accent
--color-accent
--color-accent-hover
--color-success
--color-danger
--color-border
--color-border-strong
--space-component-gap    /* space between component children */
--space-section-gap      /* space between page sections */
--radius-component
--shadow-card
--duration-short         /* 150ms */
--duration-medium        /* 250ms */
--easing-default
```

```
/* Component */
--btn-bg
--btn-bg-hover
--btn-text
--btn-radius
--btn-padding-inline
--card-bg
--card-shadow
```

3. Full Token Architecture in CSS

```
/* ═══════════════════════════════════════════════════════════ */
/* tokens/primitives.css                                       */
/* The palette — raw values, no semantic meaning               */
/* ═══════════════════════════════════════════════════════════ */
:root {
    /* Colour primitives — oklch scale, H=248 (blue) */
    --blue-950: oklch(0.18 0.18 248);
    --blue-900: oklch(0.26 0.18 248);
    --blue-700: oklch(0.42 0.18 248);
    --blue-500: oklch(0.60 0.18 248);
    --blue-400: oklch(0.70 0.18 248);
    --blue-200: oklch(0.85 0.18 248);
    --blue-50:  oklch(0.96 0.18 248);

    /* Neutral scale — very low chroma, same hue for tonal harmony */
    --grey-950: oklch(0.12 0.01 248);
    --grey-900: oklch(0.18 0.01 248);
    --grey-700: oklch(0.35 0.01 248);
    --grey-500: oklch(0.55 0.01 248);
    --grey-300: oklch(0.75 0.01 248);
    --grey-100: oklch(0.93 0.01 248);
    --grey-50:  oklch(0.97 0.01 248);

    /* Status colours */
    --green-500: oklch(0.60 0.18 145);  --green-200: oklch(0.85 0.18 145);
    --red-500:   oklch(0.60 0.18 25);   --red-200:   oklch(0.85 0.18 25);
    --amber-500: oklch(0.70 0.18 65);   --amber-200: oklch(0.88 0.18 65);

    /* Spacing — 4px base grid */
    --space-1: 0.25rem;   --space-2: 0.5rem;
    --space-4: 1rem;      --space-6: 1.5rem;
    --space-8: 2rem;      --space-12:3rem;
    --space-16:4rem;      --space-24:6rem;

    /* Radius */
    --radius-sm: 4px;  --radius-md: 8px;
    --radius-lg: 12px; --radius-xl: 16px;
    --radius-full: 9999px;

    /* Typography */
    --font-size-xs: 0.75rem;   --font-size-sm: 0.875rem;
    --font-size-md: 1rem;       --font-size-lg: 1.125rem;
    --font-size-xl: 1.25rem;    --font-size-2xl:1.5rem;
    --font-weight-normal: 400;   --font-weight-bold: 700;
```

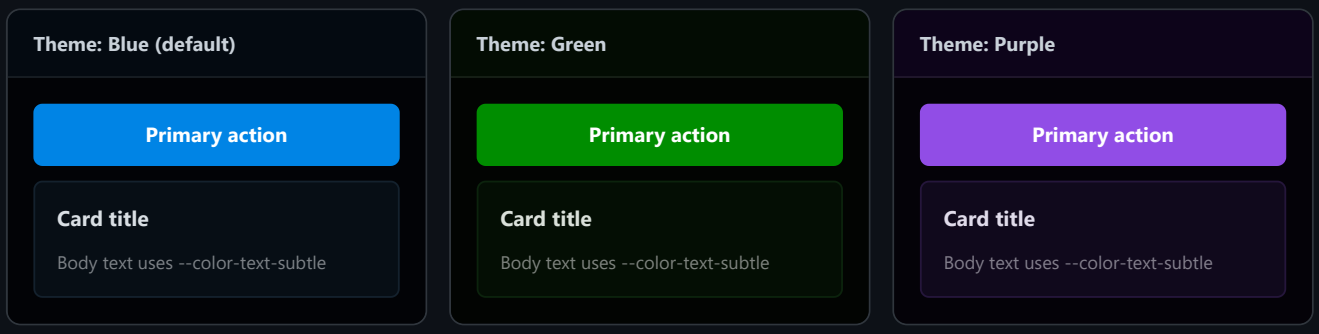
```
    /* Motion */
    --duration-short:  150ms;
    --duration-medium: 250ms;
    --duration-long:   400ms;
    --easing-default:  cubic-bezier(0.16, 1, 0.3, 1);
    --easing-in:       cubic-bezier(0.4, 0, 1, 1);
}
```

```css
/* ═══════════════════════════════════════════════════════════ */
/* tokens/semantic.css                                         */
/* The contract — maps primitives to purpose                   */
/* This file is swapped (or overridden) per theme              */
/* ═══════════════════════════════════════════════════════════ */
:root {
    color-scheme: light dark;

    /* Background layers */
    --color-bg:         light-dark(var(--grey-50),  var(--grey-950));
    --color-bg-subtle:  light-dark(var(--grey-100), var(--grey-900));
    --color-surface:    light-dark(oklch(1 0 0),     var(--grey-900));
    --color-surface-2:  light-dark(var(--grey-50),  var(--grey-950));

    /* Text */
    --color-text:         light-dark(var(--grey-950), var(--grey-100));
    --color-text-subtle:  light-dark(var(--grey-500), var(--grey-500));
    --color-text-on-accent: oklch(1 0 0);   /* always white on coloured bg */

