CSS COURSE — INTERMEDIATE LEVEL

CSS Intermediate

12 chapters — Flexbox · Grid · Media Queries · CSS Functions · Animations · Architecture

01 Flexbox Fundamentals

02 Flexbox Advanced — Alignment, Wrapping, Ordering

03 Grid Fundamentals

04 Grid Advanced — Template Areas, auto-fit/fill

05 Advanced Selectors — Combinators, Attribute Selectors, :nth-child

06 The Cascade and Specificity in Depth

07 Media Queries and Responsive Layouts

08 CSS Functions — calc(), clamp(), min(), max()

09 Transforms — Translate, Rotate, Scale, Skew

10 Keyframe Animations

11 CSS Architecture — BEM and Utility-First Thinking

12 Project: Fully Responsive Multi-Column Layout

Flexbox • Grid • clamp() • @keyframes • BEM • Project

CHAPTER 1

Flexbox Fundamentals

📦 Chapter 1 — Flexbox Fundamentals

Before Flexbox, laying out elements side by side meant fighting with floats, inline-block tricks, and table hacks — all fragile and difficult to reason about. Flexbox changed everything. It is a one-dimensional layout model that gives precise, readable control over how items are placed, sized, and aligned along a single axis. Master Flexbox and you can build navigation bars, card rows, centred heroes, and sidebar layouts with clean, declarative CSS.

1 — What is Flexbox?

The **Flexible Box Layout** (Flexbox) is a CSS layout mode designed for arranging items in a single line — either a row or a column. Unlike the normal document flow, flex items do not collapse, float away, or behave differently based on whether they are block or inline elements. They simply line up and respond to a consistent set of layout rules.

Flexbox works through two roles:

FLEX CONTAINER

The parent element

Set `display: flex` on this element. It controls the direction, alignment, and spacing of its children. Layout rules are declared here.

FLEX ITEMS

The direct children

Any direct child of a flex container automatically becomes a flex item. Grandchildren are unaffected. Items can control their own sizing with the `flex` property.

> **One dimension at a time.** Flexbox handles either a row *or* a column — not both simultaneously. For layouts that need rows *and* columns at once (like a full page grid), use CSS Grid, which is covered in Chapters 3 and 4 of this course. Flexbox and Grid are complementary, not competing.

2 — Creating a Flex Container

A single declaration transforms an element into a flex container:

🎨 **CSS — Turning on Flexbox**

```css
/* Before: children stack vertically (normal block flow) */
.nav {
    /* no special display — each child starts on its own line */
}

/* After: children line up in a row */
.nav {
    display: flex;
}
```

`display: flex` makes the container itself a block-level box (takes up the full width of its parent). Use `display: inline-flex` if you need the container to shrink-wrap its content and sit inline with surrounding text.

🌐 **HTML — A navigation with three links**

```html
<nav class="navbar">
    <a href="#">Home</a>
    <a href="#">About</a>
    <a href="#">Contact</a>
</nav>
```

🎨 **CSS — Flex turns the three links into a row**

```css
.navbar {
    display: flex;
    gap: 24px;
    padding: 12px 20px;
    background: #1a1d27;
}

.navbar a {
```

```
    color: #e2e4ec;
    text-decoration: none;
}
```

The three `<a>` elements are direct children of `.navbar` and automatically become flex items. They line up in a row with 24px gaps between them — no float, no inline-block, no clearfix.

3 — Main Axis and Cross Axis

Every flex container has two axes. All of Flexbox's alignment and distribution properties are defined in terms of these axes, so understanding them is the foundation of everything else.

- **Main axis** — runs in the direction of `flex-direction`. Items are placed along it. `justify-content` distributes space here.
- **Cross axis** — perpendicular to the main axis (vertical when direction is row). `align-items` and `align-self` act here.

When you change `flex-direction` to `column`, the axes rotate: the main axis becomes vertical (top-to-bottom) and the cross axis becomes horizontal. **justify-content** and **align-items** always follow their respective axes — they do not change meaning, only direction.

> **The axes rotate together.** A common mistake is thinking `justify-content: center` always centres horizontally. It centres along the *main axis*. In a column container, that means vertical centering. Keep the axes in mind, not compass directions.

4 — flex-direction: Which Way Do Items Flow?

`flex-direction` sets the direction of the main axis — and therefore the direction in which flex items are placed. The default is `row` (left to right).

row

★ *default*

Items flow left → right. Main axis is horizontal.

row-reverse

Items flow right → left. Same horizontal axis but start and end are swapped.

column

Items flow top → bottom. Main axis becomes vertical.

column-reverse

Items flow bottom → top. Vertical main axis, reversed.

> 💡 **column is key for responsive design.** *A row of cards on desktop becomes a stacked column on mobile with a single media query that changes just `flex-direction`. No restructuring the HTML.*

5 — justify-content: Space Along the Main Axis

`justify-content` controls how the browser distributes flex items and any leftover space along the *main axis*. It only has a visible effect when there is free space — either because items do not fill the container, or because some items have a smaller `flex-grow` value than others.

VALUE	BEHAVIOUR
`flex-start` *default*	Items packed at the start. All free space appears at the end.
`flex-end`	Items packed at the end. Free space appears at the start.
`center`	Items grouped in the centre. Equal free space at both ends.
`space-between`	First item at start, last at end. Remaining space split equally *between* items. No space at edges.
`space-around`	Equal space on each side of every item. Edge gaps are half the size of inner gaps.
`space-evenly`	Identical gaps between all items *and* at both edges.

> **Most-used in real projects:** `space-between` for navbar (logo left, links right) and card rows. `center` for hero content or centred button groups. `flex-end` for aligning modal action buttons to the

right.

6 — align-items: Alignment Along the Cross Axis

`align-items` controls how flex items are positioned along the *cross axis* (perpendicular to the main axis). In a `row` container, this means vertical alignment. This single property replaces years of vertical-centering hacks.

VALUE	BEHAVIOUR
`stretch` *default*	Items stretch to fill the container's cross-axis size. All items match the height (or width) of the tallest sibling.
`flex-start`	Items align to the cross-axis start (top, in a row). Items keep their natural size.
`flex-end`	Items align to the cross-axis end (bottom, in a row).
`center`	Items centred on the cross axis. The classic "vertically centre anything" Flexbox trick.
`baseline`	Items aligned so their text baselines line up. Useful when mixing different font sizes.

🎨 **CSS — Perfectly centred content in two lines**

```
.hero {
    display: flex;
    justify-content: center;    /* centre on main axis (horizontal) */
    align-items: center;        /* centre on cross axis (vertical)  */
    min-height: 100vh;
}
```

Before Flexbox, perfectly centring an element required absolute positioning with a negative-margin hack, or JavaScript calculations. Now it is two property declarations.

7 — flex-wrap: Handling Overflow

By default, flex items refuse to wrap — they squeeze onto a single line even if it means shrinking below their natural size. `flex-wrap` changes this behaviour.

`nowrap`

★ *default*

All items on one line. Items shrink if needed, or overflow if `flex-shrink: 0`.

`wrap`

Items that do not fit wrap to new lines, stacking top-to-bottom.

`wrap-reverse`

Same as wrap but new lines appear above the first line (bottom-to-top stacking).

> 💡 **The responsive card pattern.** *`flex-wrap: wrap` combined with `flex: 1 1 250px` on items creates a self-wrapping responsive grid — items sit side by side when there is room and drop to their own rows on small screens, with no media queries needed for the basic behaviour.*

8 — gap: Space Between Items

`gap` adds consistent space between flex items — but never at the outer edges of the container. This makes it far cleaner than adding margins to individual items.

🎨 **CSS — gap property**

```
.container {
    display: flex;

    gap: 20px;          /* same row and column gap */
    gap: 16px 32px;     /* row-gap  column-gap */

    row-gap: 16px;      /* gap between wrapped rows */
    column-gap: 32px;   /* gap between items in a row */
}
```

`gap` only creates space between items. Adding `padding` to the container handles the space between items and the container's edges — the two properties work together, not against each other.

> ⚠ **Avoid margin-right hacks.** A common older pattern is adding `margin-right` to every flex item, then using `:last-child { margin-right: 0 }` to remove the trailing gap. `gap` does this automatically — always prefer it.

9 — The flex Property: Grow, Shrink, Basis

The `flex` property is set on *items* (not the container) and controls how each item sizes itself relative to available space. It is shorthand for three sub-properties:

🎨 CSS — The flex shorthand

```
.item {
    flex:  1    1    auto;
    /*     ↑    ↑    ↑
           |    |    └── flex-basis: starting size before grow/shrink is a
           |    └─────── flex-shrink: 1 = can shrink, 0 = never shrink
           └──────────── flex-grow:   0 = won't grow, 1+ = will grow
}

/* Common shorthand values: */
.item { flex: 1; }          /* grow equally, can shrink, basis: 0 */
.item { flex: 0 0 auto; }   /* natural size, no growing, no shrinking */
.item { flex: 0 0 240px; }  /* fixed 240px sidebar that never resizes */
.item { flex: 1 1 200px; }  /* responsive: start at 200px, wrap and grow *
```

flex-grow: claiming extra space

`flex-grow` is a unitless ratio. If one item has `flex-grow: 3` and two others have `flex-grow: 1`, the first item claims three parts of the free space while each of the others claims one part. An item with `flex-grow: 0` (the default) does not grow at all.

flex-shrink: giving up space

`flex-shrink` is the mirror of `flex-grow`. When items overflow their container, items with a higher shrink value give up proportionally more space. The default is `1` — all items shrink

equally. Setting `flex-shrink: 0` prevents an item from ever shrinking below its `flex-basis` — essential for fixed-width sidebars.

flex-basis: the starting size

`flex-basis` is the item's hypothetical size before growing or shrinking is applied. It overrides `width` in a row container, or `height` in a column container. Use `auto` to use the item's natural content size, `0` to start from nothing and grow proportionally from scratch, or a fixed value like `300px`.

> **The four flex values you will write most often:**
>
> - `flex: 1` — equal-width columns filling available space
> - `flex: 0 0 auto` — item stays its natural size; does not grow or shrink
> - `flex: 0 0 240px` — fixed-width sidebar
> - `flex: 1 1 200px` — self-wrapping responsive item; starts at 200px, grows and wraps as needed

10 — align-self: Per-Item Alignment Override

`align-self` is set on an individual flex item and overrides the container's `align-items` value for that item only. It accepts the same values: `auto` (inherit from container — the default), `stretch`, `flex-start`, `flex-end`, `center`, `baseline`.

🎨 **CSS — align-self on one item**

```css
.container {
    display: flex;
    align-items: center;    /* all items vertically centred */
    height: 100px;
}

.container .badge {
    align-self: flex-end;    /* this one sits at the bottom */
}
```

A typical use case: a navbar where most items are vertically centred, but a notification badge or a "New!" chip is pinned to the bottom of the bar to visually stand out.

11 — Practical Patterns

Navigation bar: logo left, links right

🎨 CSS

```
.navbar {
    display: flex;
    justify-content: space-between;  /* logo left, links right */
    align-items: center;
    padding: 0 24px;
    height: 60px;
}

.nav-links {
    display: flex;          /* nested flex for the link group */
    gap: 24px;
    align-items: center;
}
```

Equal-height card row with bottom-pinned buttons

🎨 CSS

```
.card-row {
    display: flex;
    gap: 20px;
    /* align-items defaults to stretch — all cards equal height */
}

.card {
    flex: 1;
    display: flex;
    flex-direction: column;
```

```
    padding: 20px;
}

.card .btn {
    margin-top: auto;    /* pushes button to card bottom regardless of cont
}
```

`margin-top: auto` on the last item in a column flex container claims all remaining space above itself, effectively pinning it to the bottom. This is the standard "sticky card button" pattern.

Fixed sidebar + flexible main

🎨 CSS

```
.page {
    display: flex;
    min-height: 100vh;
}

.sidebar {
    flex: 0 0 240px;    /* fixed — never grows or shrinks */
}

.main {
    flex: 1;              /* claims all remaining width */
}
```

12 — Quick Reference

Properties set on the flex container

PROPERTY	DEFAULT	WHAT IT CONTROLS
`display: flex`	—	Activates flex layout; makes direct children flex items
`flex-direction`	`row`	Direction of the main axis (row / column)

PROPERTY	DEFAULT	WHAT IT CONTROLS
`justify-content`	`flex-start`	How items and free space are distributed along the main axis
`align-items`	`stretch`	How items are aligned on the cross axis
`flex-wrap`	`nowrap`	Whether items wrap to new lines when they overflow
`gap`	`0`	Space between items (not at container edges)
`align-content`	`normal`	Distribution of space between wrapped rows or columns

Properties set on flex items

PROPERTY	DEFAULT	WHAT IT CONTROLS
`flex-grow`	`0`	Proportion of extra space the item claims (0 = no growth)
`flex-shrink`	`1`	How much the item gives up when space is tight (0 = never shrink)
`flex-basis`	`auto`	Starting size before grow/shrink is applied
`flex`	`0 1 auto`	Shorthand for flex-grow / flex-shrink / flex-basis
`align-self`	`auto`	Per-item override of the container's align-items
`order`	`0`	Visual order within the flex line (does not affect DOM order)

🖍️ Exercises

Build each layout in a fresh HTML file and open it in your browser. Attempt the task before looking at the solution — the process of working it out is where the learning happens.

> **EXERCISE 1**
>
> Create a horizontal navigation bar: a site logo on the left and three links (**Home**, **About**, **Contact**) on the right. The logo and links should be vertically centred. The bar

should have a dark background and a height of 60px. Use Flexbox only — no floats, no absolute positioning.

Hint: `justify-content: space-between` pushes the logo and link group to opposite ends. Wrap the links in a `<div>` and make that a nested flex container with `gap`.

▶ Show Sample Solution

EXERCISE 2

Build a row of three feature cards. Each card should have a title, a short paragraph of text (make one card have much more text than the others), and a "Learn more" button at the bottom. All cards should be equal height, and the button should always sit at the very bottom of its card regardless of how much text the card contains.

Hint: make the row a flex container and each card `flex: 1`. Make each card itself a column flex container. Give the paragraph `flex: 1` so it grows, or use `margin-top: auto` on the button.

▶ Show Sample Solution

EXERCISE 3

Create a sidebar layout with a fixed **240px sidebar** on the left and a flexible **main content area** filling the rest of the viewport. Both should stretch to at least full viewport height. The sidebar should never grow wider or shrink narrower than 240px.

Hint: `flex: 0 0 240px` locks the sidebar at exactly 240px. `flex: 1` on the main area claims all remaining width. Set `min-height: 100vh` on the wrapper.

▶ Show Sample Solution

EXERCISE 4

Build a responsive tag cloud. Create 10 pill-shaped tags in a container. On a wide screen they sit in a row; when there is not enough room they should wrap to new lines with consistent gaps. Tags should never grow — they should only be as wide as their label text. Wrap should use `gap`, not margins.

Hint: `flex-wrap: wrap` on the container, `gap: 8px` for spacing. Do not set `flex-grow` on the tags — let them stay at their natural content width.

▶ Show Sample Solution

CHAPTER 2

Flexbox Advanced — Alignment, Wrapping, Ordering

📐 Chapter 2 — Flexbox Advanced

Chapter 1 covered the core Flexbox properties — `justify-content`, `align-items`, `flex-wrap`, and `flex`. This chapter goes deeper: `align-content` for multi-line containers, the `order` property for visual reordering, auto margins as a layout superpower, and the precise mechanics behind how `flex-grow` and `flex-shrink` share space. These are the tools that separate basic Flexbox usage from fluid, production-ready layouts.

1 — align-content: Space Between Wrapped Lines

In Chapter 1 you learned that `align-items` positions items on the cross axis within a single flex line. When `flex-wrap` is on and items break into multiple rows (or columns), a second property takes over: `align-content`.

`align-content` distributes space between and around *flex lines* — the rows themselves — along the cross axis. It works exactly like `justify-content` does for items on the main axis, but for wrapped rows on the cross axis.

> ⚠️ **align-content only works with multiple lines.** If `flex-wrap: nowrap` (the default) is set, all items are on a single line and `align-content` has no effect whatsoever. You must have `flex-wrap: wrap` (or `wrap-reverse`) for it to do anything. This is the most common source of confusion with this property.

VALUE	BEHAVIOUR
`flex-start`	Lines packed at the start of the cross axis. Free space at the end.
`flex-end`	Lines packed at the end. Free space at the start.
`center`	Lines grouped in the centre of the cross axis.
`space-between`	First line at start, last at end. Remaining space shared equally between lines.
`space-around`	Equal space on each side of every line. Edge gaps half the size of inner gaps.

VALUE	BEHAVIOUR
`space-evenly`	Equal gaps between all lines and at both cross-axis edges.
`stretch` *default*	Lines stretch to fill the container's cross-axis size evenly.

> **align-items vs align-content — the distinction:**
>
> - `align-items` — positions individual items within their flex line
> - `align-content` — positions the flex lines themselves within the container
>
> Both can be active simultaneously: `align-content` sets where the rows sit in the container, and `align-items` sets how items sit within each row.

2 — order: Visual Reordering Without Touching HTML

By default, flex items display in the same order they appear in the HTML (DOM order). The `order` property overrides this — it accepts any integer (negative, zero, or positive) and items are rendered from lowest to highest order value. Items with the same order value keep their relative DOM order.

🎨 **CSS — Using order**

```
/* Default: all items have order: 0 */
/* Items with equal order render in DOM order */

.item-c { order: -1; }  /* moves before all order-0 items */
.item-a { order: 1;  }  /* moves after all order-0 items  */

/* Result: C → B → (other 0s) → A */

/* Inside a media query — reorder for mobile: */
@media (max-width: 600px) {
    .sidebar { order: 2; }   /* sidebar after main on mobile */
```

```
    .main    { order: 1; }   /* main content first on mobile  */
}
```

> ⚠️ **order only changes visual order, not DOM order.** Screen readers, tab navigation, and search engines follow the HTML source order — not the visual order. If you use `order` to reorder content meaningfully, ensure the DOM order still makes sense for keyboard and assistive technology users. Use it for cosmetic adjustments, not to correct structural problems in your HTML.

3 — Auto Margins: The Alignment Superpower

Setting `margin: auto` on a flex item works differently than it does in normal block flow. In a flex container, `auto` margins absorb all available free space in their direction. This makes them a powerful tool for pushing items around — often more elegantly than wrapping things in extra containers.

The most common real-world uses:

🎨 **CSS — Auto margin patterns**

```
/* ── Pattern 1: Push one item to the far end ── */
/* Navbar: logo + links left, CTA button right */
.navbar { display: flex; align-items: center; gap: 20px; }
.nav-cta { margin-left: auto; }   /* absorbs all space, sits at far right

/* ── Pattern 2: Split a row at a specific item ── */
/* Items before .sep left-align, items after right-align */
.sep { margin-left: auto; }

/* ── Pattern 3: Centre one item in a column ── */
.column-container {
    display: flex;
    flex-direction: column;
    height: 300px;
}
.centred-item {
    margin-top: auto;
```

```
    margin-bottom: auto;   /* equal space above and below = centred */
}
```

Auto margins beat `justify-content` when you only want to push one specific item while leaving the others in their natural position.

> 💡 **Auto margins override justify-content.** *When a flex item has an `auto` margin, it consumes all free space in that direction. `justify-content` only distributes space that items have not already claimed with auto margins — so the two interact in a predictable but important way.*

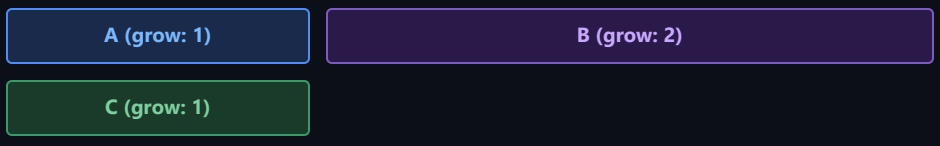
4 — How flex-grow and flex-shrink Actually Work

Understanding the math behind flex sizing helps you predict and control layouts — especially when items have different sizes or grow/shrink values.

flex-grow: distributing free space

After items are placed at their `flex-basis` sizes, the browser calculates how much space is left over. This free space is divided among items with a positive `flex-grow` value in proportion to their grow numbers.

Example: 600px container, three items with flex-basis: 0



Total grow units: 1 + 2 + 1 = **4**

Space per unit: 600px ÷ 4 = **150px**

A gets: 1 × 150px = **150px** B gets: 2 × 150px = **300px** C gets: 1 × 150px = **150px**

🎨 **CSS — flex-grow example**

```
.container { display: flex; width: 600px; }

.a { flex: 1 1 0; }   /* gets 150px */
.b { flex: 2 1 0; }   /* gets 300px */
```

```
.c { flex: 1 1 0; }    /* gets 150px */

/* Shorthand: flex: 1 sets grow:1, shrink:1, basis:0 */
/* flex: 2 sets grow:2, shrink:1, basis:0             */
```

flex-shrink: giving back space

When items overflow the container, shrinkage kicks in. The browser calculates how much overflow exists and distributes the shrinkage among items with a positive `flex-shrink` value — but weighted by their `flex-basis` size (larger items give up more space by default).

> **Preventing shrinkage is common for fixed-width elements.** A sidebar, a logo, an icon button — anything that should stay a specific size regardless of how crowded the container gets should have `flex-shrink: 0` (or equivalently, `flex: 0 0 Xpx`).

5 — flex-basis: 0 vs auto vs a Fixed Value

The difference between `flex-basis: 0` and `flex-basis: auto` is one of the most misunderstood subtleties in Flexbox.

FLEX-BASIS: AUTO

Starts from natural size

The item first takes up space for its content, then distributes any remaining free space. Items with more content start larger before growing. Used in `flex: 1 1 auto`.

FLEX-BASIS: 0

Starts from zero

All space is treated as "free space" to be distributed by `flex-grow`. Items grow proportionally from scratch, ignoring content size. Used in `flex: 1` (shorthand for `1 1 0`).

FLEX-BASIS: 200PX

Starts from a fixed size

Item begins at exactly 200px along the main axis, then grows or shrinks from there. Overrides `width` (in a row container). Use for items that need a minimum guaranteed size before growing.

FLEX-BASIS: CONTENT

Exactly the content size

Similar to `auto` but always uses the content's intrinsic size, ignoring any explicit `width` or `height`. Useful but less widely supported in older browsers.

🎨 CSS — flex-basis: 0 vs auto side-by-side

```css
/* flex: 1  →  flex-grow:1, flex-shrink:1, flex-basis:0 */
/* All space is free space. Items grow in exact proportion. */
.equal-columns { flex: 1; }

/* flex: 1 1 auto */
/* Items start at their content size, then share remaining space. */
/* A long-text item starts wider and gains less free space than a */
/* short-text item, which starts narrow and gains more.           */
.content-proportional { flex: 1 1 auto; }
```

Use `flex: 1` (basis 0) for truly equal columns. Use `flex: 1 1 auto` when you want columns that roughly match their content size but still fill the container.

6 — min-width: 0 — The Hidden Flexbox Gotcha

Flex items have an implicit `min-width: auto` by default. This means a flex item will never shrink below the size of its content — even if you set `flex-shrink: 1`. This protects content from being squashed, but it can cause layouts to overflow unexpectedly.

