

WEB DEVELOPMENT COURSE

Tailwind CSS Fundamentals

A hands-on guide to Tailwind CSS v4: every example compiled and measured in a real browser.

12 Chapters

Philip Osztromok

Generated with Claude

TAILWIND CSS FUNDAMENTALS

Contents

Chapter 1	Getting Started	2
Chapter 2	Spacing and Type	13
Chapter 3	Colour and Borders	24
Chapter 4	Flex and Grid	33
Chapter 5	Position and Stack	42
Chapter 6	State Variants	51
Chapter 7	Responsive Design	61
Chapter 8	Dark Mode	68
Chapter 9	Your Theme	76
Chapter 10	Beyond Utilities	84
Chapter 11	Motion	94
Chapter 12	Real Components	103

Tailwind CSS Fundamentals

Chapter 1 · Why Utility-First, and Getting Set Up

Tailwind CSS gives you a large set of small classes, each doing one job, and you style a page by putting them in your HTML: `p-4` for padding, `text-white` for a white text colour, `rounded-lg` for rounded corners. A tool reads your files, finds the classes you used, and writes a CSS file containing only those. This chapter does the practical part first: build something, look at exactly what Tailwind produced, and learn how it decides what to produce. Everything after this chapter is learning which classes exist and how to combine them.

> **EVERY EXAMPLE HERE WAS RUN**
>
> The commands and output in this chapter come from running them with Tailwind CSS **4.3.3**, Node.js 24.18, npm 11.16 and Vite 8.3 in a scratch project (the shell was Git Bash on Windows). Timings such as "Done in 50ms" vary from run to run, and file names such as Vite's `index-DqPXTuJK.css` change with the content. The Play CDN example was run in headless Chrome. Anything I did not run is labelled.
>
> This course uses Tailwind **v4**; the older v3 style (a `tailwind.config.js` file) is what *CSS Frameworks 3* describes.

What a Utility Class Is

A utility class does one small thing, named after what it does. Here is a box with four of them:

```
<div class="p-4 bg-blue-500 text-white rounded-lg">A card</div>
```

```
.rounded-lg {
  border-radius: var(--radius-lg);
}
.bg-blue-500 {
  background-color: var(--color-blue-500);
}
.p-4 {
  padding: calc(var(--spacing) * 4);
}
.text-white {
  color: var(--color-white);
}
```

The second box is the CSS Tailwind generated for those four classes (copied from the real output file, which you will meet in a moment). Each class is one rule, and each rule reads from a *theme variable*: `--spacing` is `0.25rem`, so `p-4` is four of those, 1rem. The names are short because you will type them hundreds of times: `p` is padding, `bg` is background, `text` is text colour (or size, depending on the value that follows), and the number is a step on a scale.

The argument for working this way, and the usual objection to it ("my markup is full of classes"), is made in *CSS Frameworks 2*, so it is not repeated here. In one sentence: you stop inventing names for things and stop switching between your HTML and a stylesheet, and in return you accept longer `class` attributes. The rest of this course shows how to keep that trade-off comfortable.

Your First Build with the CLI

The Tailwind CLI is the simplest way to see what Tailwind does, because nothing sits between you and the output. You need Node.js. In an empty folder:

```
npm init -y
npm install tailwindcss @tailwindcss/cli
mkdir src
```

Two files go in `src`. The CSS file is one line, and it is the whole of Tailwind's setup in v4:

```
/* src/input.css */
@import "tailwindcss";
```

The page uses the classes. It links `output.css`, which does not exist yet:

```
<!-- src/index.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="./output.css" rel="stylesheet">
</head>
<body>
```

```
    <h1 class="text-3xl font-bold underline">Hello world!</h1>
    <div class="p-4 bg-blue-500 text-white rounded-lg">A card</div>
  </body>
  </html>
```

Now build:

```
npx @tailwindcss/cli -i ./src/input.css -o ./src/output.css
```

```
≈ tailwindcss v4.3.3

Done in 50ms
```

Open `src/index.html` in a browser: a large bold underlined heading and a blue rounded box with white text. That is a working Tailwind project. The two flags are `-i` (input) and `-o` (output); add `--minify` for a smaller file when you publish.

What Is in the File Tailwind Wrote

Open `output.css`. For this page it is 5,676 bytes, or 5,009 bytes with `--minify`. It is organised into **cascade layers**, which you can see with one command:

```
grep "^@layer" src/output.css
```

```
@layer properties;
@layer theme, base, components, utilities;
@layer theme {
@layer base {
@layer utilities {
@layer properties {
```

The second line declares the order: `theme` first, `utilities` last, and a later layer beats an earlier one no matter how specific the selectors are. That is why a utility such as `p-4` reliably overrides the reset in the `base` layer. The layers hold:

- **theme**: the variables your page *uses*, and only those.
- **base**: Tailwind's reset (called Preflight), the same for every page.

- **utilities**: one rule for each class Tailwind found in your files.
- **properties**: a few internal variables (the `--tw-…` ones) that some utilities share.

The `theme` layer for this page is short, because the page uses few things:

```
sed -n '/^@layer theme {/,/^}/p' src/output.css
```

```
@layer theme {
  :root, :host {
    --font-sans: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue",
      "Noto Sans", Arial, sans-serif, "Apple Color Emoji", "Segoe UI Emoji",
      "Segoe UI Symbol", "Noto Color Emoji";
    --font-mono: ui-monospace, SFMono-Regular, Menlo, Monaco, Consolas, "Liberation Mono",
      "Courier New", monospace;
    --color-blue-500: oklch(62.3% 0.214 259.815);
    --color-white: #fff;
    --spacing: 0.25rem;
    --text-3xl: 1.875rem;
    --text-3xl--line-height: calc(2.25 / 1.875);
    --font-weight-bold: 700;
    --radius-lg: 0.5rem;
    --default-font-family: var(--font-sans);
    --default-mono-font-family: var(--font-mono);
  }
}
```

Two things to notice. Tailwind has hundreds of colours, sizes and radii in its default theme, but the file contains only the ones this page needs: `--color-blue-500` and `--color-white`, not the whole palette. And the colours are written in `oklch()`, a modern colour notation; browsers have only supported it for a few years, which is part of why Tailwind has browser requirements (see the end of this chapter). Chapter 9 shows how to add your own theme variables.

> **PREFLIGHT: WHY A BARE HEADING LOOKS LIKE PLAIN TEXT**
>
> The `base` layer resets what browsers do by default. Among its rules, headings lose their size and weight, and lists lose their bullets:

```
grep -A3 -E "^  (h1, h2|ol, ul)" src/output.css
```

```
    h1, h2, h3, h4, h5, h6 {
      font-size: inherit;
      font-weight: inherit;
    }
  --
    ol, ul, menu {
      list-style: none;
    }
    img, svg, video, canvas, audio, iframe, embed, object {
```

So an `<h1>` with no classes looks like body text, and a `<ul>` has no bullets: every visual decision is yours to make with a class. The same layer also zeroes margins and padding on everything, sets `box-sizing: border-box`, and makes images `display: block` with `max-width: 100%`. If a page looks "unstyled" after you add Tailwind, this layer is almost always the reason, and Exercise 3 below makes you read it.

How Tailwind Finds Your Classes

Tailwind does not run your code. It reads your project's files as plain text, looks for strings that match one of its class names, and generates a rule for each match. Two things follow. First, a class that is not in your files is not in the output: delete `p-4` from the page, rebuild, and the `.p-4` rule is gone (I checked: `grep -c "\.p-4"` printed `0`). Second, **the class has to appear in the file as a whole word**. This is the mistake that catches everyone once:

```
// badge.js
const color = "red";
el.className = `bg-${color}-500 p-2`;   // pieced together at run time
el.className2 = "bg-green-500 p-3";    // written out in full
```

```
.bg-green-500
.p-2
.p-3
```

Those three lines are the utilities Tailwind generated from that file. `bg-green-500` is there because it is written out; `bg-red-500` is *not*, because the text `bg-red-500` never appears in the file, only the pieces `bg-` and `-500`. (`p-2` is there because that part is written whole.) The fix is to write complete class names, even if it looks repetitive; Exercise 2 shows a tidy way to do it.

Chapter 1 of *Tailwind CSS Intermediate/Advanced* covers the other options for classes Tailwind cannot see.

Which files does it read? By default it scans the project folder you run it from, and it skips some things. I tested this in a small git project with five files that each used a different margin class: `src/a.html` (`m-1`), `ignored.html` (`m-9`, listed in `.gitignore`), `node_modules/pkg/index.html` (`m-8`), `notes.css` (`m-7`) and `pic.png` (`m-6`, really a text file with a picture's extension). Only one class made it into the output:

```
grep -E "^\s+\.m-[0-9]" out.css
```

```
  .m-1 {
```

Files in `.gitignore`, anything in `node_modules`, stylesheets and binary-looking files (images) are skipped. When I removed the `.gitignore` line and rebuilt, `.m-9` appeared. Keep this in mind when a class works in one place and not another: the file may be ignored.

Watch Mode

Running the build by hand after every change gets tedious. Add `--watch` and the CLI stays running and rebuilds when a file changes:

```
npx @tailwindcss/cli -i ./src/input.css -o ./src/output.css --watch
```

WHEN --WATCH EXITS STRAIGHT AWAY

In a normal terminal that command keeps running. But when I ran it from a script (so the program's standard input was not a terminal), it printed the banner, built nothing, and exited. The CLI's own help explains why: `-w, --watch[=always]`, "use `always` to keep watching when stdin is closed". With `--watch=always` it built the file, then rebuilt each time I edited the page (the log showed a "Done" line for each build). If a watch task in your editor, a task runner or a container seems to do nothing, try `--watch=always`.

Using Tailwind with Vite

Most projects use a bundler, and Tailwind has a plugin for Vite. The Tailwind installation docs list Laravel, SvelteKit, React Router, Nuxt and SolidJS as frameworks that use this route; it is a good default whenever your project already runs on Vite. Install the plugin, add it to the Vite config, and import Tailwind in your CSS. No PostCSS configuration is needed:

```
npm install tailwindcss @tailwindcss/vite vite

// vite.config.js
import { defineConfig } from 'vite'
import tailwindcss from '@tailwindcss/vite'

export default defineConfig({
  plugins: [tailwindcss()],
})

/* src/style.css */
@import "tailwindcss";
```

The page links the stylesheet (`<link href="/src/style.css" rel="stylesheet">`), and a normal Vite build does the rest. For the same one-heading page:

```
npx vite build
```

```
transforming...
✓ 3 modules transformed.
rendering chunks...
computing gzip size...
dist/index.html                 0.30 kB │ gzip: 0.23 kB
dist/assets/index-DqPXTuJK.css  4.77 kB │ gzip: 1.63 kB

✓ built in 226ms
```

The CSS comes out at 4.77 kB (1.63 kB compressed) and is already minified, with the same layers as before. Notice that a framework with its own build, such as Astro or Next.js, needs its own variation of these steps; those are covered in *Tailwind CSS Intermediate/Advanced 5*. I did not run the Vite dev server in this chapter; the build above is what I ran.

The Play CDN: Prototypes Only

For a five-minute experiment you can skip the build entirely. One script tag loads Tailwind into the page, and it generates the CSS in the browser as you go:

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
</head>
<body>
  <h1 class="text-3xl font-bold underline">Hello world!</h1>
</body>
</html>
```

```
with the script:     font-size 30px | weight 700 | text-decoration underline
without the script: font-size 32px | weight 700 | text-decoration none
```

The second box is what I measured in headless Chrome with and without the script: with it, `text-3xl` gave 30px (1.875rem) and `underline` took effect; without it, the browser's default heading showed. The Tailwind documentation says the Play CDN "is designed for development purposes only, and is not intended for production", because the browser does the work on every page view and downloads the script each time. Use it to try a class; use the CLI or Vite for anything you will publish. It also needs a network connection, which is a reason not to depend on it in a course exercise, so the exercises here use the CLI.

Which Way Should I Use?

ROUTE	WHAT IT IS	USE IT FOR	RUN HERE?
CLI	One command: input CSS in, output CSS out; `--watch`, `--minify`	Plain HTML sites, learning, small projects, back-end templates	Yes
Vite plugin	`@tailwindcss/vite` in `vite.config`; build and dev server handle the rest	Any project that already uses Vite (many frameworks do)	Build: yes

ROUTE	WHAT IT IS	USE IT FOR	RUN HERE?
Framework setups	PostCSS plugin or webpack plugin for tools that are not Vite-based (Next.js, for example)	Whatever your framework's build uses	No (Tailwind CSS Intermediate/Advanced 5)
Play CDN	One `<script>` tag; CSS generated in the browser	Quick experiments and demos only	Yes (headless Chrome)

Editor Setup

Two tools are worth installing before you write much markup. The **Tailwind CSS IntelliSense** extension for VS Code (extension ID `bradlc.vscode-tailwindcss`) gives class-name completion, shows the CSS a class produces when you hover it, and warns about conflicting classes. The **Prettier plugin for Tailwind** (npm package `prettier-plugin-tailwindcss`) sorts the classes in each `class` attribute into a consistent order, so that two people writing the same element produce the same text. Both are described, with their settings, in the *VS Code Tailwind CSS Cheat Sheet* in the Sidebar (Programming). I did not install them for this chapter.

Browser Support

Tailwind v4 is built for current browsers. The official compatibility page lists the versions the core framework depends on: **Chrome 111** (March 2023), **Safari 16.4** (March 2023) and **Firefox 128** (July 2024). You have already seen why in the output file: it uses cascade layers (`@layer`), registered custom properties (`@property`) and `oklch()` colours. A few utilities go further and use very new CSS, such as `field-sizing: content`; the same page says to use those only if your audience's browsers support them. If you must support older browsers than these, v4 is not the right tool, and the v3 line is.

One last difference from older tutorials, since you will meet them: in v3 the CSS file began with three lines (`@tailwind base;`, `@tailwind components;`, `@tailwind utilities;`) and a `tailwind.config.js` file held the settings. In v4 it is the single `@import "tailwindcss";` line you used above, and the settings live in the CSS itself. That is the subject of chapter 9.

Hands-On Exercises

EXERCISE 1

In an empty folder, set up the Tailwind CLI from scratch and build a page with a heading (`text-2xl font-semibold`) and a link styled as a button (`inline-block rounded-md bg-teal-600 px-4 py-2 text-white hover:bg-teal-700`). From the output file alone, prove three things: the rule for `bg-teal-600` exists, the hover rule is wrapped in `@media (hover: hover)`, and the two teal colours appear as variables in the theme layer.

View solution

EXERCISE 2

A script builds badge classes with `` `bg-${color}-500 text-white px-2` `` for red, green and blue. Build it and show that no background colour is generated, then change the script (without changing what it displays) so that all three are.

View solution

EXERCISE 3

Build a page with an `<h1>`, a two-item `<ul>` and an `<img>` and no classes at all. Find the three Preflight rules in the output that explain how each element will look. Then add classes to the list so it has bullets and indentation again, and report which utility rules Tailwind generated.

View solution

CHAPTER 1 QUICK REFERENCE

- A utility class does one small job (`p-4`, `text-white`, `rounded-lg`); Tailwind writes a CSS rule for each class it finds in your files, and only those
- v4 setup is one CSS line: `@import "tailwindcss";` (v3 used three `@tailwind` lines and a JS config file)
- CLI: `npm install tailwindcss @tailwindcss/cli`, then `npx @tailwindcss/cli -i ./src/input.css -o ./src/output.css`; `--minify` for production; `--watch` to rebuild on change, and `--watch=always` when standard input is closed
- Vite: `@tailwindcss/vite` in the `plugins` array, `@import "tailwindcss";` in your CSS
- Play CDN (`@tailwindcss/browser@4`): prototypes only, never production

- The output has layers in the order `theme`, `base`, `components`, `utilities`; `theme` holds only the variables you use; `base` is Preflight, the reset
- Tailwind reads files as text: write class names in full (`bg-red-500`, never `bg-${color}-500`); it skips `.gitignore`d files, `node_modules`, stylesheets and images
- Needs Chrome 111, Safari 16.4 or Firefox 128 or newer
- Editor: `bradlc.vscode-tailwindcss` (IntelliSense) and `prettier-plugin-tailwindcss` (class sorting)

COMING NEXT

Tailwind CSS Fundamentals 2 starts on the vocabulary: the spacing scale behind `p-4`, width and height, and the typography utilities for size, weight, line height and spacing.

Tailwind CSS Fundamentals

Chapter 2 · The Scale: Spacing, Sizing and Typography

Most of what you type in Tailwind is a number on a scale. `p-4`, `gap-6`, `w-64` and `text-lg` are not arbitrary: each number is a step, and the steps are the same everywhere, so a page built from them looks consistent without you measuring anything. This chapter covers the scales for space, size and text. Colour is the next chapter.

> **EVERY EXAMPLE HERE WAS RUN**
>
> The CSS shown is copied from Tailwind CSS 4.3.3's own output. The pixel values in the tables were measured in headless Chrome (a 16px root font size, as in every browser by default). Where a width depends on the font, such as `65ch`, I say so.

The Spacing Scale

Padding, margin, gap, width, height and several other things all share one scale, built from a single theme variable:

```
--spacing: 0.25rem;
```

```
.p-4 {
  padding: calc(var(--spacing) * 4);
}
```

That is the whole idea: the number in a class is a multiple of `0.25rem`, which is 4px at the default font size. `p-4` is four of those, 16px. Measured values for the steps you will use most:

NUMBER	REM	PIXELS	NUMBER	REM	PIXELS
0	0	0px	10	2.5rem	40px
px	—	1px	12	3rem	48px
0.5	0.125rem	2px	16	4rem	64px
1	0.25rem	4px	20	5rem	80px

NUMBER	REM	PIXELS	NUMBER	REM	PIXELS
1.5	0.375rem	6px	24	6rem	96px
2	0.5rem	8px	32	8rem	128px
3	0.75rem	12px	40	10rem	160px
4	1rem	16px	48	12rem	192px
5	1.25rem	20px	64	16rem	256px
6	1.5rem	24px	96	24rem	384px
8	2rem	32px			

To go from a design measurement to a class, divide the pixels by 4: 24px of padding is `p-6`, a 320px width is `w-80`. The table is a sample, not a list of the only values. In v4 any number works, because the rule is just "that number times the spacing": I compiled `p-13` and got `calc(var(--spacing) * 13)`, and `p-3.5` gave `calc(var(--spacing) * 3.5)`. Two special cases are not multiples: `p-px` is exactly 1px, and `p-0` is `0px`.

When you need a value that is not on the scale, put it in square brackets, or point at a CSS variable with parentheses:

```
<div class="p-[13px]">…</div>
<div class="p-(--gap)">…</div>
```

```
.p-\[13px\] {
  padding: 13px;
}
.p-\(--gap\) {
  padding: var(--gap);
}
```

Use these sparingly: the whole point of the scale is that most values come from it. Chapter 10 returns to arbitrary values. And because the scale comes from one variable, changing `--spacing` in your theme rescales every padding, margin, gap and size at once (chapter 9).

Padding and Margin

Padding is the space inside an element, margin is the space outside it. The letter picks which, and the suffix picks the side:

CLASS	GENERATED CSS	SIDES
p-4	`padding: calc(var(--spacing) * 4)`	all four
px-4	`padding-inline: calc(var(--spacing) * 4)`	left and right (in left-to-right text)
py-2	`padding-block: calc(var(--spacing) * 2)`	top and bottom
pt-1 pr-2 pb-3 pl-4	`padding-top`, `padding-right`, `padding-bottom`, `padding-left`	one physical side
ps-2 pe-3	`padding-inline-start`, `padding-inline-end`	the start or end of a line, which flips in right-to-left languages
m-4, mx-2, my-2, mt-2 …	`margin`, `margin-inline`, `margin-block`, `margin-top` …; `ms-2` and `me-2` are `margin-inline-start` and `-end`	the same letters as padding

Margin has two extras that padding does not. A **negative margin** puts a minus sign in front of the class: `-mt-2` compiles to `margin-top: calc(var(--spacing) * -2)`, which pulls an element up by 8px (or `-mx-1` for both sides). And `auto` margins: `mx-auto` is `margin-inline: auto`, the classic way to centre a block that has a fixed width. I measured it: a `w-64` box (256px) with `mx-auto` inside a 900px page got a 322px margin on the left, which is exactly (900 − 256) ÷ 2. `mt-auto` pushes an element away from what is above it, which becomes useful once you meet flexbox.

> **W-80 IS 320PX WIDE, PADDING INCLUDED**
>
> Padding is added inside the width, not on top of it, because Preflight (chapter 1) sets `box-sizing: border-box` on everything. An element with `w-80 p-6` measures 320px across, and its content area is 320 − 24 − 24 = 272px. This is what you want almost every time, and one reason layouts in Tailwind rarely need arithmetic.

Gap, Space-Between and Dividers

To put space *between* the children of a flex or grid container, use `gap`. It belongs to the container, so you set it once:

```
gap-4     gap-x-2     gap-y-6
```

```
gap: calc(var(--spacing) * 4);
column-gap: calc(var(--spacing) * 2);
row-gap: calc(var(--spacing) * 6);
```

`gap-x-*` is the space between columns and `gap-y-*` the space between rows. Gap also adds space between the rows when items wrap onto several lines, which margins on the children cannot do cleanly. There is an older way, still common in Tailwind code: `space-x-4` and `space-y-2`. They work on any container, including a plain block, and this is what they generate:

```
space-y-2
```

```
:where(.space-y-2 > :not(:last-child)) {
  --tw-space-y-reverse: 0;
  margin-block-start: calc(calc(var(--spacing) * 2) * var(--tw-space-y-reverse));
  margin-block-end: calc(calc(var(--spacing) * 2) * calc(1 - var(--tw-space-y-reverse)));
}
```

So `space-y-2` puts a bottom margin on every child except the last, rather than a gap on the parent. Use `gap` when the container is flex or grid (chapter 4), and `space-y-*` for a simple stack of blocks. The `divide-y` and `divide-x` classes work the same way with borders: a 1px border between children, on every child except the last.

Width and Height

Widths and heights use the same scale, plus fractions, keywords and a few viewport units. The measured results come from a 600px-wide parent:

CLASS	GENERATED CSS	MEASURED
w-64	`width: calc(var(--spacing) * 64)`	256px
w-1/2, w-1/3	`width: calc(1 / 2 * 100%)`, `calc(1 / 3 * 100%)`	300px and 200px

CLASS	GENERATED CSS	MEASURED
w-full, h-full	`width: 100%`, `height: 100%`	of the parent
size-12	`width` and `height` both `calc(var(--spacing) * 12)`	48px × 48px
w-screen, h-screen	`100vw`, `100vh`	the whole viewport
h-dvh, h-svh, min-h-dvh	`100dvh`, `100svh`, `min-height: 100dvh`	see below
w-fit, w-min, w-max	`fit-content`, `min-content`, `max-content`	sized by the content
max-w-md	`max-width: var(--container-md)` (`28rem`)	448px
max-w-prose	`max-width: 65ch`	561px here (varies with the font)
min-w-0	`min-width: 0px`	lets a flex child shrink
w-px, h-px	`1px`	hairlines

A few points. `h-screen` is `100vh`, which on phones is the height with the browser toolbar *hidden*, so a "full-height" section can run behind the toolbar. The newer units fix that: `h-svh` is the small viewport (toolbars showing), `h-lvh` the large one, and `h-dvh` the dynamic one that follows the toolbar as it moves. For a full-height page, `min-h-dvh` is usually what you want. The named maximum widths come from the container scale (`max-w-sm` is 24rem, `max-w-xl` 36rem, `max-w-3xl` 48rem, `max-w-7xl` 80rem), which is the reusable way to cap a column. And `max-w-prose` caps a column at about 65 characters, a comfortable line length for reading: `ch` is the width of the digit 0 in the current font, so the pixel width depends on your font.

Font Size and Line Height

The type scale is a set of named sizes, each with a matching line height. Measured, with each class applied to a paragraph:

CLASS	FONT SIZE	LINE HEIGHT	CLASS	FONT SIZE	LINE HEIGHT
text-xs	12px	16px	text-4xl	36px	40px
text-sm	14px	20px	text-5xl	48px	48px

CLASS	FONT SIZE	LINE HEIGHT	CLASS	FONT SIZE	LINE HEIGHT
text-base	16px	24px	text-6xl	60px	60px
text-lg	18px	28px	text-7xl	72px	72px
text-xl	20px	28px	text-8xl	96px	96px
text-2xl	24px	32px	text-9xl	128px	128px
text-3xl	30px	36px			

Each class sets two properties, and the line height is set so it can be overridden:

```
text-lg
```

```
font-size: var(--text-lg);
line-height: var(--tw-leading, var(--text-lg--line-height));
```

The `var(--tw-leading, …)` means "use the line height someone chose with a `leading-*` class, and fall back to the size's default". That is why `text-lg leading-tight` works in either order: the leading class always wins. There are two ways to change the line height:

- **A leading class**: `leading-none` (1), `leading-tight` (1.25), `leading-snug` (1.375), `leading-normal` (1.5), `leading-relaxed` (1.625), `leading-loose` (2). These are multiples of the font size: on a 16px paragraph I measured 16, 20, 22, 24, 26 and 32px, and `text-lg leading-tight` gave 22.5px (18 × 1.25). You can also give a number from the spacing scale, such as `leading-6`, which is a fixed `calc(var(--spacing) * 6)` (24px), or an arbitrary value, `leading-[1.3]`.
- **A slash after the size**: `text-lg/7` sets the size and the line height together. It compiled to `font-size: var(--text-lg)` and `line-height: calc(var(--spacing) * 7)`, and measured 18px with a 28px line. It is shorter than two classes and keeps them together.

For a size that is not on the scale, `text-[17px]` sets just `font-size: 17px` (no line height).

Weight, Family and Letter Spacing

PROPERTY	CLASSES	VALUES (MEASURED OR FROM THE GENERATED CSS)
Weight	`font-thin` … `font-black`	thin 100, extralight 200, light 300, normal 400, medium 500, semibold 600, bold 700, extrabold 800, black 900
Family	`font-sans`, `font-serif`, `font-mono`	system stacks: `--font-sans` starts with `-apple-system, BlinkMacSystemFont, "Segoe UI"`; `--font-serif` with `ui-serif, Georgia`; `--font-mono` with `ui-monospace`
Letter spacing	`tracking-tighter` … `tracking-widest`	−0.05em, −0.025em, 0, 0.025em, 0.05em, 0.1em (on 16px text: −0.8px, −0.4px, normal, 0.4px, 0.8px, 1.6px)

The weights only change anything if the font has those weights, so `font-thin` on a font with no thin face just shows the nearest one. The tracking classes are in `em`, so they scale with the text: a large heading gets a larger adjustment than small text for the same class. Large headings usually look better slightly tighter (`tracking-tight`) and small uppercase labels slightly looser (`tracking-wide` or `tracking-widest`). Chapter 9 shows how to add your own font families.

Text Layout: Alignment, Wrapping and Truncation

A group of small classes controls how text sits and breaks. Everything here is copied from the generated CSS:

CLASS	CSS
text-left, text-center, text-right, text-justify	`text-align: left | center | right | justify`
uppercase, lowercase, capitalize	`text-transform: …`
italic, not-italic	`font-style: italic | normal`
underline, no-underline	`text-decoration-line: underline | none`
text-balance	`text-wrap: balance` (lines of similar length; good for headings)
text-pretty	`text-wrap: pretty` (avoids a single word on the last line)
text-nowrap, whitespace-nowrap	`text-wrap: nowrap`, `white-space: nowrap`
break-words	`overflow-wrap: break-word` (breaks a long word, such as a URL, rather than overflowing)

CLASS	CSS
truncate	`overflow: hidden; text-overflow: ellipsis; white-space: nowrap`
line-clamp-3	`overflow: hidden; display: -webkit-box; -webkit-box-orient: vertical; -webkit-line-clamp: 3`

The Tailwind compatibility page lists `text-wrap: balance` among the newer features with limited browser support, so treat `text-balance` as an enhancement: in a browser that lacks it the heading still displays, just without balanced lines.

Truncating to one line is `truncate`, and it needs two things: a width that is smaller than the text, and (inside flexbox) a parent that is allowed to shrink. I measured the failure. A 192px-wide flex row holds a wrapper with a long title in a `truncate` paragraph:

```
<div class="flex w-48">
  <div class="flex-1">
    <p class="truncate">A very long product name that will not fit in the box</p>
  </div>
</div>
```

```
container: 192px   wrapper: 366px   paragraph: 366px   (no ellipsis; it overflows)
```

The wrapper refused to shrink below the width of its text, so the paragraph inside was never narrower than its text and there was nothing to truncate. Adding `min-w-0` to the wrapper (`class="min-w-0 flex-1"`) gave a 192px wrapper and a 192px paragraph, with the ellipsis showing. Putting `truncate` directly on the flex item also worked (192px). The rule to remember: a flex child's minimum width is its content, unless you say `min-w-0`, so any wrapper between the flex container and the truncating text needs it.

Clamping to several lines is `line-clamp-3`. A 192px paragraph of placeholder text was 240px high (ten lines of 24px); with `line-clamp-3` it was 72px, exactly three lines, with an ellipsis at the end of the third. `line-clamp-none` undoes it (for example at a larger breakpoint, once chapter 7 covers those).

Lists

Preflight removed the bullets and numbers (chapter 1, Exercise 3), so you add them back with a class:

CLASS	CSS
list-disc, list-decimal, list-none	`list-style-type: disc | decimal | none`
list-inside, list-outside	`list-style-position: inside | outside`

A typical list is `list-disc pl-5`: the bullets, and padding on the left so the bullets (which sit outside the text, `list-outside` being the default) are not cut off by the edge of the container. Use `space-y-1` or `gap` (on a flex column) for the space between items.

Putting It Together

Here is an article header built only from classes in this chapter. I measured every value in headless Chrome so you can see what each class contributed:

```
<header class="max-w-prose">
  <p class="text-xs font-semibold uppercase tracking-widest">Release notes</p>
  <h1 class="text-4xl font-bold tracking-tight leading-tight text-balance">Version 4.3 a
  <p class="text-lg/7">A short introduction that says why the article is worth reading.<
  <p class="text-base leading-relaxed">Body copy sits at the default size with a looser
</header>
```

```
eyebrow  12px / 16px   weight 600   letter-spacing 1.2px   uppercase
title    36px / 45px   weight 700   letter-spacing -0.9px  text-wrap: balance
lede     18px / 28px   weight 400
body     16px / 26px   weight 400
all four blocks 561px wide (max-w-prose, in this font)
```

Look at the title: `text-4xl` alone would give a 40px line height, but `leading-tight` replaced it with 1.25 × 36 = 45px, as the `--tw-leading` variable promised. A small uppercase label, a large tight heading, a slightly larger lede and a looser body: four levels of hierarchy from thirteen classes, all on the scale.

Hands-On Exercises

EXERCISE 1

A designer specifies a card: 320px wide, 24px of padding, a 32px gap below it, centred horizontally in its parent, with its three children stacked and 12px apart. Write the classes (start from `flex flex-col`, which chapter 4 explains), build the page, and confirm every measurement in the browser's computed values, including the width of the content area inside the padding.

View solution

EXERCISE 2

Style an article header to a spec: eyebrow 12px, uppercase, 600 weight, letter-spacing 0.1em; title 36px, bold, 1.25 line height, letter-spacing -0.025em, balanced lines; lede 18px on a 28px line; body 16px on a 26px line; every block no wider than 65 characters. Pick the classes, build, and check each value. Then find out which single class you could swap to make the lede's 28px line come from the spacing scale instead of from the type scale's default.

View solution

EXERCISE 3

In a 192px-wide row, show a product title that truncates with an ellipsis next to a price that never shrinks, and below it a description clamped to three lines. The title sits in a wrapper inside a flex row. Make it work, then remove the one class that makes the truncation fail and measure the damage.

View solution

CHAPTER 2 QUICK REFERENCE

- Spacing scale: the number is a multiple of `--spacing` (`0.25rem` = 4px): `p-4` = 16px; any number works (`p-13` , `p-3.5`); `p-px` = 1px; off-scale: `p-[13px]` , `p-(--var)`
- Sides: `p` / `m` all; `x` left+right; `y` top+bottom; `t r b l` physical sides; `s e` start/end of the line; negative margin `-mt-2` ; `mx-auto` centres a fixed-width block
- `w-80` includes padding (Preflight sets `border-box`)
- `gap-*` , `gap-x-*` , `gap-y-*` on flex and grid; `space-x/y-*` and `divide-x/y` put margins and borders on the children (all but the last)

- Sizes: `w-64`, `w-1/2`, `w-full`, `size-12`, `max-w-md` (`--container-md`), `max-w-prose` (65ch), `w-fit`; heights `h-svh`, `h-dvh`, `min-h-dvh` beat `h-screen` on phones
- Type scale `text-xs` (12/16) … `text-base` (16/24) … `text-9xl` (128/128); `text-lg/7` sets size and line height; `leading-tight` etc. override the line height in any order
- `font-thin` … `font-black` (100–900), `font-sans/serif/mono`, `tracking-tighter` … `tracking-widest` (em)
- `text-balance` (headings), `text-pretty` (paragraphs), `truncate` (one line, needs a width; a wrapper in a flex row needs `min-w-0`), `line-clamp-3`, `break-words`
- Preflight removes list markers: `list-disc pl-5` or `list-decimal pl-5`

COMING NEXT

Tailwind CSS Fundamentals 3 adds colour: the palette and its numbered shades, opacity modifiers, backgrounds and gradients, borders, rings and shadows.

Tailwind CSS Fundamentals

Chapter 3 · Colour, Backgrounds and Borders

Colour in Tailwind works like spacing: you pick from a ready-made palette with names and numbers, and the same names work on text, backgrounds, borders, rings and shadows. The edges of things (borders, rounded corners, rings, shadows) are a similar small system. This chapter covers both, including a measured look at colour contrast, which is the part people most often get wrong.

> **EVERY EXAMPLE HERE WAS RUN**
>
> CSS is copied from Tailwind CSS 4.3.3's output. Colour values, contrast ratios, computed styles and the two pixel readings were measured in headless Chrome. The contrast ratios use the WCAG 2 formula on the colours Chrome actually painted.

The Palette

The default theme has **26 colour families**, each with **11 shades** numbered 50, 100, 200, 300, 400, 500, 600, 700, 800, 900 and 950, plus `black` and `white`. I counted them in Tailwind's own `theme.css`: red, orange, amber, yellow, lime, green, emerald, teal, cyan, sky, blue, indigo, violet, purple, fuchsia, pink, rose; the greys slate, gray, zinc, neutral and stone; and mauve, olive, mist and taupe (those last four were added in v4.2). A low number is light and a high number is dark: `blue-50` is nearly white, `blue-950` nearly black, and `blue-500` is the middle.

Every colour is a theme variable, written in `oklch()`:

```
--color-blue-500: oklch(62.3% 0.214 259.815);
```

The three numbers are lightness, chroma (how vivid) and hue (the angle on the colour wheel). It is a good notation for building a palette because shades of one hue look evenly spaced. You never need to type it: you use the name, and Tailwind writes the variable only if you use the colour.

Putting Colour on Things

A colour class is a prefix, which says what to colour, followed by the family and shade. Each prefix maps to one CSS property:

PREFIX	COLOURS	CSS PROPERTY
text-blue-600	text	`color`
bg-blue-500	background	`background-color`
border-gray-200	border	`border-color`
divide-gray-200	the lines between children	`border-color` on every child but the last
ring-blue-500, outline-blue-500	rings and outlines	`--tw-ring-color`, `outline-color`
decoration-blue-500	underlines	`text-decoration-color`
accent-blue-500, caret-blue-500	checkboxes and sliders; the text cursor	`accent-color`, `caret-color`
fill-blue-500, stroke-blue-500	SVG shapes	`fill`, `stroke`
shadow-blue-500/50	the colour of a shadow	`--tw-shadow-color`

There are also a few special values that are not in the palette: `bg-transparent` (`transparent`), `bg-current` and `border-current` (`currentcolor`, meaning "the text colour"), and `text-inherit` (`inherit`). For a colour that is not in the palette use brackets, `bg-[#bada55]`, or a variable, `bg-(--brand)`, which compile to `background-color: #bada55` and `background-color: var(--brand)`. The colour of an input's placeholder text has two spellings that give the same result: `placeholder-gray-400` and the variant form `placeholder:text-gray-400` both compiled to a `::placeholder` rule setting `color`. The variant form is the one that generalises (chapter 6).

Opacity: the Slash

To make a colour partly transparent, add a slash and a percentage to any colour class: `bg-blue-500/50` is the blue at 50%. It works with every prefix in the table above, and with arbitrary values (`bg-blue-500/[0.37]` gave 37%). Here is what Tailwind wrote for `bg-blue-500/50`:

```
bg-blue-500/50
```

```
.bg-blue-500\/50 {
  background-color: color-mix(in srgb, oklch(62.3% 0.214 259.815) 50%, transparent);
  @supports (color: color-mix(in lab, red, red)) {
    background-color: color-mix(in oklab, var(--color-blue-500) 50%, transparent);
  }
}
```

It uses `color-mix()`: a plain version first for older browsers, then, where the browser supports it, a version that mixes in the `oklab` colour space using the theme variable. Chrome painted it as `oklab(0.623 -0.0378 -0.2106 / 0.5)`, which on a white page is a lighter blue (pixel value 44, 128, 255 at half alpha).

BG-BLACK/40 IS NOT THE SAME AS OPACITY-40

`bg-black/40` makes only the *background colour* 40% black. `opacity-40` (`opacity: 40%`) fades the whole element and *everything inside it*, including text. I measured both on a white page, each with the same amber heading inside a 100%-wide box: with `bg-black/40` the box was grey (153, 153, 153) and the text was the pure amber (255, 210, 48). With `bg-black opacity-40` the box was the same grey, but the text had faded to (255, 236, 171), a pale cream. For an overlay behind text, use the slash on the overlay's background and keep the text outside the faded element.

Readable Colour: Contrast

A colour scheme is only useful if people can read it. The WCAG accessibility guidelines measure the contrast between text and background as a ratio from 1:1 to 21:1. Level AA asks for at least **4.5:1** for normal text and **3:1** for large text (24px, or about 19px bold); AAA asks for 7:1. I painted common pairs in Chrome, read the real colours back, and computed the ratio:

PAIR	RATIO	NORMAL TEXT (4.5:1)	LARGE TEXT (3:1)
text-gray-400 on white	2.60	fails	fails
text-gray-500 on white	4.84	passes	passes
text-gray-600 on white	7.56	passes	passes
text-gray-700 on white	10.30	passes	passes

PAIR	RATIO	NORMAL TEXT (4.5:1)	LARGE TEXT (3:1)
text-gray-900 on white	17.75	passes	passes
text-white on bg-blue-500	3.76	fails	passes
text-white on bg-blue-600	5.25	passes	passes
text-white on bg-blue-700	6.83	passes	passes
text-blue-500 on white	3.76	fails	passes
text-blue-600 on white	5.25	passes	passes

Two habits follow. The light-grey captions that look elegant (`text-gray-400` on white) are hard to read and fail the guideline; `gray-500` is the lightest shade of this grey that passes on white. And the popular white-on-`blue-500` button fails for ordinary text size; moving one step darker, to `bg-blue-600`, fixes it. In general, for light backgrounds use shades 600 or darker for text, and for a coloured button with white text use shade 600 or darker. (Always check your own pairs: a different family has different numbers.) Exercise 1 gives you a snippet to do the measurement.

Backgrounds and Gradients

Colour is one kind of background; images and gradients are others. The small utilities map directly to properties:

CLASS	CSS
bg-cover, bg-contain	`background-size: cover | contain`
bg-center, bg-no-repeat, bg-fixed	`background-position: center`, `background-repeat: no-repeat`, `background-attachment: fixed`
bg-[url('/img/hero.jpg')]	`background-image: url('/img/hero.jpg')`
bg-clip-text text-transparent	`background-clip: text` and `color: transparent`: lets a gradient show through the letters

A **gradient** is made from a direction class plus colour stops:

```
<div class="h-10 bg-linear-to-r from-cyan-500 via-blue-500 to-indigo-600"></div>
```

```
computed background-image:
linear-gradient(to right, oklch(0.715 0.143 215.221) 0%, oklch(0.623 0.214 259.815) 50%,
```

`bg-linear-to-r` sets the direction (`to right`; there are eight, such as `-to-br` for bottom-right), `from-*`, `via-*` and `to-*` set the colours, and the three stops sit at 0%, 50% and 100%. Move a stop with a percentage: `from-10%` and `to-90%` are `--tw-gradient-from-position: 10%` and `--tw-gradient-to-position: 90%`. The same family gives you `bg-linear-45` (an angle), `bg-radial` and `bg-conic`. In the generated CSS the colours are mixed in the `oklab` colour space (`to right in oklab`), which keeps the middle of a gradient from going grey; add `/oklch` (as in `bg-linear-to-r/oklch`) to ask for a different space. The older class name `bg-gradient-to-r` still compiles in v4, but `bg-linear-to-r` is the current name.

Borders

A border has a width, a style and a colour, and Tailwind sets each by class. The width classes:

CLASS	CSS
border	`border-style: var(--tw-border-style); border-width: 1px`
border-2, border-4, border-0	the same with `2px`, `4px`, `0px`
border-t, border-x-4	`border-top-width: 1px`; `border-inline-width: 4px` (sides: `t r b l x y s e`, as in the padding classes)
border-solid, border-dashed, border-dotted	`border-style: solid | dashed | dotted`

A BORDER WITH NO COLOUR CLASS TAKES THE TEXT COLOUR

In Tailwind v4 the default border colour is `currentcolor`, the colour of the element's text. I put `border` on an element with `text-red-600` and Chrome reported a border colour of `oklch(0.577 0.245 27.325)`, which is the red of the text. In Tailwind v3 the default was a light grey, so tutorials and code you copy from older projects often write `border` and expect grey. In v4 always add a colour: `border border-gray-200` gave `oklch(0.928 0.006 264.531)`, a light grey. (Tailwind CSS Intermediate/Advanced 8 covers moving v3 code to v4.)

Rounded corners come from the radius scale:

CLASS	RADIUS	CLASS	RADIUS
rounded-xs	0.125rem (2px)	rounded-xl	0.75rem (12px)
rounded-sm, rounded	0.25rem (4px)	rounded-2xl	1rem (16px)
rounded-md	0.375rem (6px)	rounded-3xl, rounded-4xl	1.5rem, 2rem
rounded-lg	0.5rem (8px)	rounded-full	`calc(infinity * 1px)` (a circle on a square, a pill on a rectangle)

A bare `rounded` is 0.25rem, the same as `rounded-sm` in v4. Round one side or corner with a suffix: `rounded-t-lg` (both top corners), `rounded-s-md` (both start-side corners, which flip in right-to-left text), or an exact value with `rounded-[10px]`. `rounded-none` removes rounding.

Rings and Outlines

A **ring** is a solid line around an element that takes up no space, which makes it ideal for focus and selected states, because the layout does not move when it appears. It is drawn with `box-shadow`, so `ring-2 ring-blue-500` is a 2px blue ring:

```
ring-2 ring-blue-500
```

```
--tw-ring-shadow: var(--tw-ring-inset,) 0 0 0 calc(2px + var(--tw-ring-offset-width)) va
--tw-ring-color: var(--color-blue-500);
```

Two things are different from older Tailwind, according to the official v4 upgrade guide, and I checked the generated CSS. A bare `ring` is now **1px** (v3's was 3px), and its default colour is `currentcolor` (v3's was blue-500), so a ring with no colour class matches the text. For the old width write `ring-3`, which compiled to a 3px ring. `ring-offset-2` leaves a 2px gap between the element and the ring. `inset-ring` draws the ring inside the element.

An **outline** is the browser's own outline: `outline` (1px), `outline-2`, `outline-blue-500`, `outline-offset-2`. Browsers show an outline on keyboard focus by default, and the usual Tailwind pattern for custom focus styles is to replace it with your own, and v4 has two classes for

"no outline". `outline-hidden` compiled to `outline-style: none` *plus* a rule that restores a transparent 2px outline in forced-colours mode (Windows High Contrast, `@media (forced-colors: active)`), so the focus indicator is not lost for people who use that mode. `outline-none` compiled to just `outline-style: none`. The upgrade guide says v3's `outline-none` behaved like today's `outline-hidden`, so older code that uses `outline-none` for focus styles should be `outline-hidden` now. Never remove the focus indicator without giving a visible replacement.

Shadows and Filters

Shadows come in a size scale. Each is made of one or two `box-shadow` layers at low opacity:

CLASS	GENERATED SHADOW LAYERS
shadow-2xs	`0 1px rgb(0 0 0 / 0.05)`
shadow-xs	`0 1px 2px 0 rgb(0 0 0 / 0.05)`
shadow-sm, shadow	`0 1px 3px 0` and `0 1px 2px -1px`, both `rgb(0 0 0 / 0.1)`
shadow-md	`0 4px 6px -1px` and `0 2px 4px -2px`
shadow-lg	`0 10px 15px -3px` and `0 4px 6px -4px`
shadow-xl	`0 20px 25px -5px` and `0 8px 10px -6px`
shadow-2xl	`0 25px 50px -12px rgb(0 0 0 / 0.25)`
shadow-none	`0 0 #0000`

Bigger offset and blur means the element looks higher above the page. The clever part is how they combine. Every shadow and ring class writes into its own variable, and `box-shadow` reads all five, so classes stack rather than replace each other. A card with `shadow-md ring-2 ring-blue-500` has the blue ring *and* the shadow, which Chrome showed as one list: four ring and offset layers (three empty, then the 2px blue ring) followed by `0 4px 6px -1px` and `0 2px 4px -2px`. Without the ring classes the same list has an empty slot where the ring was. Other shadow classes: `inset-shadow-sm` (a shadow inside), and a coloured shadow with `shadow-blue-500/50`.

A warning for code you copy from v3 projects: v4 renamed the scales so that every size has a name. The upgrade guide's table: v3 `shadow-sm` is now `shadow-xs` and v3 `shadow` is now

`shadow-sm`; the same shift applies to `rounded`, `blur`, `drop-shadow` and `backdrop-blur`. The bare names still work (I saw that `shadow` and `shadow-sm` generate the same CSS in 4.3.3), but copied `shadow-sm` classes will look slightly stronger than they did in v3.

Filters. `drop-shadow-md` is a different tool: it is a `filter` that traces the *shape* of what is drawn, so it follows a transparent PNG or SVG outline where `box-shadow` would draw a rectangle. `blur-sm` blurs the element (`blur(8px)`); `backdrop-blur-md` blurs what is *behind* the element (`blur(12px)`), which gives the "frosted glass" panel when combined with a partly transparent background such as `bg-white/50`. The blur sizes come from the theme: xs 4px, sm 8px, md 12px, lg 16px, xl 24px, 2xl 40px, 3xl 64px.

Hands-On Exercises

EXERCISE 1

A page has a "Subscribe" button (`bg-blue-500 text-white`) and a grey caption (`text-gray-400`) on white. Measure the contrast of both pairs in the browser, say which fail WCAG AA for normal text, and change the smallest possible thing in each to make them pass. Show the new ratios.

View solution

EXERCISE 2

Make a card with padding, a 1px light-grey border, 12px rounded corners and a medium shadow. First write it with plain `border` and no colour, and read the computed border colour; then fix it. Finally add a "selected" state with a 2px blue ring and show that the shadow is still there.

View solution

EXERCISE 3

Build a hero banner: a left-to-right gradient from cyan-500 through blue-500 to indigo-600, with a darkening overlay at 40% and white heading text above it. Print the computed gradient. Then rebuild the overlay the wrong way (`bg-black opacity-40` around the text) and prove from the pixels what happens to coloured text.

View solution

CHAPTER 3 QUICK REFERENCE

- 26 colour families × 11 shades (50–950) plus `black` and `white`; low number = light; colours are `oklch()` theme variables
- Prefixes: `text-` `bg-` `border-` `divide-` `ring-` `outline-` `decoration-` `accent-` `caret-` `fill-` `stroke-` `shadow-`; specials `transparent`, `current`, `inherit`; off-palette `bg-[#bada55]`, `bg-(--brand)`
- Placeholder colour: `placeholder-gray-400` or `placeholder:text-gray-400` (same `::placeholder` rule)
- Opacity: `bg-blue-500/50` changes only that colour (`color-mix`); `opacity-50` fades the element and everything in it
- Contrast (WCAG AA): 4.5:1 normal text, 3:1 large. On white: `gray-500` passes (4.84), `gray-400` fails (2.60); white on `blue-600` passes (5.25), on `blue-500` fails (3.76)
- Gradients: `bg-linear-to-r from-X via-Y to-Z`, `bg-linear-45`, `bg-radial`, `bg-conic`, stops `from-10%`; `bg-clip-text text-transparent` for gradient text
- Borders: `border` 1px, `border-2`, sides `border-t`, style `border-dashed`; **v4 border colour defaults to the text colour**: always add `border-gray-200` or similar
- Radius: `rounded-xs` 2px, `sm` / `rounded` 4px, `md` 6px, `lg` 8px, `xl` 12px, `2xl` 16px, `full` circle/pill; `rounded-t-lg`, `rounded-s-md`
- Rings are `box-shadow`, take no space: `ring-2 ring-blue-500`, `ring-offset-2`, `inset-ring`; bare `ring` is 1px and `currentcolor` in v4 (`ring-3` for the old width); outlines: `outline-2 outline-offset-2`; `outline-hidden` for custom focus (keeps a forced-colours fallback; v4's `outline-none` removes it)
- Shadows `shadow-2xs` … `shadow-2xl` stack with rings in one `box-shadow` (v3's `shadow-sm` is v4's `shadow-xs`); `drop-shadow-*` follows the shape; `backdrop-blur-md` + `bg-white/50` = frosted glass

COMING NEXT

Tailwind CSS Fundamentals 4 is layout: `display`, flexbox and grid, with the centring and sidebar recipes you will use constantly.

Tailwind CSS Fundamentals

Chapter 4 · Layout: Display, Flexbox and Grid

Layout is where Tailwind earns its keep. Flexbox and grid are CSS features, and the classes are thin names for their properties, so this chapter is as much about knowing what flexbox and grid do as about knowing the class names. For each family I give the class, the CSS it generates, and numbers measured in a real browser, because the clearest way to understand `justify-between` or `flex-auto` is to see where the boxes land. Positioning and overflow are chapter 5.

> **EVERY EXAMPLE HERE WAS RUN**
>
> CSS is copied from Tailwind CSS 4.3.3's output. Every position and width in the tables was measured in headless Chrome, using fixed-size boxes so the numbers can be checked by hand. Fractions such as 189.33 are real: the browser divides space in thirds without rounding.

Display

`display` decides how an element lays out its children and how it sits among its neighbours. The classes use the CSS names:

CLASS	CSS	WHAT IT DOES
block, inline, inline-block	`display: block | inline | inline-block`	the normal flow values
flex, inline-flex	`display: flex | inline-flex`	children become flex items (full-width box or inline box)
grid, inline-grid	`display: grid | inline-grid`	children become grid items
hidden	`display: none`	removed from the layout entirely
contents	`display: contents`	the element's own box disappears; its children behave as if they were its parent's children
flow-root	`display: flow-root`	a box that contains its floats and stops margins collapsing through it
invisible	`visibility: hidden`	not painted, but still takes up its space

The last two rows are easy to confuse, so I measured them. Three 40px-high blocks, the middle one hidden: with `hidden` the third block started at 40px (the middle one took no space); with `invisible` it started at 80px (the middle one was a 40px gap). Use `hidden` to take something out of the page and `invisible` to keep the layout stable while hiding it.

Flexbox: One Row or Column

Put `flex` on a container and its children are laid out along one line, the **main axis**; the other direction is the **cross axis**. Everything below is set on the container, except where noted.

- **Direction.** `flex-row` (the default, `flex-direction: row`), `flex-col` (`column`), `flex-row-reverse`, `flex-col-reverse`. The main axis follows the direction.
- **Wrapping.** `flex-wrap` lets items move onto a second line; `flex-nowrap` (the default) squeezes them into one; `flex-wrap-reverse` wraps upwards. Combine with `gap-4` (chapter 2).

Along the main axis, `justify-*` spreads the free space. I measured three 100px-wide items in a 600px container (so 300px free), and the left edge of each item:

CLASS	CSS	LEFT EDGES OF ITEMS 1, 2, 3
justify-start	`justify-content: flex-start`	0, 100, 200
justify-center	`center`	150, 250, 350
justify-end	`flex-end`	300, 400, 500
justify-between	`space-between`	0, 250, 500
justify-around	`space-around`	50, 250, 450
justify-evenly	`space-evenly`	75, 250, 425

`justify-between` puts the first item at the start and the last at the end with equal gaps between (150px each here). `justify-around` gives each item the same space on both sides, so the outer gaps are half the inner ones (50 and 100). `justify-evenly` makes all four gaps equal (75).

Across the cross axis, `items-*` aligns the items in the container; `self-*` overrides it for one item. I measured three items of different heights (20px, 40px and 60px) in a 96px-high row; each cell is the item's top offset:

CLASS	CSS	TOP OFFSETS (ITEM 1, 2, 3)
items-start	`align-items: flex-start`	0, 0, 0
items-center	`center`	38, 28, 18
items-end	`flex-end`	76, 56, 36
items-stretch	`stretch` (the default)	0, 0, 0, and items that have no height of their own grow to 96px
items-baseline	`baseline`	8, 8, 0 (the items' text baselines line up; the third item has bigger text)

Stretch is why flex children in a row end up the same height without you asking, as in a row of cards. An item with its own height (the 20px one) keeps it; the others stretched to 96px. There is also `content-*` (`align-content`) for the spacing of wrapped lines.

How Items Share Space: flex-1, flex-auto, grow, shrink

These classes go on the *items*. Three items in a 600px row with widths `w-20`, `w-32` and `w-48` (80, 128 and 192px, 400px in total, 200px free):

CLASS ON EACH ITEM	CSS	MEASURED WIDTHS	WHAT HAPPENED
flex-1	`flex: 1`	200, 200, 200	starts from 0 and shares everything equally, ignoring their widths
flex-auto	`flex: auto`	146.66, 194.67, 258.67	starts from each item's own size, then shares the 200px of free space equally (+66.67 each)
grow	`flex-grow: 1`	146.66, 194.67, 258.67	same as `flex-auto` here
flex-none	`flex: none`	80, 128, 192	fixed: no growing and no shrinking
flex-initial	`flex: 0 auto`	80, 128, 192	can shrink but never grows

So `flex-1` is for "equal columns" and for "take whatever is left" (a search box between a logo and a button); `flex-auto` or `grow` is for "let content decide first, then fill". `shrink-0` stops an item being squeezed, which you saw on the price in chapter 2, and `grow-0` and `shrink` are the

opposite switches. `basis-1/2` and `basis-64` set the starting size (`flex-basis`) as a fraction or a spacing step. And remember `min-w-0` from chapter 2 for flex items that contain text which should truncate.

`order-*` changes the visual order without moving the HTML: `order-1` is `order: 1`, `order-first` is `order: -9999`, `order-last` is `order: 9999`, and `order-none` is `0`. Keep the HTML in a sensible reading order first: screen readers and the Tab key follow the HTML, not the visual order.

Grid: Rows and Columns Together

Use grid when you care about both directions at once. `grid` makes the container, and `grid-cols-3` defines three equal columns:

```
<div class="grid grid-cols-3 gap-4 w-[600px]">
  <div>1</div>
  <div class="col-span-2">2</div>
  <div>3</div>
</div>
```

```
.grid-cols-3 {
  grid-template-columns: repeat(3, minmax(0, 1fr));
}
.col-span-2 {
  grid-column: span 2 / span 2;
}

measured widths of items 1, 2, 3:  189.33, 394.67, 189.33
```

The 600px is split as (600 − 2 × 16) ÷ 3 = 189.33px per column, with 16px gaps (`gap-4`). Item 2 spans two columns: 2 × 189.33 + 16 = 394.67px, and item 3 wraps to the next row. The related classes:

- `grid-cols-1` … `grid-cols-12` (`repeat(N, minmax(0, 1fr))`), `grid-cols-none`, and `grid-cols-subgrid` (`grid-template-columns: subgrid`, so a nested grid lines up with its parent's columns; I compiled it but did not test it in a browser).
- `grid-rows-3` (the same for rows), `auto-rows-fr` (every implicit row the same height), `auto-cols-min`.

- `col-span-2` (`span 2 / span 2`), `col-span-full` (`1 / -1`, from the first line to the last), `col-start-2`, `col-end-4`, and the same with `row-`.
- `grid-flow-col` (fill columns first), `grid-flow-row-dense` (`row dense`: back-fill gaps left by spans).

Your own column layout. When the columns are not equal, write the template in square brackets, using an underscore where CSS has a space: `grid-cols-[200px_1fr]` compiles to `grid-template-columns: 200px 1fr`. In a 600px container with `gap-4` I measured the two columns at 200px and 384px: a fixed sidebar and a main area that takes the rest. And the most useful arbitrary template needs no breakpoints at all:

```
<div class="grid grid-cols-[repeat(auto-fill,minmax(12rem,1fr))] gap-4">…</div>
```

```
in a 900px container: 4 columns, each 213px
(900 - 3 x 16) / 4 = 213
```

That reads "as many columns as fit, each at least 12rem (192px), sharing the space". At 900px, four columns fit (a fifth would need 5 × 192 + 4 × 16 = 1024px) and they stretch to 213px; narrower screens get fewer columns, wider ones more. It is the simplest responsive card grid.

> **1FR CAN BLOW UP: WHY GRID-COLS-2 USES MINMAX(0, 1FR)**
>
> The plain CSS value `1fr` means "a share of the free space, but never smaller than the content". With a long unbreakable string in one cell, that "never smaller" wins. I put a 120-character word in the first cell of a 600px grid: with `grid-cols-[1fr_1fr]` the columns measured **1238.44px and 9.41px**, so the page overflowed and the second column was crushed; with `grid-cols-2` they were **300px and 300px**. Tailwind's numbered columns are `minmax(0, 1fr)` precisely to stop this. If you write your own template, use `minmax(0,1fr)` rather than bare `1fr` for columns that may hold long content, and pair it with `break-words` (chapter 2).

Placing Items and Centring

The `place-*` classes set two alignments at once: `place-items-center` is `place-items: center` (align and justify the items), `place-content-center` is `place-content: center` (the whole grid within its container), and `place-self-center` centres one item. `justify-items-center` is the horizontal half alone. They are mostly used for the question everyone asks: how do I centre

something? I measured a 100 × 50px box in a 600 × 300px container with three recipes (offsets from the container's top-left corner):

RECIPE	CLASSES ON THE CONTAINER (OR THE BOX)	BOX OFFSET (LEFT, TOP)
Flexbox	`flex items-center justify-center`	250, 125
Grid	`grid place-items-center`	250, 125
Auto margins (horizontal only)	`mx-auto` on the box, which needs a width	250, 0

Both centre exactly: (600 − 100) ÷ 2 = 250 and (300 − 50) ÷ 2 = 125. Use `grid place-items-center` when there is a single child (it is two classes); use flex when there are several items to centre as a group, or when you need `flex-col`. A third way, absolute positioning with `inset-0 m-auto`, is in chapter 5.

Container and Aspect Ratio

The `container` class sets a width of 100% and then a `max-width` at each breakpoint:

```
container
```

```
.container {
  width: 100%;
  @media (width >= 40rem) { max-width: 40rem; }
  @media (width >= 48rem) { max-width: 48rem; }
  @media (width >= 64rem) { max-width: 64rem; }
  @media (width >= 80rem) { max-width: 80rem; }
  @media (width >= 96rem) { max-width: 96rem; }
}
```

Two things to see in that output. It does not centre itself or add side padding (the generated rule has no margin or padding), so a page wrapper is normally `container mx-auto px-4`. And the steps are the breakpoints from chapter 7. Many projects skip `container` and use a plain `mx-auto max-w-5xl px-4` (one fixed maximum width), which is simpler to reason about.

Aspect ratio keeps a box in proportion as its width changes. `aspect-video` uses the theme variable `--aspect-video`, which is `16 / 9` (Chrome reported `16 / 9`); `aspect-square` is `1 / 1`, and your own ratio goes in brackets (`aspect-[4/3]`) or in the shorter form `aspect-3/2`;

`aspect-auto` removes it. A 256px-wide box measured 256 × 144 with `aspect-video`, 256 × 256 with `aspect-square` and 256 × 192 with `aspect-[4/3]`. The height follows the width, so you set one of them.

Two Recipes Worth Memorising

A navigation bar: the logo at the left, the links next to it, and a button pushed to the far right. The trick is `ml-auto`, a margin on the left that absorbs all the free space:

```
<nav class="flex items-center gap-6 px-6 py-3 border-b border-gray-200">
  <a class="text-lg font-bold">Acme</a>
  <ul class="flex gap-4"><li>Home</li><li>Docs</li><li>Pricing</li></ul>
  <a class="ml-auto rounded-md bg-blue-600 px-3 py-1.5 text-white">Sign in</a>
</nav>
```

```
in a 900px bar:
logo starts 24px from the left (px-6)
links start 24px after the logo (gap-6)
the button ends 24px from the right edge (px-6)
logo, links and button all centred on the same line, 30px from the top (items-center)
```

A sticky footer: on a short page the footer should sit at the bottom of the screen, not float in the middle. Make the body a column that is at least one screen tall, and let the main area grow:

```
<body class="min-h-dvh flex flex-col">
  <header class="h-16">…</header>
  <main class="flex-1 p-4">Short page.</main>
  <footer class="h-12">Footer</footer>
</body>
```

```
viewport height 405px
with flex-1 on main:    main 293px high, footer bottom = 405px (the bottom of the screen
without flex-1:         main 56px high,  footer bottom = 168px (just under the content)
```

`min-h-dvh` makes the body at least as tall as the screen, `flex-col` stacks the three parts, and `flex-1` lets `main` take all the leftover height (293 = 405 − 64 − 48). Remove `flex-1` and the leftover space has nowhere to go, so the footer rides up under the content.

Hands-On Exercises

EXERCISE 1

Build the navigation bar above in a 900px-wide page, then prove with measurements that the logo, link group and button share a centre line, that the button's right edge is exactly as far from the bar's edge as the logo is from its left edge, and what happens to the button's position if you delete `ml-auto`.

View solution

EXERCISE 2

Build the sticky-footer page and show with measurements that the footer touches the bottom of the viewport. Then change the page so the footer stays at the bottom *and* the main area is vertically centred content: a heading and a paragraph in the middle of the leftover space.

View solution

EXERCISE 3

Make a gallery in a 900px container: a three-column grid with 16px gaps, 16:9 tiles, five tiles, the first spanning two columns. Report the size and position of every tile. Then change it to a grid that chooses its own number of columns (each at least 12rem wide) and report how many columns you get at 900px.

View solution

CHAPTER 4 QUICK REFERENCE

- Display: `block inline inline-block flex grid contents flow-root`; `hidden` (removed from layout) vs `invisible` (keeps its space)
- Flex container: `flex`, `flex-col`, `flex-wrap`, `gap-*`; main axis `justify-start|center|end|between|around|evenly`; cross axis `items-start|center|end|stretch|baseline`; one item `self-*`
- `justify-between` with 3×100px in 600px: 0, 250, 500; `justify-evenly`: 75, 250, 425
- Flex items: `flex-1` (equal shares from 0), `flex-auto`/`grow` (own size, then share the rest), `flex-none` (fixed), `shrink-0`, `basis-*`, `order-*`, `ml-auto` pushes an item to the end

- Grid: `grid grid-cols-3 gap-4` (columns are `minmax(0, 1fr)`); `col-span-2`, `col-span-full`, `col-start-*`; custom `grid-cols-[200px_1fr]` (underscore = space); responsive without breakpoints `grid-cols-[repeat(auto-fill,minmax(12rem,1fr))]`
- Plain `1fr` can overflow on long content (measured 1238px + 9px); `grid-cols-2` held 300px + 300px
- Centring: `flex items-center justify-center` or `grid place-items-center` (both 250, 125 in 600×300); `mx-auto` centres horizontally only
- `container` = `width: 100%` plus max-widths at 40, 48, 64, 80 and 96rem; add `mx-auto px-4`; `aspect-video` 16/9, `aspect-square`, `aspect-[4/3]`
- Sticky footer: `min-h-dvh flex flex-col` on the page and `flex-1` on `main`

COMING NEXT

Tailwind CSS Fundamentals 5 covers positioning and stacking: `relative`, `absolute`, `fixed` and `sticky`, `inset-*`, `z-*`, overflow and scrolling, and images.

Tailwind CSS Fundamentals

Chapter 5 · Positioning, Stacking and Overflow

Flexbox and grid arrange things in rows and columns. Positioning is for the exceptions: a badge on the corner of a card, a header that stays at the top while you scroll, a dialog that floats above the page. These effects look simple and fail in confusing ways (a sticky header that does not stick, a dialog hidden behind the navigation), so this chapter measures the failures as well as the successes, and says what causes each.

> **EVERY EXAMPLE HERE WAS RUN**
>
> CSS is copied from Tailwind CSS 4.3.3's output. Positions, scroll results and "which element is on top" answers were measured in headless Chrome (a viewport 869px wide, the page scrolled by script). Where I describe standard CSS behaviour that I did not measure, I say so.

The position Values and the Offsets

CLASS	CSS	THE ELEMENT…
static	`position: static`	sits in the normal flow (the default); offsets and z-index do nothing
relative	`position: relative`	stays in the flow, can be nudged, and becomes the reference box for absolute children
absolute	`position: absolute`	leaves the flow and is placed against its nearest positioned ancestor
fixed	`position: fixed`	leaves the flow and is placed against the viewport
sticky	`position: sticky`	flows normally until it would scroll past its offset, then sticks

The offsets say how far from each edge. They use the spacing scale from chapter 2: `top-0` (`top: 0px`), `right-4` (`calc(var(--spacing) * 4)`), fractions `left-1/2` (`calc(1 / 2 * 100%)`), `top-full` (`100%`), a negative `-top-2`, and brackets for anything else. `inset-0` is all four sides at once (`inset: 0px`); `inset-x-0` is `inset-inline: 0px` and `inset-y-0` is `inset-block: 0px` ; `inset-1/2` and `inset-px` work too. For right-to-left languages there are logical versions: `start-0` is `inset-inline-start` and `end-4` is `inset-inline-end`.

Relative: Nudged, but Its Space Is Kept

`relative top-4 left-4` moves an element 16px down and right from where it would have been, but the layout still treats it as being in its original place. Three 40px-high boxes, A, B and C, with B nudged:

```
<div class="h-10 w-20">A</div>
<div class="relative top-4 left-4 h-10 w-20">B</div>
<div class="h-10 w-20">C</div>
```

```
A at (0, 0)    B at (16, 56)    C at (0, 80)
```

B's normal position would be (0, 40); it was drawn 16px further down and right at (16, 56). C is at 80, exactly where it would be if B had not moved: B's slot is still reserved. In practice you rarely nudge things with `relative`. Its main job is the next one.

Absolute: Placed Against a Reference Box

An absolutely positioned element is taken out of the flow, so its neighbours close up around it, and its offsets are measured from its **nearest positioned ancestor**: the closest ancestor whose `position` is anything but `static`. If there is none, it falls back to the page itself. I put the same badge (`absolute top-0 right-0 size-8`) in two parents of the same size:

```
<div class="h-24 w-72">                <!-- not positioned -->
  <div class="absolute top-0 right-0 size-8"></div>
</div>
<div class="relative h-24 w-72">         <!-- the reference box -->
  <div class="absolute top-0 right-0 size-8"></div>
</div>
```

```
parent has no position:  badge right edge 869, top 0     (parent: right 328, top 220)
parent is relative:      badge right edge 328, top 336   (parent: right 328, top 336)
```

With no positioned ancestor the badge jumped to the top-right corner of the page (869px is the width of the viewport here), nowhere near its parent. With `relative` on the parent it sat exactly in the parent's top-right corner. That is the pattern to memorise: **put `relative` (with no**

offsets) on the container, `absolute` **plus offsets on the child.** It is the recipe for badges, close buttons, captions over images, and the "stretched link" that makes a whole card clickable: an empty `<a class="absolute inset-0">` inside a `relative` card covers the card exactly (I measured a link of 300 × 150px in a 300 × 150px card).

Centring with absolute. Chapter 4 promised a third centring recipe. There are two, and both put a 100 × 50px box at offset (250, 125) in a 600 × 300px parent, the exact centre:

```
<div class="relative h-[300px] w-[600px]">
  <div class="absolute inset-0 m-auto h-[50px] w-[100px]"></div>
</div>

<div class="relative h-[300px] w-[600px]">
  <div class="absolute left-1/2 top-1/2 h-[50px] w-[100px] -translate-x-1/2 -translate-y
</div>
```