    /* Accent (brand colour) */
    --color-accent:       light-dark(var(--blue-500), var(--blue-400));
    --color-accent-hover: oklch(from var(--color-accent) calc(l + 0.07) c h);
    --color-accent-subtle:color-mix(in oklch, var(--color-accent) 12%, var(--color-bg));

    /* Status */
    --color-success: light-dark(var(--green-500), oklch(0.72 0.18 145));
    --color-danger:  light-dark(var(--red-500),   oklch(0.72 0.18 25));
    --color-warning: light-dark(var(--amber-500), oklch(0.82 0.18 65));

    /* Border */
    --color-border:       light-dark(var(--grey-300), var(--grey-700));
    --color-border-strong:light-dark(var(--grey-700), var(--grey-300));

    /* Shadow */
    --shadow-sm:   0 1px 3px oklch(0 0 0 / 0.08);
    --shadow-card: 0 4px 16px oklch(0 0 0 / 0.12);
    --shadow-lg:   0 16px 48px oklch(0 0 0 / 0.18);

    /* Semantic spacing */
    --space-component-gap:  var(--space-4);
    --space-section-gap:    var(--space-12);
    --radius-component:     var(--radius-md);
    --radius-input:         var(--radius-sm);
}
```

4. Multi-Brand and Multi-Theme Support

A properly layered token system lets you ship multiple brands or colour themes by overriding only the semantic layer. The components and primitive layer are identical across all themes — only the mapping changes.

Same component markup, three brand themes — only the token override layer changes:



```css
/* ── Multi-brand theme via data attribute ──────────────────────── */
/* <html data-theme="green"> — one attribute switches the whole UI */

:root,
[data-theme="blue"] {
    --brand-hue:  248;
    --brand-chroma: 0.18;
}
[data-theme="green"] {
    --brand-hue:  145;
    --brand-chroma: 0.20;
}
[data-theme="purple"] {
    --brand-hue:  300;
    --brand-chroma: 0.22;
}

/* Semantic tokens derive from the brand variables */
:root {
    --color-accent:
        light-dark(
            oklch(0.55 var(--brand-chroma) var(--brand-hue)),
            oklch(0.70 var(--brand-chroma) var(--brand-hue))
        );
    --color-accent-hover:
        light-dark(
            oklch(0.62 var(--brand-chroma) var(--brand-hue)),
            oklch(0.77 var(--brand-chroma) var(--brand-hue))
        );
    --color-accent-subtle:
        oklch(from var(--color-accent)
            light-dark(0.95, 0.20) calc(c * 0.3) h);
}

/* JavaScript to switch theme */
document.documentElement.dataset.theme = 'green';
/* Everything updates instantly — no class replacement, no JS style injection */
```

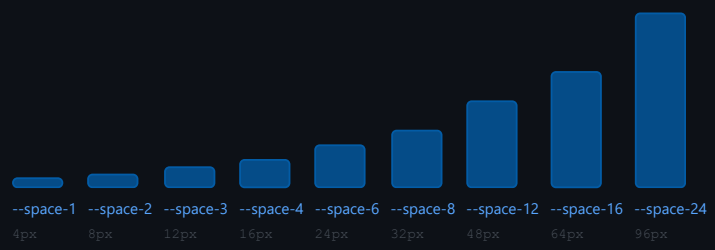
5. Component Tokens and Overrides

```
/* ── Component tokens defined at :root, consuming semantic tokens ─ */
:root {
    /* Button */
    --btn-bg:           var(--color-accent);
    --btn-bg-hover:     var(--color-accent-hover);
    --btn-text:         var(--color-text-on-accent);
    --btn-radius:       var(--radius-component);
    --btn-padding-block:var(--space-2);
    --btn-padding-inline:var(--space-4);
    --btn-font-size:    var(--font-size-sm);

    /* Card */
    --card-bg:          var(--color-surface);
    --card-border:      1px solid var(--color-border);
    --card-radius:      var(--radius-md);
    --card-padding:     var(--space-6);
    --card-shadow:      var(--shadow-card);

    /* Input */
    --input-bg:         var(--color-surface);
    --input-border:     1px solid var(--color-border);
    --input-border-focus:2px solid var(--color-accent);
    --input-radius:     var(--radius-input);
    --input-padding:    var(--space-2) var(--space-4);
}

/* ── Component CSS consumes only component tokens ──────────────── */
.btn {
    background:     var(--btn-bg);
    color:          var(--btn-text);
    border-radius:  var(--btn-radius);
    padding:        var(--btn-padding-block) var(--btn-padding-inline);
    font-size:      var(--btn-font-size);
    transition:     background var(--duration-short) var(--easing-default);
}
.btn:hover {
    background: var(--btn-bg-hover);
}

/* ── Component-level override in a specific context ─────────────── */
/* Override at the component token level — the component CSS is untouched */
.marketing-hero {
    --btn-radius:          var(--radius-full);   /* pill shape in hero only */
```