🎨 CSS — Fixing overflow caused by min-width: auto

```css
/* Problem: a flex item containing a long word or a wide element */
/* refuses to shrink below its content width, causing overflow.  */

/* Fix 1: override min-width on the item */
.flex-item {
```

```
    flex: 1;
    min-width: 0;   /* allows the item to shrink below content width */
}

/* Fix 2: add overflow: hidden to the item (also clips content) */
.flex-item {
    flex: 1;
    overflow: hidden;   /* establishes a new block-formatting context */
}

/* Both fixes let long text truncate with text-overflow: ellipsis */
.flex-item p {
    white-space: nowrap;
    overflow: hidden;
    text-overflow: ellipsis;
}
```

If you have a flex item that contains a code block, a long URL, a table, or any other wide content, and the layout breaks unexpectedly, `min-width: 0` is almost always the fix.

7 — Nested Flex Containers

Any flex item can itself be a flex container. Nested flex is the standard approach for complex layouts — the outer container handles the macro structure (columns, rows), and inner containers handle the detail within each cell.

🎨 CSS — A three-panel layout with nested flex

```
/* Outer: horizontal split — sidebar + content area */
.page {
    display: flex;
    min-height: 100vh;
}

.sidebar { flex: 0 0 240px; }

/* Inner: content area — stacks header, main, footer vertically */
.content {
    flex: 1;
```

```
    display: flex;
    flex-direction: column;
}

.content header { flex: 0 0 60px;  }
.content main   { flex: 1;         }  /* takes remaining height */
.content footer { flex: 0 0 48px;  }

/* Innermost: card grid inside main */
.card-grid {
    display: flex;
    flex-wrap: wrap;
    gap: 20px;
}
.card-grid .card { flex: 1 1 280px; }
```

> 💡 **When to stop nesting and use Grid instead.** *If you find yourself nesting more than two or three flex containers to create a layout, consider switching to CSS Grid for the outer structure. Grid handles two-dimensional layouts more naturally. Flexbox excels for one-dimensional component interiors — button groups, nav links, form rows, card contents.*

8 — Responsive Flexbox Patterns

Pattern 1: Row → column on small screens

🎨 CSS

```
.feature-row {
    display: flex;
    gap: 24px;
}

.feature-row .item { flex: 1; }

@media (max-width: 640px) {
    .feature-row {
        flex-direction: column;   /* items stack vertically */
```

```
    }
}
```

Pattern 2: Auto-wrapping card grid (no media query needed)

🎨 CSS

```css
.card-grid {
    display: flex;
    flex-wrap: wrap;
    gap: 20px;
}

.card-grid .card {
    flex: 1 1 280px;    /* start at 280px, grow to fill, wrap when needed *
    max-width: 100%;     /* prevent single card filling row beyond 100% */
}
```

With `flex: 1 1 280px`, items sit side by side when the container is wide enough, and automatically drop to their own rows as the viewport narrows. No media queries required for the wrapping behaviour itself.

Pattern 3: Stacked mobile nav → horizontal desktop nav

🎨 CSS

```css
.nav {
    display: flex;
    flex-direction: column;    /* mobile-first: stack links */
    gap: 4px;
}

@media (min-width: 768px) {
    .nav {
        flex-direction: row;     /* desktop: links in a row */
        gap: 24px;
        align-items: center;
    }
}
```

Pattern 4: Reordering with order in a media query

CSS

```css
/* Desktop: image | text (DOM order) */
.feature {
    display: flex;
    gap: 32px;
    align-items: center;
}

/* Mobile: text first, image second (better UX) */
@media (max-width: 640px) {
    .feature { flex-direction: column; }
    .feature .image { order: 2; }
    .feature .text  { order: 1; }
}
```

9 — Quick Reference

Properties covered in this chapter

PROPERTY	SET ON	WHAT IT DOES
`align-content`	Container	Distributes wrapped flex lines along the cross axis. Requires `flex-wrap: wrap` and multiple lines.
`order`	Item	Integer controlling visual position. Default: 0. Lower = earlier. Does not change DOM order.
`margin-[side]: auto`	Item	Absorbs all free space in that direction. Can push other items away or centre a single item.
`min-width: 0`	Item	Overrides the default `min-width: auto` so the item can shrink below its content size.

align-content values (same pattern as justify-content)

VALUE	LINES DISTRIBUTED AS
`stretch` *default*	Lines expand to fill available cross-axis space
`flex-start`	Lines packed at cross-axis start
`flex-end`	Lines packed at cross-axis end
`center`	Lines grouped in the middle
`space-between`	First/last lines at edges, equal gaps between remaining lines
`space-around`	Equal space on each side of every line
`space-evenly`	Identical gaps between all lines and at edges

🖍️ Exercises

Build each layout in a fresh HTML file. Try the task before checking the solution.

EXERCISE 1

Create a fixed-height container (300px) with `flex-wrap: wrap` containing eight items. Use `align-content: space-between` to push the first row to the top and the last row to the bottom, with equal space distributed between any middle rows.

Hint: Set a fixed height on the container, `flex-wrap: wrap`, and `align-content: space-between`. Give each item a `flex: 0 0 100px` so they have a defined width and wrap naturally.

▶ Show Sample Solution

EXERCISE 2

Build a navigation bar with four links on the left (**Home**, **About**, **Services**, **Blog**) and a **Sign In** button on the far right — all without using `justify-content: space-between`. Use an auto margin instead.

Hint: Put all five items in a single flex row. Apply `margin-left: auto` to the Sign In button — it absorbs all free space to its left, pushing it to the right edge.

▶ Show Sample Solution

EXERCISE 3

Create a flex row with four items where the widths are proportional: the second item should be twice as wide as the first and third, and the fourth should be three times as wide. All items should grow from zero (not from their content size). No fixed widths — use `flex-grow` only.

Hint: Set `flex: 1 1 0` on the first and third items (or just `flex: 1`), `flex: 2 1 0` (or `flex: 2`) on the second, and `flex: 3 1 0` (or `flex: 3`) on the fourth. The basis must be 0 for pure proportional sizing.

▶ Show Sample Solution

EXERCISE 4

Build a mobile-first feature section. On small screens (default), three feature blocks should stack in a column with text content first. On wide screens (min-width: 768px), they should sit in a row. Add a media query that also reverses the order of the first block — so its image appears on the right on desktop — without changing the HTML.

Hint: Start with `flex-direction: column`. The media query switches to `row`. Use `order` on the image and text inside the first feature block to swap them on desktop.

▶ Show Sample Solution

CHAPTER 3

Grid Fundamentals

🔲 Chapter 3 — Grid Fundamentals

Flexbox is one-dimensional — it lays items out in a row or a column. CSS Grid is two-dimensional — it controls both rows and columns at the same time. Grid is how you build page layouts, dashboard panels, image galleries, and anything that requires items to be aligned in both directions simultaneously. This chapter covers the foundational Grid concepts: defining tracks, the fr unit, auto placement, explicit item placement, named areas, and alignment.

1 — Grid vs Flexbox: When to Use Which

Grid and Flexbox are complementary, not competing. A common rule of thumb:

USE FLEXBOX WHEN…

Layout flows in one direction

Navigation links, button groups, card interiors, tag lists — anything where items line up in a single row or column and you mainly care about how they distribute along that line.

USE GRID WHEN…

Layout needs rows AND columns

Page layouts, galleries, dashboards, form grids — anything where items need to align in two dimensions and you want precise control over both axes at once.

> **In practice, you use both.** A page uses Grid for the outer structure (header / sidebar / main / footer). Inside each section, Flexbox handles the one-dimensional components — a navbar, a row of cards, a form row. Nesting them is completely normal and expected.

2 — Creating a Grid Container

Like Flexbox, Grid is activated with a single declaration on the parent element. Direct children automatically become **grid items**.

🎨 CSS — display: grid

```css
/* The container becomes a grid context */
.container {
    display: grid;
}

/* Without grid-template-columns, all items stack in one column */
/* That is the implicit grid's default behaviour                */
```

`display: inline-grid` also works — the container becomes inline-level (shrinks to content width) but its children still use grid layout.

Two key terms you need before going further:

- **Grid track** — a single row or column in the grid (the space between two grid lines).
- **Grid cell** — the intersection of a row track and a column track. The smallest addressable unit in a grid.
- **Grid area** — one or more grid cells that an item occupies (can span multiple rows and/or columns).
- **Grid lines** — the dividing lines between tracks. Columns have vertical lines numbered left-to-right from 1. Rows have horizontal lines numbered top-to-bottom from 1. Lines can also be numbered from the end using negative numbers (-1 is the last line).

3 — grid-template-columns and grid-template-rows

These two properties define the *explicit* grid — the tracks you declare up front. Each space-separated value creates one track of that size.

🎨 **CSS — Defining column and row tracks**

```css
.grid {
    display: grid;

    /* Three fixed-width columns */
    grid-template-columns: 200px 200px 200px;

    /* Same thing using repeat() */
```

```
    grid-template-columns: repeat(3, 200px);

    /* Three equal flexible columns */
    grid-template-columns: 1fr 1fr 1fr;
    grid-template-columns: repeat(3, 1fr);

    /* Fixed sidebar + flexible main */
    grid-template-columns: 240px 1fr;

    /* Two explicit rows: 60px header, flexible content */
    grid-template-rows: 60px 1fr;
}
```

> 💡 **repeat(count, size)** *is a shorthand that saves typing.* `repeat(4, 1fr)` *is identical to* `1fr 1fr 1fr 1fr` *— it creates four equal-width tracks. You can also mix:* `repeat(3, 1fr) 200px` *creates three flexible columns plus one fixed column.*

4 — The fr Unit: Fractions of Available Space

`fr` stands for *fraction*. One `fr` represents one share of the space left over after fixed-size tracks have claimed their portion. It works like `flex-grow` but for grid tracks.

🎨 **CSS — How fr is calculated**

```
.grid {
    display: grid;
    width: 900px;

    /* Fixed column takes 200px, remaining 700px split 1:2 */
    grid-template-columns: 200px 1fr 2fr;
    /*                       ↑     ↑    ↑
                           200px  233px 467px
                           (700px ÷ 3 units × 1 = 233px)
                           (700px ÷ 3 units × 2 = 467px) */
}
```

```
/* fr tracks expand to fill the container. */
/* Unlike %, they automatically account for gap spacing. */
```

`fr` units respect gap — they distribute space after gaps are subtracted. Using percentages with gaps requires manual calculation. Always prefer `fr` for flexible tracks.

> **fr vs % — why fr wins.** If you have three columns with `gap: 20px` and use percentages, you need to calculate: `calc((100% - 40px) / 3)` for each column. With `fr`, you just write `1fr 1fr 1fr` and the browser handles the gap subtraction automatically.

5 — gap: Space Between Tracks

`gap` works identically in Grid as in Flexbox — it adds space between cells but not at the outer edges. In Grid you can set row and column gaps independently.

🎨 **CSS — gap in a grid**

```css
.grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);

    gap: 20px;          /* same row and column gap */
    gap: 16px 24px;     /* row-gap  column-gap */

    row-gap: 16px;      /* vertical space between rows */
    column-gap: 24px;   /* horizontal space between columns */
}
```

6 — The Implicit Grid and Auto Placement

When you have more items than your defined tracks can hold, the browser creates new rows automatically. These extra rows form the **implicit grid**. You can control the size of these auto-created rows with `grid-auto-rows`.

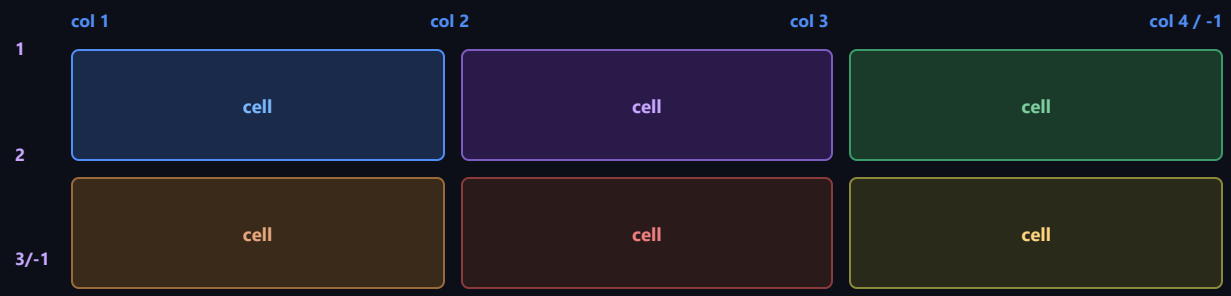
🎨 CSS — Controlling the implicit grid

```css
.card-grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    gap: 20px;

    /* All auto-created rows: at least 120px, can grow */
    grid-auto-rows: minmax(120px, auto);
}

/* grid-auto-flow controls whether new items go across (row)
   or down (column) when they overflow the explicit grid */
.grid {
    grid-auto-flow: row;    /* default: fill rows first */
    grid-auto-flow: column; /* fill columns first instead */
    grid-auto-flow: dense;  /* back-fill gaps left by spanning items */
}
```

`minmax(min, max)` defines a track that is at least `min` and at most `max`. `minmax(120px, auto)` means: at least 120px tall, but grow if content is taller. This is perfect for card grids where content length varies.

7 — Grid Lines: Placing Items Explicitly

Every track boundary is a numbered grid line. Columns have vertical lines numbered 1, 2, 3… from left to right. Rows have horizontal lines numbered 1, 2, 3… from top to bottom. You can also address lines from the end using negative numbers: **-1 is the last line**, -2 is the second to last, and so on.



• Column lines (vertical) — numbered left → right • Row lines (horizontal) — numbered top → bottom

Use `grid-column` and `grid-row` on items to place them at specific line intersections, or use `span` to make an item cover multiple tracks.

🎨 CSS — Placement with line numbers and span

```
/* Line-number syntax: start-line / end-line */
.header {
    grid-column: 1 / 4;   /* from line 1 to line 4 = full width (3-col gri
    grid-column: 1 / -1;  /* same — -1 always means the last line */
}

/* Span keyword: "start here, cross N tracks" */
.wide-card {
    grid-column: span 2;   /* 2 columns wide, auto-placed */
}
.featured {
    grid-column: span 2;
    grid-row: span 2;      /* 2 columns wide AND 2 rows tall */
}

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

8 — grid-template-areas: Naming Regions

`grid-template-areas` lets you draw your layout as ASCII art in your CSS. Each quoted string represents a row; each word in the string is a named zone. Use a dot (`.`) for an empty cell. Items then reference their zone by name with `grid-area`.

🎨 CSS — The layout above in code

```css
.page {
    display: grid;
    grid-template-columns: 200px 1fr;
    grid-template-rows: 60px 1fr 48px;
    min-height: 100vh;
    gap: 0;

    grid-template-areas:
        "header  header"
        "sidebar main"
        "footer  footer";
}

/* Each direct child references its zone by name */
header  { grid-area: header;  }
.sidebar{ grid-area: sidebar; }
main    { grid-area: main;    }
footer  { grid-area: footer;  }
```

The names in `grid-template-areas` implicitly define named grid lines too — "header" creates lines named `header-start` and `header-end` on both axes.

🎨 CSS — Responsive: sidebar below main on mobile

```css
@media (max-width: 640px) {
    .page {
        grid-template-columns: 1fr;        /* single column */
        grid-template-rows: auto;           /* all rows auto-sized */
        grid-template-areas:
            "header"
            "main"
            "sidebar"
            "footer";   /* sidebar moves below main */
    }
}
```

Redefining `grid-template-areas` in a media query completely reorganises the layout with no HTML changes. This is the killer feature of named areas.

> ⚠️ **Area names must form rectangles.** Every name in `grid-template-areas` must occupy a contiguous rectangular region — no L-shapes or disconnected cells. An invalid shape causes the entire declaration to be ignored. Use a dot (`.`) for intentionally empty cells.

9 — Alignment in Grid

Grid has six alignment properties — three on the container and two on items — plus a distinction between aligning *items within their cells* vs aligning the *grid tracks within the container*.

PROPERTY	SET ON	CONTROLS	AXIS
`justify-items`	Container	Items within their cells (inline = horizontal)	Inline (horizontal)
`align-items`	Container	Items within their cells (block = vertical)	Block (vertical)
`justify-content`	Container	Grid tracks within the container (if tracks don't fill it)	Inline (horizontal)
`align-content`	Container	Grid tracks within the container	Block (vertical)
`justify-self`	Item	That item within its cell (overrides justify-items)	Inline (horizontal)
`align-self`	Item	That item within its cell (overrides align-items)	Block (vertical)

🎨 **CSS — Common alignment patterns**

```
.grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);

    /* Items fill their cells (default for both) */
    justify-items: stretch;   /* default */
    align-items: stretch;     /* default */
```

```css
    /* Centre items within their cells */
    justify-items: center;
    align-items: center;
}

/* Centre a specific item within its cell */
.special {
    justify-self: center;
    align-self: end;
}

/* Centre the entire grid inside the container */
/* (only visible when tracks are smaller than the container) */
.grid-centered {
    justify-content: center;
    align-content: center;
}
```

> 💡 **place-items and place-self.** *The shorthands `place-items: center` and `place-self: center` set both the inline and block axes at once — they are equivalent to writing `align-items: center; justify-items: center` and save a line of code when both values are the same.*

10 — minmax(): Flexible Track Bounds

`minmax(min, max)` defines a track that is at least `min` wide/tall and at most `max` — the browser picks a size within that range. It unlocks truly responsive track sizing without media queries.

🎨 **CSS — minmax() examples**

```css
/* ── In grid-template-columns ── */
.grid {
    /* Column at least 200px, can grow to 1fr */
    grid-template-columns: minmax(200px, 1fr) 1fr;

    /* Fixed sidebar, content never narrower than 300px */
    grid-template-columns: 240px minmax(300px, 1fr);
```

```
}

/* ── In grid-auto-rows ── */
.card-grid {
    grid-template-columns: repeat(3, 1fr);

    /* Rows: at least 120px, but expand for taller content */
    grid-auto-rows: minmax(120px, auto);
}

/* ── Common special values ── */
/* minmax(0, 1fr)   — like 1fr but can shrink to zero (avoids overflow) */
/* minmax(min-content, 1fr) — at least as wide as the content */
/* minmax(200px, max-content) — at least 200px, grows to fit content */
```

You will see `minmax(0, 1fr)` often in advanced grids. The default `min` of an `fr` track is `auto` (content size), which can cause overflow. Setting the min to `0` removes that lower bound.

11 — Quick Reference

Container properties

PROPERTY	WHAT IT DEFINES
`display: grid`	Creates a grid formatting context; direct children become grid items
`grid-template-columns`	Explicit column tracks (sizes, fr units, repeat())
`grid-template-rows`	Explicit row tracks
`grid-template-areas`	Named region map — draw the layout as ASCII strings
`gap` / `row-gap` / `column-gap`	Space between tracks (not at edges)
`grid-auto-rows`	Size for implicitly created rows (overflow items)
`grid-auto-columns`	Size for implicitly created columns
`grid-auto-flow`	How auto-placed items fill the grid: `row`, `column`, `dense`

PROPERTY	WHAT IT DEFINES
`justify-items` / `align-items`	Default alignment of all items within their cells
`justify-content` / `align-content`	Distribution of tracks within the container

Item properties

PROPERTY	WHAT IT CONTROLS
`grid-column: s / e`	Start and end column lines (or `span N`)
`grid-row: s / e`	Start and end row lines (or `span N`)
`grid-area: name`	Assigns item to a named area from `grid-template-areas`
`grid-area: r/c/re/ce`	Shorthand for row-start / col-start / row-end / col-end
`justify-self` / `align-self`	Per-item alignment within its cell

Key functions

FUNCTION	USAGE
`repeat(n, size)`	Create *n* tracks of *size*. Avoids repetition: `repeat(4, 1fr)`
`minmax(min, max)`	Track is at least *min*, at most *max*. E.g. `minmax(120px, auto)`

🖍️ Exercises

Build each layout in a fresh HTML file. Try to write the CSS before peeking at the solution.

EXERCISE 1

Create a 3-column card grid. The grid should have 9 cards. All columns should be equal width. There should be a 20px gap between cards. The cards should have a fixed minimum height of 120px but grow taller if content requires it. Use `grid-auto-rows` and `minmax()`.

Hint: `repeat(3, 1fr)` for equal columns. `grid-auto-rows: minmax(120px, auto)` for the row height.

▶ Show Sample Solution

EXERCISE 2

Build a full-page layout using `grid-template-areas` with four zones: **header** (full width, 60px tall), **sidebar** (200px wide, fills remaining height), **main** (flexible, fills remaining space), and **footer** (full width, 48px tall). The page should fill at least the full viewport height.

Hint: Use a 2-column template (`200px 1fr`) and a 3-row template (`60px 1fr 48px`). Define three area strings. Each HTML element gets `grid-area: zonename`.

▶ Show Sample Solution

EXERCISE 3

Create an editorial grid with 12 items on a 4-column grid. The first item should span all 4 columns (a featured hero card). The next three items should each span 1 column. The remaining 8 items fill in normally with auto placement. Use a fixed row height of `minmax(180px, auto)`.

Hint: Set `repeat(4, 1fr)` for columns. Give the first item `grid-column: 1 / -1` (or `span 4`). The rest need no explicit placement.

▶ Show Sample Solution

EXERCISE 4

Take the page layout from Exercise 2 and make it responsive: on screens narrower than 700px, collapse to a single column with the order: header → main → sidebar → footer. Redefine `grid-template-areas` inside a media query — no HTML changes.

Hint: In the media query, set `grid-template-columns: 1fr` and rewrite `grid-template-areas` with four single-cell rows. You can remove `grid-template-rows` and let it go `auto`.

▶ Show Sample Solution

CHAPTER 4

Grid Advanced — Template Areas, auto-fit/fill

⚡ Chapter 4 — Grid Advanced

Chapter 3 covered the mechanics of CSS Grid — defining tracks, placing items, and naming areas. This chapter pushes into the techniques that make Grid feel like a superpower: `auto-fit` and `auto-fill` for truly responsive layouts without a single media query, named grid lines for surgical placement, dense packing for gallery-style grids, and stacking items inside the same cell. By the end, you will be writing professional-grade layouts that adapt intelligently to any screen size.

1 — auto-fill vs auto-fit

Both keywords go inside `repeat()` as a replacement for a fixed count. Instead of telling the browser *how many* tracks to create, you tell it to create *as many as fit*:

🎨 CSS — syntax

```css
/* Instead of a fixed number… */
grid-template-columns: repeat(3, 200px);

/* …let the browser decide how many tracks fit */
grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
grid-template-columns: repeat(auto-fit,  minmax(200px, 1fr));
```

The difference only shows up when items **do not fill all available space** — for example, you have 3 cards but the container is wide enough for 6.

- **auto-fill** — creates as many tracks as the minimum allows, *including empty ones*. The filled items share space with those phantom tracks.
- **auto-fit** — creates the same tracks, but then *collapses the empty ones to zero width*. The filled items expand to take up all the available space.

> **When does the difference matter?** In practice, `auto-fit` is almost always what you want for card grids and galleries — you want items to fill the available width. Use `auto-fill` when you want

the grid to reserve space for items that might appear later (e.g., a fixed-width grid that gets items added dynamically), or when you want to prevent cards from stretching too wide when there are only a few of them.

2 — The Intrinsically Responsive Grid

Combine `auto-fit` with `minmax()` and you get the most powerful layout one-liner in CSS — a grid that automatically adjusts its column count to the container width, with *zero media queries*:

🎨 **CSS — the RAM pattern (Repeat, Auto, Minmax)**

```css
.card-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(240px, 1fr));
    gap: 24px;
}

/* Reading this declaration:
   "Create as many 1fr columns as will fit,
    where each column is at least 240px wide"