```
inset-0 m-auto:             (250, 125)
left-1/2 top-1/2 -translate: (250, 125)
```

The first works because an absolute box with all four offsets at 0, a fixed size and `margin: auto` splits the leftover space equally on every side. The second moves the box's top-left corner to the centre and then pulls it back by half its own size with `translate` (`-translate-x-1/2` compiles to `--tw-translate-x: calc(calc(1 / 2 * 100%) * -1)` and `translate: var(--tw-translate-x) var(--tw-translate-y)`), which also works when the box's size is unknown. Transforms are chapter 11.

Fixed and Sticky

Fixed elements ignore scrolling. A `fixed top-2 right-2` square had its right edge at 861px and its top at 8px; after the page was scrolled down by 300px it was still at 861 and 8, while an ordinary element on the same page had moved up by exactly 300px. Use it for things that must stay on screen: a cookie bar, a "back to top" button, a modal's backdrop. (By standard CSS rules a fixed element is relative to the viewport unless an ancestor has a `transform`, `filter` or similar, which makes that ancestor the reference instead; I did not measure that.)

Sticky is the more useful one for headers and table headings: the element flows normally, then sticks at its offset once scrolled that far. `sticky top-0` inside a 600px-high container, measured

as the page scrolled:

SITUATION	HEADER'S TOP EDGE
Scrolled 200px into its container	0 (stuck to the top of the viewport)
Scrolled until 20px of the container is left	-20 (it leaves with its container: sticky never escapes its parent)
Same, but the container has `overflow-hidden`	**-200: it did not stick at all**
Same, but the container has `overflow-clip`	0 (stuck)

STICKY STOPS WORKING INSIDE OVERFLOW-HIDDEN

A sticky element sticks relative to its nearest ancestor that can scroll. Any `overflow` value other than `visible` (including `hidden`, `auto` and `scroll`) makes the element a scroll container, so the header tries to stick inside that box, which does not scroll, and seems to do nothing. This is the usual reason "sticky does not work": an `overflow-hidden` on a wrapper added to clip something else. Remove it, or use `overflow-clip`, which clips in the same way but does not create a scroll container (the measured result above: 0 instead of -200). The difference is visible from script too: setting `scrollTop = 50` on a 96px box with 240px of content took effect with `overflow-hidden` (it stayed at 50) and did nothing with `overflow-clip` (still 0).

Stacking: z-index and Stacking Contexts

When positioned elements overlap, `z-*` decides which is on top: `z-0`, `z-10`, `z-20` ... `z-50` (`z-index: 10` and so on), `z-auto`, a negative `-z-10` (`calc(10 * -1)`) and anything else in brackets, `z-[100]`. It only has an effect on positioned elements (anything but `static`) and on flex and grid items. Two behaviours cause nearly all the puzzles.

1. Later positioned elements paint over earlier ones, unless z-index says otherwise. A sticky header with no z-index was covered by a `relative` card that scrolled up under it: at the header's position I asked the browser which element was on top and it answered the card. Adding `z-10` to the header made the header the answer. So: **give a sticky or fixed header a `z-10`** (or higher) or content with `relative` will slide over it.

2. z-index only competes inside a stacking context. Some elements start a new *stacking context*: everything inside is stacked as a group, and the group takes one place in its parent's

stacking order. `isolate` (`isolation: isolate`) does exactly this, and so do a positioned element with a `z-index` other than auto, anything with a `transform` or `opacity` below 1, and a few others. I made a red box `z-10` overlap a sibling that held a blue child with `z-50`, and asked which was on top where they overlapped:

BLUE CHILD'S PARENT	ON TOP WHERE THEY OVERLAP
plain positioned box (no stacking context)	the blue child (z-50 beats z-10)
the same box with `isolate`	the **red** box: the child's z-50 only counts inside its parent

A high z-index cannot lift an element out of its parent's stacking context. If something stays hidden behind another thing however big its z-index, look at its ancestors for `isolate`, `transform`, `opacity`, a `z-*` on a positioned parent, and so on. Used deliberately, `isolate` is useful: it stops a component's internal z-indexes from leaking out and fighting the rest of the page.

Overflow, Scrolling and Snapping

CLASS	CSS	RESULT
overflow-visible	`overflow: visible`	content spills out of the box (the default)
overflow-hidden	`overflow: hidden`	clipped; still scrollable by script; creates a scroll container
overflow-clip	`overflow: clip`	clipped; no scrolling at all; no scroll container
overflow-auto	`overflow: auto`	scrollbars only when the content is bigger
overflow-scroll	`overflow: scroll`	scrollbars always
overflow-x-auto, overflow-y-hidden	`overflow-x: auto`, `overflow-y: hidden`	one axis at a time

Measured on a 96px-high box with 240px of content: `overflow-hidden` reported `scrollHeight` 240 and `clientHeight` 96 (the content is there, just clipped). `overflow-auto` took 15px of width for a scrollbar (`offsetWidth` 192, `clientWidth` 177) when the content was taller, and none (192 and 192) when it was short; the scrollbar's width depends on the browser and

operating system, and this was Chrome on Windows. The common use of `overflow-x-auto` is a wide table or code block inside a narrow column, so the box scrolls rather than the whole page. Also useful: `overscroll-contain` (`overscroll-behavior: contain`) stops a scrolled-to-the-end list from scrolling the page behind it, and in 4.3 `scrollbar-thin`, `scrollbar-none` and `scrollbar-auto` set `scrollbar-width`.

Scroll snapping makes a scrolling row (a carousel) settle neatly on items. The container gets `snap-x snap-mandatory` (`scroll-snap-type: x var(--tw-scroll-snap-strictness)` and `mandatory`) and each item gets `snap-start` (`scroll-snap-align: start`; there are also `snap-center` and `snap-end`):

```
<div class="flex w-[400px] overflow-x-auto snap-x snap-mandatory">
  <div class="w-[300px] shrink-0 snap-start">1</div>
  <div class="w-[300px] shrink-0 snap-start">2</div>
  <div class="w-[300px] shrink-0 snap-start">3</div>
  <div class="w-[300px] shrink-0 snap-start">4</div>
</div>
```

```
scrolling to 120px:  with snap-mandatory 0     without snap 120
scrolling to 200px:  with snap-mandatory 300   without snap 200
```

I scrolled the row to 120px and to 200px by script: the snapping row ended at the start of the nearest item (0 and 300), the plain one stayed where it was. `snap-proximity` is the gentler version that only snaps when you are close. Two related classes: `scroll-smooth` animates scrolling to anchors (`scroll-behavior: smooth`), and `scroll-mt-16` (`scroll-margin-top`) leaves room above a link target so a sticky header does not cover it; `scroll-p-4` sets `scroll-padding` on the scrolling container.

Images, Hidden Text and Click-Through

Fitting an image to a box. The `object-*` classes set `object-fit`: `object-cover` fills the box and crops the overflow, `object-contain` shows the whole image with empty space, `object-fill` stretches it, `object-none` keeps its size, and `object-scale-down` picks the smaller of none and contain. `object-center`, `object-top` and `object-bottom` choose which part of a cropped image stays visible. The usual card image is `aspect-video w-full object-cover` (chapter 4 for the ratio). I confirmed the CSS each class generates (for example `object-fit: cover`) but did not measure how an image crops.

Text for screen readers only. `sr-only` hides something visually but leaves it available to assistive technology: an icon-only button needs a name. Its generated CSS is a 1px box that is clipped away:

```
<button><svg …></svg><span class="sr-only">Close dialog</span></button>
```

```
.sr-only {
  position: absolute; width: 1px; height: 1px; padding: 0; margin: -1px;
  overflow: hidden; clip-path: inset(50%); white-space: nowrap; border-width: 0;
}
measured: 1 x 1px, position absolute, clip-path inset(50%)
.not-sr-only measured: position static, clip-path none, normal size
```

`not-sr-only` undoes it, which is how a "skip to content" link (hidden until it receives keyboard focus) is built: `sr-only focus:not-sr-only` (variants are chapter 6).

Clicks passing through. A decorative overlay on top of a button swallows the clicks. I put a translucent overlay over a button and asked which element is at the button's corner: without `pointer-events-none` the answer was the overlay; with it, the button. Put `pointer-events-none` on anything purely decorative that sits over something clickable (`pointer-events: none`), and `pointer-events-auto` on a child that must be clickable again.

A Modal Overlay, and Why It Gets Trapped

The standard modal is a full-screen backdrop with the dialog centred on it:

```
<div class="fixed inset-0 z-50 grid place-items-center bg-black/50">
  <div class="h-32 w-64 bg-white">dialog</div>
</div>
```

```
viewport 869 x 605
overlay:  left 0, top 0, 869 x 605     (exactly the viewport)
dialog centre (434.5, 302.5)           (exactly the viewport's centre)
```

`fixed inset-0` stretches the overlay over the viewport, `z-50` puts it above ordinary content, and `grid place-items-center` (chapter 4) centres the dialog. Now the trap. A fixed header with

`z-40` sits at the top of the page, and the modal's markup lives inside a container with `relative z-0` (a stacking context). I asked which element was on top at the header's position:

WHERE THE OVERLAY'S MARKUP IS	ON TOP AT THE HEADER
inside a `relative z-0` parent	the **header** (the overlay's z-50 only counts inside the parent, which has level 0)
same, after removing `z-0` from the parent	the overlay (the parent is no longer a stacking context)
moved to be a direct child of `<body>`	the overlay

That is why component libraries render dialogs and menus at the very end of `<body>` (often called a portal): out there, nothing can trap them under a parent's stacking context. If you must keep the dialog in place, make sure no ancestor starts a stacking context with a lower level than the things you want to cover.

Hands-On Exercises

EXERCISE 1

Build a 300 × 150px card with a small red badge in its top-right corner (8px from each edge) and a link that stretches over the whole card. Measure the badge's distance from the card's corner. Then remove the one class that makes the badge belong to the card, and say where the badge goes.

View solution

EXERCISE 2

A page wrapper has `overflow-hidden` and a sticky header that does not stick. Measure the header's top edge after scrolling 200px into the wrapper, fix it without losing the clipping, and measure again. Then show that a `relative` card scrolling under the sticky header covers it, and fix that with one class.

View solution

EXERCISE 3

Build a modal overlay (`fixed inset-0 z-50` , centred dialog) inside a parent that has `relative z-0` , with a `z-40` fixed header outside it. Show that the header is on top of the overlay, then fix it two different ways, proving each with the element found at the top of the screen. Confirm the overlay fills the viewport and the dialog is centred.

View solution

CHAPTER 5 QUICK REFERENCE

- `relative` keeps its slot and is the reference box; `absolute` leaves the flow and is placed against the nearest non-static ancestor (none: the page); `fixed` against the viewport; `sticky` sticks at its offset but never leaves its parent
- Offsets: `top-0 right-4 left-1/2 -top-2` , `inset-0` , `inset-x-0` , `start-0 end-4`
- Pattern: `relative` parent, `absolute top-2 right-2` child; stretched link `absolute inset-0` ; centring `absolute inset-0 m-auto` (needs a size) or `left-1/2 top-1/2 -translate-x-1/2 -translate-y-1/2`
- Sticky fails inside `overflow-hidden/auto/scroll` wrappers: remove the overflow or use `overflow-clip`
- `z-*` needs a positioned element (or a flex/grid item); give sticky and fixed headers `z-10` +; `isolate` (also `transform` , `opacity` <1, a z-indexed positioned parent) makes a stacking context that traps children's z-index
- Overflow: `hidden` (clips, still scrollable by script), `clip` (clips, no scroll container), `auto` (scrollbar only if needed), `scroll` ; `overflow-x-auto` for wide tables; `overscroll-contain` ; `scrollbar-thin/none` (4.3)
- Snap: container `snap-x snap-mandatory` , items `snap-start shrink-0` ; `scroll-smooth` , `scroll-mt-16`
- `object-cover/contain/fill/none/scale-down` , `object-center/top` ; `sr-only` / `not-sr-only` ; `pointer-events-none` on decorative overlays
- Modal: `fixed inset-0 z-50 grid place-items-center bg-black/50` ; render it at the end of `<body>` so no parent's stacking context traps it

COMING NEXT

Tailwind CSS Fundamentals 6 is state variants: `hover:` , `focus-visible:` , `group` and `peer` , `aria-*` and `data-*` , and the structural variants such as `first:` , `has-*` and `not-*` .

Tailwind CSS Fundamentals

Chapter 6 · State Variants: Hover, Focus and Friends

So far every class has applied all the time. A **variant** is a prefix that makes a class apply only in certain circumstances: `hover:bg-blue-700` is "the blue-700 background, but only while the pointer is over it". Variants are how Tailwind does everything that plain CSS does with `:hover`, `:focus`, `:checked`, `:nth-child()`, `::before` and attribute selectors, without leaving your HTML. This chapter covers the state-based ones. Responsive and dark-mode variants, which work the same way, are chapters 7 and 8.

EVERY EXAMPLE HERE WAS RUN

CSS is copied from Tailwind CSS 4.3.3's output. For this chapter I drove headless Chrome with real mouse and keyboard events (through the Chrome DevTools Protocol), so "hover", "pressed", "Tab" and "click" below are real input, and the colours and opacities were read back from the browser. Colours are shown as shade names; I checked the underlying values.

How a Variant Works

A variant puts a colon between a condition and a utility. The utility is unchanged; Tailwind wraps the same declaration in a selector for the condition:

```
<button class="bg-blue-500 hover:bg-blue-700 active:bg-blue-900 px-4 py-2 text-white">Bu
```

```
@media (hover: hover) {
  .hover\:bg-blue-700:hover {
    background-color: var(--color-blue-700);
  }
}
.active\:bg-blue-900:active {
  background-color: var(--color-blue-900);
}

measured with real input:
at rest      blue-500
```

```
  pointer over blue-700
  mouse down   blue-900
```

The class name itself is escaped in the CSS (`hover\:bg-blue-700`), and the rule only matches in the state. Note the `@media (hover: hover)` around the hover rule: in v4 `hover:` applies only on devices whose main input can really hover. On a touch screen it is skipped, so a tap does not leave a button stuck in its hover style. (The headless Chrome I tested with reported `matchMedia('(hover: hover)').matches` as `true`.) The upgrade guide says that if you built a site that depended on hover firing on tap, you can restore the old behaviour with `@custom-variant hover (&:hover);`, but it recommends treating hover as an enhancement and never relying on it.

The Interaction and Form States

Each of these is a pseudo-class with the same name, taken from the generated CSS:

VARIANT	SELECTOR IT GENERATES	WHEN IT APPLIES
hover:	`:hover` inside `@media (hover: hover)`	pointer over the element, on devices that can hover
active:	`:active`	while pressed
focus:	`:focus`	the element has focus, however it got it
focus-visible:	`:focus-visible`	focus the browser decides should be shown (below)
focus-within:	`:focus-within`	the element or anything inside it has focus
disabled: enabled:	`:disabled`, `:enabled`	form controls
checked:	`:checked`	checkboxes, radios, options
required: invalid: read-only:	`:required`, `:invalid`, `:read-only`	form validation and editability
placeholder-shown:	`:placeholder-shown`	an input still showing its placeholder
visited:	`:visited`	links already followed

VARIANT	SELECTOR IT GENERATES	WHEN IT APPLIES
open:	`:is([open], :popover-open, :open)`	a `<details>`, popover or dialog that is open
empty:	`:empty`	no children at all

I tested several with real state. `disabled:bg-gray-400` on a button with the `disabled` attribute gave the grey, and the enabled twin stayed blue. An email input holding `nope` with `invalid:border-red-500` had a red border, and after I changed its value to `a@b.co` the border went back to grey. A `<details open:bg-blue-100>` was grey until I clicked its summary, and then blue (`open` became `true`).

Focus: focus, focus-visible and focus-within

Focus is the one to get right, because it is how keyboard users see where they are. The two variants differ, and I measured the difference. A button with `focus:text-yellow-300 focus-visible:text-red-500`, and a text input with the same classes:

ACTION	`:FOCUS`	`:FOCUS-VISIBLE`	WHICH CLASS APPLIED
Click the button with the mouse	true	**false**	`focus:` (yellow)
Press Tab to reach the button (fresh page)	true	true	both; `focus-visible:` (red) came last, so it wins
Click into the text input	true	true	both (red)

A mouse click gives a button focus but not `:focus-visible`, because a person who just clicked it does not need an indicator; keyboard users do. So the pattern for buttons and links is to style `focus-visible:`: a ring or outline for keyboard users, nothing extra after a click. Text inputs always match `:focus-visible` (you are about to type), so either variant works on them. I measured a button with `focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outline-blue-600`: after a mouse click its outline style was `none`; after Tab it was `2px solid` with a 2px offset in blue. Never remove the outline without such a replacement (chapter 3).

`focus-within:` styles a *container* when something inside it has focus: a wrapper with `border-gray-300 focus-within:border-blue-500` around an input was grey, and turned blue as soon as I clicked into the input. It is how you give a fancy input (an icon, a prefix) a focus state on the outer box.

> **HOVER: STILL FIRES ON A DISABLED BUTTON**
>
> I put `hover:bg-blue-700` and `disabled:opacity-50` on a button with the `disabled` attribute and hovered it: the background still changed to blue-700 (while dimmed). A disabled button is still under the pointer. The fix is the `enabled:` variant: `enabled:hover:bg-blue-700` compiles to `:enabled:hover`; the disabled button stayed blue-600 when hovered, and an enabled button with the same classes still changed. Use it (and `enabled:active:`) on anything that can be disabled.

Group and Peer: Styling One Thing from Another's State

A plain variant looks at the element's own state. Often you want to style one element based on the state of *another*, and Tailwind has two markers for that.

`group` marks a parent; `group-hover:`, `group-focus:` and so on style its *descendants*:

```
<div class="group/card border p-2">
  <span class="opacity-20 group-hover/card:opacity-100">revealed by the card</span>
  <div class="group/row bg-gray-100 p-2">
    <span class="opacity-20 group-hover/row:opacity-100">revealed by the row</span>
    <span class="opacity-20 group-hover/card:opacity-100">also by the card</span>
  </div>
</div>
```