```
    --btn-padding-inline:  var(--space-8);        /* wider padding in hero only */
}
```

6. Spacing and Type Scales

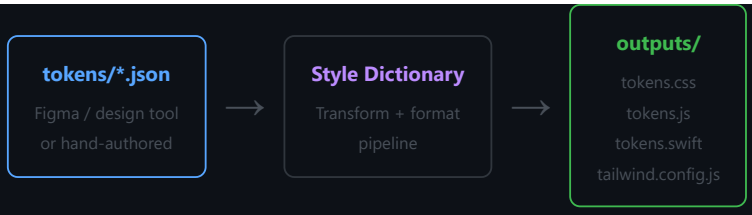
4px base grid — spacing tokens on a geometric scale:




```
/* ── Fluid type scale with clamp() ──────────────────────────── */
/* Type scales using clamp() + rem units adapt to viewport and user font-size */

:root {
    /* Fluid scale: clamp(min, preferred, max) */
    /* preferred: vw unit creates proportional scaling */
    --font-size-xs:  clamp(0.70rem, 0.65rem + 0.2vw, 0.75rem);
    --font-size-sm:  clamp(0.85rem, 0.80rem + 0.2vw, 0.90rem);
    --font-size-md:  clamp(1.00rem, 0.95rem + 0.3vw, 1.10rem);
    --font-size-lg:  clamp(1.15rem, 1.05rem + 0.5vw, 1.35rem);
    --font-size-xl:  clamp(1.30rem, 1.10rem + 0.9vw, 1.70rem);
    --font-size-2xl: clamp(1.60rem, 1.20rem + 1.8vw, 2.20rem);
    --font-size-3xl: clamp(2.00rem, 1.40rem + 2.8vw, 3.00rem);
}
```

7. Style Dictionary: Tokens as a Single Source of Truth

Style Dictionary (by Amazon) takes a JSON or JavaScript token definition and transforms it into platform-specific outputs: CSS custom properties, iOS Swift constants, Android XML, Tailwind config, JavaScript objects, and more. It makes tokens the authoritative single source that feeds every platform.


tokens/*.json
Figma / design tool
or hand-authored
Style Dictionary
Transform + format
pipeline
outputs/
tokens.css
tokens.js
tokens.swift
tailwind.config.js

```
/* ── tokens/color.json — the source token file ──────────────── */
{
    "color": {
        "blue": {
            "500": { "$value": "oklch(0.60 0.18 248)", "$type": "color" },
            "400": { "$value": "oklch(0.70 0.18 248)", "$type": "color" }
        },
        "accent": {
            "default": {
                "$value": "{color.blue.500}",   /* reference to another token */
                "$type":  "color"
            }
        }
    },
    "space": {
        "4": { "$value": "16px", "$type": "dimension" }
    }
}

/* ── style-dictionary.config.js ──────────────────────────────── */
export default {
    source: ['tokens/**/*.json'],
    platforms: {
        css: {
            transformGroup: 'css',
            files: [{
                destination: 'dist/tokens.css',
                format:      'css/variables',
                options: { outputReferences: true },
            }],
        },
        js: {
            transformGroup: 'js',
            files: [{
                destination: 'dist/tokens.js',
                format:      'javascript/esm',
            }],
        },
    },
};

/* ── Generated dist/tokens.css ──────────────────────────────── */
:root {
    --color-blue-500: oklch(0.60 0.18 248);
    --color-blue-400: oklch(0.70 0.18 248);
    --color-accent-default: var(--color-blue-500);   /* reference preserved */
```

```
    --space-4: 16px;
}
```

8. Versioning, Documentation, and Governance

```
/* ── Semantic versioning for design tokens ──────────────────────── */
/* MAJOR: removing or renaming a token (breaking — consumers must update) */
/* MINOR: adding a new token (safe — consumers can opt in) */
/* PATCH: changing a token's value within acceptable range (e.g. L+0.02) */

/* ── Deprecation pattern ────────────────────────────────────────── */
/* Never remove a token without a deprecation period */
:root {
    /* @deprecated — use --color-accent instead. Will be removed in v5. */
    --primary-blue: var(--color-accent);   /* alias pointing to new token */
}

/* ── Token documentation in JSON (W3C Design Token format) ──── */
{
    "color-accent": {
        "$value":       "{color.blue.500}",
        "$type":        "color",
        "$description": "Primary interactive colour. Use on buttons, links, and focus ri
        "$extensions": {
            "com.myco.tokens": {
                "status":    "stable",
                "since":     "2.0.0",
                "usedBy": ["Button", "Link", "Input", "Badge"]
            }
        }
    }
}

/* ── Integrating tokens into Tailwind ──────────────────────────── */
/* Map CSS custom properties into Tailwind's theme */

export default {
    theme: {
        colors: {
            bg:      'var(--color-bg)',
            surface: 'var(--color-surface)',
            text:    'var(--color-text)',
            accent:  'var(--color-accent)',
            border:  'var(--color-border)',
            success: 'var(--color-success)',
            danger:  'var(--color-danger)',
        },
        spacing: {
            1:  'var(--space-1)',
```