   At 1200px: fits 4 columns (300px each)
   At  900px: fits 3 columns (300px each)
   At  560px: fits 2 columns (280px each)
   At  250px: fits 1 column  (250px)

   No media queries. No breakpoints. Just math. */
```

*This is sometimes called the "RAM" pattern — **R**epeat, **A**uto-fit, **M**inmax. It's one of the most widely used CSS Grid patterns in production.*

⚠️ **Don't use 0 as the minimum.** `repeat(auto-fit, minmax(0, 1fr))` gives you the same behaviour as `repeat(auto-fit, 1fr)` — columns never collapse because 0 is always satisfiable. Always use a meaningful pixel minimum that reflects the actual smallest usable size for the content inside.

> 💡 **Capping item width.** *To stop cards from growing too wide on very large screens, pair the RAM pattern with* `max-width` *on the grid:* `.card-grid { max-width: 1200px; margin-inline: auto; }`. *The grid reflows automatically up to 1200px, then the container centres.*

3 — Named Grid Lines

Grid lines have numbers by default (1, 2, 3… and -1, -2, -3 from the end). You can also give them **names** using square brackets in `grid-template-columns` or `grid-template-rows`. Named lines make placement declarations read like English instead of coordinates.

🎨 **CSS — defining and using named lines**

```css
.layout {
    display: grid;

    /* Name the lines in brackets before/after each track size */
    grid-template-columns:
        [full-start]
            1fr
        [main-start]
            minmax(0, 720px)
        [main-end]
            1fr
        [full-end];

    grid-template-rows:
        [header-start] 60px
        [header-end content-start] 1fr
        [content-end];

    /* A line can have multiple names — separate them with a space */
}

/* Place items by name instead of number */
.hero {
    grid-column: full-start / full-end;    /* full bleed */
    grid-row: header-start / content-end;
}
```

```
.article {
    grid-column: main-start / main-end;    /* constrained centred column *
}
```

Named lines are especially valuable in complex editorial layouts where you have a "full-bleed" outer zone and a centred "content" zone — you give items one of two names and they snap to the right zone without counting line numbers.

Named lines from grid-template-areas. When you define `grid-template-areas`, the browser automatically creates named lines for you — every area named `foo` generates column lines `foo-start` and `foo-end`, and the same for rows. You can use those auto-generated names in `grid-column` and `grid-row` even if you are not using `grid-area` on the item itself.

🎨 **CSS — using the auto-named lines from grid-template-areas**

```
.page {
    display: grid;
    grid-template-columns: 200px 1fr;
    grid-template-areas:
        "header  header"
        "sidebar main"
        "footer  footer";
}

/* These named lines are created automatically:
   column: header-start, header-end, sidebar-start,
           sidebar-end / main-start, main-end, footer-start, footer-end
   row:    header-start, header-end, sidebar-start,
           sidebar-end, main-start, main-end, footer-start, footer-end */

/* A decorative overlay that sits in the "main" column */
.overlay {
    grid-column: main-start / main-end;
    grid-row: header-start / footer-end;
}
```

4 — grid-template-areas in Depth

You covered the basics in Chapter 3. Let's go deeper with a real magazine-style layout that mixes multiple area sizes, uses dots for empty cells, and rewires completely on mobile.

🎨 CSS — complex magazine layout

```css
.magazine {
    display: grid;
    grid-template-columns: 2fr 1fr 1fr;
    grid-template-rows: auto auto auto auto;
    gap: 16px;

    grid-template-areas:
        "hero    hero    promo"    /* row 1 */
        "hero    side1   promo"    /* row 2 — hero spans 2 rows */
        "feature side1   side2"    /* row 3 — side1 spans 2 rows */
        "feature .       side2";   /* row 4 — dot = empty cell   */
}

.hero    { grid-area: hero;    }
.promo   { grid-area: promo;   }
.side1   { grid-area: side1;   }
.side2   { grid-area: side2;   }
.feature { grid-area: feature; }

/* On tablet: 2-column with different region arrangement */
@media (max-width: 800px) {
    .magazine {
        grid-template-columns: 1fr 1fr;
        grid-template-areas:
            "hero    hero"
            "promo   side1"
            "feature side2";
    }
}

/* On mobile: single column, reading order */
@media (max-width: 500px) {
    .magazine {
```

```
        grid-template-columns: 1fr;
        grid-template-areas:
            "hero"
            "feature"
            "promo"
            "side1"
            "side2";
    }
}
```

The dot (.) creates an intentionally empty cell — useful for maintaining structural alignment when not every zone has content at every breakpoint.

> ⚠ **Each named area must be a rectangle.** The names in your area map must tile into contiguous rectangles — no L-shapes, no disconnected islands. The browser silently ignores an invalid `grid-template-areas` declaration, so a layout that looks correct may be falling back silently. Check DevTools grid overlay when things look off.

5 — Dense Packing with grid-auto-flow: dense

When some items span multiple columns or rows, the auto-placement algorithm leaves **gaps** — cells it cannot fill because the next item is too wide. By default those gaps remain empty. Adding `dense` instructs the browser to go back and fill them with later (smaller) items.

> ⚠ **Dense changes the visual order.** Items are reordered visually (but not in the DOM) to fill gaps. This creates a mismatch between the reading/tab order (DOM) and the visual order, which is an accessibility concern. Use dense only for purely decorative grids (image galleries, tile walls) where the visual order does not need to match the logical order. Never use it where each item has a meaningful position.

🎨 **CSS — dense packing for an image gallery**

```
.gallery {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    grid-auto-rows: 200px;
```

```
    grid-auto-flow: dense;      /* back-fill any gaps */
    gap: 8px;
}

/* Landscape photos span 2 columns, portraits are 1×1 */
.gallery .landscape { grid-column: span 2; }
.gallery .tall      { grid-row:    span 2; }
.gallery .featured  { grid-column: span 2; grid-row: span 2; }

.gallery img {
    width: 100%;
    height: 100%;
    object-fit: cover;
}
```

6 — Overlapping Items and z-index

Unlike Flexbox, CSS Grid lets you place multiple items into the **same cell or area**. They stack on top of each other. Use `z-index` to control which appears in front — and unlike static positioned elements, grid items respond to `z-index` without needing `position: relative`.

🎨 **CSS — text overlay on a card using Grid stacking**

```
.card {
    display: grid;
    grid-template: 1fr / 1fr;  /* one row, one column */
    height: 240px;
    border-radius: 8px;
    overflow: hidden;
}

/* All three children share the same cell — they stack */
.card > * {
    grid-column: 1;
    grid-row: 1;
}

.card__image   { z-index: 1; object-fit: cover; width: 100%; height: 100%;
```

```
.card__overlay { z-index: 2; background: linear-gradient(to top, rgba(0,0,
.card__text    {
    z-index: 3;
    display: flex;
    flex-direction: column;
    justify-content: flex-end;
    padding: 20px;
}
```

`grid-template: 1fr / 1fr` is a shorthand for `grid-template-rows: 1fr; grid-template-columns: 1fr;` — a single row and single column, each taking all available space.

7 — Advanced Responsive Patterns

Pattern 1 — Sidebar that wraps below content

🎨 CSS — sidebar that collapses automatically

```
.with-sidebar {
    display: grid;
    grid-template-columns: minmax(260px, 1fr) minmax(0, 2fr);
    gap: 24px;
}

/* When the container is narrower than 520px (260+260),
   the second column can no longer maintain its minimum and
   the layout naturally breaks to a single column.
   No media query needed. */
```

This pattern uses `minmax` on both columns. When the container cannot satisfy both minimums simultaneously, the grid implicitly wraps — but only works when `grid-template-columns` creates 2 columns in the first place. Pair with a media query if you need guaranteed wrapping at a specific breakpoint.

Pattern 2 — Full-bleed sections in a constrained layout

🎨 CSS — three-zone layout for article pages

```css
/*  Three columns: edge | content | edge
    Content is capped at 720px and centred.
    Some sections can "break out" to full width. */
.article-page {
    display: grid;
    grid-template-columns:
        [full-start]
        1fr
        [content-start]
        minmax(0, 720px)
        [content-end]
        1fr
        [full-end];
}

/* All direct children default to the content zone */
.article-page > * {
    grid-column: content;    /* shorthand: content-start / content-end */
}

/* Full-bleed hero, pull-quotes, or banners break out */
.full-bleed {
    grid-column: full;       /* full-start / full-end */
}
```

When you use a named area like `content` or `full` in a shorthand, the browser expands it to `content-start / content-end` automatically — this works because named lines ending in `-start` and `-end` are treated as named areas by the placement algorithm.

Pattern 3 — Equal-height rows across multiple cards

🎨 CSS — vertically aligned card interiors using nested Grid

```css
.card-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(220px, 1fr));
    grid-auto-rows: 1fr;      /* all rows same height */
    align-items: stretch;
}
```

```
/* Each card is itself a grid — interior rows align across cards */
.card {
    display: grid;
    grid-template-rows: auto 1fr auto;  /* header | body | footer */
}
/* card__header: auto height  */
/* card__body:   grows (1fr)   */
/* card__footer: auto (button) — always pinned to bottom */
```

8 — Quick Reference

auto-fill vs auto-fit

KEYWORD	EMPTY TRACKS	ITEM WIDTH WITH FEW ITEMS	BEST FOR
`auto-fill`	Kept (phantom slots)	Stays at minimum — does not stretch	Grids where empty slots should reserve space
`auto-fit`	Collapsed to 0	Expands to fill the container	Card grids, galleries — most common use case

grid-auto-flow values

VALUE	BEHAVIOUR	WARNING
`row`	Fill rows left→right, new row when full (default)	—
`column`	Fill columns top→bottom, new column when full	—
`dense`	Back-fill gaps left by spanning items	Visual ≠ DOM order — accessibility risk
`row dense`	Row direction + dense back-fill	Same accessibility warning

Named line syntax

PATTERN	EFFECT
`[name]` before a track	Names the line before that track
`[name1 name2]`	One line can have multiple names
`grid-column: name-start / name-end`	Place item using line names
`grid-column: name` (single name)	Shorthand — expands to name-start / name-end if both exist
Auto-names from `grid-template-areas`	Area "foo" creates foo-start, foo-end on both axes

🖍️ Exercises

Each exercise builds on techniques from this chapter. Attempt the solution before revealing the answer.

EXERCISE 1

Create a responsive image gallery using the RAM pattern. Cards should be at least 180px wide and expand to fill available space. All rows should be 220px tall. Images should use `object-fit: cover`. Every third card (nth-child) should span 2 columns (a featured photo). Use `grid-auto-flow: dense` so the featured cards don't leave gaps.

Hint: `repeat(auto-fit, minmax(180px, 1fr))`. Give `.card:nth-child(3n)` a `grid-column: span 2`. Add `grid-auto-flow: dense` to the container.

▶ Show Sample Solution

EXERCISE 2

Build an article page layout where all content is constrained to a 680px centred column, but any element with class `.full-bleed` breaks out to the full container width. Use named grid lines (`[full-start]`, `[content-start]`, `[content-end]`, `[full-end]`) and set all direct children to the content zone by default.

Hint: Three columns — `1fr [content-start] minmax(0, 680px) [content-end] 1fr`. Set `.article > *` to `grid-column: content-start / content-end`. Override with `.full-bleed { grid-column: full-start / full-end; }`.

▶ Show Sample Solution

EXERCISE 3

Create a card component where a background image, a gradient overlay, and a text block all share the same grid cell, stacking visually. The text should always appear at the bottom of the card. The card should be 280px tall. Use only `display: grid` for the stacking — no absolute positioning.

Hint: Set the card to a single-cell grid (`grid-template: 1fr / 1fr`). Give all children `grid-column: 1; grid-row: 1`. Use `z-index` to layer them. Make the text container `display: flex; align-items: flex-end`.

▶ Show Sample Solution

EXERCISE 4

Define a 3-column magazine grid using `grid-template-areas` with these zones: **hero** (spans all 3 columns, 2 rows), **lead** (column 1, rows 3–4), **featured** (columns 2–3, row 3), **aside** (column 2, row 4), **promo** (column 3, row 4). Then write a media query for screens under 700px that collapses it to a single column in reading order: hero → lead → featured → aside → promo.

Hint: Draw the 4-row area map first. Hero spans rows 1–2 with the same name in both rows across all columns. Then define a 5-row single-column version in the media query.

▶ Show Sample Solution

CHAPTER 5

Advanced Selectors — Combinators, Attribute Selectors, :nth-child

🎯 Chapter 5 — Advanced Selectors

Basic selectors (element, class, id) get you 80% of the way. The remaining 20% — fine-grained control without adding extra classes — comes from combinators, attribute selectors, and structural pseudo-classes. Master these and you will write CSS that reacts to the structure of your HTML: "style every other list item," "target links that go to PDFs," "select the last card in a row." No JavaScript required.

1 — Combinators

A combinator is a character that sits *between* two simple selectors and describes the relationship between the elements they match. There are four.

COMBINATOR	SYMBOL	MEANING
Descendant	`A B` (space)	B is anywhere inside A (any depth)
Child	`A > B`	B is a *direct* child of A only
Adjacent sibling	`A + B`	B immediately follows A (same parent)
General sibling	`A ~ B`	B follows A anywhere (same parent, any distance)

🎨 CSS — practical combinator uses

```
/* ── Descendant — broad, targets all nested matches ── */
.card p      { color: #666; }   /* all p inside .card */
.nav a       { color: #fff; }   /* all links in nav */

/* ── Child — precise, avoids unintended nested matches ── */
.dropdown > li { border-bottom: 1px solid #eee; }
/* only first-level li items, not li inside nested ul */

/* ── Adjacent sibling — styling "the thing after" ── */
```

```
h2 + p         { margin-top: 0; }   /* first paragraph after a heading */
input + label  { font-weight: bold; }

/* ── General sibling — all following matches ── */
.error ~ .field { border-color: red; }  /* all .field after .error */

/* ── Chaining combinators ── */
.sidebar > ul > li > a { display: block; }  /* precise drill-down */
```

> ⚠ **The descendant combinator is the slowest to evaluate.** The browser reads selectors right-to-left — it first finds all elements matching the right side, then checks ancestors. A selector like `.nav a` is fine, but `div div div a` forces the engine to check three ancestor levels. In large DOMs, prefer adding a class or using a child combinator to limit traversal depth.

2 — Attribute Selectors

Attribute selectors match elements based on their HTML attributes and attribute values. They use square brackets: `[attr]`. They are especially powerful for targeting links, inputs, and custom `data-` attributes without adding extra classes.

SELECTOR	MATCHES WHEN…	EXAMPLE
`[attr]`	Attribute exists (any value or empty)	`[disabled]`
`[attr="val"]`	Exact match	`[type="checkbox"]`
`[attr^="val"]`	Value **starts with** val	`[href^="https"]`
`[attr$="val"]`	Value **ends with** val	`[href$=".pdf"]`
`[attr*="val"]`	Value **contains** val anywhere	`[class*="icon"]`
`[attr~="val"]`	Space-separated list **contains word** val	`[class~="btn"]`
`[attr|="val"]`	Equals val or starts with val- (language codes)	`[lang|="en"]`
`[attr="val" i]`	Case-insensitive match (add `i` flag)	`[href$=".PDF" i]`

🎨 CSS — practical attribute selector patterns

```css
/* Automatically add an icon after external links */
a[href^="https"]::after {
    content: " ↗";
    font-size: 0.75em;
    opacity: 0.7;
}

/* PDF link: different colour + icon */
a[href$=".pdf"],
a[href$=".PDF" i] {  /* case-insensitive — matches .PDF too */
    color: #e8a87d;
}

/* Style all disabled form controls */
input[disabled],
button[disabled],
select[disabled] {
    opacity: 0.45;
    cursor: not-allowed;
}

/* data-* attributes as style hooks — no class needed */
[data-status="active"]   { border-left: 3px solid #7ecfa0; }
[data-status="inactive"] { opacity: 0.5; }

/* Input types */
input[type="text"]   { border-radius: 4px; }
input[type="submit"] { background: #4f8ef7; }
```

💡 **[attr~="val"] vs [attr*="val"].** *The tilde `~=` treats the attribute value as a space-separated word list — `[class~="btn"]` matches `class="btn btn-primary"` but NOT `class="btn-primary"`. The asterisk `*=` does a substring match anywhere — `[class*="btn"]` matches both. For class attributes specifically, the `~=` operator behaves exactly like checking for a class name.*

3 — Structural Pseudo-classes

Structural pseudo-classes select elements based on their **position in the document tree** — their relationship to their siblings and parents — without needing any attribute or class.

Position pseudo-classes

PSEUDO-CLASS	SELECTS
`:first-child`	Element that is the first child of its parent
`:last-child`	Element that is the last child of its parent
`:only-child`	Element that is the only child of its parent
`:first-of-type`	First element of its tag type among siblings
`:last-of-type`	Last element of its tag type among siblings
`:only-of-type`	Only element of its tag type among siblings
`:nth-child(n)`	Element at position n (see An+B below)
`:nth-last-child(n)`	Same but counting from the end
`:nth-of-type(n)`	nth sibling of the same tag type
`:nth-last-of-type(n)`	Same but counting from the end
`:empty`	Element with no children (not even text nodes)

🎨 **CSS — structural pseudo-class patterns**

```css
li:first-child { font-weight: bold; }  /* first list item */
li:last-child  { border-bottom: none; }  /* remove last separator */

/* Remove top margin from the very first element in an article */
.article > *:first-child { margin-top: 0; }
.article > *:last-child  { margin-bottom: 0; }

/* Hide a badge when it has no content */
.badge:empty { display: none; }
```

```
/* Only-child: different treatment for solo items */
.card:only-child { max-width: 480px; margin: 0 auto; }
```

4 — The An+B Formula

`:nth-child()` and its relatives accept an expression in the form `An+B` where **n** is a counter that starts at 0 and increments by 1 forever. Plug in each value of n to find the matching positions.

> **Reading An+B:** `A` is the cycle size (step), `B` is the offset. The formula `3n+1` generates the sequence: 3(0)+1=**1**, 3(1)+1=**4**, 3(2)+1=**7**, 3(3)+1=**10**… — every third element starting from the first. Negative values of A or B clip the sequence: `-n+3` generates 3, 2, 1 — then 0 and negative numbers are ignored, giving you elements 1, 2, 3 only (the first three).

🎨 **CSS — An+B patterns decoded**

```
li:nth-child(odd)      /* 1, 3, 5, 7… — alias for 2n+1 */
li:nth-child(even)     /* 2, 4, 6, 8… — alias for 2n   */
li:nth-child(3)        /* exactly the 3rd item          */
li:nth-child(3n)       /* 3, 6, 9, 12… — every 3rd      */
li:nth-child(3n+1)     /* 1, 4, 7, 10… — 1st of each 3  */
li:nth-child(-n+3)     /* 1, 2, 3     — first three      */
li:nth-child(n+4)      /* 4, 5, 6, 7… — from 4th onward  */
li:nth-last-child(1)   /* the very last item            */
li:nth-last-child(-n+2) /* last two items               */

/* Combine to create a range: from 4th to 8th */
li:nth-child(n+4):nth-child(-n+8) { background: #1a2a4a; }
```

5 — :nth-child vs :nth-of-type

This is the most common source of confusion with structural selectors. The difference is subtle but important.

- `:nth-child(n)` — counts *all* siblings regardless of tag, then checks if the element at position n also matches the rest of the selector.
- `:nth-of-type(n)` — counts only siblings with the *same tag*, then selects the nth one. The tag is part of what is counted.

HTML — mixed siblings

```
<article>
    <h2>Title</h2>           <!-- child 1, h2 #1 -->
    <p>First para.</p>       <!-- child 2, p #1  -->
    <p>Second para.</p>      <!-- child 3, p #2  -->
    <blockquote>…</blockquote> <!-- child 4 -->
    <p>Third para.</p>       <!-- child 5, p #3  -->
</article>
```

CSS — how the two selectors differ on the HTML above

```
/* article p:nth-child(2)
   → finds child #2 of article, checks if it's a p
   → matches "First para." ✓ (it IS both 2nd child AND a p) */

/* article p:nth-child(3)
   → finds child #3, checks if it's a p
   → matches "Second para." ✓ */

/* article p:nth-child(4)
   → finds child #4, checks if it's a p
   → NO match — child #4 is a blockquote, not a p ✗ */

/* article p:nth-of-type(2)
   → among p siblings only (ignores h2, blockquote)
   → selects the 2nd p: "Second para." ✓ */

/* article p:nth-of-type(3)
   → selects the 3rd p: "Third para." ✓
   → works even though it's child #5, not #3 */
```

Rule of thumb: *If your list/grid is a sequence of the same tag (all `li`, all `div.card`), both work identically. If siblings are mixed tags (like an article with h2, p, blockquote), use `:nth-of-type` to target a specific tag by its own count. Or better*

yet, add a class and use `:nth-child` with the class selector.

> ⚠️ **:nth-of-type is tag-specific.** `p:nth-of-type(2)` counts only `p` elements. `.card:nth-of-type(2)` does NOT count by class — it counts by tag, then filters by class. So if your cards are all `div.card`, it counts all div siblings, which is rarely what you want. For class-based patterns, stick with `:nth-child`.

6 — :not(), :is(), and :where()

:not() — the negation pseudo-class

`:not(selector)` selects elements that do *not* match the given selector. Modern browsers support a comma-separated list inside `:not()`.

🎨 **CSS — :not() patterns**

```css
/* All links except those inside nav */
a:not(nav a) { text-decoration: underline; }

/* All list items except the last (no trailing border) */
li:not(:last-child) { border-bottom: 1px solid #eee; }

/* Inputs that are neither disabled nor read-only */
input:not([disabled]):not([readonly]) {
    border-color: #4f8ef7;
}

/* Modern: comma list inside :not() */
:not(h1, h2, h3, h4) { font-size: 1rem; }

/* Cards except .featured */
.card:not(.featured) { opacity: 0.7; }
```

:is() — match-any grouping (with specificity)

`:is()` takes a selector list and matches any element that matches any selector in the list. It eliminates repetitive compound selectors and is **forgiving** — invalid selectors inside are silently ignored rather than breaking the whole rule.

🎨 CSS — :is() cleans up repetition

```
/* Before :is() — four separate rules */
.card h1,
.card h2,
.card h3,
.card h4 { color: #fff; }

/* After :is() — one rule */
.card :is(h1, h2, h3, h4) { color: #fff; }

/* :is() on the ancestor side */
:is(article, section, aside) p {
    line-height: 1.8;
}
/* equivalent to:
   article p, section p, aside p { line-height: 1.8; } */
```

Specificity of :is(): *takes the specificity of its most specific argument. `:is(h1, .title)` has class-level specificity (0,1,0) because `.title` is the most specific argument inside.*

:where() — zero-specificity grouping

`:where()` is identical to `:is()` in what it matches, but it always contributes **zero specificity**. Use it for base/reset styles that you want to be easily overrideable.

🎨 CSS — :is() vs :where() specificity

```
/* :is() — uses highest specificity of its arguments */
:is(h1, h2, h3) { margin-top: 1.5rem; }
/* specificity: 0,0,1 (element-level — h1/h2/h3 are type selectors) */

:is(h1, .page-title) { font-size: 2rem; }
/* specificity: 0,1,0 (class-level — .page-title dominates) */
```

```
/* :where() — always zero specificity */
:where(h1, h2, h3, h4, h5, h6) { font-family: system-ui; }
/* specificity: 0,0,0 — any later rule beats this */

/* Great for resets and base styles */
:where(ul, ol) { list-style: none; padding: 0; }
/* user can easily override: nav ul { list-style: disc; } */
```

> **Coming in the Advanced course: :has().** The `:has()` pseudo-class is the "parent selector" that CSS lacked for decades — `div:has(img)` targets any div that contains an image. It is fully supported in modern browsers and is one of the most transformative selectors added to CSS in years. It is covered in CSS Advanced Chapter 4.

7 — Quick Reference

Combinators

COMBINATOR	SYNTAX	MATCHES
Descendant	`A B`	B anywhere inside A
Child	`A > B`	B that is a direct child of A
Adjacent sibling	`A + B`	B immediately after A, same parent
General sibling	`A ~ B`	All B siblings after A, same parent

Attribute selectors

SELECTOR	MATCH TYPE
`[attr]`	Attribute exists
`[attr="val"]`	Exact value
`[attr^="val"]`	Starts with
`[attr$="val"]`	Ends with

SELECTOR	MATCH TYPE
`[attr*="val"]`	Contains substring
`[attr~="val"]`	Contains word (space-separated)
`[attr="val" i]`	Case-insensitive match

An+B cheat sheet

EXPRESSION	MEANING	MATCHES (OF 12)
`odd` / `2n+1`	Odd-numbered	1,3,5,7,9,11
`even` / `2n`	Even-numbered	2,4,6,8,10,12
`3n`	Every 3rd	3,6,9,12
`4n+1`	1st of each group of 4	1,5,9
`-n+3`	First three	1,2,3
`n+4`	From 4th onwards	4,5,6,...12
`n+3` combined with `-n+7`	Range: 3rd to 7th	3,4,5,6,7

Functional pseudo-classes

PSEUDO-CLASS	SPECIFICITY	PURPOSE
`:not(A)`	Specificity of A	Exclude matches
`:is(A, B, C)`	Highest in list	Match-any grouping, forgiving
`:where(A, B, C)`	Always 0	Match-any with no specificity weight

🖍️ Exercises

Solve each exercise using only CSS selectors — no added classes or HTML changes.

EXERCISE 1

You have a `<ul class="menu">` containing several `<li>` elements separated by a bottom border. Write one selector to add the border to every item **except the last**, instead of adding a border to all and resetting the last. Then write a second rule that makes the very first item bold, and a third that styles every other item (starting from the 2nd) with a subtle background.

Hint: `:not(:last-child)` for the border. `:first-child` for bold. `:nth-child(even)` for alternating rows.

▶ Show Sample Solution

EXERCISE 2

Write CSS that automatically adds a small arrow icon (↗) after every link that points to an external URL (starts with `https`), and a lock icon (🔒) after every link whose `data-auth="required"` attribute is set. Do not use JavaScript or add classes — use only attribute selectors and `::after`.

Hint: `a[href^="https"]::after { content: " ↗"; }` and `a[data-auth="required"]::after`.

▶ Show Sample Solution

EXERCISE 3

You have a grid of 9 `.card` elements. Style the layout so: (a) every card gets a base border; (b) the first card spans 2 columns and gets a highlighted background; (c) every 3rd card gets an orange accent border; (d) the last two cards get reduced opacity. Use only structural pseudo-classes — no extra classes on the cards.

Hint: Use `:first-child` for (b), `:nth-child(3n)` for (c), and `:nth-last-child(-n+2)` for (d).

▶ Show Sample Solution

EXERCISE 4

Rewrite this repetitive CSS using `:is()` to collapse the rules into fewer lines. Then write an alternative version using `:where()` and explain in a comment why you might prefer `:where()` for reset-style rules.

Hint: Both sides of the combinator can use `:is()`. The specificity difference is the key for the :where() explanation.