```
.group-hover\/card\:opacity-100:is(:where(.group\/card):hover *) { opacity: 100%; }   (i

opacities of the three spans:
at rest                        0.2   0.2   0.2
pointer over the card only     1     0.2   1
pointer over the row           1     1     1
```

The selector reads "this element, when it is inside a `.group` that is hovered". When groups nest, a bare `group` is ambiguous (the nearest one? all of them?), so you **name** them, `group/card` and `group/row`, and write `group-hover/card:`. In the measurement the card-hover span lit up

for the whole card, while the row-hover span lit up only when the pointer was inside the row (and, because the row is inside the card, hovering the row also counted as hovering the card).

`peer` marks an element so that its *later siblings* can style themselves from its state: `peer-checked:`, `peer-invalid:`, `peer-focus-visible:`. The generated selector uses the sibling combinator: `.peer-checked\:bg-red-500:is(:where(.peer):checked ~ *)`. The `~` is the catch: **the peer must come *before* the element that reacts to it**. I built both orders and clicked both labels:

```
<!-- works: the peer comes first -->
<input id="a" type="checkbox" class="peer">
<label for="a" class="bg-gray-200 peer-checked:bg-blue-600">after the peer</label>

<!-- does not work: the label comes before the peer -->
<label for="b" class="bg-gray-200 peer-checked:bg-blue-600">before the peer</label>
<input id="b" type="checkbox" class="peer">
```

```
after clicking both labels:
both checkboxes: checked = true
label after the peer:   blue-600  (peer-checked applied)
label before the peer:  gray-200  (nothing happened)
```

Both checkboxes were checked; only the label that came after its peer reacted. This is the whole trick behind CSS-only toggles, tabs and accordions (Exercise 2). The group version has the same limit in the other direction: `group-*` can only style descendants, not ancestors or earlier siblings.

aria-* and data-*: Styling from Attributes

Many components expose their state as attributes: `aria-expanded="true"` on a menu button, `data-state="open"` on a dialog (headless component libraries do this). Tailwind turns those attributes into variants, so no JavaScript has to add and remove classes:

```
<div class="bg-gray-200 aria-checked:bg-blue-600 data-active:bg-green-600 data-[state=op
```

```
.aria-checked\:bg-blue-600[aria-checked="true"]
.data-active\:bg-green-600[data-active]
```

```
.data-\[state\=open\]\:bg-red-600[data-state="open"]

element starts with none of the attributes:  gray-200
after aria-checked="true":                   blue-600
after adding data-active:                    green-600
after adding data-state="open":              red-600
```

The built-in `aria-` variants (I compiled `aria-checked` and `aria-expanded`; the official docs list more of the same kind) match the attribute set to `"true"`; for another value use brackets, as in `aria-[sort=ascending]:` (`[aria-sort="ascending"]`). `data-active:` matches the bare attribute being present, and `data-[state=open]:` matches a specific value. When several apply, the one that comes last in Tailwind's generated CSS wins, which is why the red one showed at the end.

Structural Variants: Position Among Siblings

VARIANT	SELECTOR
first: last: only:	`:first-child`, `:last-child`, `:only-child`
odd: even:	`:nth-child(odd)`, `:nth-child(even)`
first-of-type:	`:first-of-type`
nth-3:, nth-[3n+1]:	`:nth-child(3)`, `:nth-child(3n+1)`

Four list items with `odd:bg-gray-100 even:bg-white first:rounded-t-lg last:rounded-b-lg` measured: backgrounds gray-100, white, gray-100, white; the first item with an 8px radius at its top corners and the last with 8px at its bottom corners, and 0px elsewhere. This is zebra striping and rounded ends without a wrapper or extra classes.

Two variants style the *children* from the parent, so you do not repeat a class on every child: `*:` targets direct children (`:is(.\*\:p-2 > *)`) and `**:` targets all descendants. A `<ul class="*:p-2">` gave each of its two items 8px of padding. A trade-off: the parent's rule is a more specific selector than the child's own class. With `class="*:first:pt-0"` on the list and `pt-3` on each item, the first item measured `0px` and the second `12px`: the parent's variant overrode the child's own `pt-3` where it applied.

STACKED VARIANTS READ LEFT TO RIGHT IN V4

You can combine variants (`md:hover:bg-red-500`, `group-hover:md:underline`), and they apply in the order written, left to right. This matters mainly for the structural ones, where order changes the meaning. The two classes below look alike and compile to different selectors:

`*:first:pt-0` → `:is(.\*\:first\:pt-0 > *):first-child` (the first child of the list)

`first:*:pt-0` → `:is(.first\:\*\:pt-0:first-child > *)` (the children of a list that is itself a first child)

The upgrade guide notes that v3 read stacked variants right to left, so v3 code written as `first:*:pt-0` should be reversed to `*:first:pt-0` in v4. Combinations of hover with breakpoints, such as `md:hover:` and `hover:md:`, nest the same two conditions in either order and behave the same.

has-*, not-* and in-*

`has-*` styles an element based on what it *contains*, which was impossible in CSS before `:has()`: `has-checked:bg-blue-100` compiles to `:has(:checked)`, and any selector goes in brackets, `has-[:checked]:` or `has-[img]:`. I made a `<label class="has-checked:bg-blue-100">` around a checkbox: it had no background, and after I clicked the checkbox the whole label became blue-100. The same works from a group: `group-has-[a]:block` (compiled to `:is(:where(.group):has(:is(a)) *)`).

`not-*` is the negation: `not-first:mt-2` is `:not(:first-child)`, the classic "space between items but not before the first". For states it composes: `not-hover:opacity-50` generated two rules, `:not(:hover)` and `@media not (hover: hover)`, so the dimmed style applies on touch screens too. Measured: 0.5 at rest, 1 while hovered.

`in-*` is like `group-*` without the `group` marker: `in-hover:opacity-100` compiles to `:where(:hover) .in-hover\:opacity-100`, "inside *any* hovered element". That "any" bites. Every page's `<html>` element is `:hover` while the pointer is anywhere over the page, so I measured an element with `opacity-50 in-hover:opacity-100`: 0.5 with the pointer outside the window, **1 with the pointer over an unrelated button** (`html:hover` was true). For "this parent is hovered", use a named `group`. Treat `in-hover:` with suspicion; I did not test the other `in-*` variants, so check any you use against the same trap (does some ancestor, up to `<html>`, always match?).

Pseudo-elements

Variants also reach into the parts of an element that are not elements: `before:` and `after:` (`::before`, `::after`), `placeholder:` (`::placeholder`), `file:` (`::file-selector-button`, the "Choose file" button), `marker:` (list bullets and numbers, applied to the element and its children), `selection:` (selected text), `first-line:`, `first-letter:` and `backdrop:` (a dialog's backdrop). For `::before` and `::after` you must give the pseudo-element some content, with the `content-*` class:

```
<span class="after:content-['*'] after:text-red-500">label</span>
<span class="after:block after:size-4">no content utility</span>
```

```
.before\:content-\[\'\'\]::before { --tw-content: ''; content: var(--tw-content); }

computed content of ::after:
with after:content-['*']        "*"
with only after:block (no content class)   ""   and display: block
```

The second line is a v4 convenience: the `before:` and `after:` variants add `content: var(--tw-content)` themselves and `--tw-content` defaults to an empty string, so `after:block after:size-4` already creates a box (the measured content was `""`). You only write `content-['…']` when you want actual text. A `placeholder:text-red-500` input measured the red on its placeholder colour.

The ! Modifier and Variant Order

To make one utility win over everything, put `!` at the *end* of the class: `bg-red-500!` compiles to `background-color: var(--color-red-500) !important`, and with a variant, `hover:bg-red-600/50!`. In v3 the `!` went at the start; the upgrade guide says the old position still works but is deprecated. Reach for it rarely: needing it usually means two rules are fighting that should not be.

Hands-On Exercises

EXERCISE 1

Style a primary button for six states: resting (blue-600), hover (blue-700), pressed (blue-800), a visible keyboard focus ring (2px outline, 2px offset), and disabled (half opacity, `not-allowed` cursor). Using

real mouse and keyboard input, show which state appears in each case, and prove that a mouse click does not show the focus outline but Tab does. Then show that a disabled button still changes colour on hover, and fix it.

View solution

EXERCISE 2

Build a toggle switch with no JavaScript: a visually hidden checkbox, a rounded track that turns blue when checked, and a white knob that slides right by 20px. Show the state before and after a click (colour, knob position), that Tab and Space also work, and that the focus ring shows for the keyboard but not after a mouse click.

View solution

EXERCISE 3

Build a three-row table with zebra striping, a row highlight on hover, a row that turns blue when its checkbox is ticked, and an "Edit" button in each row that appears only when the row is hovered. Measure each state. Then check what a keyboard user sees: Tab to the second row's button and report its opacity, and make sure it is visible.

View solution

CHAPTER 6 QUICK REFERENCE

- A variant is `condition:utility`; the same declaration is wrapped in a selector. `hover:` is inside `@media (hover: hover)` (no hover on touch); treat hover as an enhancement
- States: `hover active focus focus-visible focus-within disabled enabled checked required invalid read-only placeholder-shown visited open empty`
- Keyboard focus: use `focus-visible:` for buttons and links (a mouse click gives `:focus` but not `:focus-visible`); `focus-within:` on a wrapper for inputs
- `disabled` controls still match `hover`: use `enabled:hover:`
- `group` on a parent, `group-hover:` on descendants; name nested groups `group/card` and `group-hover/card:`. `peer` on an element, `peer-checked:` on LATER siblings only
- `aria-checked:` / `aria-expanded:` (= `"true"`), `aria-[sort=ascending]:`, `data-active:` (attribute present), `data-[state=open]:` (specific value)

- `first: last: only: odd: even: nth-3: nth-[3n+1]:` ; `*:` direct children and `**:` all descendants (the parent's rule beats the child's own class)
- Stacked variants read left to right in v4: `*:first:pt-0` is not `first:*:pt-0`
- `has-checked:` , `has-[img]:` (what it contains); `not-first:` , `not-hover:` ; `in-hover:` matches ANY hovered ancestor, including `<html>` : use `group` instead
- Pseudo-elements: `before: after: placeholder: file: marker: selection: first-line: first-letter: backdrop:` ; `content-['x']` only needed for text (empty content is the default)
- `!` goes at the end: `bg-red-500!`

COMING NEXT

Tailwind CSS Fundamentals 7 uses the same variant mechanism for screen size: mobile-first breakpoints, `max-*` variants, container queries (`@container`), and designing a layout upwards from the smallest screen.

Tailwind CSS Fundamentals

Chapter 7 · Responsive Design and Container Queries

A page that works at one width is a page that works on one device. Responsive design means the layout adapts: one column on a phone, two on a tablet, four on a desktop. In Tailwind this is the same variant mechanism as chapter 6, with a screen size as the condition: `md:grid-cols-2` is "two columns, from the `md` width upwards". There is one rule to learn (it is mobile-first) and one newer tool to add (container queries, which respond to the size of a parent rather than the screen).

> **EVERY EXAMPLE HERE WAS RUN**
>
> CSS is copied from Tailwind CSS 4.3.3's output. The breakpoint results come from resizing the viewport of headless Chrome to the exact widths named (using Chrome's device emulation, driven through the DevTools Protocol) and reading back computed styles. Widths are CSS pixels.

Mobile-First: The One Rule

Tailwind's breakpoints are five named widths, defined in the theme and compiled to `min-width` media queries:

PREFIX	THEME VALUE	PIXELS	GENERATED MEDIA QUERY
sm:	`--breakpoint-sm: 40rem`	640px	`@media (width >= 40rem)`
md:	`48rem`	768px	`@media (width >= 48rem)`
lg:	`64rem`	1024px	`@media (width >= 64rem)`
xl:	`80rem`	1280px	`@media (width >= 80rem)`
2xl:	`96rem`	1536px	`@media (width >= 96rem)`

A class with no prefix applies at every width. A prefixed class applies at that width and everything wider. So you write the phone layout with plain classes first, and use the prefixes to change it as the screen grows. I put six background colours on one element and resized the viewport:

```
<div class="bg-red-500 sm:bg-green-500 md:bg-blue-500 lg:bg-purple-500 xl:bg-pink-500 2x
```

```
viewport width   colour
600, 639         red        (the unprefixed class)
640, 767         green      (sm: from 640)
768, 1023        blue       (md: from 768)
1024, 1279       purple     (lg: from 1024)
1280, 1535       pink       (xl: from 1280)
1536             yellow     (2xl: from 1536)
```

The boundaries are inclusive: at exactly 640px the `sm:` colour is already in force, at 639px it is not. And each colour holds until the next prefix takes over. This is the idea that trips people up: `sm:` **does not mean "on small screens"**. It means "from the small breakpoint (640px) up". If you want something on phones only, the plain class is the phone style, and you cancel it above:

```
<!-- wrong: nothing happens on a phone, because sm: starts at 640px -->
<div class="sm:text-center">…</div>

<!-- right: centred on phones, left-aligned from 640px up -->
<div class="text-center sm:text-left">…</div>
```

WITHOUT THE VIEWPORT META TAG, PHONES LIE ABOUT THEIR WIDTH

Every responsive page needs this line in its `<head>`: `<meta name="viewport" content="width=device-width, initial-scale=1">`. Without it a mobile browser pretends the screen is about 980px wide and shrinks the result, so none of your small-screen styles apply. I emulated a phone 390px wide and loaded the same page twice. Without the tag: `innerWidth` was **980**, `matchMedia('(min-width: 768px)')` was **true**, and the `md:` colour applied, on a 390px phone. With the tag: `innerWidth` was 390, the media query was false, and only the plain class applied. If your responsive layout "does nothing" on a real phone, check this first.

Max-Width, Ranges and Arbitrary Breakpoints

Mobile-first covers most needs, but you can also target "below" a width:

VARIANT	MEDIA QUERY	MEASURED
max-md:	`@media (width < 48rem)`	applied at 767px, not at 768px (strictly below)
max-sm:, max-lg:, max-xl:, max-2xl:	`width < 40rem`, `< 64rem`, `< 80rem`, `< 96rem`	the same pattern
md:max-lg:	`@media (width >= 48rem) { @media (width < 64rem) { … } }`	applied from 768 to 1023 only: blue at 768 and 1023, not at 767 or 1024
min-[600px]:	`@media (width >= 600px)`	not at 599px, applied at 600px
max-[600px]:	`@media (width < 600px)`	(compiled; the same strict rule)

Use `max-*` and ranges sparingly: they are for the case where a style is needed only in a middle band, such as hiding something between tablet and small-laptop sizes. A one-off width that is not on the scale goes in brackets. Adding your own named breakpoint to the theme (`--breakpoint-xs`) is chapter 9. The same mechanism also has a few other conditions: `landscape:` and `portrait:` (`orientation`), `print:` (`@media print`), and `pointer-coarse:` and `pointer-fine:` (`@media (pointer: coarse | fine)`, handy for making touch targets larger on touch screens). Dark mode, reduced motion and contrast are chapter 8.

Designing Upwards from a Phone

The workflow follows from mobile-first. Build the layout at about 360 to 390px wide with plain classes. Widen the window; the moment it looks wrong, add one breakpoint class that fixes that. Three patterns cover most pages, all measured:

A card grid that gains columns:

```
<div class="grid grid-cols-1 gap-4 sm:grid-cols-2 lg:grid-cols-4">…eight cards…</div>
```

```
viewport   columns
400        1
639        1
640        2
1023       2
1024       4
1100       4
```

A navigation bar that shows its links from `md` and a menu button below it. The links are `hidden md:flex` (not displayed, then a flex row from 768px) and the button is `md:hidden` (shown, then hidden from 768px). Measured: at 767px the links had `display: none` and the button `display: block`; at 768px the links were `flex` and the button `none`. The open-and-close behaviour of the menu needs a few lines of JavaScript or a `<details>` element; Tailwind handles only the showing and hiding.

Type and spacing that grow: `p-4 text-2xl md:p-8 md:text-4xl` measured 16px of padding and 24px text up to 767px, then 32px and 36px from 768px.

Images already behave, thanks to Preflight (chapter 1): `img { max-width: 100%; height: auto }`, so a large image shrinks to fit a small screen. For a fixed shape use `aspect-video w-full object-cover` (chapters 4 and 5), and to show different images at different sizes combine `hidden md:block` with `md:hidden`. Serving smaller files to smaller screens is HTML's job (`srcset` and `<picture>`), not Tailwind's.

Container Queries: Responding to the Parent

Breakpoints ask "how wide is the *screen*?". That is the wrong question for a reusable component. The same product card might sit in a wide main column and in a narrow sidebar on one page, at one screen width. A **container query** asks "how wide is the *box I am in*?". Mark the parent with `@container` and use the `@`-prefixed variants on its descendants:

```
<div class="@container">
  <div class="flex flex-col gap-2 @md:flex-row">
    <div class="h-24 bg-blue-200 @md:w-40 @md:shrink-0"></div>
    <div class="flex-1">Text</div>
  </div>
</div>
```