```
            2:  'var(--space-2)',
            4:  'var(--space-4)',
            8:  'var(--space-8)',
        },
        borderRadius: {
            component: 'var(--radius-component)',
            full:      'var(--radius-full)',
        },
    },
};
/* Now Tailwind classes like bg-accent, text-text, p-4, rounded-component */
/* all resolve to the CSS custom properties — theme switch works in Tailwind too */
```

Chapter Summary

Concept	Key point
Three tiers	Primitive tokens (raw values, the palette), semantic tokens (purpose-named references to primitives — the contract), component tokens (scoped to a component, references to semantics). Components consume semantic or component tokens only — never primitives directly.
Naming convention	category-role-variant-scale-state. Read left to right from general to specific. Semantic tokens drop the scale and describe purpose: --color-accent, --color-border-strong, --space-component-gap. Consistent naming enables IDE autocomplete and documentation generation.
light-dark() theming	Define all semantic colour tokens using light-dark() — one declaration covers both colour schemes. Declare color-scheme: light dark on :root. Use color-scheme: dark on a subtree to force dark values in a specific section (e.g. a promotional dark banner inside a light page).
Multi-brand theming	Extract the brand's hue and chroma into --brand-hue and --brand-chroma primitives. Derive all accent-family semantic tokens from those variables. Switch theme by setting data-theme="green" on the html element — no JavaScript style injection, no class toggling across hundreds of elements.
Component tokens	Optional third tier. Define --btn-bg, --card-padding, --input-border-focus at :root from semantic tokens. Components consume component tokens. Context-specific overrides change component tokens locally without touching component CSS: .marketing-hero { --btn-radius: var(--radius-full); }
Spacing scale	4px (0.25rem) base grid. Multiples: 1, 2, 3, 4, 6, 8, 12, 16, 24. Map to semantic tokens: --space-component-gap: var(--space-4). Use clamp() for fluid type scales that adapt to both viewport width and user font-size preferences.
Style Dictionary	Takes JSON token files (using the W3C Design Token Community Group format) and transforms them into CSS custom properties, JavaScript exports, iOS/Android constants, and Tailwind config. Token references ({color.blue.500}) are preserved as var() in CSS output. Enables tokens as a true single source of truth across platforms.
Versioning	Treat tokens like an API. Major version: rename or remove (breaking). Minor version: add new tokens (safe). Patch: adjust a value within acceptable bounds. Deprecate with alias tokens before removal. Document $description and $status in the JSON source.

EXERCISES

1. **Build the full three-tier token stack:** Create three CSS files: `primitives.css`, `semantic.css`, and `components.css`. In primitives, define a complete blue scale (50–950 in oklch), a neutral scale, and spacing tokens (1–24 on a 4px grid). In semantic, map those to purpose-named tokens using `light-dark()` for all colour values. In components, define `--btn-*` and `--card-*` tokens referencing semantic tokens. Write a Button and Card component using only their respective component tokens. Change the neutral hue in primitives from 248 to 30 (warm) and observe that all three tiers update without touching component CSS.

2. **Multi-brand theme switcher:** Implement three brand themes (blue, green, purple) using `data-theme` on `<html>`. Each theme sets `--brand-hue` and `--brand-chroma`. Derive all accent tokens from those variables. Add a JavaScript theme switcher that reads the current theme from `localStorage` on page load and applies it. Verify that a page reload preserves the selected theme and that every coloured element updates instantly on switch.

3. **Fluid type scale:** Define a seven-step fluid type scale using `clamp()` for sizes xs through 3xl. Use `utopia.fyi` or the formula `clamp(min, min + (max-min) * (100vw - 320px) / (1280-320), max)` to calculate exact values for a 320px–1280px fluid range. Apply the scale to a prose stylesheet via semantic typography tokens (`--font-size-body`, `--font-size-heading-lg`, etc.). Verify by resizing the browser: text should scale smoothly between the min and max values with no media query breakpoints.

4. **Style Dictionary pipeline:** Set up a minimal Style Dictionary project. Write a `tokens/color.json` using the W3C Design Token Community Group `$value`/`$type` format. Include at least five colour tokens and three spacing tokens with a token reference (`{color.blue.500}`). Configure Style Dictionary to output a `dist/tokens.css` file with CSS custom properties and a `dist/tokens.js` ES module. Run the build and confirm references in CSS output use `var()` syntax. Then add a second JSON file for dark-mode overrides and wire it as a separate platform output.

5. **Token audit — find primitive leaks:** Take an existing stylesheet (your own or an open-source UI library). Search for any raw colour values, px values, or hardcoded numbers used directly in component rules rather than referencing a token. List every "primitive leak" — a value that should be a token but isn't. Propose a token name for each one following the category-role-variant convention. Convert the three worst offenders to proper token references and verify the visual output is identical.

Next: Chapter 12 — Project: Component Library with Modern CSS. The capstone chapter. Build a production-ready component library from scratch using the techniques from all eleven chapters: a full three-tier token system, @layer architecture, container queries on every component, logical properties throughout, oklch colour theming, scroll-driven animations, will-change and contain for performance, CSS Modules for scoping, and light-dark() for zero-JavaScript dark mode.

CHAPTER 12

Project: Component Library with Modern CSS

Chapter 12 — Project: Component Library with Modern CSS

This capstone project brings together every technique from the course into a single, coherent component library. You'll build a production-ready set of components backed by a three-tier token system, scoped with `@layer`, sized with container queries, internationalized with logical properties, animated with scroll-driven and `@property` transitions, and performance-hardened with `contain` and `will-change`. No framework required — pure, modern CSS.

@layer | Container queries | Subgrid | :has() | Scroll-driven | Logical props | oklch colors | @property | contain | Token system | light-dark()

1. Project Structure

TOKENS/
- primitives.css
- semantic.css
- components.css

BASE/
- reset.css
- typography.css
- houdini.css

COMPONENTS/
- button.css
- badge.css
- card.css
- input.css
- modal.css
- disclosure.css
- progress.css

LAYOUTS/
- grid.css
- stack.css
- cluster.css
- sidebar.css

UTILITIES/
- spacing.css
- typography.css
- display.css
- a11y.css

INDEX.CSS
- @layer declarations
- @import ordering