```css
/* Original — repetitive */
.card h2, .card h3, .card h4 { color: #fff; }
.modal h2, .modal h3, .modal h4 { color: #fff; }
.sidebar h2, .sidebar h3, .sidebar h4 { color: #fff; }
```

▶ Show Sample Solution

CHAPTER 6

The Cascade and Specificity in Depth

🌊 Chapter 6 — The Cascade and Specificity in Depth

CSS stands for Cascading Style Sheets — the cascade is not an incidental feature, it is the central algorithm that makes CSS work. When two rules both try to set the same property on the same element, the cascade decides which one wins. Understanding this algorithm precisely — not just roughly — is the difference between CSS that behaves predictably and CSS that feels like a fight you keep losing. This chapter covers the full cascade, specificity scoring, inheritance, and practical strategies for writing CSS that is easy to reason about.

1 — What the Cascade Does

Every CSS property on every element has exactly one computed value. When multiple declarations compete to set the same property, the cascade resolves the conflict in a fixed order. **The cascade does not care about your file structure or your intent — it cares about three things, in order of priority:**

1. Origin and Importance — where the rule came from, and whether it carries `!important`
2. Specificity — the "weight" of the selector that matched
3. Source Order — when specificity ties, whichever declaration comes last wins

Only when step 1 cannot decide a winner does the cascade move to step 2. Only when step 2 ties does it move to step 3. Steps are evaluated in order until a winner is found.

> **Inheritance is not the cascade.** These two mechanisms are often confused. The cascade resolves conflicts between competing declarations. Inheritance is a separate mechanism: when no declaration sets a property on an element, the browser checks if that property *inherits* from the parent — and if so, uses the parent's computed value. Section 7 covers inheritance in detail.

2 — Step 1: Origin and Importance

Every CSS declaration comes from one of three **origins**:

- **User-agent stylesheet** — the browser's built-in defaults (buttons look button-like, links are blue and underlined, etc.).
- **User stylesheet** — custom styles the person browsing has applied (accessibility overrides, custom fonts). Rare in practice but part of the spec.
- **Author stylesheet** — CSS you write as a developer. This is almost always the only origin you work with.

In the normal (non-`!important`) cascade, author styles override user styles, which override user-agent styles. **Adding `!important` reverses the order** — user-agent `!important` is strongest, then user `!important`, then author `!important`.

> **Why does !important reverse the order?** The reversal is deliberate. If author `!important` always beat everything, users with visual impairments who set high-contrast colours in their operating system would have their settings overridden by websites. The spec protects user overrides by making user `!important` stronger than author `!important`.

In practice — working entirely in author stylesheets — `!important` just overrides other author declarations for that property. But it sits at a completely different layer in the algorithm, *not* in the specificity calculation. This is why a low-specificity rule with `!important` beats a high-specificity rule without it.

3 — Step 2: Specificity — the A·B·C Score

Specificity is a three-part score written as **A · B · C**. Each part is an independent integer. They are compared left-to-right — A takes absolute priority over B, B takes absolute priority over C. You cannot "add up" C values to beat a B value: 0·0·1000 is still less specific than 0·1·0.

A — ID COLUMN	B — CLASS COLUMN	C — ELEMENT COLUMN
+1 for each **ID selector** (`#name`)	+1 for each **class** (`.name`), **attribute** (`[attr]`), or **pseudo-class** (`:hover` , `:nth-child()`)	+1 for each **element** type selector (`div` , `p`) or **pseudo-element** (`::before`)

A — ID COLUMN	B — CLASS COLUMN	C — ELEMENT COLUMN

The universal selector `*` adds 0 to every column. Combinators (+, >, ~, space) add nothing. `:not()`, `:is()`, `:has()` contribute the specificity of their most specific argument. `:where()` always contributes 0.

Scoring examples

SELECTOR	A	B	C	NOTES
`*`	0	0	0	Universal selector — zero specificity
`div`	0	0	1	One element type
`div p`	0	0	2	Two element types (combinator adds 0)
`.card`	0	1	0	One class
`.card p`	0	1	1	One class + one element
`.card.active`	0	2	0	Two classes (chained — no space)
`a[href]`	0	1	1	Attribute selector counts as class-level
`a:hover`	0	1	1	Pseudo-class counts as class-level
`p::first-line`	0	0	2	Pseudo-element counts as element-level
`#nav`	1	0	0	ID — column A, always beats B and C
`#nav .link:hover`	1	2	0	ID + class + pseudo-class
`:not(.card)`	0	1	0	`:not()` contributes its argument's specificity
`:is(h1, .title)`	0	1	0	`:is()` takes highest — `.title` wins
`:where(h1, .title)`	0	0	0	`:where()` always 0,0,0

4 — Step 3: Source Order

When two declarations have **identical specificity and the same origin**, the one that appears *later* in the stylesheet wins. This is the tiebreaker of last resort — and it is why the

order of your CSS rules and `<link>` tags matters.

CSS — source order in practice

```
/* File order matters — later rules win ties */
.btn          { background: #4f8ef7; }  /* overridden */
.btn          { background: #3a9a6a; }  /* wins — same spec, later */

/* Link tag order matters too */
<link rel="stylesheet" href="base.css">       <!-- loaded first -->
<link rel="stylesheet" href="components.css"> <!-- overrides base for ties
<link rel="stylesheet" href="overrides.css">  <!-- has the last word -->

/* Practical rule: put more specific/later styles at the bottom of your fi
/* Reset → Base → Layout → Components → Utilities → Page-specific
```

5 — !important in Depth

`!important` elevates a declaration out of the normal specificity stack entirely. It sits in a separate "important" tier of the cascade — above all normal declarations regardless of their specificity.

CSS — !important syntax and behaviour

```
/* Syntax: goes inside the declaration, before the semicolon */
.card {
    color: red !important;
}

/* This beats even a highly specific rule with no !important */
#app #main .card.active span {  /* 2·2·1 — very high specificity */
    color: blue;
}
/* .card { color: red !important } wins — !important is in a different tie

/* When two !important rules compete, specificity decides between them */
```

```
.card   { color: red  !important; }  /* 0·1·0 */
#hero   { color: blue !important; }  /* 1·0·0 → wins: higher A */
```

When !important is legitimate

SITUATION	VERDICT
Utility classes that must always apply (`.visually-hidden` , `.sr-only` , `.d-none`)	✓ **Appropriate**
Overriding third-party CSS (external library, widget, embed) you cannot modify	✓ **Appropriate**
Accessibility overrides that *must not* be overridden	✓ **Appropriate**
Fixing specificity you created yourself but can't be bothered to refactor	✗ Warning — technical debt
Fighting against another `!important` you wrote earlier	✗ Specificity war — refactor instead
Making something "more important" as a general habit	✗ Almost certainly a design smell

> ⚠ **The !important escalation trap.** Using `!important` to win a specificity fight creates a new problem: the next person who needs to override you must also use `!important`, then with higher specificity, then you respond with higher specificity again. CSS becomes unmaintainable. The root fix is almost always to reduce the specificity of the original rule, not to add more `!important`.

6 — Inheritance

Inheritance is a separate mechanism from the cascade. When **no declaration sets a property** on an element (the cascade finds no winner), the browser checks if that property is defined as **inheritable**. If so, it uses the computed value of the nearest ancestor that has it.

Not all properties inherit. Properties that control *typography and text* generally inherit; properties that control *box layout and decoration* generally do not.

Common inherited properties

CATEGORY	INHERITED PROPERTIES
Typography	`color`, `font`, `font-family`, `font-size`, `font-weight`, `font-style`, `font-variant`, `line-height`, `letter-spacing`, `word-spacing`, `text-align`, `text-transform`, `text-indent`
Lists	`list-style`, `list-style-type`, `list-style-position`
Tables	`border-collapse`, `border-spacing`, `caption-side`
Visibility	`visibility`, `cursor`

Common non-inherited properties

CATEGORY	NON-INHERITED PROPERTIES
Box model	`width`, `height`, `margin`, `padding`, `border`
Background	`background`, `background-color`, `background-image`
Layout	`display`, `position`, `top/right/bottom/left`, `float`
Flex/Grid	`flex`, `grid-column`, `align-self`, and all container properties

Explicit inheritance keywords

You can force or reset inheritance with four special keywords that work on *any* property:

🎨 **CSS — inherit, initial, unset, revert**

```css
.child {
    /* inherit: force inheritance even for non-inherited properties */
    background: inherit;   /* background doesn't inherit — this forces it
    color: inherit;         /* colour inherits anyway, but explicit is cle

    /* initial: reset to the CSS specification's initial value */
```

```
    /* (not the browser default — the spec-defined initial value) */
    color: initial;          /* = 'canvastext' (browser-dependent black) */
    display: initial;        /* = 'inline' (spec initial, not block) */

    /* unset: inherits if property is inheritable, else uses initial */
    /* behaves like 'inherit' for color, like 'initial' for display */
    all: unset;               /* reset ALL properties at once — useful for

    /* revert: reset to the browser's default stylesheet value */
    /* (the value the browser would apply if you wrote no CSS) */
    display: revert;         /* div → block, span → inline, etc. */
}

/* all: unset is common in component resets */
.widget {
    all: unset;  /* wipe the slate — set every property to unset state */
}
```

`initial` is the spec-defined default, which is not always the browser default. For example, `display: initial` gives you `inline` — the spec's initial value — even on a `div`. If you want the browser's default (`block` for `div`), use `revert` instead.

7 — @layer: A New Tier in the Cascade

CSS now supports a `@layer` rule that adds a new level to the cascade, sitting *between* origin and specificity. Layers let you define ordered buckets of CSS where the **layer order matters more than specificity** — a rule in a higher-priority layer wins even if a rule in a lower-priority layer has higher specificity.

🎨 **CSS — @layer basics (preview)**

```
/* Declare layer order — first listed = lowest priority */
@layer reset, base, components, utilities;

/* Assign rules to a layer */
@layer base {
    a { color: blue; }
}
```

```
@layer utilities {
    /* utilities has higher priority than base — */
    /* this wins even if .text-red has lower specificity than the base rul
    .text-red { color: red; }
}

/* Rules outside any @layer beat all layered rules */
```

`@layer` is a major addition to the cascade that solves specificity wars at the architecture level. It is covered in depth in the CSS Advanced course, Chapter 1. Browser support is excellent (all modern browsers since 2022).

8 — Practical: Writing Manageable CSS

Keep specificity low and consistent

The hardest CSS to maintain is CSS with high specificity variance — some rules at 0·0·1, others at 2·3·2. When you need to override something, you have to match or beat the highest score already in your stylesheet. Start flat and stay flat:

🎨 **CSS — specificity strategies**

```
/* ✗ Avoid nesting that builds specificity */
#sidebar .widget .widget-title a.active {  /* 1·3·1 — very hard to overrid
    color: red;
}

/* ✓ Give the element its own class — target it directly */
.sidebar-active-link {  /* 0·1·0 — easy to override or extend */
    color: red;
}

/* ✓ Avoid IDs in selectors entirely — save them for anchors/JS */
/* ✗ #main h2 { … } — the ID pollutes every h2 inside #main */
/* ✓ .page-main h2 { … } — class-level, much easier to manage  */

/* ✓ Use :where() for base styles — zero specificity */
:where(h1, h2, h3) {
```

```
    font-family: system-ui;  /* easily overrideable — 0·0·0 */
}
```

The order that prevents conflicts

🎨 **CSS — recommended file/section order**

```
/* 1. Reset / normalise — zero or near-zero specificity */
@import "reset.css";

/* 2. Design tokens — custom properties (inherited, low spec) */
:root { --color-primary: #4f8ef7; }

/* 3. Base styles — element selectors, low specificity */
body { font-family: Georgia, serif; }

/* 4. Layout — single class selectors */
/* 5. Components — single class selectors */
/* 6. Utilities — single class, !important where needed */
/* 7. Page-specific overrides — last, highest source order */
```

> 💡 **DevTools is your best teacher.** *In Chrome or Firefox DevTools, clicking an element and viewing the "Styles" panel shows every competing declaration for the selected element, each crossed out (overridden) or active. The source of each rule — file name and line number — is shown on the right. This is the fastest way to understand why a style is or is not applying.*

9 — Quick Reference

Cascade resolution order (highest wins)

PRIORITY	CATEGORY
1 (highest)	Transitions (in progress)
2	`!important` user-agent declarations

PRIORITY	CATEGORY
3	`!important` user declarations
4	`!important` author declarations (your CSS)
5	Animations (in progress)
6	Normal author declarations → resolved by specificity → then source order
7	Normal user declarations
8 (lowest)	Normal user-agent declarations (browser defaults)

Specificity scoring

A (ID)	B (CLASS/ATTR/PSEUDO-CLASS)	C (ELEMENT/PSEUDO-ELEMENT)
`#id`	`.class` · `[attr]` · `:hover` · `:nth-child()`	`div` · `p` · `::before`

Explicit inheritance keywords

KEYWORD	EFFECT
`inherit`	Force: use parent's computed value (even for non-inheritable properties)
`initial`	Reset to the CSS spec's initial value
`unset`	`inherit` if the property inherits, else `initial`
`revert`	Reset to the browser's default stylesheet value
`all: unset`	Apply `unset` to every property at once

🖍️ Exercises

Predict the outcome before revealing the answer — the goal is to reason through the cascade, not just check the result.

EXERCISE 1

Given the HTML `<p id="intro" class="lead highlight">` and these four CSS rules all setting `color`, predict which wins and explain why. Then calculate the specificity score (A·B·C) for each rule.

Rule A: `p { color: gray; }`

Rule B: `.lead { color: blue; }`

Rule C: `.lead.highlight { color: green; }`

Rule D: `#intro { color: red; }`

Hint: Compare column A first. Any rule with an ID selector immediately beats all rules with no ID, regardless of what is in columns B and C.

▶ Show Sample Solution

EXERCISE 2

A junior developer added `color: purple !important` to `.card` to fix a specificity problem, but now they need to override it in `.card.featured`. They try adding `color: gold` to `.card.featured` and it does not work. Explain why, and describe two ways to fix it — one using `!important` and one without.

Hint: Remember where !important sits in the cascade. The second fix should involve refactoring the original rule to remove !important.

▶ Show Sample Solution

EXERCISE 3

You have a `.widget` component with paragraphs inside it. The widget is placed in two different contexts: inside `.sidebar` and inside `.content`. Write CSS that gives `.widget p` a default `font-size: 0.9rem`, overrides it to `1rem` when inside `.content`, and overrides it to `0.8rem` when inside `.sidebar`. Achieve this using only the cascade and specificity — no `!important`.

Hint: Add context to the selector: `.content .widget p` has higher specificity than `.widget p` alone. The same principle applies for `.sidebar .widget p`.

▶ Show Sample Solution

EXERCISE 4

A `.btn` element inside a form has `font-size` set to `16px` on `body`. The button shows at `12px` instead. No CSS explicitly sets `font-size` on `.btn`. Investigate: (a) explain how it could show at 12px if no rule targets it; (b) write a rule using `inherit` to force the button to use the body's font size; (c) explain what `all: revert` would do on `.btn`.

Hint: Browser user-agent stylesheets set `font-size` on form elements to a small size. `font-size` is an inherited property, but user-agent declarations trump inheritance from body.

▶ Show Sample Solution

CHAPTER 7

Media Queries and Responsive Layouts

📱 Chapter 7 — Media Queries and Responsive Layouts

Responsive design is the practice of making a single HTML document adapt to different screen sizes, device capabilities, and user preferences — without serving different HTML to different devices. The core tool is the CSS media query: a conditional block of CSS that applies only when specific conditions about the environment are true. This chapter covers the full media query syntax, the mobile-first philosophy, breakpoint strategy, preference-based queries, and responsive layout patterns that work in production.

1 — The Viewport Meta Tag (Non-Negotiable)

Before a single media query works on a mobile device, you need this tag in your HTML `<head>`:

📄 **HTML — viewport meta tag**

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

Without this tag, mobile browsers render the page at a "virtual viewport" of ~980px and then zoom out to show the full page — completely bypassing your media queries. This single line is the prerequisite for all responsive work.

> **Why does the virtual viewport exist?** Before responsive design existed, websites were built for 980px desktops. When smartphones appeared, mobile browsers had to show those sites somehow — they invented a virtual desktop-sized viewport and zoomed out. Now that responsive design exists, `width=device-width` tells the browser to use the real physical screen width instead, making your media queries meaningful.

- **`width=device-width`** — set the viewport to the device's actual width in CSS pixels.
- **`initial-scale=1`** — do not zoom in or out when the page first loads.

- **Do not add** `user-scalable=no` — it prevents users from zooming, which is an accessibility violation.

2 — @media Syntax

A media query wraps a block of CSS that only applies when its condition evaluates to true. The condition is made up of a **media type** (optional) and one or more **media features** in parentheses.

🎨 CSS — anatomy of a media query

```css
/*  @media  [type]  [and (feature: value)]  { declarations }  */

@media screen and (min-width: 768px) {
    /* applies on screens 768px or wider */
}

/* Media type is optional — defaults to 'all' */
@media (min-width: 768px) {
    /* same effect — 'all' is the default type */
}

/* Print stylesheet — only when printing */
@media print {
    .no-print { display: none; }
    body { font-size: 12pt; color: #000; }
}
```

Media types

TYPE	APPLIES TO
`all`	All devices (default when type is omitted)
`screen`	Screens: computers, phones, tablets
`print`	Print preview and printing

TYPE	APPLIES TO
`speech`	Screen readers (rarely used directly)

3 — Width-Based Media Features

`min-width` and `max-width` are by far the most used media features. They test the *viewport* width — the width of the browser window, not the screen resolution.

🎨 CSS — width features

```css
/* min-width: applies when viewport is AT LEAST this wide */
@media (min-width: 600px) { /* 600px and wider */ }

/* max-width: applies when viewport is AT MOST this wide */
@media (max-width: 599px) { /* 599px and narrower */ }

/* Targeting a range (old syntax) */
@media (min-width: 600px) and (max-width: 959px) { /* tablet only */ }

/* Targeting a range (Level 4 range syntax — see Section 7) */
@media (600px <= width <= 959px) { /* same — cleaner */ }

/* height, orientation */
@media (min-height: 800px) { /* tall viewports */ }
@media (orientation: landscape) { /* wider than tall */ }
@media (orientation: portrait)  { /* taller than wide */ }
```

Viewport width is measured in CSS pixels, not physical pixels. On a Retina (2×) display, a 750-pixel-wide phone has a CSS viewport width of 375px. Media queries always use CSS pixels.

4 — Mobile-First vs Desktop-First

MOBILE-FIRST

Start small — add with min-width

DESKTOP-FIRST

Start large — strip with max-width

Write the base CSS for the smallest viewport. Use `min-width` queries to progressively add complexity as the screen gets wider. Each breakpoint *adds* to what was already there.

Write the base CSS for large screens. Use `max-width` queries to remove or simplify as the screen gets smaller. Each breakpoint *takes away* from the default.

> **Mobile-first is the professional default** — and for good reason. Mobile users are the majority of web traffic. Writing for mobile first means your base styles are simpler (less CSS = faster download). Desktop enhancements are layered on top. Mobile-first also tends to produce cleaner cascades: you add properties, rather than overriding and removing them. Desktop-first leads to more `unset` and `none` overrides.

5 — Breakpoint Strategy

Content-based vs device-based breakpoints

The most common mistake with breakpoints is choosing them by device (*"768 for iPad"*, *"375 for iPhone"*). Devices change every year — your CSS should not be tied to 2024 screen sizes. Instead, **choose breakpoints where your content breaks**: drag the browser window narrow until the layout looks bad, then add a breakpoint there.

🎨 **CSS — common breakpoint values (content-based)**

```css
/* A reasonable starting set — adjust to suit your content */

@media (min-width: 480px)  { /* large phones, landscape */ }
@media (min-width: 640px)  { /* small tablets, phablets */ }
@media (min-width: 768px)  { /* tablets */ }
@media (min-width: 1024px) { /* small laptops */ }
@media (min-width: 1280px) { /* desktops */ }
@media (min-width: 1536px) { /* wide screens */ }