```
.\@container { container-type: inline-size; }
@container (width >= 28rem) { .\@md\:flex-row { flex-direction: row; } }
```

`@container` sets `container-type: inline-size`, which tells the browser to track the element's width so queries can use it. `@md:` then applies when the nearest container is at least `28rem` (448px) wide. Note that the container sizes are *not* the screen breakpoints: they come from the container scale, which is smaller:

VARIANT	CONTAINER WIDTH	VARIANT	CONTAINER WIDTH
@3xs:	16rem (256px)	@xl:	36rem (576px)
@2xs:	18rem (288px)	@2xl:	42rem (672px)
@xs:	20rem (320px)	@3xl … @7xl:	48, 56, 64, 72 and 80rem
@sm:	24rem (384px)	@max-md:	width < 28rem
@md:	28rem (448px)	@min-[400px]:	width >= 400px
@lg:	32rem (512px)		

Here is the difference from breakpoints, measured. The viewport was 1200px wide in every case. A grid with `grid-cols-1 @md:grid-cols-2` was placed inside a container, and beside it a grid with `grid-cols-1 md:grid-cols-2` (a screen breakpoint):

CONTAINER WIDTH	COLUMNS WITH `@MD:` (CONTAINER)	COLUMNS WITH `MD:` (SCREEN)
300px	**1**	2 (the screen is wide, whatever the box)
500px	2	2

The container-query version adapted to its 300px box; the screen-breakpoint version squeezed two columns into it. The same component dropped anywhere on the page lays itself out correctly. Three more details, all measured:

- **The query measures the container's content width, inside its padding.** A container `w-[460px] p-4` has 428px of content (460 − 32), which is below 448px, so `@md:` did not apply (1 column). `w-[480px] p-4` has exactly 448px of content, and `@md:` applied (2 columns).
- **Named containers** let a nested layout pick which one it listens to: `@container/card` (`container-type: inline-size; container-name: card`) and `@md/card:` (`@container card (width >= 28rem)`). A 500px named container gave 2 columns.
- **It is not the `container` class from chapter 4.** That one is a centred page wrapper with screen-width limits; `@container` declares a query container. The names are similar and the two are unrelated.

By the rules of CSS containment (not something I measured), a container with `container-type: inline-size` cannot take its width from its own contents, so give it a width from its parent or from layout, as the examples above do. As a rule of thumb: use screen breakpoints for the

page's overall structure (header, columns, navigation) and container queries for components that appear in more than one place.

Testing Responsive Layouts

In Chrome and Edge, press **Ctrl+Shift+M** in DevTools to open the device toolbar; Firefox has the same shortcut for its Responsive Design Mode. Drag the width handle slowly from 320px up to 1600px and watch each breakpoint fire (the toolbar can show the media-query bands), then try a phone preset to see the viewport meta tag working. Two checks are worth doing on every page: that nothing scrolls sideways at 320px (a common cause is a fixed width wider than the screen, or the 1fr issue from chapter 4), and that the touch targets are comfortable at phone sizes. An emulator is not a phone: test on a real device before you ship something important.

Hands-On Exercises

EXERCISE 1

Build a page wrapper (`mx-auto max-w-6xl p-4`) holding a grid of eight 80px-high cards that has one column on phones, two from 640px and four from 1024px, with 16px gaps. Report the number of columns and the width of a card at viewport widths 390, 700, 1100 and 1500, and explain each card width from the classes.

View solution

EXERCISE 2

Build one product card that stacks its image above its text in narrow spaces and puts them side by side in wide ones, using a container query. Place the identical card in a 700px main column and a 300px sidebar on the same 1200px-wide page. Show that the two copies lay out differently, and say what would have happened with `md:` instead of `@md:` .

View solution

EXERCISE 3

Read-the-classes audit. For each element below, say what it resolves to at viewport widths 360, 640, 768 and 1024 before you measure, then check with a resized viewport: (a) `text-sm md:text-base lg:text-lg` (font size), (b) `hidden sm:block md:hidden` (display), (c) `p-2 sm:p-4 max-sm:p-1` (padding), (d) `text-red-500 md:max-lg:text-blue-500` (colour). Explain each surprise.

View solution

CHAPTER 7 QUICK REFERENCE

- Always `<meta name="viewport" content="width=device-width, initial-scale=1">`; without it a 390px phone reported `innerWidth` 980 and applied `md:`
- Breakpoints: `sm` 640, `md` 768, `lg` 1024, `xl` 1280, `2xl` 1536 (`width >= …`, inclusive). **Unprefixed = all sizes; prefixed = that size and up.** `sm:` is not "on phones"
- Write the phone layout first, add `sm: md: lg:` only where it breaks: `text-center sm:text-left`, `grid-cols-1 sm:grid-cols-2 lg:grid-cols-4`, `hidden md:flex` + `md:hidden`
- `max-md:` (`width < 48rem`, strict), range `md:max-lg:` (768 to 1023), arbitrary `min-[600px]:` / `max-[600px]:`
- Other conditions: `landscape: portrait: print: pointer-coarse: pointer-fine:`
- Container query: parent `@container` (`container-type: inline-size`), children `@md:` etc. (`@3xs` 16rem … `@md` 28rem … `@7xl` 80rem), `@max-md:`, `@min-[400px]:`
- The query measures the container's CONTENT width (inside its padding); name containers with `@container/card` and `@md/card:`
- Screen breakpoints for page structure, container queries for reusable components; DevTools device toolbar is `Ctrl+Shift+M`; check 320px for sideways scrolling

COMING NEXT

Tailwind CSS Fundamentals 8 covers the user-preference variants: `dark:` and how to switch dark mode by a class or attribute, then `motion-safe:` and `motion-reduce:`, `contrast-more:`, `forced-colors:` and `print:`.

Tailwind CSS Fundamentals

Chapter 8 · Dark Mode and Preference Variants

Chapter 7's variants responded to the size of the screen. This chapter's respond to the *person*: whether they have chosen a dark theme, asked their system to reduce animation, asked for stronger contrast, or are printing the page. They use the same `variant:utility` pattern, and each is a CSS media feature that the browser sets from the user's operating-system settings. The main one is `dark:`, and most of the chapter is about using it well: designing colours that stay readable, and letting the visitor override the system choice without a flash of the wrong theme.

> **EVERY EXAMPLE HERE WAS RUN**
>
> CSS is copied from Tailwind CSS 4.3.3's output. To test preferences I used Chrome's media emulation (the same switches as DevTools' Rendering panel) to set the colour scheme, reduced motion, contrast and forced-colours settings, and to switch to print media, then read computed styles back. The contrast ratios are the same WCAG measurement as chapter 3.

dark: and the System Preference

By default `dark:` is the `prefers-color-scheme: dark` media query. The usual pattern is to write the light colours as plain classes and override them in dark mode:

```
<div class="bg-white text-gray-900 dark:bg-gray-900 dark:text-gray-100">…</div>
```

```
@media (prefers-color-scheme: dark) {
  .dark\:bg-gray-900 { background-color: var(--color-gray-900); }
}

system set to light:  background white,    text gray-900
system set to dark:   background gray-900, text gray-100
```

I switched the emulated system between the two and read the colours back (`matchMedia("(prefers-color-scheme: dark)").matches` went from `false` to `true`); the page restyled itself with no script. The `dark:` prefix combines with other variants: `dark:hover:bg-gray-800` compiled to a `@media (prefers-color-scheme: dark)` block containing the `@media (hover:`

`hover)` rule, and `dark:md:p-4` to the dark block containing the `md` block (and `md:dark:p-5` the other way round, with the same effect).

Designing a Dark Palette

Dark mode is not "invert everything". Three things change, and the first is measurable. **Shades that pass on white often fail on dark.** Chapter 3 showed that `gray-500` is the lightest grey that passes on white (4.84:1). On a `gray-900` background the same colours behave very differently, measured the same way:

TEXT ON `BG-GRAY-900`	RATIO	NORMAL TEXT (4.5:1)
text-gray-100	16.13	passes
text-gray-300	12.05	passes
text-gray-400	6.82	passes
text-gray-500	3.67	**fails**
text-gray-600	2.35	**fails**
text-blue-400	6.72	passes
text-blue-500	4.72	passes (just)
text-blue-600	3.38	**fails**
text-red-400	6.14	passes
text-red-600	3.72	**fails**

The pattern: a colour that is darkened (600) for a white page needs the lighter end (300 to 400) on a dark page. A secondary text colour of `text-gray-500` in both modes is therefore a real bug: `text-gray-500 dark:text-gray-400` fixes it (4.84 on white, 6.82 on gray-900). A link needs the same: `text-blue-600 dark:text-blue-400` (5.25 and 6.72). The second change is **depth**: shadows (chapter 3) are black at about 10% opacity, and black on a near-black page is close to invisible, so in dark mode separate surfaces with a lighter background or a border (`dark:border-gray-700`) rather than relying on `shadow-md`. The third is **the browser's own parts**: form controls, scrollbars and the default page background follow the CSS `color-scheme` property. The `scheme-dark` class sets `color-scheme: dark` (`scheme-light` and `scheme-light-dark` are

the others), and the computed value reported `dark`; set it on `<html>` when the page is dark so checkboxes and scrollbars match. (I verified the property, not how each control looks.)

Switching by Class or Attribute

The system setting is a good default, but many sites let the visitor choose. For that, change what `dark:` means with one line of CSS, using the `@custom-variant` directive. The official documentation gives two forms. The class version:

```
@import "tailwindcss";
@custom-variant dark (&:where(.dark, .dark *));
```

```
.dark\:bg-gray-900:where(.dark, .dark *) {
  background-color: var(--color-gray-900);
}
```

Now `dark:` applies when the element is inside something with the class `dark` (or has it itself). Put the class on `<html>` and the whole page is dark. The `:where()` adds no specificity, so these rules compete with the light ones on equal terms. The system setting is no longer consulted. I tested all four combinations of the (emulated) system setting and the class:

SYSTEM SETTING	`DARK` CLASS ON `<HTML>`	BACKGROUND
dark	absent	white (light)
dark	present	gray-900 (dark)
light	present	gray-900 (dark)
light	absent	white (light)

A dark system setting with no class gave a light page: with the custom variant, only the class counts. That is why a class-based site needs a small script to decide which to apply (next section). The attribute version is the same with a selector for `data-theme`: `@custom-variant dark (&:where([data-theme=dark], [data-theme=dark] *));`. It also lets you make a dark **island** on a light page: I put `data-theme="dark"` on one `<div>`; a white-by-default box inside it measured dark, and an identical box outside it stayed white. Setting the attribute on `<html>` then turned the outside dark too.

A Light / Dark / System Toggle, Without a Flash

A good theme switch has three choices: light, dark, and "follow my system" (the default). It remembers the choice, and it follows the system live when set to system. This is the script, adapted from the official one, and I ran it with the class variant above:

```
<head>
  …
  <script>
    function applyTheme() {
      const t = localStorage.theme;
      const dark = t === 'dark' || (t !== 'light' && matchMedia('(prefers-color-scheme:
      document.documentElement.classList.toggle('dark', dark);
    }
    applyTheme();
    matchMedia('(prefers-color-scheme: dark)').addEventListener('change', applyTheme);
    function setTheme(choice) {
      if (choice === 'system') localStorage.removeItem('theme'); else localStorage.theme
      applyTheme();
    }
  </script>
</head>
<body>
  <button onclick="setTheme('light')">Light</button>
  <button onclick="setTheme('dark')">Dark</button>
  <button onclick="setTheme('system')">System</button>
  …
```

Measured, step by step, with the emulated system setting changing underneath the page:

STEP	PAGE IS DARK?	STORED `THEME`
1. Fresh visit, system light	no	nothing
2. System switches to dark while the page is open	**yes** (the `change` listener)	nothing
3. Visitor chooses Light (system still dark)	no	`light`
4. System light, visitor chooses Dark	yes	`dark`

STEP	PAGE IS DARK?	STORED `THEME`
5. Reload the page	yes, and already dark when the body started	`dark`
6. Visitor chooses System (system light)	no	removed
7. System switches to dark	yes (following the system again)	nothing

Step 5 is the point of placing the script **inline in the** `<head>`. I compared two pages that run the same code, one with the script in the head and one at the end of the body, and recorded whether the `dark` class was set at the moment the browser started parsing the body: **true** for the head script, **false** for the end-of-body script (the class was only added afterwards). With the later script the browser is free to paint the light page first and switch a moment later, which is the "flash of the wrong theme" you have seen on some sites. (I measured when the class was applied, not the paint itself, so I can show the cause but not a flash.) Keep the script small and put it before anything that paints.

Reduced Motion, Contrast, Forced Colours and Print

The other preference variants work like `dark:`. Each compiles to a media query and I switched the corresponding setting on in the emulator:

VARIANT	MEDIA QUERY	MEASURED
motion-safe:	`(prefers-reduced-motion: no-preference)`	`motion-safe:animate-pulse`: animation `pulse` with no preference, `none` when reduction was requested
motion-reduce:	`(prefers-reduced-motion: reduce)`	`animate-spin motion-reduce:animate-none`: `spin` normally, `none` with reduction on
contrast-more: contrast-less:	`(prefers-contrast: more | less)`	`border-0 contrast-more:border-2`: 0px, then 2px with more contrast requested
forced-colors: not-forced-colors:	`(forced-colors: active)`, `not (forced-colors: active)`	`p-1 forced-colors:p-8`: 4px, then 32px with forced colours active
print:	`print`	`print:hidden`: `display: block` on screen, `none` in print media

VARIANT	MEDIA QUERY	MEASURED
inverted-colors:	`(inverted-colors: inverted)` ,	(compiled; not measured)
noscript:	`(scripting: none)`	

Why bother. **Reduced motion**: some people get dizzy or ill from large movement; spinners and sliding panels should respect the setting. The safest way is `motion-safe:`, which applies animation only when the visitor has not asked for less, rather than adding `motion-reduce:` to every animated class. **Contrast**: a visitor who has asked for more contrast may need your faint borders strengthened, as in `contrast-more:border-2`. **Forced colours** (Windows High Contrast): the browser replaces your colours with a small system palette, so anything defined only by a background colour (a button with no border) can lose its edges; `forced-colors:` lets you add a border or outline there. That is standard behaviour of the mode; I verified that the variant applies, not how a page looks inside it. **Print**: `print:hidden` for navigation and ads, `print:text-black` and similar for ink-friendly pages.

Testing Preferences

You do not need to change your operating system. In Chrome's DevTools open the command menu (Ctrl+Shift+P), run **Show Rendering**, and use the "Emulate CSS media feature" selectors for `prefers-color-scheme`, `prefers-reduced-motion`, `prefers-contrast` and `forced-colors`, and "Emulate CSS media type" for print. These are the same settings I switched with the emulator above (I drove them by script, not through the panel, and the exact list of switches in the panel depends on your Chrome version). For a manual theme toggle also test with local storage cleared, with a stored choice, and on a slow connection, which is where a flash is most visible.

Hands-On Exercises

EXERCISE 1

Style a card that works in both system modes: a surface, a border, a heading, a grey caption and a blue link. Start with the caption as `text-gray-500` and the link as `text-blue-600` in both modes, measure the contrast ratio of every text pair in light and in dark, find which fail, and fix them with the smallest changes.

View solution

EXERCISE 2

Add the class-based `dark` variant and a Light / Dark / System toggle to a page. Using emulated system changes, prove that the choice is remembered across a reload, that it is applied before the body is parsed, that "System" follows live changes, and that an explicit choice ignores them.

View solution

EXERCISE 3

Make a small component respect four preferences: a spinner that stops spinning when the visitor asks for reduced motion, a card whose border thickens for higher contrast, a decorative banner that disappears in print, and a button that gets extra padding in forced-colours mode. Switch each preference on in the emulator and report the before and after computed values.

View solution

CHAPTER 8 QUICK REFERENCE

- `dark:` = `@media (prefers-color-scheme: dark)` by default; write light as the plain class: `bg-white dark:bg-gray-900`; it stacks (`dark:hover:`, `dark:md:`)
- Dark palettes: text on `gray-900`: gray-100 16.13, 300 12.05, 400 6.82 pass; gray-500 3.67 and gray-600 2.35 fail. blue-400/500 pass, blue-600 fails; red-400 passes, red-600 fails. Use `text-gray-500 dark:text-gray-400`, `text-blue-600 dark:text-blue-400`
- Shadows vanish on dark surfaces: use `dark:border-gray-700` or a lighter surface; `scheme-dark` (`color-scheme: dark`) for controls and scrollbars
- Manual switching: `@custom-variant dark (&:where(.dark, .dark *));` (class) or `…([data-theme=dark], [data-theme=dark] *)` (attribute, allows dark islands); the system setting is then ignored
- Toggle: inline script in `<head>` that sets the class from `localStorage.theme` or `matchMedia`; listen for `change`; `removeItem('theme')` for System
- `motion-safe:` / `motion-reduce:` (reduced motion), `contrast-more:` / `contrast-less:`, `forced-colors:` / `not-forced-colors:`, `print:`, `noscript:`, `inverted-colors:`

- Test with DevTools: Show Rendering, then "Emulate CSS media feature …" and "Emulate CSS media type: print"

COMING NEXT

Tailwind CSS Fundamentals 9 opens the theme itself: `@theme` and its namespaces, extending and overriding the defaults, adding fonts and breakpoints, and keyframes for your own animations.

Tailwind CSS Fundamentals

Chapter 9 · Your Theme: @theme, Variables and Fonts

Every class you have used so far reads from a *theme*: `p-4` from `--spacing`, `bg-blue-500` from `--color-blue-500`, `rounded-lg` from `--radius-lg`. In Tailwind v3 you changed the theme in a JavaScript file, `tailwind.config.js`. In v4 you change it in your CSS, with a directive called `@theme`. This chapter shows how to add your own colours, fonts, sizes and animations, how to remove or replace the defaults, and the one option (`inline`) that decides whether theming by context works.

> **EVERY EXAMPLE HERE WAS RUN**
>
> Every theme file in this chapter was compiled with Tailwind CSS 4.3.3 and the output read, and the behaviour was checked in headless Chrome (computed styles, the fonts the browser loaded, animations running, and the page resized to test breakpoints). The official `@theme` reference was used for the concepts, and each claim below was tested rather than assumed. One font file (a Windows system font) was used only for the local test, not published.

What @theme Does

A **theme variable** is a CSS variable written inside `@theme`. It does two jobs at once: it becomes a normal CSS variable on the page, *and* its name (the part before the first dash after `--`) tells Tailwind which classes to create. That first part is the **namespace**:

NAMESPACE	CREATES CLASSES SUCH AS	EXAMPLE VARIABLE
`--color-*`	`bg-mint-500`, `text-mint-500`, `border-mint-500`, `ring-`, `fill-` …	`--color-mint-500: oklch(0.72 0.11 178)`
`--font-*`	`font-display` (font family)	`--font-display: "Georgia", serif`
`--text-*`	`text-huge` (font size), with `--text-huge--line-height`	`--text-huge: 5rem`
`--spacing`, `--spacing-*`	every padding, margin, gap, width and height class; `p-gutter`, `gap-gutter`, `w-gutter`	`--spacing-gutter: 1.5rem`
`--radius-*`	`rounded-pill`	`--radius-pill: 9999px`

NAMESPACE	CREATES CLASSES SUCH AS	EXAMPLE VARIABLE
--shadow-*	`shadow-glow`	`--shadow-glow: 0 0 20px rgb(34 211 238 / 0.5)`
--ease-*	`ease-snappy`	`--ease-snappy: cubic-bezier(0.2, 0.9, 0.3, 1)`
--animate-*	`animate-wiggle`	`--animate-wiggle: wiggle 1s ease-in-out infinite`
--breakpoint-*	screen variants `xs:`, `3xl:` (chapter 7)	`--breakpoint-xs: 30rem`
--container-*, --font-weight-*, --blur-*, ...	`@md:` and `max-w-md`; `font-bold`; `blur-sm`	the other default namespaces

I put one variable in each of those namespaces into a theme file, used a class for each, and compiled it. Each class produced a rule that reads its variable (shadows are the one exception, see the note after the list):