```css
/* ── index.css — @layer order is declared first, always ──────── */
/* This is the single most important architectural decision.     */
/* Layers are resolved in this order regardless of import order. */

@layer reset, tokens, base, layouts, components, utilities, overrides;

/* Tokens load first — all custom properties available globally */
@import 'tokens/primitives.css' layer(tokens);
@import 'tokens/semantic.css'   layer(tokens);
@import 'tokens/components.css' layer(tokens);

@import 'base/reset.css'        layer(reset);
@import 'base/typography.css'   layer(base);
@import 'base/houdini.css'      layer(base);

@import 'layouts/grid.css'      layer(layouts);
@import 'layouts/stack.css'     layer(layouts);
@import 'layouts/cluster.css'   layer(layouts);
@import 'layouts/sidebar.css'   layer(layouts);

@import 'components/button.css'     layer(components);
@import 'components/badge.css'      layer(components);
@import 'components/card.css'       layer(components);
@import 'components/input.css'      layer(components);
@import 'components/modal.css'      layer(components);
@import 'components/disclosure.css' layer(components);
@import 'components/progress.css'   layer(components);

@import 'utilities/spacing.css'  layer(utilities);
@import 'utilities/a11y.css'     layer(utilities);
```

Layer resolution order — lowest (first declared) to highest (overrides):

1. reset	Normalize browser defaults
2. tokens	All CSS custom properties
3. base	Typography, Houdini @property
4. layouts	Grid, Stack, Cluster, Sidebar
5. components	Button, Card, Input, Modal…
6. utilities	Spacing, display, a11y helpers
7. overrides	Per-page, theme, a/b test

2. Reset and Base Styles

```
/* ── base/reset.css ──────────────────────────────────────────── */
@layer reset {
    *, *::before, *::after {
        box-sizing:  border-box;
        margin:      0;
        padding:     0;
    }
    html {
        color-scheme:       light dark;  /* unlocks light-dark() */
        -webkit-text-size-adjust: none;
        tab-size:           4;
    }
    body {
        min-block-size:  100dvb;          /* logical, dynamic viewport */
        background:      var(--color-bg);
        color:           var(--color-text);
        font-family:     var(--font-family-body);
        font-size:       var(--font-size-md);
        line-height:     1.6;
    }
    img, video, svg {
        max-inline-size: 100%;       /* logical property — replaces max-width */
        display:         block;
    }
    button {
        font:    inherit;
        cursor: pointer;
    }
    :focus-visible {
        outline:        2px solid var(--color-accent);
        outline-offset: 3px;
        border-radius:  2px;
    }
    @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/houdini.css — @property registrations ──────────────── */
```

```
/* Registered before use so transitions animate correctly        */
@layer base {
    @property --progress-pct {
        syntax:         '<percentage>';
        inherits:       false;
        initial-value:  0%;
    }
    @property --overlay-opacity {
        syntax:         '<number>';
        inherits:       false;
        initial-value:  0;
    }
    @property --ripple-scale {
        syntax:         '<number>';
        inherits:       false;
        initial-value:  0;
    }
}
```

3. Layout Primitives

Layout primitives are small, composable CSS classes built on a handful of properties each. They handle *flow* rather than visual appearance, and they compose cleanly with any component.

```css
/* ── layouts/stack.css — vertical rhythm ──────────────────────── */
/* .stack > * + * gets a top margin — the "lobotomised owl" */
@layer layouts {
    .stack {
        display:        flex;
        flex-direction: column;
        gap:            var(--stack-gap, var(--space-4));
    }

    /* ── layouts/cluster.css — wrapping row of items ──────────── */
    .cluster {
        display:   flex;
        flex-wrap: wrap;
        gap:       var(--cluster-gap, var(--space-2));
        align-items:center;
    }

    /* ── layouts/sidebar.css — sidebar + main content ─────────── */
    .sidebar-layout {
        display:   flex;
        flex-wrap: wrap;
        gap:       var(--space-6);
    }
    .sidebar-layout > .sidebar {
        flex-basis: var(--sidebar-width, 16rem);
        flex-grow:  1;
    }
    .sidebar-layout > .main {
        flex-basis:  0;
        flex-grow:   999;
        min-inline-size: 60%;  /* breakpoint: wraps when main < 60% */
    }

    /* ── layouts/grid.css — auto-responsive card grid ─────────── */
    .auto-grid {
        display:               grid;
        grid-template-columns: repeat(auto-fit, minmax(var(--min-col, 16rem), 1fr));
        gap:                   var(--grid-gap, var(--space-4));
    }
    /* Subgrid card rows — all cards share the same row heights */
    .auto-grid.subgrid-rows > * {
        display:               grid;
        grid-row:              span 3;
        grid-template-rows:    subgrid;  /* Ch. 3 — card rows align across columns */
    }
    .auto-grid.subgrid-rows {
        grid-template-rows:    auto 1fr auto;   /* header / body / footer */
```