/* You almost certainly don't need all six — start with two or three. */
/* Most layouts need: small (base) + medium (600–768) + large (960+). */
```

> 💡 **Use custom properties to store breakpoint values.** *CSS custom properties cannot be used inside `@media` condition strings, but you can define them as reference tokens in a comment or use a preprocessor variable. In plain CSS, define breakpoints in one place (a `:root` comment block or a dedicated `_breakpoints.css` file) and never write the pixel values from memory — search-and-replace beats trying to remember whether you used 768 or 769.*

6 — Logical Operators

🎨 **CSS — and, comma (or), not**

```css
/* and — ALL conditions must be true */
@media (min-width: 600px) and (orientation: landscape) {
    /* 600px+ AND landscape orientation */
}

/* , (comma = OR) — ANY condition is true */
@media (max-width: 480px), (orientation: portrait) {
    /* narrow screens OR portrait — either triggers this */
}

/* not — negates the entire query (not just the feature) */
@media not (hover: hover) {
    /* devices without a hover capability (touch-only) */
}

/* Combining all three */
@media screen and (min-width: 1024px),
       print  and (min-resolution: 300dpi) {
    /* wide screen OR high-res print */
}
```

not pitfall: *`not` negates the entire query, not just the next condition. `@media not screen and (min-width: 768px)` is read as `not (screen and (min-width: 768px))` — it applies to everything that is NOT (a screen wider than 768px). Wrap with parentheses in complex queries.*

7 — Level 4 Range Syntax

CSS Media Queries Level 4 introduced a cleaner range syntax using comparison operators. It is supported in all modern browsers since 2023 and eliminates the awkward `min-width/max-width` naming.

🎨 CSS — range syntax comparison

```css
/* ── Old syntax ── */
@media (min-width: 600px)                                  { }
@media (max-width: 599px)                                  { }
@media (min-width: 600px) and (max-width: 959px)           { }

/* ── New range syntax ── */
@media (width >= 600px)                                    { }
@media (width <  600px)                                    { }
@media (600px <= width < 960px)                            { }

/* Read the last one as: "when 600px ≤ width < 960px"     */
/* The range syntax also supports height, aspect-ratio: */
@media (45rem <= width <= 80rem) {   /* rem-based breakpoints */  }
```

Many developers prefer `rem`-based breakpoints over `px`: if the user has set a larger browser default font size, `rem` breakpoints scale with it. `48rem` ≈ 768px at the default 16px base, but scales proportionally with user font size preferences.

8 — User Preference Queries

Beyond layout, media queries can detect **user preferences set at the operating system level**. These unlock genuinely responsive experiences — not just different layouts, but fundamentally different visual treatments based on what the user has told their device they need.

`prefers-color-scheme`

OS dark/light mode preference. The most widely implemented preference query — all major platforms expose this.

`dark · light`

`prefers-reduced-motion`

The user has turned on "Reduce Motion" in accessibility settings. Disable or minimise animations.

`reduce · no-preference`

hover

Whether the primary input can hover over elements. `none` = touch-only device. Use to show/hide hover-dependent UI.

`hover` · `none`

pointer

Precision of the primary input. `coarse` = finger/stylus (needs bigger tap targets). `fine` = mouse/trackpad.

`fine` · `coarse` · `none`

prefers-contrast

The user prefers higher (or lower) contrast than normal. Useful for accessibility overlays.

`more` · `less` · `no-preference`

forced-colors

Windows High Contrast mode is active. The OS forces a limited colour palette on all elements.

`active` · `none`

🎨 CSS — preference queries in practice

```
/* ── Dark mode ── */
:root {
    --bg: #fff;
    --text: #111;
}
@media (prefers-color-scheme: dark) {
    :root { --bg: #0d0f18; --text: #e2e4ec; }
}
/* All elements that use var(--bg) and var(--text) switch automatically */

/* ── Reduced motion — IMPORTANT for accessibility ── */
.card {
    transition: transform 0.3s ease;
}
.card:hover { transform: translateY(-4px); }

@media (prefers-reduced-motion: reduce) {
    /* Replace motion with a simpler fade, or remove it */
    .card { transition: opacity 0.15s; }
    .card:hover { transform: none; opacity: 0.85; }
}

/* ── Touch targets for coarse pointer ── */
.btn { padding: 6px 14px; }  /* default: fine pointer */
@media (pointer: coarse) {
    .btn { padding: 12px 20px; min-height: 44px; }  /* 44px = Apple's tap
```

```
}

/* ── Hover: hide hover-only UI from touch devices ── */
.tooltip { display: none; }
@media (hover: hover) {
    .tooltip-trigger:hover .tooltip { display: block; }
}
```

⚠️ **prefers-reduced-motion is an accessibility requirement.** Some users with vestibular disorders experience nausea, vertigo, or migraines from animated interfaces. `prefers-reduced-motion: reduce` is set by real users who need it. Implementing it is a matter of basic accessibility — not just a nice-to-have enhancement.

💡 **Testing preference queries in DevTools.** *In Chrome: DevTools → more options (⋮) → Rendering tab → scroll to "Emulate CSS media feature". You can force `prefers-color-scheme: dark`, `prefers-reduced-motion: reduce`, and more without changing your OS settings. Firefox has the same in DevTools → Inspector → Responsive Design Mode.*

9 — Responsive Layout Patterns

Pattern 1 — Stack to side-by-side

🎨 **CSS — the most common responsive transformation**

```
/* Mobile: stacked */
.page {
    display: grid;
    grid-template-columns: 1fr;
    gap: 20px;
}

@media (min-width: 720px) {
    .page {
        grid-template-columns: 1fr 300px;
        grid-template-areas: "main sidebar";
```

```
    }
}
```

Pattern 2 — Responsive navigation

🎨 CSS — nav that collapses on mobile

```
/* Mobile: stacked links, hidden by default */
.nav__links {
    display: none;
    flex-direction: column;
}
.nav__links.open { display: flex; }
.nav__toggle { display: block; }  /* hamburger button */

@media (min-width: 768px) {
    .nav__links {
        display: flex !important;  /* always visible on desktop */
        flex-direction: row;
    }
    .nav__toggle { display: none; }  /* no hamburger on desktop */
}
```

The `!important` on the desktop nav is one of the legitimate uses from Chapter 6: it prevents JS toggle logic (which sets `display: flex` via class) from accidentally hiding the nav when resizing from mobile to desktop with the menu open.

Pattern 3 — Fluid typography

🎨 CSS — font size that grows with viewport

```
/* Approach 1: stepped — breakpoints change font size */
html { font-size: 16px; }
@media (min-width: 768px)  { html { font-size: 17px; } }
@media (min-width: 1280px) { html { font-size: 18px; } }

/* Approach 2: fluid — clamp() (covered in Chapter 8) */
html {
    font-size: clamp(16px, 1.2vw + 12px, 20px);
    /* minimum 16px, grows with viewport, maximum 20px */
```

```
}

/* Approach 3: Tailwind-style — use modular scale per breakpoint */
h1 { font-size: 1.75rem; }
@media (min-width: 768px) {
    h1 { font-size: 2.5rem; }
}
```

Pattern 4 — Responsive images

🎨 **CSS — preventing images from breaking layouts**

```
/* The universal image reset */
img {
    max-width: 100%;    /* never wider than its container */
    height: auto;       /* maintain aspect ratio */
    display: block;     /* eliminate inline-block gap below image */
}

/* Hide/show different art direction at breakpoints */
.hero-img-mobile  { display: block;  }
.hero-img-desktop { display: none;   }

@media (min-width: 768px) {
    .hero-img-mobile  { display: none;  }
    .hero-img-desktop { display: block; }
}

/* Better: use the HTML <picture> element for art direction
   CSS cannot change which image src the browser downloads.
   <picture> tells the browser to pick the right source. */
```

Always add `max-width: 100%; height: auto` to all images as a global reset — it is one of the most important two lines in any stylesheet. Without it, a 1200px-wide image renders at 1200px regardless of its container, causing horizontal overflow.

10 — Quick Reference

Key media features

FEATURE	TESTS	COMMON VALUES
`width` / `min-width` / `max-width`	Viewport width	px, rem, em
`height` / `min-height` / `max-height`	Viewport height	px, rem
`orientation`	Portrait vs landscape	`portrait` · `landscape`
`prefers-color-scheme`	OS colour mode	`dark` · `light`
`prefers-reduced-motion`	Reduce motion setting	`reduce` · `no-preference`
`hover`	Primary input can hover	`hover` · `none`
`pointer`	Pointer precision	`fine` · `coarse` · `none`
`prefers-contrast`	Contrast preference	`more` · `less` · `no-preference`

Logical operators

OPERATOR	MEANING	EXAMPLE
`and`	All conditions true	`(min-width: 600px) and (orientation: landscape)`
`,` (comma)	Any condition true (OR)	`(max-width: 480px), (orientation: portrait)`
`not`	Negates the query	`not (hover: hover)`

Old vs new range syntax

OLD (LEVEL 3)	NEW (LEVEL 4)
`(min-width: 600px)`	`(width >= 600px)`

OLD (LEVEL 3)	NEW (LEVEL 4)
`(max-width: 959px)`	`(width < 960px)`
`(min-width: 600px) and (max-width: 959px)`	`(600px <= width < 960px)`

🖍️ Exercises

Write the CSS from scratch before checking the solution. Test in a browser by resizing the window — DevTools Responsive Mode (F12 → Toggle device toolbar) makes this easy.

EXERCISE 1

Build a mobile-first card grid. Base (mobile): single column, 16px gap, cards with 20px padding. At 600px: two columns. At 960px: three columns, 24px gap. At 1280px: four columns. Use `min-width` queries throughout. Include the viewport meta tag.

Hint: Start with `grid-template-columns: 1fr` as the base, then progressively add columns inside each `min-width` breakpoint. Only the column count and gap need to change — padding stays the same.

▶ Show Sample Solution

EXERCISE 2

Add a dark mode to a simple page. Define CSS custom properties for `--bg`, `--surface`, `--text`, and `--accent` in light mode on `:root`. Then write a `prefers-color-scheme: dark` query that redefines all four variables. Apply the variables to `body`, a `.card` component, and all `a` link elements.

Hint: Only the variable definitions need to go inside the media query — any rule that uses `var()` updates automatically. You don't need to repeat the full rules inside the query.

▶ Show Sample Solution

EXERCISE 3

Write a `.hero` section with a large animated background gradient. Add a `prefers-reduced-motion` query that disables the animation entirely and substitutes a static background for users who have enabled Reduce Motion. Then add a `hover: none` query that removes a `:hover` scale transform from a `.hero-btn` inside the hero (touch devices cannot hover).

Hint: Use `animation: none` inside the reduced-motion query to cancel any named keyframe animation. Use `pointer-events: auto` but remove the transform in the hover-none query.

▶ Show Sample Solution

EXERCISE 4

Rewrite these three desktop-first media queries as mobile-first equivalents (using `min-width`) that produce the same visual result. Then rewrite the range query using the Level 4 range syntax.

```
/* Desktop-first */
.grid { grid-template-columns: repeat(4, 1fr); }
@media (max-width: 1023px) { .grid { grid-template-columns: repeat(3, 1fr); } }
@media (max-width: 767px) { .grid { grid-template-columns: repeat(2, 1fr); } }
@media (max-width: 479px) { .grid { grid-template-columns: 1fr; } }
```

Hint: In mobile-first, the base style is the mobile (1 column) version. Each `min-width` breakpoint adds a column. For the range syntax, target the tablet range using `600px <= width < 960px`.

▶ Show Sample Solution

CHAPTER 8

CSS Functions — calc(), clamp(), min(), max()

🔢 Chapter 8 — CSS Functions: calc(), clamp(), min(), max()

CSS was originally a list of property-value pairs with no arithmetic — if you needed a width that was "the full container minus a 280px sidebar", you had to calculate it yourself and hardcode the result. CSS functions changed that. `calc()` lets CSS do the math at render time. `min()`, `max()`, and `clamp()` add comparisons. Together they unlock truly adaptive values that respond to the viewport, to their container, and to other CSS values — without a single media query needed.

1 — calc()

`calc()` evaluates a mathematical expression and produces a CSS value. Its superpower is **mixing units that CSS cannot otherwise combine** — like percentages and pixels.

🎨 CSS — calc() syntax and operators

```css
/* Basic arithmetic — + - * / */
.box {
    width:   calc(100% - 2rem);       /* subtract fixed gutter from fluid
    height:  calc(100vh - 64px);      /* full viewport minus a fixed nav h
    padding: calc(1rem + 4px);        /* rarely useful, but valid */
    font-size: calc(1rem * 1.25);    /* multiply by unitless number */
    width:   calc(100% / 3);          /* divide by unitless number */
}

/* Custom properties inside calc() */
:root { --sidebar-width: 260px; }
.main { width: calc(100% - var(--sidebar-width)); }
/* Change --sidebar-width and .main updates automatically */
```

Whitespace rule: *the `+` and `-` operators MUST be surrounded by spaces: `calc(100% - 2rem)` ✓ — `calc(100%-2rem)` ✗ (parsed as "100%" minus "2rem" fails, because the parser sees it as a single token). Multiplication and division do not require spaces but spaces are still good practice.*

The four operators and their rules

+	−	*	/
Addition Needs spaces around it	Subtraction Needs spaces around it	Multiplication One operand must be unitless	Division Right operand must be unitless

> **The big deal: mixing units.** Pure CSS had no way to express "100% of the container, minus 16px of padding." You either had to use `box-sizing: border-box` (which handles padding only), or JavaScript, or approximate values. `calc()` lets you freely combine percentages, px, rem, em, vw, vh, and any other unit in a single expression — calculated fresh every time the layout changes.

Common calc() patterns

CSS — calc() in the real world

```
/* Full-bleed content area with capped max-width */
.content {
    width:   calc(min(100%, 1200px) - 2 * 1.5rem);
    margin:  0 auto;
}

/* Offset a sticky element below a fixed nav */
.sticky-sidebar {
    position:  sticky;
    top:        calc(var(--nav-height) + 1rem);
    max-height: calc(100vh - var(--nav-height) - 2rem);
}

/* Grid column that leaves room for gap */
.grid-item {
    width: calc(33.333% - (2 * 16px / 3));
    /* 3 items per row, 16px gap between them */
}

/* Negative calc() for overlapping pulls */
.pull-left {
    margin-left: calc(-1 * var(--container-padding));
```

```
    width:       calc(100% + var(--container-padding));
}
```

2 — min() and max()

`min()` and `max()` take two or more comma-separated values and return the **smallest** or **largest** respectively. This gives you responsive constraints without media queries.

🎨 CSS — min() and max()

```
/* min() — "use the smaller of these values" */
.container {
    width: min(500px, 100%);
    /* On wide screens: 500px (100% would be bigger, so min picks 500px) *
    /* On narrow screens: 100% (100% is smaller than 500px, min picks it)
    /* Equivalent to:  width: 100%;  max-width: 500px; */
}

/* max() — "use the larger of these values" */
.sidebar {
    width: max(200px, 25%);
    /* On wide screens: 25% (25% is bigger than 200px) */
    /* On narrow screens: 200px floor (25% shrinks below 200px) */
    /* Equivalent to:  width: 25%;  min-width: 200px; */
}

/* Font size that never drops below 16px */
p { font-size: max(16px, 2vw); }
/* At 600px viewport: 2vw = 12px → max picks 16px (floor) */
/* At 1000px viewport: 2vw = 20px → max picks 20px (grows) */

/* More than two arguments */
.box { width: min(300px, 50%, 80vw); }  /* smallest of three */
```

Equivalences — min/max as shorthand for min/max-width

CLASSIC (TWO PROPERTIES)

```
.box {
    width:     100%;
    max-width: 800px;
}
```

MODERN (ONE EXPRESSION)

```
.box {
    width: min(800px, 100%);
}
```

3 — clamp()

`clamp(minimum, preferred, maximum)` is the most powerful of the three. It locks a value between a floor and a ceiling, with a preferred middle value that can be fluid. Think of it as "min and max combined into one expression."

🎨 CSS — clamp() anatomy

```
/*  clamp( MINIMUM, PREFERRED, MAXIMUM )                    */
/*         ↑ floor   ↑ ideal       ↑ ceiling                */
/*                                                           */
/*  The browser evaluates preferred.                         */
/*  If preferred < minimum → uses minimum.                   */
/*  If preferred > maximum → uses maximum.                   */
/*  Otherwise → uses preferred.                              */

h1 { font-size: clamp(1.5rem, 4vw, 3rem); }
/* At 320px viewport:  4vw = 12.8px  → 1.5rem (24px) floor applies */
/* At 768px viewport:  4vw = 30.7px  → fluid (30.7px) in between   */
/* At 1200px viewport: 4vw = 48px    → 3rem (48px) ceiling applies */

/* Mathematical identity — these are equivalent: */
clamp(a, b, c)    /* same as → */   max(a, min(b, c))
```

Use `rem` for the minimum and maximum values (not `px`). This respects the user's browser font-size preference — if they've set a larger default, your minimum and maximum scale accordingly. A user who needs larger text for accessibility will get it.

> 💡 **Colour coding in the demo above:** *purple = clamped to minimum (floor active), blue = fluid (preferred value in range), green = clamped to maximum (ceiling active). A good clamp*

value spends most viewport widths in the fluid blue zone.

4 — Fluid Typography with clamp()

The most celebrated use of `clamp()` is fluid typography: font sizes that smoothly scale with the viewport instead of jumping at breakpoints. A single declaration replaces a font-size rule plus multiple `@media` overrides.

🎨 **CSS — a complete fluid type scale**

```
/* Fluid type scale using clamp()
   Formula: clamp(min, preferred-vw, max)
   Preferred ≈ the vw fraction where the size feels right at "normal" view

:root {
    /* base text — 16px → 20px */
    --text-base:  clamp(1rem,   1.5vw + 0.5rem, 1.25rem);

    /* small label — 12px → 14px */
    --text-sm:    clamp(0.75rem, 1vw + 0.25rem, 0.875rem);

    /* h4 — 18px → 22px */
    --text-h4:    clamp(1.125rem, 2vw + 0.25rem, 1.375rem);

    /* h3 — 22px → 28px */
    --text-h3:    clamp(1.375rem, 2.5vw + 0.5rem, 1.75rem);

    /* h2 — 28px → 40px */
    --text-h2:    clamp(1.75rem,  3.5vw + 0.5rem, 2.5rem);

    /* h1 — 36px → 60px */
    --text-h1:    clamp(2.25rem,  5vw + 0.5rem,  3.75rem);

    /* Hero title — 48px → 96px */
    --text-hero:  clamp(3rem,     7vw + 0.5rem,  6rem);
}

body    { font-size: var(--text-base); }
```

```
h1      { font-size: var(--text-h1);  }
/* etc. — no media queries needed for typography */
```

The "preferred" value format `Xvw + Yrem` is a linear interpolation: the `vw` part provides the slope (grows with viewport), the `rem` part shifts the baseline up or down. Adjust the vw coefficient for steeper or gentler growth.

> ⚠ **Don't use vw alone for font sizes.** `font-size: 4vw` (no clamp) means on a 300px screen the text is 12px (illegible), and on a 2400px screen it's 96px (absurd). `clamp()` is not optional here — the min floor guarantees readability, the max ceiling prevents absurdly large text.

5 — Fluid Spacing

The same clamp technique applies to `padding`, `margin`, and `gap`. Spacing tokens that grow with the viewport create layouts that feel proportional at every size — more breathable on desktop, tighter and more efficient on mobile.

🎨 **CSS — fluid spacing tokens**

```
:root {
    /* Space scale — each step roughly doubles */
    --space-xs:  clamp(0.25rem, 0.5vw,  0.5rem);
    --space-sm:  clamp(0.5rem,  1vw,    0.875rem);
    --space-md:  clamp(1rem,    2vw,    1.5rem);
    --space-lg:  clamp(1.5rem,  3.5vw,  3rem);
    --space-xl:  clamp(2.5rem,  6vw,    5rem);
    --space-2xl: clamp(4rem,    10vw,   8rem);
}

/* Use tokens throughout — spacing adapts automatically */
.section    { padding: var(--space-xl) var(--space-lg); }
.card       { padding: var(--space-md); }
.card-grid  { gap: var(--space-md); }
h2          { margin-bottom: var(--space-sm); }
```

6 — Nesting Functions

`calc()`, `min()`, `max()`, and `clamp()` can all be nested inside each other. This enables compound expressions that would otherwise require JavaScript or multiple CSS rules.

🎨 **CSS — nested function expressions**

```css
/* Container with fluid padding AND a maximum width */
.container {
    width:   min(calc(100% - 2 * var(--space-lg)), 1200px);
    margin:  0 auto;
    /* "width minus gutters, but never wider than 1200px" */
}

/* Sidebar with a fluid preferred width, floored and ceilinged */
.sidebar {
    width: clamp(180px, calc(25% + 2rem), 320px);
    /* preferred: 25% of parent + 2rem, clamped 180px→320px */
}

/* Safe area aware padding (notch/rounded-corner phones) */
.page {
    padding-bottom: max(env(safe-area-inset-bottom), 1rem);
    /* at least 1rem, but respect the phone's safe area */
}

/* Responsive border-radius that scales but stays reasonable */
.card {
    border-radius: clamp(8px, calc(0.5vw + 4px), 20px);
}

/* Line-height that scales with font-size (relative calc) */
p {
    font-size:   clamp(1rem, 1.5vw, 1.25rem);
    line-height: calc(1.4 + 0.2 * (1.25rem - 1rem) / 0.25rem);
    /* line-height tightens slightly as font size grows */
}
```

calc() inside clamp preferred values is especially common. The pattern `clamp(min, Xvw + Yrem, max)` is itself a nested expression: `Xvw + Yrem` is a calc()-style expression written inline (CSS allows the arithmetic inside min/max/clamp without an explicit `calc()` wrapper). This is often written without `calc()`: `clamp(1rem, 4vw + 0.5rem, 3rem)` — the browser recognises the expression automatically.

7 — Level 4 Math Functions CSS LEVEL 4

CSS Values and Units Level 4 defines additional math functions. Browser support is growing (most landed in 2023–2024). You may encounter these in modern codebases.

FUNCTION	RETURNS	EXAMPLE
`round(strategy, value, step)`	value rounded to nearest step	`round(nearest, 17px, 4px)` → `16px`
`mod(a, b)`	Remainder (same sign as divisor)	`mod(10px, 3px)` → `1px`
`rem(a, b)`	Remainder (same sign as dividend)	`rem(-10px, 3px)` → `-1px`
`abs(a)`	Absolute value	`abs(-2rem)` → `2rem`
`sign(a)`	`-1`, `0`, or `1`	`sign(-5px)` → `-1`
`sin(a)` / `cos(a)`	Trigonometric (unitless 0–1)	Used in animations and transforms
`pow(a, b)`	a raised to power b	`pow(2, 3)` → `8`
`sqrt(a)`	Square root	`sqrt(9)` → `3`

🎨 **CSS — Level 4 math in practice**

```
/* round() — snap values to a grid (e.g. 4px design grid) */
.box {
    width: round(nearest, 33.333%, 4px);
```

```
    /* Snaps the 1/3 width to the nearest 4px increment */
}

/* sin/cos — position elements on a circle (no JS needed) */
.clock-hand {
    /* Rotate items around a centre using trigonometric placement */
    transform: translate(
        calc(var(--radius) * cos(var(--angle))),
        calc(var(--radius) * sin(var(--angle)))
    );
}