```
    }
}
```

4. Button Component

Live output

BUTTON VARIANTS

Primary Secondary Ghost Danger

```css
/* ── components/button.css ──────────────────────────────────── */
@layer components {

    /* Base — logical properties throughout (Ch. 6) */
    .btn {
        display:        inline-flex;
        align-items:    center;
        gap:            var(--space-2);
        padding-block:  var(--btn-padding-block);
        padding-inline: var(--btn-padding-inline);
        border-radius:  var(--btn-radius);
        font-size:      var(--btn-font-size);
        font-weight:    600;
        border:         none;
        cursor:         pointer;
        text-decoration:none;
        white-space:    nowrap;
        user-select:    none;
        contain:        layout style;       /* Ch. 9 — isolates layout impact */
        will-change:    transform;          /* Ch. 9 — promotes on hover */
        transition:
            background   var(--duration-short) var(--easing-default),
            transform    var(--duration-short) var(--easing-default),
            box-shadow   var(--duration-short) var(--easing-default);
    }

    /* :has() — button containing a spinner is loading (Ch. 4) */
    .btn:has(.spinner) {
        opacity:  0.72;
        pointer-events: none;
        cursor:   not-allowed;
    }

    /* Variants — each overrides only component tokens (Ch. 11) */
    .btn--primary {
        background: var(--btn-bg);
        color:      var(--btn-text);
    }
    .btn--primary:hover {
        background: var(--btn-bg-hover);
        transform:  translateY(-1px);
        box-shadow: 0 4px 12px oklch(from var(--btn-bg) l c h / 0.35);
    }
    .btn--secondary {
        --btn-bg:      var(--color-accent-subtle);
        --btn-text:    var(--color-accent);
        background:    var(--btn-bg);
```

```
        color:          var(--btn-text);
        border:         1px solid var(--color-border);
    }
    .btn--ghost {
        background: transparent;
        color:      var(--color-accent);
        border:     1px solid currentColor;
    }
    .btn--danger {
        --btn-bg:       var(--color-danger);
        --btn-bg-hover:oklch(from var(--color-danger) calc(l + 0.07) c h);
        background:     var(--btn-bg);
        color:          var(--color-text-on-accent);
    }

    /* Size modifier */
    .btn--sm {
        --btn-padding-block:  var(--space-1);
        --btn-padding-inline: var(--space-2);
        --btn-font-size:      var(--font-size-xs);
    }
    .btn--lg {
        --btn-padding-block:  var(--space-4);
        --btn-padding-inline: var(--space-8);
        --btn-font-size:      var(--font-size-lg);
        --btn-radius:         var(--radius-lg);
    }
}
```

5. Card Component with Container Queries

Live output

Card Title

Secondary content fills this area. Grows to fill available rows when subgrid is used.

New View

Another Card

When these cards are in a subgrid, all footers align automatically — no JavaScript height matching needed.

Active Open

Third Card

Short body.

Draft Edit

```css
/* ── components/card.css ──────────────────────────────────────── */
@layer components {

    .card {
        container-type: inline-size;  /* Ch. 2 — card reports its own width */
        container-name: card;

        background:   var(--card-bg);
        border:       var(--card-border);
        border-radius:var(--card-radius);
        padding:      var(--card-padding);
        box-shadow:   var(--card-shadow);

        /* Subgrid: card becomes a three-row subgrid participant (Ch. 3) */
        display:             grid;
        grid-template-rows:  subgrid;
        grid-row:            span 3;
        contain:             layout style paint;   /* Ch. 9 */
    }

    .card__title {
        font-size:   var(--font-size-lg);
        font-weight: 700;
        color:       var(--color-text);
    }
    .card__body {
        color:       var(--color-text-subtle);
        font-size:   var(--font-size-sm);
        line-height: 1.5;
    }
    .card__footer {
        display:         flex;
        align-items:     center;
        justify-content: space-between;
        padding-block-start: var(--space-2);  /* logical — replaces padding-top */
        border-block-start:  1px solid var(--color-border);
    }

    /* Container query — wide card switches to horizontal layout (Ch. 2) */
    @container card (min-inline-size: 480px) {
        .card {
            grid-template-columns: auto 1fr;
            grid-template-rows:    auto 1fr auto;
        }
        .card__media {
            grid-row:           1 / -1;
            border-radius:      calc(var(--card-radius) - 2px) 0 0
```

```
                                    calc(var(--card-radius) - 2px);
        }
    }

    /* :has() — card with a featured badge gets elevated shadow (Ch. 4) */
    .card:has(.badge--featured) {
        border-color: var(--color-accent);
        box-shadow:   0 0 0 1px var(--color-accent), var(--shadow-card);
    }
}
```

6. Progress Bar — `@property` Animated

PROGRESS BARS (ANIMATED VIA @PROPERTY)

72% complete

38% complete

90% complete

```css
/* ── components/progress.css ──────────────────────────────── */
@layer components {

    /* @property registered in base/houdini.css — enables smooth    */
    /* transition of --progress-pct (Ch. 8). Without registration,   */
    /* the browser can't interpolate a custom property value.       */