/* abs() — ensure a positive distance regardless of direction */
.tooltip {
    margin-top: abs(var(--offset));
}
```

*Check **caniuse.com** before using Level 4 math functions in production. `round()`, `mod()`, `abs()`, and trig functions are supported in Chrome 111+, Firefox 118+, Safari 15.4+ — but verify for your specific target browsers.*

8 — Quick Reference

FUNCTION	TAKES	RETURNS	PRIMARY USE
`calc(expr)`	Math expression with units	Computed value	Mix units, subtract fixed amounts from fluid widths
`min(a, b, …)`	2+ comma-separated values	Smallest value	Maximum constraint — "no wider than X"
`max(a, b, …)`	2+ comma-separated values	Largest value	Minimum constraint — "never smaller than X"
`clamp(min, pref, max)`	3 values: floor, ideal, ceiling	Clamped preferred	Fluid typography, fluid spacing, responsive sizing

Whitespace rules in calc()

OPERATOR	SPACES REQUIRED?	EXAMPLE
`+` and `-`	**Yes — mandatory**	`calc(100% - 2rem)` ✓ `calc(100%-2rem)` ✗
`*` and `/`	No — optional	`calc(2 * 1rem)` = `calc(2*1rem)`

clamp() — fluid typography quick formula

🎨 CSS — choosing clamp parameters

```
/* For font sizes: min in rem, vw for preferred, max in rem     */
/* Preferred rule of thumb: choose vw so the preferred falls    */
/* nicely between min and max at a typical 768px–1024px screen. */
/*                                                               */
/* At 800px: 4vw = 32px → if target heading is 30–34px, good.   */

/* Verify your values: */
/*   At min: vw_at_min_viewport ≤ minimum                       */
/*   At max: vw_at_max_viewport ≥ maximum                       */

clamp( 1.5rem,  4vw,   3rem  )   /* h2: 24px → 48px */
clamp( 1rem,    2vw,   1.25rem)  /* body: 16px → 20px */
clamp( 1rem,    2vw + 0.5rem, 1.5rem) /* compound preferred */
```

🖍️ Exercises

Write the CSS from scratch before checking the solution. Focus on picking the right function for each constraint — not all problems need `clamp()`.

EXERCISE 1

Create a centred content container that: (a) is always 90% of its parent width, (b) is never wider than 1100px, (c) never narrower than 280px, all in a single CSS declaration using the appropriate function. Then write a sticky header that sits 0px from the top of the viewport, and a content area whose top padding ensures it is never hidden behind the header — using `calc()` with a `--header-height` custom property set to `64px`.

Hint: For the container, `clamp()` handles both the floor and ceiling. For the content padding, `calc(var(--header-height) + 1rem)` adds comfort space below the header.

▶ Show Sample Solution

EXERCISE 2

Write a full set of fluid typography custom properties for a project. Define: `--text-sm` (12px → 14px), `--text-base` (16px → 20px), `--text-lg` (20px → 28px), `--text-xl` (28px → 40px), `--text-2xl` (40px → 64px). Use `rem` units for min and max. Apply each to the corresponding element type (`small`, `body`, `h3`, `h2`, `h1`).

Hint: Convert px to rem by dividing by 16 (default root font size). A suitable preferred vw for each step: sm≈0.8vw, base≈1.5vw, lg≈2.5vw, xl≈4vw, 2xl≈6.5vw. You may add a `rem` offset to the preferred to shift the curve.

▶ Show Sample Solution

EXERCISE 3

Write three rules that replace common two-property patterns with a single expression: (a) replace `width: 100%; max-width: 720px` with `min()` — (b) replace `font-size: 2vw; min-width: 18px (font-size)` effectively → use `max()` so the font never goes below 18px — (c) replace a three-property responsive sidebar (`width: 25%; min-width: 180px; max-width: 300px`) with a single `clamp()`.

Hint: min() = "never wider than", max() = "never smaller than", clamp() = "floor, preferred, ceiling" in one expression.

▶ Show Sample Solution

EXERCISE 4

Build a hero section using only CSS functions (no fixed pixel values except for the header height). The hero should: fill the viewport height minus a 70px header (`calc()`), have fluid padding top/bottom that scales from 2rem to 5rem (`clamp()`), contain a heading with fluid type from 2rem to 5rem (`clamp()`), and a subtitle from 1rem to 1.5rem (`clamp()`). Define `--nav-height: 70px` as a custom property and use it in the height calc.

Hint: `height: calc(100vh - var(--nav-height))` for the hero height. Choose vw values for the clamp preferred that feel good at a typical 900px viewport.

▶ Show Sample Solution

CHAPTER 9

Transforms — Translate, Rotate, Scale, Skew

🔄 Chapter 9 — Transforms

CSS transforms move, rotate, resize, and distort elements visually — without affecting layout flow. A translated element still occupies its original space in the document; nothing around it shifts. This makes transforms the right tool for hover effects, entrance animations, card flips, parallax, and any visual motion where you don't want to disturb surrounding content.

1 — What Transforms Do

The `transform` property applies one or more **transform functions** to an element. These functions change how the element is painted to the screen — but the element still occupies its original position in the layout flow. Think of it as moving a sticker on top of an already-laid-out page.

> **Transforms vs. layout properties.** Setting `margin-left: 50px` shifts an element and pushes its siblings. Setting `transform: translateX(50px)` shifts the element visually but leaves a "ghost" in its original position — siblings and parent containers are completely unaffected. This is why transforms are used for animation: changing `transform` never triggers layout recalculation.

🎨 CSS — the transform property

```
/* Single function */
.card { transform: rotate(45deg); }

/* Multiple functions (applied right-to-left — see Section 6) */
.card { transform: translateY(-10px) scale(1.05) rotate(2deg); }

/* Remove a transform */
.card { transform: none; }

/* The transform creates a new stacking context — the element
```

```
    becomes a containing block for position:fixed children,
    and is composited independently by the GPU. */
```

2 — translate()

`translate(x, y)` moves an element along the X and Y axes. Both arguments accept any CSS length or percentage — percentages are relative to the element's **own size**, not the parent's, which enables the classic centering trick.

🎨 CSS — translate variants

```
/* Two-axis shorthand */
.box { transform: translate(100px, -50px); }  /* right 100px, up 50px */

/* Single-axis helpers */
.box { transform: translateX(2rem); }
.box { transform: translateY(-1em); }
.box { transform: translateZ(50px); }  /* Z-axis — needs perspective */

/* Percentage = % of the element's own size */
.tooltip { transform: translateX(-50%); }  /* shift left by half its own w

/* The centering trick — no need to know the element's size */
.centered {
    position:  absolute;
    top:       50%;
    left:      50%;
    transform: translate(-50%, -50%);
    /* top/left move the top-left corner to centre, */
    /* translate(-50%,-50%) then shifts back by half its own size */
}
```

3 — rotate()

`rotate(angle)` spins an element clockwise for positive angles and counter-clockwise for negative ones. It accepts `deg`, `rad`, `grad`, and `turn` units.

CSS — rotate variants

```css
/* Angle units — all equivalent to 90° clockwise */
rotate(90deg)      /* degrees          */
rotate(1.5708rad)  /* radians          */
rotate(0.25turn)   /* turns (0–1)      */
rotate(100grad)    /* gradians (0–400)*/

/* 3D rotation variants (require perspective for visual depth) */
rotateX(45deg)    /* tilt towards/away from viewer — horizontal axis */
rotateY(45deg)    /* spin left/right — vertical axis */
rotateZ(45deg)    /* same as 2D rotate() */

/* Hover rotation effect */
.icon       { transition: transform 0.3s ease; }
.icon:hover { transform: rotate(180deg); }

/* Collapsed/expanded indicator (common accordion arrow) */
.arrow           { transform: rotate(0deg);   transition: transform 0.25s;
.open .arrow     { transform: rotate(90deg);  }
```

4 — scale() and skew()

CSS — scale() and skew()

```css
/* ── scale() ── */
/* Values: 1 = no change, 2 = double size, 0.5 = half size */
scale(1.5)          /* uniform — same as scale(1.5, 1.5) */
scale(2, 0.5)       /* double width, half height */
scaleX(-1)          /* flip horizontally (mirror) */
scaleY(-1)          /* flip vertically (upside down) */

/* Pulsing button effect */
```

```css
.btn:active { transform: scale(0.95); }

/* ── skew() ── */
/* Shears/distorts — positive X-angle tilts right, positive Y tilts down */
skew(20deg)          /* X-axis skew only */
skew(15deg, 5deg)   /* X and Y axis */
skewX(-30deg)        /* single axis */

/* Diagonal banner / parallelogram button */
.banner {
    transform:   skewX(-15deg);
}
.banner-inner {
    transform:   skewX(15deg);  /* un-skew the text inside */
}
```

5 — transform-origin

`transform-origin` sets the pivot point around which transforms are applied. The default is `50% 50%` — the centre of the element. Change it to rotate around a corner, an edge, or even a point outside the element entirely.

🎨 **CSS — transform-origin values**

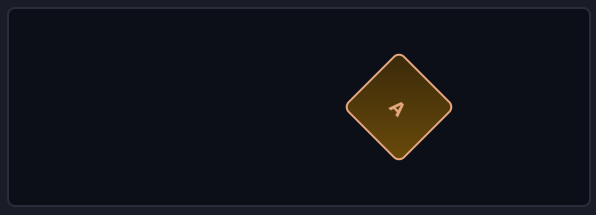
```css
/* Keywords */
transform-origin: center;          /* default — 50% 50% */
transform-origin: top left;       /* 0% 0% */
transform-origin: bottom right;  /* 100% 100% */

/* Lengths or percentages */
transform-origin: 0 0;             /* top-left corner */
transform-origin: 100% 0;         /* top-right corner */
transform-origin: -20px 50%;     /* 20px left of the element */
transform-origin: 150% 50%;     /* point to the right — orbiting effect */

/* Three values include the Z-axis (for 3D transforms) */
transform-origin: 50% 50% 100px;
```

```css
/* Hinge effect — door swinging from its left edge */
.door {
    transform-origin: left center;
    transform:        rotateY(0deg);
    transition:       transform 0.5s ease;
}
.door.open { transform: rotateY(-80deg); }
```

6 — Combining Transforms — Order Matters

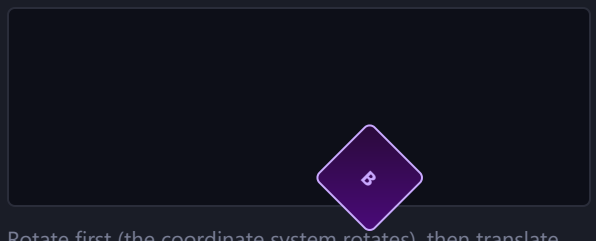
When multiple transform functions appear in one `transform` declaration, they are applied **right to left**. This mirrors matrix multiplication, and the order dramatically changes the result — because each function operates in the *coordinate system established by all previous transforms*.

translateX(50px) → rotate(45°)



Translate first (moves along original X axis), then rotate. The box ends up to the right of centre, then spun.

rotate(45°) → translateX(50px)



Rotate first (the coordinate system rotates), then translate. The "X axis" is now diagonal, so translateX moves diagonally.

🎨 **CSS — transform order and multiple functions**

```css
/* Box A — translate right, then rotate */
.a { transform: translateX(50px) rotate(45deg); }
/* Applied order: rotate(45°) THEN translateX(50px)
   → translation is along the ROTATED x-axis (diagonal direction) */

/* Box B — rotate, then translate */
.b { transform: rotate(45deg) translateX(50px); }
/* Applied order: translateX(50px) THEN rotate(45°)
   → translation happens along original x-axis (straight right) */
```

```
/* Tip: write transforms in the ORDER you'd narrate them
   ("move it right, then tilt it") but remember the CSS
   declaration reads right-to-left in application. */

/* Common combined hover effect */
.card:hover {
    transform: translateY(-6px) scale(1.03) rotate(1deg);
}
```

7 — 3D Transforms

3D transforms add depth to layouts. To see 3D perspective, you need three things: a **perspective** value on the parent, **transform-style: preserve-3d** on the container element so children exist in a shared 3D space, and 3D transform functions on the children.

🎨 CSS — the 3D setup

```
/* Perspective — how far the viewer is from the screen.
   Smaller = more dramatic. Applied to the parent. */
.scene {
    perspective:        600px;   /* 300-800px = dramatic; 1200px+ = subtle
    perspective-origin: 50% 50%; /* viewpoint position (default) */
}

/* preserve-3d — children share the scene's 3D space */
.container {
    transform-style: preserve-3d;
}

/* backface-visibility — hide an element when it faces away from viewer */
.card-face {
    backface-visibility: hidden;
}

/* 3D transform functions */
rotateX(45deg)          /* tilt towards/away */
rotateY(180deg)         /* spin (card flip) */
translateZ(100px)       /* move towards viewer */
```

```
translate3d(x, y, z)    /* three-axis move in one function */
scale3d(sx, sy, sz)     /* three-axis scale */

/* perspective() as a function (applied inline on the element) */
.item { transform: perspective(600px) rotateY(45deg); }
/* Use this when you want per-element perspective (no shared scene) */
```

The card flip pattern

🎨 **CSS — full card flip markup**

```
/* HTML structure:
   <div class="scene">
     <div class="card">
       <div class="card-face card-front">Front</div>
       <div class="card-face card-back">Back</div>
     </div>
   </div> */

.scene {
    perspective: 600px;
}

.card {
    position:        relative;
    transform-style: preserve-3d;
    transition:      transform 0.6s ease;
}
.card.flipped { transform: rotateY(180deg); }

.card-face {
    position:           absolute;
    inset:              0;
    backface-visibility: hidden;  /* hide when facing away */
}

.card-back {
```

```
    transform: rotateY(180deg);  /* pre-rotated so it starts facing away *
}
```

8 — Transforms and Transitions

Transforms become motion when combined with `transition`. The browser smoothly interpolates the transform values between states. Chapter 10 (Keyframe Animations) covers complex multi-step motion — here we focus on the state-to-state pattern.

🎨 **CSS — transform + transition patterns**

```
/* Hover lift + shadow */
.card {
    transition: transform 0.25s ease, box-shadow 0.25s ease;
}
.card:hover {
    transform:  translateY(-6px) scale(1.02);
    box-shadow: 0 12px 32px rgba(0,0,0,0.25);
}

/* Icon spin on hover */
.btn .icon       { transition: transform 0.4s ease; }
.btn:hover .icon { transform:  rotate(360deg);    }

/* Press / click feedback */
.btn         { transition: transform 0.1s ease; }
.btn:active  { transform:  scale(0.95);          }

/* Slide-in navigation panel */
.nav       { transform: translateX(-100%); transition: transform 0.35s eas
.nav.open  { transform: translateX(0);     }

/* Staggered entrance (JS adds .visible to each in sequence) */
.item         { transform: translateY(20px); opacity: 0; transition: trans
.item.visible { transform: translateY(0);    opacity: 1;  }
```

9 — Performance: Why Transforms Are Fast

> **Transforms and opacity are the two CSS properties that the browser can animate entirely on the GPU compositor thread.** Every other property that changes geometry (`width`, `height`, `top`, `left`, `padding`, `margin`…) forces a *layout* recalculation — the browser must re-measure every affected element before painting. Transforms skip layout and paint entirely; the compositor just moves an already-painted texture, which is essentially free.

⚠ SLOW — TRIGGERS LAYOUT

```
.btn:hover {
    width:  110%;
    top:    -4px;
    left:   -4px;
}
```

✓ FAST — COMPOSITOR ONLY

```
.btn:hover {
    transform: scale(1.05);
    transform: translateY(-4px);
}
```

🎨 CSS — will-change hint

```
/* Hint the browser to promote the element to its own compositor
   layer BEFORE the animation starts — eliminates the initial
   "jank" frame when the layer is first promoted. */
.animated-card {
    will-change: transform, opacity;
}

/* ⚠ Overuse is harmful: every will-change element uses GPU memory.
   Add it only to elements you know will animate. Remove it after
   animation ends via JavaScript if the animation is one-shot. */

/* The three-property GPU rule — animate ONLY these for max perf: */
/*   transform    (position, size, rotation)                      */
/*   opacity      (fade in/out)                                   */
/*   filter       (blur, brightness — varies by browser/hardware) */
```

10 — Individual Transform Properties MODERN CSS

Modern CSS provides `translate`, `rotate`, and `scale` as standalone properties. They apply on top of `transform` and let you animate each axis independently in `@keyframes` or transitions — something previously impossible because changing `transform` in two different rules would overwrite each other.

translate	rotate	scale
translate: 20px 10px	rotate: 45deg	scale: 1.2

🎨 CSS — individual transform properties

```
/* Old approach — hover and focus both set transform,
   the second declaration overwrites the first completely */
.card:hover  { transform: translateY(-8px);          }
.card:focus  { transform: scale(1.05); }  /* wipes out translateY! */

/* New approach — individual properties compose independently */
.card        { transition: translate 0.25s, scale 0.25s; }
.card:hover  { translate: 0 -8px;  }  /* doesn't affect scale */
.card:focus  { scale:     1.05;    }  /* composes with translate */
/* Hover + focus together: card lifts AND scales — no conflict */

/* Syntax differences from functions */
/* translate: x y      (space-separated, not comma)   */
/* rotate:    angle    (single value; or x y z angle) */
/* scale:     x       (or x y)                        */
translate: 50px 20px;   /* = translateX(50px) translateY(20px) */
rotate:    45deg;
scale:     1.5;          /* uniform */
scale:     1.5 1;       /* X and Y */

/* Application order (always): translate → rotate → scale
   Regardless of declaration order in CSS */

/* Browser support: Chrome 104+, Firefox 72+, Safari 14.1+ */
```

11 — Quick Reference

FUNCTION	WHAT IT DOES	UNITS / NOTES
`translate(x, y)`	Move element	Length or %; % = % of self
`translateX(x)` / `translateY(y)`	Single-axis move	—
`translateZ(z)` / `translate3d(x,y,z)`	Move along Z-axis	Needs `perspective`
`rotate(angle)`	Spin clockwise (positive)	deg, rad, turn, grad
`rotateX(a)` / `rotateY(a)` / `rotateZ(a)`	3D rotation	Needs `perspective`
`scale(n)`	Resize uniformly	Unitless ratio; 1 = no change
`scale(x, y)`	Resize each axis	-1 = flip
`scaleX(n)` / `scaleY(n)`	Single-axis scale	—
`skew(x, y)`	Shear/distort	deg
`skewX(x)` / `skewY(y)`	Single-axis skew	—

PROPERTY	FUNCTION	NOTES
`transform-origin`	Sets the pivot point	Default: `50% 50%`
`transform-style`	`preserve-3d` for 3D children	Applied to container, not element
`perspective`	Depth illusion — set on parent	300–1200px typical range
`backface-visibility`	`hidden` hides back of flipped element	Essential for card flips
`will-change`	Hint GPU layer promotion	Use sparingly
`translate` / `rotate` / `scale`	Individual transform props	Chrome 104+, Firefox 72+, Safari 14.1+

🖍️ Exercises

Each exercise can be done with just CSS — no JavaScript needed. Focus on which function creates the right visual effect, and which properties need to be on the parent vs. the element.

EXERCISE 1

Create a button with these three interactive states using only `transform` and `transition`: (a) on hover — lift up 4px and scale to 1.04; (b) on active (click) — press down to scale 0.97; (c) ensure the transition feels snappy on press (0.1s) but smooth on hover (0.25s). Use `cubic-bezier` or named easings of your choice.

Hint: `:hover` and `:active` are separate rules. `:active` wins over `:hover` when both match. Set `transition` on the base element so both states animate.

▶ Show Sample Solution

EXERCISE 2

A loading spinner: create a `<div class="spinner">` that is a 40×40 circle with a partial border (transparent on one side). Use a CSS animation (`@keyframes spin`) to rotate it 360 degrees continuously. Set `transform-origin` to its centre. Then write a second version (`.spinner-orbit`) where the spinning dot travels around a circle by combining `rotate()` with `translateY()` and an external `transform-origin`.

Hint: For the orbit, set `transform-origin: 50% 200%` (or similar) so the dot rotates around a centre point below itself. The `@keyframes` just goes from `rotate(0deg)` to `rotate(360deg)`.

▶ Show Sample Solution

EXERCISE 3

Build a CSS-only tooltip that slides in from the bottom. The tooltip is a `::after` pseudo-element on a `.has-tooltip` span. At rest: `opacity: 0` and `translateY(6px)`. On hover: `opacity: 1` and `translateY(0)`. Use `transform-origin: bottom center` and position the tooltip above the trigger element. Animate both opacity and transform simultaneously.

Hint: Set `position: absolute; bottom: 100%; left: 50%` on the pseudo-element, then use `translate(-50%, 0)` to centre it horizontally. The `transform` can contain both centering and the slide animation using multiple functions.

▶ Show Sample Solution

EXERCISE 4

Build a 3D "tilt card" that reacts to mouse position using JavaScript. The card should have a perspective container, and when the mouse moves over the card, apply `rotateX` and `rotateY` transforms proportional to the cursor position within the card (max ±15 degrees). On mouse leave, smoothly animate back to no rotation. Add a subtle scale up on hover (1.03) and a glare highlight effect using a `::before` pseudo-element with a radial gradient that follows the mouse.

Hint: In the `mousemove` handler, calculate `x = (e.offsetX / card.offsetWidth - 0.5) * 30` for the Y-axis rotation and the inverse for X. On `mouseleave`, set `transform: rotateX(0) rotateY(0)`. The `transition` on the card handles the smooth return.

▶ Show Sample Solution

CHAPTER 10

Keyframe Animations

🎬 Chapter 10 — Keyframe Animations

Transitions animate between two states — the element either has a property or it doesn't, and the browser interpolates between them. Keyframe animations go further: they define an entire timeline of states — at 0%, at 40%, at 100% — and the browser plays through them automatically. They can loop, reverse, pause, and run independently of any user interaction.

1 — Transitions vs. Animations

TRANSITIONS

- Triggered by a state change (hover, class, focus)
- Two endpoints only — from and to
- Play once per trigger
- Reverse automatically on exit
- Simpler to write

KEYFRAME ANIMATIONS

- Self-starting — can run on page load
- Multiple intermediate keyframes
- Loop, repeat, alternate, pause
- Full control over timing per-keyframe
- Reusable — define once, apply anywhere

2 — @keyframes Syntax

A `@keyframes` rule defines a named animation timeline. Each keyframe (percentage stop) declares the CSS state at that point in time. The browser interpolates between adjacent keyframes.

🎨 CSS — @keyframes syntax

```css
/* Simplest form — from and to (= 0% and 100%) */
@keyframes slide-in {
    from { transform: translateX(-100%); opacity: 0; }
    to   { transform: translateX(0);      opacity: 1; }
}
```

```css
/* Multiple stops with per-keyframe timing */
@keyframes bounce {
    0%   { transform: translateY(0);     animation-timing-function: ease-
    40%  { transform: translateY(-40px); animation-timing-function: ease-
    60%  { transform: translateY(-28px); }
    80%  { transform: translateY(-40px); animation-timing-function: ease-
    100% { transform: translateY(0);     }
}

/* Comma-list: two stops share the same declarations */
@keyframes pulse {
    0%, 100% { transform: scale(1);    opacity: 1;    }
    50%      { transform: scale(1.25); opacity: 0.75; }
}

/* Only the properties you declare are animated — anything
   not mentioned in the keyframes inherits its normal value. */
```

Name your animations with a verb or description of what they do: `slide-in`, `fade-out`, `spin` — not `animation1`. The name is reused on every element that plays it.

3 — Applying Animations

The `animation` shorthand packs all eight sub-properties into one line. The only order rule: the **first time value** is always `animation-duration` and the **second** (if present) is `animation-delay`.

🎨 **CSS — animation shorthand**