    .progress {
        block-size:    6px;                  /* logical — replaces height */
        background:    var(--color-surface-2);
        border-radius:var(--radius-full);
        position:      relative;
        overflow:      hidden;
        contain:       strict;              /* Ch. 9 — completely isolated */
    }

    .progress::after {
        content:           '';
        position:          absolute;
        inset-block:       0;               /* logical — replaces top/bottom */
        inset-inline-start:0;               /* logical — replaces left */
        inline-size:       var(--progress-pct);  /* set via data attr or JS */
        background:        var(--progress-color, var(--color-accent));
        border-radius:     inherit;
        transition:        inline-size 600ms var(--easing-default);
    }

    /* Animated stripe pattern using oklch (Ch. 7) */
    .progress--striped::after {
        background-image: repeating-linear-gradient(
            -45deg,
            oklch(1 0 0 / 0.1) 0px,
            oklch(1 0 0 / 0.1) 10px,
            transparent 10px,
            transparent 20px
        );
        animation: progress-stripes 600ms linear infinite;
    }
    @keyframes progress-stripes {
        from { background-position-x: 0; }
        to   { background-position-x: 28px; }
    }
}

/* JavaScript — update progress with CSS custom property */
function setProgress(el, pct) {
```

```
    el.style.setProperty('--progress-pct', pct + '%');
}
```

7. Input and Form Components

INPUT VARIANTS

Default input — click to see focus ring

Disabled input

```css
/* ── components/input.css ──────────────────────────────────────── */
@layer components {
    .field {
        display:        flex;
        flex-direction: column;
        gap:            var(--space-1);
    }
    .field__label {
        font-size:   var(--font-size-sm);
        font-weight: 600;
        color:       var(--color-text);
    }
    .input {
        background:    var(--input-bg);
        border:        var(--input-border);
        border-radius: var(--input-radius);
        padding:       var(--input-padding);
        color:         var(--color-text);
        font-size:     var(--font-size-md);
        inline-size:   100%;              /* logical — replaces width */
        outline:       none;
        transition:    border-color var(--duration-short) ease,
                       box-shadow   var(--duration-short) ease;
    }
    .input:focus {
        border-color: var(--color-accent);
        box-shadow:    0 0 0 3px color-mix(in oklch,
                           var(--color-accent) 20%,
                           transparent);   /* Ch. 7 */
    }
    .input:disabled {
        opacity:        0.4;
        cursor:         not-allowed;
    }

    /* :has() — field with an invalid input shows error state (Ch. 4) */
    .field:has(.input:invalid:not(:placeholder-shown)) {
        .field__label   { color: var(--color-danger); }
        .input          {
            border-color: var(--color-danger);
            box-shadow:   0 0 0 3px color-mix(in oklch,
                              var(--color-danger) 20%,
                              transparent);
        }
        .field__hint    { display: block; }   /* reveal error hint */
```

```
        }
    }
```

8. Modal — View Transition and Enter Animation

```
/* ── components/modal.css ──────────────────────────────────────── */
@layer components {

    /* Overlay — @property makes opacity transition animatable (Ch. 8) */
    .modal-overlay {
        position:   fixed;
        inset:      0;                  /* logical shorthand for all four edges */
        background: oklch(0 0 0 / var(--overlay-opacity));
        transition: --overlay-opacity var(--duration-medium) ease;
        --overlay-opacity: 0;
        display:    grid;
        place-items:center;
    }
    .modal-overlay[aria-hidden="false"] {
        --overlay-opacity: 0.6;
    }

    /* Dialog panel */
    .modal {
        background:     var(--color-surface);
        border:         1px solid var(--color-border);
        border-radius:  var(--radius-xl);
        padding:        var(--space-8);
        max-inline-size:min(90vw, 540px);    /* logical + fluid constraint */
        box-shadow:     var(--shadow-lg);
        contain:        layout style paint;    /* Ch. 9 */

        /* Enter animation via @keyframes + will-change */
        animation:  modal-enter var(--duration-medium) var(--easing-default) both;
        will-change:transform, opacity;    /* Ch. 9 */
    }
    @keyframes modal-enter {
        from {
            opacity:   0;
            transform: translateY(24px) scale(0.96);
        }
    }

    /* View Transition API — smooth cross-page modal navigation */
    ::view-transition-old(modal) {
        animation: 200ms ease out both fade-out;
    }
    ::view-transition-new(modal) {
        animation: 300ms var(--easing-default) both modal-enter;
```

```
    }
    .modal {
        view-transition-name: modal;      /* Ch. 5 technique */
    }
}
```

9. Scroll-Driven Content Reveal

```
/* ── utilities/a11y.css — screen-reader and scroll-driven utils — */
@layer utilities {

    /* Screen reader only */
    .sr-only {
        position:   absolute;
        inline-size:1px;
        block-size: 1px;
        overflow:   hidden;
        clip:        rect(0,0,0,0);
        white-space:nowrap;
    }

    /* Scroll-driven entry reveal — no IntersectionObserver needed (Ch. 5) */
    .reveal {
        animation:            reveal-in linear both;
        animation-timeline:   view();
        animation-range:      entry 0% entry 30%;
    }
    @keyframes reveal-in {
        from {
            opacity:   0;
            transform: translateY(32px);
        }
    }