```css
/* Full shorthand — name dur timing delay count dir fill play-state */
.box {
    animation: bounce 0.8s ease 0s infinite normal none;
}

/* Equivalent individual properties */
.box {
    animation-name:            bounce;
```

```
    animation-duration:         0.8s;
    animation-timing-function: ease;
    animation-delay:            0s;
    animation-iteration-count: infinite;
    animation-direction:        normal;
    animation-fill-mode:        none;
    animation-play-state:       running;
}
```

4 — Timing Functions and steps()

Each keyframe-to-keyframe segment can have its own `animation-timing-function`. The global one (set on the element) applies to every segment by default. Override it inside a `@keyframes` stop to control individual segments — the bounce demo above uses this for the asymmetric up/down feel.

🎨 **CSS — timing functions in animations**

```
/* Named easings */
animation-timing-function: linear;
animation-timing-function: ease;          /* default */
animation-timing-function: ease-in;
animation-timing-function: ease-out;
animation-timing-function: ease-in-out;

/* Custom cubic bezier */
animation-timing-function: cubic-bezier(0.34, 1.56, 0.64, 1);  /* spring *
animation-timing-function: cubic-bezier(0.4, 0, 0.2, 1);     /* Material e

/* steps() — discrete jumps (sprite animation, typewriter) */
animation-timing-function: steps(4);           /* 4 equal jumps */
animation-timing-function: steps(4, end);      /* default: jump at end of
animation-timing-function: steps(4, start);    /* jump at start */

/* Typewriter effect with steps() */
@keyframes typewriter {
    from { width: 0; }
    to   { width: 20ch; }       /* ch = width of "0" character */
```

```
}
.typewriter {
    white-space:  nowrap;
    overflow:     hidden;
    border-right: 2px solid;
    animation:    typewriter 3s steps(20, end) 1 forwards,
                  blink-cursor 0.75s step-end infinite;
}
@keyframes blink-cursor {
    0%,100% { border-color: currentColor; }
    50%     { border-color: transparent;  }
}
```

`step-start` is shorthand for `steps(1, start)` and `step-end` is shorthand for `steps(1, end)` — useful for the blinking cursor pattern above.

5 — animation-direction

`normal`

Plays 0%→100% each iteration. Default.

`reverse`

Plays 100%→0% each iteration. Timing functions also reverse.

`alternate`

Odd iterations forward (0→100), even iterations backward (100→0). Creates a natural back-and-forth.

`alternate-reverse`

Like alternate but starts backward. First iteration is 100→0.

🎨 **CSS — direction examples**

```
/* Infinite ping-pong — alternate is cleaner than two @keyframes */
.loader-dot {
    animation: slide-right 0.6s ease-in-out infinite alternate;
}

/* Breathing / glow effect — scale up, scale back */
.glow {
```

```
    animation: pulse 2s ease-in-out infinite alternate;
}
```

6 — animation-fill-mode

`fill-mode` controls what styles apply to the element *outside* the animation's active period — before a delay kicks in, and after the last iteration ends.

none

Element returns to its own CSS styles both before (delay) and after. Default.

forwards

After the last keyframe, the element **keeps** those styles. Use for one-shot entrance animations.

backwards

During the delay, the element gets the **first keyframe** styles immediately — so there's no jump when the animation starts.

both

Both forwards and backwards. The most commonly useful value for entrance animations with a delay.

🎨 **CSS — fill-mode in practice**

```
/* Without fill-mode: forwards — element jumps back to opacity:1 after fad
@keyframes fade-in {
    from { opacity: 0; transform: translateY(20px); }
    to   { opacity: 1; transform: translateY(0);    }
}
/* ✗ Without forwards: element visible (opacity:1) during delay, then
       jumps to opacity:0 at start, then fades in, then jumps back. Ugly.
.bad  { animation: fade-in 0.5s ease 1s 1 none; }

/* ✓ With both: invisible during delay, fades in, stays visible after */
.good { animation: fade-in 0.5s ease 1s 1 both; }

/* The pattern: entrance animation + delay → always use fill-mode: both */
```

7 — animation-play-state

`animation-play-state: paused` freezes the animation at its current point. The most common use: pause infinite animations on hover for better user experience.

🎨 CSS — play-state patterns

```css
/* Pause on hover — user can inspect or stop a spinning element */
.spinner       { animation: spin 1s linear infinite;       }
.spinner:hover { animation-play-state: paused;             }

/* Toggle play/pause with a class via JavaScript */
.animated         { animation: pulse 1.5s infinite;        }
.animated.paused  { animation-play-state: paused;           }

/* JS: toggle the .paused class */
// btn.addEventListener('click', () => el.classList.toggle('paused'));

/* Pause everything during loading, release when ready */
.page-loading * { animation-play-state: paused; }
```

8 — Multiple Animations

Separate multiple animations with commas. Each runs independently — they don't interact unless they target the same property, in which case the *last one listed wins*.

🎨 CSS — multiple animations

```css
/* Spin and pulse simultaneously */
.loader {
    animation:
        spin  1s   linear      infinite,
        pulse 2s   ease-in-out infinite;
}

/* Slide in, then blink cursor after */
```

```css
.typewriter {
    animation:
        typewriter   3s    steps(30, end)  1        forwards,
        blink-cursor 0.75s step-end         3s infinite;
        /* cursor blinks only after typing finishes (3s delay) */
}

/* Float + gentle sway — two animations on different properties */
@keyframes float  { 0%,100% {transform:translateY(0)}   50% {transform:tra
@keyframes shadow { 0%,100% {opacity:0.6;transform:scale(1)}   50% {opacit
.floating-icon   { animation: float  3s ease-in-out infinite; }
.floating-shadow { animation: shadow 3s ease-in-out infinite; }
```

9 — Staggered Animations

Apply the same animation to multiple elements but give each one an increasing `animation-delay`. The result is a cascade where items appear one after another, creating a sense of choreography and flow.

CSS — stagger patterns

```css
/* Pattern 1: nth-child selectors (small, fixed lists) */
.item              { animation: fade-in 0.4s ease both; }
.item:nth-child(2) { animation-delay: 0.1s; }
.item:nth-child(3) { animation-delay: 0.2s; }
.item:nth-child(4) { animation-delay: 0.3s; }

/* Pattern 2: CSS custom property set by JS (scales to any length) */
.item {
    animation:       slide-up 0.45s ease both;
    animation-delay: calc(var(--i, 0) * 80ms);
}
/* JS: items.forEach((el, i) => el.style.setProperty('--i', i)); */

/* Pattern 3: Negative delay — items already mid-animation on load */
.wave-item {
    animation:       wave 1.2s ease-in-out infinite alternate;
```

```
    animation-delay: calc(var(--i, 0) * -200ms);
    /* Negative delay starts the animation already partway through — */
    /* creates a wave that looks like it's been running forever.     */
}
```

10 — Common Animation Patterns

🎨 CSS — ready-to-use @keyframes

```
/* Fade in from below */
@keyframes fade-up {
    from { opacity: 0; transform: translateY(24px); }
    to   { opacity: 1; transform: translateY(0);    }
}

/* Shake / error feedback */
@keyframes shake {
    0%,100% { transform: translateX(0);       }
    20%     { transform: translateX(-8px);  }
    40%     { transform: translateX(8px);   }
    60%     { transform: translateX(-5px);  }
    80%     { transform: translateX(5px);   }
}
.input.error { animation: shake 0.5s ease 1; }

/* Skeleton loading shimmer */
@keyframes shimmer {
    from { background-position: -200% 0; }
    to   { background-position: 200% 0;  }
}
.skeleton {
    background: linear-gradient(
        90deg,
        #1a1d27 25%,
        #2a2d3a 50%,
        #1a1d27 75%
    );
    background-size: 400% 100%;
```

```
    animation: shimmer 1.5s ease-in-out infinite;
}

/* Attention-getter: wiggle a notification dot */
@keyframes ping {
    from { transform: scale(1);   opacity: 1;   }
    to   { transform: scale(2.5); opacity: 0;   }
}
/* Place .ping behind the dot using position:absolute — it expands and fad
.notification-ping { animation: ping 1.5s ease-out infinite; }
```

11 — prefers-reduced-motion

> **Accessibility requirement.** Vestibular disorders, epilepsy, and motion sensitivity affect a significant portion of users. On Windows, macOS, iOS, and Android, users can enable "Reduce Motion" in system settings. Always respect this preference — unchecked motion is a WCAG 2.1 failure criterion in some contexts and at minimum is inconsiderate to these users.

🎨 CSS — reduced motion patterns

```
/* Nuclear option — disable all animations site-wide */
@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;
    }
}

/* Better — replace large motion with a subtle alternative */
.hero-animation {
    animation: slide-in-left 0.8s ease both;
}
@media (prefers-reduced-motion: reduce) {
    .hero-animation {
        animation: fade-in 0.3s ease both;
```

```
        /* Keep opacity fade — it's the big translate that's disorienting
    }
}

/* No-preference (motion is OK) — opt-in to a more elaborate animation */
@media (prefers-reduced-motion: no-preference) {
    .card { animation: bounce-in 0.6s cubic-bezier(0.34, 1.56, 0.64, 1) bo
}
```

Opacity fades are generally safe for motion-sensitive users. Large translates, rotations, and scaling (especially fast ones) are the triggers to avoid. When in doubt, replace the motion with a simple opacity fade rather than removing all animation.

12 — Quick Reference

PROPERTY	VALUES	NOTES
`animation-name`	Keyframe name or `none`	Matches a `@keyframes` rule name
`animation-duration`	Time in `s` or `ms`	Required — defaults to `0s` (instant)
`animation-timing-function`	`ease`, `linear`, `steps(n)`, `cubic-bezier()`	Applies to each keyframe segment
`animation-delay`	Time; negative = start partway through	Negative values useful for stagger waves
`animation-iteration-count`	Number or `infinite`	Can be a decimal: `1.5` = 1½ plays
`animation-direction`	`normal`, `reverse`, `alternate`, `alternate-reverse`	`alternate` = natural ping-pong loop
`animation-fill-mode`	`none`, `forwards`, `backwards`, `both`	Use `both` for delayed entrance animations

PROPERTY	VALUES	NOTES
`animation-play-state`	`running`, `paused`	Toggle with a class or `:hover`

PATTERN	KEY PROPERTIES
Entrance (one-shot)	`iteration-count: 1; fill-mode: both`
Infinite loop	`iteration-count: infinite; fill-mode: none`
Ping-pong loop	`iteration-count: infinite; direction: alternate`
Stagger cascade	Increasing `animation-delay` per element
Pause on hover	`.el:hover { animation-play-state: paused }`
Reduced motion	`@media (prefers-reduced-motion: reduce)`

🖍️ Exercises

All exercises should wrap their animations in a `prefers-reduced-motion: reduce` query that either removes or simplifies the motion.

EXERCISE 1

Write a skeleton loading card component. The card should have three placeholder lines (a wide title bar and two narrower text lines). Each placeholder is a `<div>` with a shimmer animation — a gradient that sweeps left to right giving the illusion of light passing over the surface. The gradient should be 400% wide so it travels fully across. Add a stagger so each bar's shimmer starts slightly offset from the previous.

Hint: Use `background: linear-gradient(90deg, #eee 25%, #ddd 50%, #eee 75%)` with `background-size: 400% 100%` and animate `background-position` from `-200% 0` to `200% 0`. Stagger via `animation-delay` on each nth-child.

▶ Show Sample Solution

EXERCISE 2

Create a typewriter animation for the string "Hello, World!" (13 characters). The text should appear character by character using `steps()`. After typing completes, a blinking cursor (the right border of the element) should blink indefinitely. The cursor should start blinking only after the typing animation finishes — use `animation-delay` on the cursor animation equal to the typing duration. Wrap everything in a `prefers-reduced-motion` override that shows the full text immediately.

Hint: `overflow: hidden; white-space: nowrap` on the element, animate `width` from 0 to 13ch using `steps(13, end)`. The cursor uses a separate `@keyframes` toggling `border-right-color` between the text colour and transparent.

▶ Show Sample Solution

EXERCISE 3

Build a notification bell icon that plays a "ring" animation once when a class `.ringing` is added via JavaScript. The ring: rotate from -15deg to 15deg rapidly (4–6 oscillations), using a single `@keyframes` with percentage stops. Set `transform-origin: top center` so it swings from its top. After the animation finishes, the class should be removed — use JavaScript's `animationend` event. The animation should not repeat.

Hint: Write the shake as percentages 0%→15%→30%→45%→60%→75%→100% alternating ±15deg, ending at 0deg. Set `transform-origin: top center` on the icon. JS: `el.addEventListener('animationend', () => el.classList.remove('ringing'))`.

▶ Show Sample Solution

EXERCISE 4

Build a staggered card grid that animates in when a `.loaded` class is added to the parent container. Six `.card` items should each fade up (opacity 0→1 + translateY 24px→0) with a cascade delay: 0, 80ms, 160ms, 240ms, 320ms, 400ms. Use CSS custom properties and `calc()` for the delays — set `--i` on each card via JavaScript. Each card starts invisible and only becomes visible when the animation begins (use `fill-mode: both`). Include a reduced-motion fallback that shows all cards immediately without animation.

Hint: The base rule has `animation: fade-up 0.45s ease calc(var(--i, 0) * 80ms) both`. JS: `cards.forEach((c, i) => c.style.setProperty('--i', i))`. The `.loaded` class triggers the animation only when added to the parent.

▶ Show Sample Solution

CHAPTER 11

CSS Architecture — BEM and Utility-First Thinking

📐 Chapter 11 — CSS Architecture: BEM and Utility-First Thinking

CSS has no enforced structure — you can name a class anything, nest selectors anywhere, and override anything with specificity. For a personal project, that freedom is fine. On a team or a codebase that grows over months, it becomes a liability: specificity wars, fear of deleting "dead" code that might secretly be used somewhere, and styles that only work because of the exact order they were written in. CSS architecture methodologies provide naming conventions and structural rules that tame this chaos.

1 — The Problem with Unstructured CSS

Imagine three developers separately adding styles for a button over six months:

🎨 CSS — the specificity war

```
/* Developer A, Month 1 */
button              { background: blue; }        /* 0,0,1 */

/* Developer B, Month 3 — needed it red in the sidebar */
.sidebar button     { background: red;  }        /* 0,1,1 */

/* Developer C, Month 6 — needs it blue again in sidebar */
.sidebar .btn       { background: blue !important; }  /* nuclear option */

/* Now nobody can safely change anything without checking
   every combination of .sidebar, button, .btn, !important... */
```

The result is CSS that only works because of its *order* and because of *implicit knowledge* of what elements exist in the HTML. Architecture methodologies eliminate these problems before they start.

Specificity scores of common selectors

`.button`	`0,1,0`	BEM — single class, flat and safe
`div.button`	`0,1,1`	Tag qualifier — harder to override
`.sidebar .button`	`0,2,0`	Context dependency — breaks when HTML changes
`.nav .sidebar .button`	`0,3,0`	Deep nesting — brittle, hard to test
`#header .nav .button`	`1,2,0`	ID selector — almost impossible to override cleanly
`.button { color: red !important; }`	∞	!important — last resort, causes escalation

2 — BEM: Block Element Modifier

BEM, developed at Yandex in 2005, solves the specificity problem with a strict naming convention. Every CSS rule targets exactly one class, keeping specificity at 0,1,0 everywhere. The naming convention encodes the component structure into the class names themselves, making the HTML self-documenting.

The three parts

🎨 **CSS — BEM naming rules**

```
/* BLOCK — a standalone, reusable component */
.card    { }  /* the card as a whole */
.nav     { }
.button  { }
.modal   { }

/* ELEMENT — a child of a block, separated by __ (double underscore) */
.card__image      { }  /* .card's image */
.card__title      { }
.card__footer     { }
.nav__item        { }
.modal__overlay   { }
```

```
/* MODIFIER — a variant or state, separated by -- (double dash) */
.card--featured  { }  /* a featured card */
.card--compact   { }  /* a smaller card */
.button--primary { }
.button--disabled{ }
.nav__item--active{ } /* element modifier */
```

A modifier is always used in addition to the base class: `<div class="card card--featured">` — not `<div class="card--featured">` alone. The modifier only overrides the parts that change.

BEM rules and anti-patterns

🎨 CSS — BEM: good vs bad

```
/* ✓ GOOD — flat specificity, predictable */
.card            { border-radius: 8px;    }   /* 0,1,0 */
.card__title     { font-size: 1.2rem;  }   /* 0,1,0 */
.card--featured  { border-color: gold; }   /* 0,1,0 */

/* ✗ ANTI-PATTERN: nesting element inside block selector */
.card .card__title { }  /* specificity is now 0,2,0 — harder to override *

/* ✗ ANTI-PATTERN: three levels deep */
.card__body__paragraph { }  /* wrong — use .card__text instead */

/* ✗ ANTI-PATTERN: element selector inside BEM */
.card h2 { }  /* breaks when h2 becomes h3 or a div */

/* ✗ ANTI-PATTERN: BEM + utility chaos */
.card--featured.is-active.has-image { }  /* two different systems collidin

/* ✓ GOOD: The MIX pattern — one element, two blocks */
/* <div class="card  layout__item"> */
/* .card controls the card's appearance                    */
/* .layout__item controls how it sits in its container     */
```

🏷️ HTML — BEM markup in practice

```
<!-- Navigation component -->
<nav class="nav">
    <ul class="nav__list">
        <li class="nav__item">
            <a class="nav__link" href="/">Home</a>
        </li>
        <li class="nav__item nav__item--active">
            <a class="nav__link" href="/about">About</a>
        </li>
        <li class="nav__item">
            <a class="nav__link nav__link--cta" href="/contact">Contact</a
        </li>
    </ul>
</nav>
<!-- The HTML alone tells you: nav (block), nav__item (elements),
     --active and --cta (modifiers). No reading the CSS file needed. -->
```

3 — Utility-First CSS

Utility-first takes the opposite approach: instead of naming components and writing CSS for them, you compose dozens of single-purpose classes directly in the HTML. Every class does exactly one thing: `.flex`, `.p-4`, `.text-blue-500`, `.rounded-lg`, `.hover:shadow-md`. Tailwind CSS popularised this approach and is now the most-downloaded CSS framework.

HTML — BEM vs utility-first: same button

```
<!-- BEM — semantic class name, styling in CSS file -->
<button class="button button--primary button--large">
    Submit
</button>

<!-- Utility-first (Tailwind) — styling in HTML itself -->
<button class="bg-blue-600 text-white px-6 py-3 rounded-lg
               font-semibold text-base hover:bg-blue-700
               active:scale-95 transition-all">
    Submit
```

```
</button>

<!-- Tailwind component abstraction — best of both:
     extract to a component in your JS framework -->
<PrimaryButton>Submit</PrimaryButton>
<!-- The component internally uses utilities but hides them -->
```

Writing your own utility classes

🎨 **CSS — a simple utility layer**

```css
/* Spacing utilities (u- prefix marks utility) */
.u-mt-0  { margin-top:  0;        }
.u-mt-1  { margin-top:  0.5rem;   }
.u-mt-2  { margin-top:  1rem;     }
.u-mt-4  { margin-top:  2rem;     }

/* Display utilities */
.u-flex  { display: flex;         }
.u-grid  { display: grid;         }
.u-hidden{ display: none;         }

/* Text utilities */
.u-truncate {
    overflow:     hidden;
    text-overflow:ellipsis;
    white-space:  nowrap;
}
.u-sr-only {  /* screen-reader only */
    position:   absolute;
    width:      1px;
    height:     1px;
    clip:       rect(0,0,0,0);
    overflow:   hidden;
}
```

4 — Other Methodologies (Overview)

OOCSS

Object-Oriented CSS — Nicole Sullivan, 2009

Two rules: separate *structure from skin* (layout vs appearance) and separate *container from content* (styles not tied to location). Predates BEM; introduced the "object" mental model.

SMACSS

Scalable and Modular Architecture for CSS — Jonathan Snook, 2011

Organises rules into five categories: **Base** (resets), **Layout** (l- prefix), **Module** (components), **State** (is- prefix), **Theme**. Good for large projects.

ITCSS

Inverted Triangle CSS — Harry Roberts

Orders CSS from generic to specific in layers: Settings → Tools → Generic → Elements → Objects → Components → Utilities. Specificity only increases, preventing cascade conflicts.

CUBE CSS

Composition Utility Block Exception — Andy Bell, 2020

A modern hybrid: lean into the cascade (C), use utility classes (U), define blocks for components (B), handle exceptions with modifiers (E). Embraces CSS rather than fighting it.

5 — The Hybrid Approach (Most Practical)

Most real projects land somewhere between pure BEM and pure utility-first. A practical hybrid: **BEM for components, utilities for layout and one-offs.**

🏷️ **HTML — hybrid: component + utility helpers**

```
<!-- Component class handles the card's appearance -->
<!-- Utility classes handle spacing, layout, one-offs -->
<article class="card card--featured  u-mt-4">
    <h2 class="card__title  u-truncate">Long Title That Gets Cut Off</h2>
    <p  class="card__excerpt">Excerpt...</p>
</article>

<!-- Layout shell is utility-driven -->
<div class="u-grid u-grid-cols-3 u-gap-4">
    <article class="card">…</article>
    <article class="card card--featured">…</article>
</div>
```

6 — CSS Custom Properties as Architecture

Regardless of which naming methodology you choose, CSS custom properties (variables) provide the foundation that makes any system consistent. Design tokens — the atoms of a design system — live in `:root` and are consumed everywhere.

🎨 **CSS — design tokens in :root**

```css
/* Layer 1: Raw values (primitive tokens) */
:root {
    /* Color palette */
    --blue-500:  #4f8ef7;
    --blue-600:  #3a7ae5;
    --gray-100:  #f3f4f6;
    --gray-900:  #111827;

    /* Layer 2: Semantic tokens (what things ARE) */
    --color-primary:     var(--blue-500);
    --color-primary-hover:var(--blue-600);
    --color-surface:     var(--gray-100);
    --color-text:        var(--gray-900);

    /* Spacing scale */
    --space-1: clamp(0.5rem,  1vw,   0.75rem);
    --space-2: clamp(1rem,    2vw,   1.5rem);
    --space-4: clamp(2rem,    4vw,   3rem);

    /* Typography */
    --text-base: clamp(1rem, 1.5vw + 0.25rem, 1.25rem);
    --text-lg:   clamp(1.25rem, 2.5vw, 1.75rem);

    /* Component tokens (consume semantic tokens) */
    --card-padding:      var(--space-2);
    --card-radius:       8px;
    --card-border:       1px solid var(--color-border);
}

/* Dark mode — just re-map the semantic tokens */
@media (prefers-color-scheme: dark) {
    :root {
        --color-surface: var(--gray-900);
        --color-text:    var(--gray-100);
        /* --blue-500 stays the same — only semantic tokens change */
```