    /* Scroll progress indicator (Ch. 5) */
    .scroll-progress {
        position:           fixed;
        inset-block-start:  0;
        inset-inline-start: 0;
        inline-size:        100%;
        block-size:         3px;
        background:         var(--color-accent);
        transform-origin:   0 0;

        animation:           scroll-track linear both;
        animation-timeline:  scroll(root);
    }
    @keyframes scroll-track {
        from { transform: scaleX(0); }
        to   { transform: scaleX(1); }
```

```
    }
}
```

10. Technique Reference Map

Technique (Chapter)	Where it lives in the library	What it solves
`@layer (Ch. 1)`	index.css — seven-layer declaration	Predictable specificity. Third-party CSS imports go into a named layer and can never win against local component styles, even with low-specificity selectors.
`Container queries (Ch. 2)`	components/card.css, components/sidebar.css	Cards switch between vertical and horizontal layout based on their own width, not the viewport. No media-query breakpoints on the component — consumers place it anywhere.
`Subgrid (Ch. 3)`	layouts/grid.css + .card	Card headers, bodies, and footers align across columns without JavaScript height matching.
`:has() (Ch. 4)`	components/button.css, components/input.css, components/card.css	Loading state (button with spinner), form error state (field containing invalid input), featured styling (card containing featured badge) — all without JavaScript class toggling.
`Scroll-driven (Ch. 5)`	utilities/a11y.css (.reveal), any layout (.scroll-progress)	Entry animations driven by scroll position with no JavaScript or IntersectionObserver. The browser's compositor thread handles it — zero JS thread cost.
`Logical properties (Ch. 6)`	All components and layouts	Every sizing, spacing, and border property uses logical equivalents (inline-size, padding-block, inset-inline-start…). RTL languages get correct layout for free.
`oklch / color-mix() (Ch. 7)`	tokens/primitives.css, tokens/semantic.css, everywhere	Perceptually uniform colour scale. light-dark() semantic tokens. Focus ring glow derived from the accent with color-mix(). Hover shadows computed from button background via relative colour syntax.
`@property (Ch. 8)`	base/houdini.css, components/progress.css, components/modal.css	CSS can animate --progress-pct and --overlay-opacity because they are registered with a type. Without registration, transition on custom properties does nothing.
`contain / will-change (Ch. 9)`	All interactive components (.btn, .card, .modal, .progress)	contain: layout style isolates each component from triggering ancestor reflows. will-change: transform promotes hover targets to their own compositor layer — animations stay at 60fps.
`Token system (Ch. 11)`	tokens/ directory — three files, three tiers	All visual values live in tokens. Swapping data-theme attribute changes the brand colour everywhere. Semantic tokens use light-dark() so dark mode is zero extra code.

CAPSTONE EXERCISES

1. **Complete the library:** Using the patterns established in this chapter, implement the remaining components: `Badge`, `Disclosure/Accordion`, `Tooltip`, `Toast`, and `Tabs`. Each must consume only semantic or component tokens (no raw values), use logical properties throughout, and be placed in the `@layer components` layer. The Disclosure should use `:has()` to style the trigger based on whether the panel is open.
2. **Add a second brand theme:** Implement a `data-theme="warm"` theme using an amber/orange hue (H≈55 in oklch). Override only `--brand-hue` and `--brand-chroma` in the selector. Verify that every component — buttons, progress bars, focus rings, card borders — switches colour without any component CSS being modified. Add a theme toggle button that saves the preference to `localStorage` and restores it on load.
3. **Build a dashboard layout:** Compose the layout primitives into a full page: a `.sidebar-layout` wrapping a navigation sidebar and a main content area, with an `.auto-grid.subgrid-rows` of cards inside the main area. Add the `.scroll-progress` bar to the layout. All cards should use the subgrid row alignment so their footers are flush. Use container queries on the card so it switches to horizontal layout at 480px. Verify RTL layout by adding `dir="rtl"` to `<html>` — nothing should look broken.
4. **Performance audit:** Open Chrome DevTools Performance panel and record a 3-second scroll of your library demo page. Identify any frames that drop below 60fps. Use the Layers panel to verify that animated elements (buttons on hover, modals, progress bars) are on their own compositor layers. If any hover animation triggers a paint, add `will-change: transform` to the relevant rule and confirm the paint disappears from the profile.
5. **Export to Style Dictionary:** Take the token files from this library and convert them to the W3C Design Token Community Group `$value`/`$type` JSON format. Set up a Style Dictionary config that outputs CSS custom properties (for the web), a JavaScript ESM export (for React components using inline styles or CSS-in-JS), and a Tailwind `theme.extend` config. Run the build and verify all three outputs are consistent. Change one primitive (e.g. `--blue-500` hue from 248 to 240) and confirm the change propagates correctly to all three outputs.

CSS Advanced Course — Complete

You've worked through all twelve chapters of the CSS Advanced course: cascade control with `@layer`, component-aware layouts with container queries and subgrid, powerful selectors with `:has()`, declarative animations with scroll-driven timelines, internationalisation-ready layouts with logical properties, perceptually uniform colour with oklch, low-level API access with CSS Houdini, rendering-pipeline performance, framework integration patterns, and production-grade design tokens. These are the tools that separate CSS written for today's web from CSS written for the web of ten years ago. Build something with them.