```
    }
}

/* Components reference tokens — never raw values */
.card {
    background:    var(--color-surface);
    color:         var(--color-text);
    padding:       var(--card-padding);
    border-radius:var(--card-radius);
}
/* Swap themes by re-mapping :root tokens — no component CSS changes */
```

7 — Choosing an Approach

CONTEXT	RECOMMENDATION	WHY
Small personal project	Whatever feels natural	Architecture pays off at scale, not worth the overhead on solo projects
Team project (no framework)	BEM + design tokens	Predictable naming prevents cross-developer conflicts
React / Vue / Svelte project	CSS Modules or utility framework	Component scoping solves the global namespace problem; Tailwind integrates naturally
Large-scale site / CMS	ITCSS or SMACSS + BEM	Layer-based architecture handles hundreds of components cleanly
Design system / component library	Design tokens + BEM	Tokens provide consistency; BEM makes components self-contained
Rapid prototyping	Utility-first (Tailwind)	No context switching between HTML and CSS — fastest iteration speed

> 💡 **The rule that matters most:** *Pick one system and apply it consistently. A project using BEM consistently for six months is vastly better than one that uses BEM in some files, SMACSS*

in others, and no methodology in the rest. Consistency is more valuable than choosing the "best" methodology.

8 — Quick Reference

CONCEPT	SYNTAX	RULE
Block	`.card`	Standalone reusable component
Element	`.card__title`	Part of a block, double underscore
Modifier	`.card--featured`	Variant or state, double dash — always used alongside base class
Element modifier	`.card__title--truncated`	State of an element — keep depth to two levels maximum
MIX	`class="card layout__cell"`	One HTML element plays two roles — different concerns, different classes
Utility	`.u-truncate`	Single-purpose class; prefix to distinguish from components
Design token	`--color-primary`	Semantic custom property; always reference tokens not raw values

🖍️ Exercises

Good architecture is a habit, not a one-time choice. These exercises build the muscle memory of naming things intentionally.

EXERCISE 1

Name every element in a blog post component using BEM. The component contains: the outer article wrapper, a hero image, a category badge (on top of the image), the content area, a heading, a meta row (author name + date + read time), the body text, a tags row (list of tag pills), and a share buttons row. Write only the class names — the CSS can be blank. Use modifiers where relevant (featured post, pinned post).

Hint: Start with the block name (e.g. `.post`). Elements use `.post__*`. Go only two levels deep — if a tag pill contains an icon, that icon is `.post__tag-icon` not `.post__tags__tag__icon`.

▶ Show Sample Solution

EXERCISE 2

Convert this "messy CSS" into BEM. The original uses tag selectors, ID selectors, context-dependent rules, and `!important`. Rewrite it as flat BEM rules — all targeting a single class at 0,1,0 specificity. Add modifier rules where the original had context-based overrides.

Hint: Remove all ID selectors, element selectors, and descendant/child combinators. Every selector should be a single class.

▶ Show Sample Solution

EXERCISE 3

Design a design token system for a simple project. Define: (a) a palette of 3 brand colours with 3 shades each in numbered custom properties; (b) semantic tokens mapping those to purpose (`--color-primary`, `--color-danger`, `--color-surface`, `--color-text`, `--color-border`); (c) a dark-mode override using `@media (prefers-color-scheme: dark)` that only re-maps the semantic tokens; (d) a spacing scale using `clamp()` from `--space-1` to `--space-6`.

Hint: The primitive tokens (raw colors) stay the same in dark mode. Only the semantic tokens change — `--color-surface` flips from a light to dark shade.

▶ Show Sample Solution

EXERCISE 4

Audit a real CSS file (any file from a previous exercise or your own project) and identify: (a) any selectors with specificity higher than 0,1,0; (b) any use of `!important`; (c) any selectors that depend on HTML structure (descendant selectors); (d) any duplicate property declarations that differ only in context. Write a refactored version of the most problematic section using either BEM naming or design tokens, and explain in a comment block at the top why each change was made.

Hint: DevTools > Elements > Styles shows specificity if you hover over selectors. Or use the Specificity Calculator at specificity.keegan.st. Look for rules where you had to add more selectors

"to make it win."

▶ Show Sample Solution

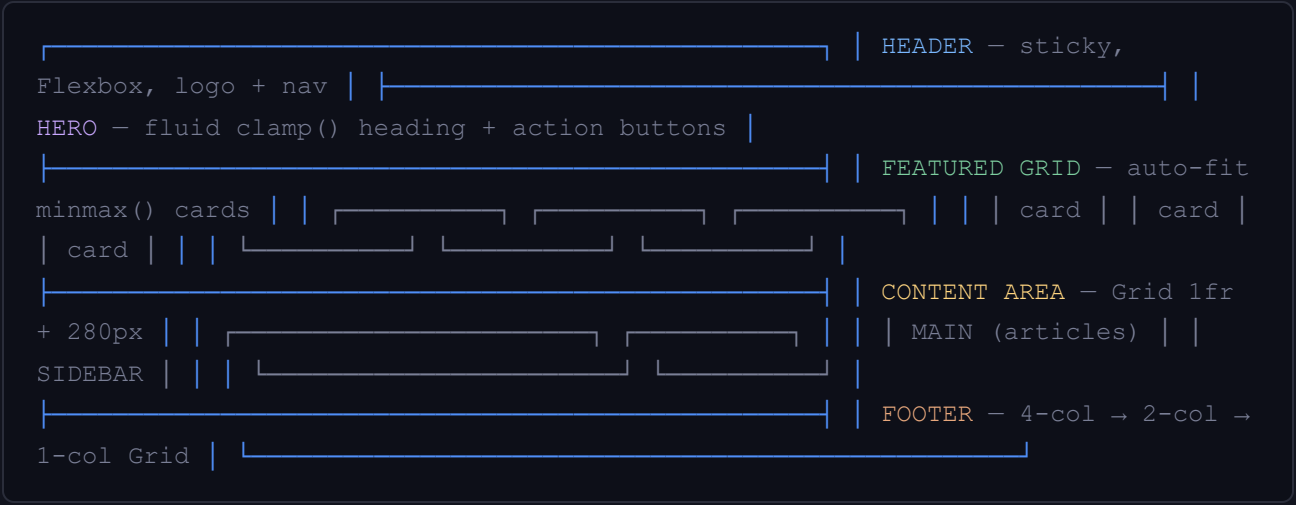
CHAPTER 12

Project: Fully Responsive Multi-Column Layout

📐 Chapter 12 — Project: Fully Responsive Multi-Column Layout

This is the capstone project for the CSS Intermediate course. We'll build a magazine-style blog layout from scratch, combining every technique from the previous eleven chapters: design tokens, Flexbox headers, Grid post grids, fluid typography, transforms on hover, keyframe animations, and BEM naming. Work through each step in order — by Step 9 you'll have a production-quality responsive layout.

> **What we're building:** A blog with a sticky header, a hero section, a three-column featured posts grid, a two-column content area (articles + sidebar), and a responsive footer. It collapses cleanly from 1100px down to 360px mobile without a single media query hack.



Step 1 — Design Tokens

Chapter 9 skill: CSS custom properties

Everything else references these tokens. Define them first and nothing downstream ever needs a raw value — dark mode, rebranding, or spacing tweaks become single-line changes.

🎨 **CSS — design tokens in :root**

```css
:root {
    /* ── Colors ── */
    --clr-bg:          #0d0f18;
    --clr-surface:     #1a1d27;
    --clr-surface-2:   #12141e;
    --clr-border:      #2a2d3a;
    --clr-primary:     #4f8ef7;
    --clr-primary-dk:  #3a7ae5;
    --clr-accent:      #c9a8ff;
    --clr-text:        #e2e4ec;
    --clr-text-muted:  #7a7f96;

    /* ── Fluid spacing scale ── */
    --sp-1: clamp(0.5rem,  1.5vw, 0.875rem);
    --sp-2: clamp(1rem,    2.5vw, 1.5rem);
    --sp-3: clamp(1.5rem,  4vw,   2.5rem);
    --sp-4: clamp(2rem,    6vw,   4rem);
    --gap:  clamp(1rem,    2.5vw, 1.75rem);

    /* ── Fluid type scale ── */
    --text-sm:   clamp(0.75rem, 1.5vw,  0.875rem);
    --text-base: clamp(0.9rem,  2vw,    1rem);
    --text-lg:   clamp(1.1rem,  2.5vw,  1.25rem);
    --text-xl:   clamp(1.4rem,  4vw,    2rem);
    --text-hero: clamp(1.8rem,  6vw,    3.5rem);

    /* ── Layout ── */
    --max-w:        1100px;
    --header-h:     60px;
    --sidebar-w:    280px;
}
```

Step 2 — Base Reset

Foundation

🎨 CSS — base reset

```css
*, *::before, *::after {
    box-sizing: border-box;
    margin:     0;
    padding:    0;
}

html {
    scroll-behavior: smooth;
}

body {
    background:  var(--clr-bg);
    color:       var(--clr-text);
    font-family: system-ui, -apple-system, sans-serif;
    font-size:   var(--text-base);
    line-height: 1.6;
}

img { display: block; max-width: 100%; }
a   { color: inherit; text-decoration: none; }
ul  { list-style: none; }

/* Shared max-width container */
.container {
    max-width: var(--max-w);
    margin:    0 auto;
    padding:   0 var(--sp-2);
}

/* Section spacing */
.section { padding: var(--sp-3) 0; }
.section__heading {
    font-size:    var(--text-xl);
    font-weight:  700;
    margin-bottom:var(--sp-2);
    padding-bottom:var(--sp-1);
    border-bottom:2px solid var(--clr-primary);
}
```

Step 3 — Sticky Header

Flexbox · position: sticky

The header uses Flexbox with `margin-left: auto` on the nav to push it to the right — the simplest push technique. `position: sticky; top: 0` keeps it visible without JavaScript.

🎨 CSS — .site-header

```
.site-header {
    position:    sticky;
    top:         0;
    z-index:     100;
    height:      var(--header-h);
    background:  var(--clr-surface-2);
    border-bottom:1px solid var(--clr-border);
    display:     flex;
    align-items: center;
}

.site-header__inner {
    display:     flex;
    align-items: center;
    gap:         var(--sp-2);
    width:       100%;
}

.site-header__logo {
    font-size:   var(--text-lg);
    font-weight: 800;
    color:       var(--clr-primary);
}

.site-header__nav {
    display:     flex;
    align-items: center;
    gap:         var(--sp-2);
    margin-left: auto;   /* pushes nav to right edge */
}
```

```
.site-header__link     { color: var(--clr-text-muted); font-size: var(--te
.site-header__link:hover{ color: var(--clr-text);       }
.site-header__link--cta{
    background:   var(--clr-primary);
    color:        #fff;
    padding:      6px 16px;
    border-radius:4px;
    font-weight:  600;
    transition:   background 0.2s;
}
.site-header__link--cta:hover { background: var(--clr-primary-dk); }

@media (max-width: 640px) {
    .site-header__nav { display: none; }  /* add hamburger menu for full b
}
```

Step 4 — Hero Section

clamp() · Flexbox buttons

🎨 CSS — .site-hero

```
.site-hero {
    background: linear-gradient(135deg, var(--clr-bg) 0%, #1a2a4a 100%);
    padding:    var(--sp-4) var(--sp-2);
    text-align: center;
}

.site-hero__inner {
    max-width: 680px;
    margin:    0 auto;
}

.site-hero__label {
    font-size:      var(--text-sm);
    text-transform: uppercase;
    letter-spacing: 0.15em;
    color:          var(--clr-primary);
```

```
    margin-bottom:  var(--sp-1);
}

.site-hero__heading {
    font-size:   var(--text-hero);   /* clamp(1.8rem, 6vw, 3.5rem) */
    font-weight: 800;
    color:       #fff;
    line-height: 1.15;
    margin-bottom:var(--sp-1);
}

.site-hero__sub {
    font-size:   var(--text-base);
    color:       var(--clr-text-muted);
    margin-bottom:var(--sp-2);
}

.site-hero__actions {
    display:         flex;
    gap:             var(--sp-1);
    justify-content: center;
    flex-wrap:       wrap;  /* stack on very narrow screens */
}

/* Shared button styles (BEM block) */
.btn {
    display:      inline-flex;
    align-items:  center;
    gap:          6px;
    padding:      10px 22px;
    border-radius:5px;
    font-weight:  600;
    font-size:    var(--text-sm);
    border:       2px solid transparent;
    cursor:       pointer;
    transition:   background 0.2s, transform 0.15s;
}
.btn:hover         { transform: translateY(-2px);   }
.btn:active        { transform: translateY(0);     }
.btn--primary      { background: var(--clr-primary);  color: #fff; }
.btn--primary:hover{ background: var(--clr-primary-dk);  }
```

```
.btn--ghost        { border-color: var(--clr-border); color: var(--clr-tex
.btn--ghost:hover  { border-color: var(--clr-primary); color: var(--clr-te
```

Step 5 — Featured Posts Grid

Grid · auto-fit · minmax()

The key trick: `minmax(min(280px, 100%), 1fr)`. The inner `min()` clamps the minimum column width to 100% when the viewport is narrower than 280px — preventing overflow on tiny screens without a media query.

🎨 CSS — .posts-grid + .post-card (BEM)

```css
.posts-grid {
    display:               grid;
    grid-template-columns: repeat(auto-fit, minmax(min(280px, 100%), 1fr))
    gap:                   var(--gap);
}

/* BEM block */
.post-card {
    background:    var(--clr-surface);
    border:        1px solid var(--clr-border);
    border-radius: 8px;
    overflow:      hidden;
    display:       flex;
    flex-direction:column;  /* column so body can grow */
    transition:    transform 0.25s, box-shadow 0.25s;
}
.post-card:hover {
    transform:  translateY(-5px);
    box-shadow: 0 12px 32px rgba(0,0,0,.3);
}

.post-card__image {
    aspect-ratio: 16 / 9;
    overflow:     hidden;
```

```
}
.post-card__image img {
    width:      100%;
    height:     100%;
    object-fit: cover;
    transition: transform 0.4s;
}
.post-card:hover .post-card__image img {
    transform: scale(1.05);  /* subtle zoom on hover */
}

.post-card__body   { padding: var(--sp-2); flex: 1; }  /* flex:1 pushes fo
.post-card__tag    {
    display:         inline-block;
    font-size:       0.7rem;
    font-weight:     700;
    text-transform: uppercase;
    letter-spacing: 0.06em;
    color:           var(--clr-primary);
    margin-bottom:   6px;
}
.post-card__title  { font-size: var(--text-lg); font-weight: 700; margin-b
.post-card__excerpt{ font-size: var(--text-sm); color: var(--clr-text-mute
.post-card__footer {
    padding:     var(--sp-1) var(--sp-2);
    border-top:  1px solid var(--clr-border);
    display:     flex;
    justify-content:space-between;
    align-items: center;
    font-size:   var(--text-sm);
    color:       var(--clr-text-muted);
}
```

Step 6 — Content Area: Main + Sidebar

Grid · align-items: start

`align-items: start` prevents the sidebar from stretching to match the main content height — it stays at its own natural height. Without it, the sidebar would fill the entire grid row.

🎨 CSS — .content-area

```css
.content-area {
    display:               grid;
    grid-template-columns: 1fr var(--sidebar-w);  /* main grows, sidebar f
    gap:                   var(--gap);
    align-items:           start;  /* sidebar stays own height */
}

/* Sidebar is sticky so it stays visible while scrolling main */
.content-area__sidebar {
    position: sticky;
    top:      calc(var(--header-h) + var(--sp-2));  /* below sticky header
}

@media (max-width: 768px) {
    .content-area {
        grid-template-columns: 1fr;  /* one column on mobile */
    }
    .content-area__sidebar {
        position: static;  /* un-sticky on small screens */
    }
}
```

Step 7 — Sidebar Widgets

BEM · reusable component

🎨 CSS — .widget (BEM block)

```css
.widget {
    background:    var(--clr-surface);
    border:        1px solid var(--clr-border);
    border-radius: 8px;
```

```
    overflow:      hidden;
    margin-bottom: var(--sp-2);
}

.widget__heading {
    font-size:    var(--text-sm);
    font-weight:  700;
    text-transform:uppercase;
    letter-spacing:0.07em;
    padding:      var(--sp-1) var(--sp-2);
    border-bottom:1px solid var(--clr-border);
    color:        var(--clr-text);
}

.widget__body {
    padding: var(--sp-1) var(--sp-2);
}

.widget__item {
    display:         flex;
    justify-content: space-between;
    align-items:     center;
    padding:         6px 0;
    border-bottom:   1px solid var(--clr-border);
    font-size:       var(--text-sm);
    color:           var(--clr-text-muted);
}
.widget__item:last-child { border-bottom: none; }
.widget__count {
    background:    var(--clr-surface-2);
    color:         var(--clr-accent);
    padding:       1px 7px;
    border-radius: 10px;
    font-size:     0.75em;
    font-weight:   600;
}

/* Tag cloud widget variant */
.widget--tags .widget__body {
    display:   flex;
    flex-wrap: wrap;
    gap:       6px;
```

```
}
.widget__tag {
    background:    var(--clr-surface-2);
    color:         var(--clr-accent);
    padding:       3px 10px;
    border-radius: 12px;
    font-size:     var(--text-sm);
    transition:    background 0.2s;
}
.widget__tag:hover { background: var(--clr-primary); color: #fff; }
```

Step 8 — Responsive Footer

Grid · multi-column → single

🎨 CSS — .site-footer

```
.site-footer {
    background:  var(--clr-surface-2);
    border-top:  1px solid var(--clr-border);
    padding:     var(--sp-3) var(--sp-2);
}

.footer__grid {
    display:               grid;
    grid-template-columns: 2fr 1fr 1fr 1fr;  /* brand col is wider */
    gap:                   var(--gap);
}

/* Collapses to 2 columns on tablet */
@media (max-width: 768px) {
    .footer__grid { grid-template-columns: 1fr 1fr; }
}

/* Single column on mobile */
@media (max-width: 480px) {
    .footer__grid { grid-template-columns: 1fr; }
}
```

```css
/* Footer bottom bar */
.footer__bottom {
    display:         flex;
    justify-content: space-between;
    align-items:     center;
    flex-wrap:       wrap;
    gap:             var(--sp-1);
    margin-top:      var(--sp-3);
    padding-top:     var(--sp-2);
    border-top:      1px solid var(--clr-border);
    font-size:       var(--text-sm);
    color:           var(--clr-text-muted);
}
```

Step 9 — Responsive Strategy Summary

BREAKPOINT	WHAT CHANGES	TECHNIQUE
≥ 769px (Desktop)	Full layout: header nav visible, 3-col grid, 2-col content, 4-col footer	Default styles
≤ 768px (Tablet)	Content area collapses to 1 col; sidebar un-stickies; footer becomes 2-col	`@media (max-width: 768px)`
≤ 640px	Header nav hides (hamburger shown); hero buttons stack	`@media (max-width: 640px)`
≤ 480px	Footer becomes 1 column	`@media (max-width: 480px)`
Any width	Post card grid reflows automatically	`auto-fit minmax(min(280px,100%), 1fr)` — no breakpoint needed
Any width	Type and spacing scale fluidly	`clamp()` on all spacing + font-size tokens

> **Key insight:** *Most of the responsiveness comes from intrinsic sizing — `auto-fit`, `clamp()`, `flex-wrap` — not from media queries. Only structural changes (collapsing the sidebar, hiding the nav)*

> *need breakpoints. Favour intrinsic techniques; add breakpoints only when structure must change.*

Live Preview — Resize the Layout

The preview uses CSS container queries (`container-type: inline-size`) on the frame div — the layout responds to the frame's width, not the viewport width. This is exactly how container queries will work in your real layouts (Chapter 2 of CSS Advanced).

Key Design Decisions

SECTION	TOOL CHOSEN	WHY NOT THE ALTERNATIVE
Header layout	Flexbox	One-dimensional row — Grid would work but adds no value here
Hero buttons	Flexbox + flex-wrap	Inline-block would need media queries to stack; flex-wrap handles it automatically
Featured cards	Grid auto-fit	Flexbox can't guarantee equal-width columns without media queries; auto-fit does it in one line
Main + Sidebar	Grid (explicit tracks)	Flexbox can't hold the sidebar at a fixed 280px while the main grows — Grid `1fr var(--sidebar-w)` does exactly this
Cards (internal)	Flexbox column	One-dimensional vertical stack; `flex: 1` on body pushes footer to bottom — Grid overkill here
Footer	Grid (explicit tracks)	Four columns with unequal widths (brand 2fr, rest 1fr) — Grid template columns are cleaner than flex
Spacing	`clamp()` tokens	Fixed values need breakpoints for every size; clamp scales continuously
Typography	`clamp()` tokens	Same — no extra media queries needed for text sizing

🚀 Go Further — Optional Challenges

- **Hamburger menu:** Add a hamburger button that appears at ≤ 640px and toggles the nav with a CSS class + `max-height` transition.
- **Skeleton loading:** Add a `.is-loading` modifier to cards that shows a shimmer animation (from Ch 10 Exercise 1) while content loads.
- **Dark / light toggle:** Add a toggle button that switches a `data-theme="light"` attribute on `<html>`, and remap the color tokens for light mode.
- **Scroll reveal:** Use `IntersectionObserver` to add a `.revealed` class to cards as they enter the viewport, triggering a `translateY(0)` + `opacity: 1` animation.
- **Reading progress bar:** A fixed `1px` bar at the top of the header that grows with `scaleX()` as the user scrolls, driven by `scroll` event and transform.

🖍️ Exercises

EXERCISE 1

Implement the sticky sidebar with a `calc()` top offset. The sidebar should stick below the header without overlapping it. The header is 60px tall and you want 16px of breathing room. Write the CSS, and explain why `top: calc(var(--header-h) + 16px)` is better than `top: 76px`.

Hint: When the header height changes (e.g. in a media query you set `--header-h: 48px`), the calc value auto-updates. A hard-coded 76px would need a second override in that media query.

▶ Show Sample Solution

EXERCISE 2

Add a `.post-card--featured` modifier that makes one card span all columns in the grid and display horizontally (image left, content right) rather than the default vertical stack. Use `grid-column: 1 / -1` to span all columns, and switch the card to a row-direction Flexbox layout internally.

Hint: The card itself uses `flex-direction: column`. Override it to `row` in the modifier. You'll also need to constrain the image width (`flex: 0 0 40%`) so it doesn't take all available space.

▶ Show Sample Solution

EXERCISE 3

Add a scroll-reveal animation to the post cards. When a card enters the viewport, it should animate from `opacity: 0; transform: translateY(24px)` to its natural position. Use `IntersectionObserver` in JavaScript to add a `.revealed` class when each card becomes visible. Include a `prefers-reduced-motion` CSS override that skips the animation entirely.

Hint: Set initial styles on `.post-card` (opacity 0, translateY), then style `.post-card.revealed` to restore them. The IntersectionObserver callback adds `.revealed` when the threshold is crossed. Use `observer.unobserve(entry.target)` so the animation only fires once.

▶ Show Sample Solution

EXERCISE 4 — CAPSTONE EXTENSION

Add a light/dark theme toggle to the layout. Store the palette in two sets of `:root` custom properties — one for dark (default) and one for `[data-theme="light"]` on `<html>`. The toggle button switches the attribute. Requirements: (a) only the semantic tokens change between themes (`--clr-bg`, `--clr-surface`, `--clr-text`, etc.) — the primitive palette (raw hex values) stays unchanged; (b) all transitions in the layout should run smoothly when the theme switches; (c) the chosen theme should persist in `localStorage` and be restored on page load.

Hint: Add `transition: background 0.3s, color 0.3s, border-color 0.3s` to `*, *::before, *::after` in the base reset (or on the key layout elements). Use `localStorage.setItem('theme', 'light')` and `document.documentElement.dataset.theme`.

▶ Show Sample Solution

Course complete! You've finished the CSS Intermediate course — 12 chapters covering Flexbox, Grid, cascade, media queries, CSS functions, transforms, animations, architecture, and a full project build. The CSS Advanced course picks up from here with `@layer`, container queries, subgrid, `:has()`, scroll-driven animations, and building a complete design system.