```
@import "tailwindcss";

@theme {
  --color-mint-500: oklch(0.72 0.11 178);
  --font-display: "Georgia", serif;
  --radius-pill: 9999px;
  --shadow-glow: 0 0 20px rgb(34 211 238 / 0.5);
  --text-huge: 5rem;
  --text-huge--line-height: 1;
  --ease-snappy: cubic-bezier(0.2, 0.9, 0.3, 1);
  --spacing-gutter: 1.5rem;
  --animate-wiggle: wiggle 1s ease-in-out infinite;
  @keyframes wiggle {
    0%, 100% { transform: rotate(-3deg); }
    50% { transform: rotate(3deg); }
  }
  --breakpoint-xs: 30rem;
  --breakpoint-3xl: 120rem;
}
```

```
bg-mint-500     background-color: var(--color-mint-500)
font-display    font-family: var(--font-display)
rounded-pill    border-radius: var(--radius-pill)
text-huge       font-size: var(--text-huge); line-height: var(--tw-leading, var(--text-hu
```

```
ease-snappy     transition-timing-function: var(--ease-snappy)
p-gutter        padding: var(--spacing-gutter)       (also gap-gutter, w-gutter)
animate-wiggle  animation: var(--animate-wiggle)
```

A detail: `shadow-glow` did not read a variable. The generated rule has the shadow written into it (`--tw-shadow: 0 0 20px var(--tw-shadow-color, rgb(34 211 238 / 0.5))`), because shadows are built from layers that combine with rings (chapter 3). Use `@theme` rather than `:root` when you want classes: a plain `:root { --brand: … }` variable is just a variable and creates nothing.

Extending, Overriding and Removing

The three things you can do to the defaults, all measured:

- **Extend:** use a new name. `--color-mint-500` and the others above sit alongside the defaults; every default class still works.
- **Override:** reuse an existing name with a new value. With `@theme { --spacing: 0.5rem; }`, `p-4` measured **32px** (it is 4 × 0.5rem; it was 16px) and `w-64` measured **512px**. The whole spacing scale doubled, because every size class multiplies that one variable.
- **Remove:** set a whole namespace to `initial`. After `--color-*: initial; --color-white: #fff; --color-midnight: #121063;` I checked the compiled classes: `bg-white`, `bg-midnight` and `text-midnight` exist, and `bg-blue-500` and `text-red-500` were **not generated** (an element with `text-red-500` kept the plain black text). This is how you replace Tailwind's palette with your brand's, so a stray `bg-blue-500` stops working instead of quietly slipping through.

> **--*: INITIAL REMOVES EVERYTHING, INCLUDING THE SPACING SCALE**
>
> `@theme { --*: initial; }` clears every default variable. With `--*: initial; --color-midnight: #121063;` the class `bg-midnight` still compiled, but `p-4` and `w-64` produced **no rules at all**, because `--spacing` was gone too. It is meant for building a design system from nothing; if you only want your own colours, remove just `--color-*`.

Theme Variables Are CSS Variables, and Only Used Ones Are Written

Because they are CSS variables, you can use them anywhere: in your own CSS (`color: var(--color-mint-500)`), in arbitrary values (`rounded-[calc(var(--radius-lg)-1px)]`), in inline styles, and in JavaScript with `getComputedStyle(document.documentElement).getPropertyValue('--color-mint-500')`. One catch: Tailwind only writes out the theme variables that something *uses*. In the theme above, `--color-mint-500` was in the compiled file, while `--color-unused-500: #123456`, which no class used, was not. Usually that is what you want (a small file), but not when JavaScript or plain CSS needs a variable that no class mentions. Mark such variables with `static`:

```
@theme static {
  --color-primary: var(--color-red-500);
}
```

```
a page where no class uses "primary":
with @theme static   getPropertyValue("--color-primary")  oklch(63.7% 0.237 25.331)
with plain @theme    getPropertyValue("--color-primary")  ""   (not emitted)
```

Two more rules, both checked. **Theme variables must be at the top level.** The documentation says they cannot be nested under a selector or a media query. I put `@theme { --color-nested: #ff0000; }` inside `@media (min-width: 40rem)`: Tailwind raised no error, and it emitted `--color-nested` unconditionally on `:root`, so the media condition was silently ignored. If you want a value that changes with screen size, use a normal variable in a normal media query. And `@theme` **can live in another file**: a `brand.css` containing only the `@theme` block, pulled in with `@import "./brand.css";` after `@import "tailwindcss";`, produced `bg-brand-500` as expected.

@theme inline: Theming by Context

This is the option that most often goes wrong, so I measured it. Suppose you define the brand colour as an ordinary variable that changes in part of the page, and point a theme variable at it:

```
@import "tailwindcss";
@theme { --color-primary: var(--brand); }            /* A: plain */
/* @theme inline { --color-primary: var(--brand); }  B: inline */

:root { --brand: oklch(0.62 0.21 260); }             /* blue */
[data-theme=green] { --brand: oklch(0.72 0.22 150); }
```

```
<div class="bg-primary">…</div>
<div data-theme="green"><div class="bg-primary">…</div></div>
```

```
                     outside the island    inside data-theme="green"
A: plain @theme      blue                  blue    (the theme did not apply)
B: @theme inline     blue                  green
```

With the plain form the theme variable `--color-primary` is declared once, on `:root`, and its value is worked out there (the blue `--brand`) and then inherited as a finished value, so changing `--brand` lower down does nothing to it. With `inline` the utility skips that intermediate variable. I compared the two compiled rules: the plain form generates `.bg-primary { background-color: var(--color-primary); }`, the inline form generates `.bg-primary { background-color: var(--brand); }`, and that `var(--brand)` is looked up on each element, so the green island gets green. The rule: **use `@theme inline` whenever a theme value is a `var()` of something that changes by context**: a `data-theme` attribute, a dark-mode override, or a font variable supplied by a framework. It is also what lets you build the multi-brand setups in *Tailwind CSS Intermediate/Advanced 2*.

Fonts

A font family is a theme variable like any other, in the `--font-*` namespace. Loading the font is separate: that is plain CSS, written *outside* `@theme`. For a font file you have:

```
@import "tailwindcss";

@font-face {
  font-family: "MyDisplay";
  src: url("./fonts/mydisplay.woff2");
}

@theme {
  --font-display: "MyDisplay", serif;
}
```

```
<span class="font-sans">       width 312px
<span class="font-display">    width 322px   fonts the browser loaded: ["MyDisplay"]
```

```
same page, but --font-display: "NoSuchFontAnywhere", serif
<span class="font-display">    width 302px   fonts loaded: []   (the plain serif fallbac
```

I used a system font file as `MyDisplay` to test this. With the `@font-face` present the browser loaded the font and the text came out at a different width than the default sans; with a name that matches nothing it silently used the generic `serif` from the fallback list (a third width again). So a misspelled font name does not fail, it falls back, and the quickest check is the list of loaded fonts (`Array.from(document.fonts)` with a status of `loaded`), or the "Rendered Fonts" list at the bottom of DevTools' Computed panel. (`document.fonts.check()` returned `true` in both cases, so do not rely on it.)

To change the font of the whole page, override `--font-sans`: Preflight (chapter 1) sets the page font to `--default-font-family`, which is `var(--font-sans)`. With `@theme { --font-sans: "Inter", sans-serif; }` a paragraph with no classes at all had the computed font family `Inter, sans-serif`, while `font-mono` was unchanged. For a hosted font such as Google Fonts, the stylesheet import has to be the first thing in the file: `@import url("https://fonts.googleapis.com/…");` followed by `@import "tailwindcss";` compiled with the font import kept first (I compiled this; I did not fetch the font). When a framework's font loader gives you a CSS variable, that is the `inline` case above: `@theme inline { --font-sans: var(--font-inter); }`.

Your Own Animations

Define the animation as a variable and put its `@keyframes` inside the same `@theme` block (as in the first example). Tailwind then writes both the `animate-wiggle` class and the keyframes. In Chrome the element had `animation-name: wiggle`, one running animation (`getAnimations().length` was 1), and its `transform` was a different rotation 350ms later: `matrix(0.9988, 0.0486, -0.0486, 0.9988, 0, 0)` then `matrix(0.9989, -0.0465, 0.0465, 0.9989, 0, 0)`, swinging from about +2.8° to −2.7°. Respect the visitor's reduced-motion setting (chapter 8) with `motion-safe:animate-wiggle`. The other examples in the first table work the same way: `--ease-snappy` becomes `ease-snappy` for transitions, `--radius-pill` becomes `rounded-pill`, and a named spacing such as `--spacing-gutter` is available to padding, margin, gap and size classes alike.

Your Own Breakpoints

The `--breakpoint-*` namespace controls the screen variants. I added `--breakpoint-xs: 30rem` and `--breakpoint-3xl: 120rem` (next to the five defaults) and resized the viewport on an element with `bg-gray-300 xs:bg-red-500 3xl:bg-green-500`:

VIEWPORT WIDTH	COLOUR	WHY
479px	gray	below `xs` (30rem = 480px)
480px	red	`xs:` applies from 480
1919px	red	`xs:` still applies; `3xl` (120rem = 1920px) has not started
1920px	green	`3xl:` applies and, being larger, overrides `xs:`

Tailwind orders the generated media queries by size, not by the order you declared the variables, so a new breakpoint slots into the right place. Changing an existing one is the same: `--breakpoint-sm: 30rem` moves `sm:` (the documentation shows this; I tested adding names, not changing `sm`).

Hands-On Exercises

EXERCISE 1

Replace Tailwind's palette with a brand palette: keep only `white`, `black` and three brand shades (50, 500 and 900), add a `font-display` family loaded with `@font-face` and a `rounded-pill` radius. Build a pill button using them and show from the compiled output and the browser that the brand classes work and that `bg-blue-500` and `text-red-500` no longer do.

View solution

EXERCISE 2

Build a two-theme page: `:root` blue and `[data-theme=green]` green, with a `bg-primary` class. Do it first with a plain `@theme` and show that the green section stays blue, then fix it with the one-word change and show the green section turns green.

View solution

EXERCISE 3

Add a `wiggle` animation, an `xs` breakpoint at 30rem, and a `--color-primary` that JavaScript must be able to read even though no class uses it. Prove the animation is running, show the breakpoint boundary at 479 and 480px, and show the difference between `@theme` and `@theme static` for the unused variable.

View solution

CHAPTER 9 QUICK REFERENCE

- `@theme { --namespace-name: value; }` makes a CSS variable *and* the classes for its namespace: `--color-*`, `--font-*`, `--text-*` (+ `--text-x--line-height`), `--spacing` / `--spacing-*`, `--radius-*`, `--shadow-*`, `--ease-*`, `--animate-*`, `--breakpoint-*`, `--container-*`. Use `:root` for variables that should NOT create classes
- Extend with a new name; override with an existing name (`--spacing: 0.5rem`: `p-4` = 32px); remove a namespace with `--color-*: initial` (`bg-blue-500` stops existing); `--*: initial` removes everything, including spacing
- Only used theme variables are written to the output; `@theme static` forces them (JavaScript can then read them); `@theme` must be top level (inside `@media` the condition is silently ignored); it can live in an imported file
- `@theme inline` when a theme value is `var(--something)` that changes by context (`data-theme`, dark overrides, framework font variables); plain `@theme` resolves it once on `:root`
- Fonts: `@font-face` outside `@theme`, then `--font-display: "Name", serif`; a wrong name silently falls back; override `--font-sans` to change the page font; `@import url()` before `@import "tailwindcss"`
- Animations: `--animate-wiggle: wiggle 1s infinite` plus `@keyframes wiggle {…}` inside `@theme`; add `motion-safe:`
- Breakpoints: `--breakpoint-xs: 30rem` gives `xs:` (from 480px); generated in size order

COMING NEXT

Tailwind CSS Fundamentals 10 goes beyond the generated utilities: arbitrary values and variants, `@utility` and `@variant`, `@layer`, `@apply` and when to write ordinary CSS instead.

Tailwind CSS Fundamentals

Chapter 10 · Going Beyond the Utilities

Sooner or later you need something Tailwind does not ship: a 372px width, a selector that targets the children, a utility your team uses everywhere, a component class for markup you cannot edit. Tailwind gives you a ladder of escape hatches, and the skill is to take the lightest one that works: an arbitrary value, then an arbitrary property or variant, then your own `@utility` or `@variant`, then a class in `@layer components` (with `@apply`), and only then a plain stylesheet. This chapter climbs the ladder, and it explains the cascade-layer rule that decides whether your own CSS beats Tailwind's or loses to it.

> **EVERY EXAMPLE HERE WAS RUN**
>
> Every class and directive was compiled with Tailwind CSS 4.3.3 and the output read; behaviour (widths, computed values, which rule wins, what happens when hovering or when the system is in dark mode) was checked in headless Chrome. The official documentation was used for the directive syntax; where I say a behaviour is not in the documentation, I tested it myself.

Arbitrary Values

Put a value in square brackets and Tailwind uses it as written. The class name is the utility plus the value; you saw this in chapters 2 and 3 with `p-[13px]` and `bg-[#bada55]`. From the compiled output:

CLASS	GENERATED CSS
w-[372px]	`width: 372px`
top-[117px]	`top: 117px`
bg-[#bada55]	`background-color: #bada55`
grid-cols-[1fr_auto]	`grid-template-columns: 1fr auto` (an underscore is a space)
content-['a_b']	`content: 'a b'` (the underscore again; `before:content-['Festivus']` adds text)
p-[calc(1rem+2px)]	`padding: calc(1rem + 2px)` (Tailwind adds the spaces CSS needs around `+`)
aspect-[4/3]	`aspect-ratio: 4/3`

CLASS	GENERATED CSS
bg-[url('/x.png')]	`background-image: url('/x.png')`

I measured a `w-[calc(100%-2rem)]` box inside a 600px parent: `568px` (600 − 32). The writing rules are small: no spaces inside the brackets (use `_` where you need one), and the whole class name must still appear in your files as one piece, which is the chapter 1 rule again.

CSS Variables as Values

To use a CSS variable, write it in parentheses: `w-(--w)` is shorthand for `w-[var(--w)]` and adds the `var()` for you. This is the right answer to a value that changes at run time, because a class name cannot be built from a variable (chapter 1) but a variable can be set from anywhere, including an inline style:

```
<div class="w-(--w) h-4" style="--w: 200px"></div>
```

```
.w-\(--w\) { width: var(--w); }

measured width: 200px
```

The one trap is **type ambiguity**. Several utilities share a prefix: `text-` sets font size or colour, `bg-` sets colour, image or size. Given only a variable, Tailwind cannot tell which you mean, so it picks *colour*, and I compiled the proof:

```
text-(--v)          text-[var(--v2)]      text-(length:--sz)      text-(color:--clr)
bg-(--bgv)          bg-(image:--img)      bg-(length:--sz)
```

```
text-(--v)          color: var(--v)
text-[var(--v2)]    color: var(--v2)
text-(length:--sz)  font-size: var(--sz)      (measured with --sz: 30px: 30px)
text-(color:--clr)  color: var(--clr)         (measured with --clr: rgb(255, 0, 0): re
bg-(--bgv)          background-color: var(--bgv)
bg-(image:--img)    background-image: var(--img)
bg-(length:--sz)    background-size: var(--sz)
```

So a variable holding a size needs the hint, `text-(length:--sz)`; the official docs show the same two hints for `text-`.

Arbitrary Properties and Variants

If there is no utility at all for a CSS property, write the property itself in brackets: `[mask-type:luminance]` compiled to `mask-type: luminance` and the element's computed value was `luminance`. Variants work in front, so `hover:[mask-type:alpha]` is a hover rule. A good use is setting a variable that other classes read, differently at different sizes:

```
<div class="[--scroll-offset:56px] lg:[--scroll-offset:44px]"></div>
```

```
viewport 1000px:  --scroll-offset = 56px
viewport 1100px:  --scroll-offset = 44px      (lg: starts at 1024px)
```

Arbitrary variants do the same for selectors and conditions: a bracketed variant is a selector in which `&` stands for the element itself. All of these were measured or compiled:

CLASS	MEANING	RESULT
[&>*]:p-2	the element's direct children	a child's padding measured 8px
[&_p]:mt-2	any `p` inside (underscore is a space)	a nested `p` measured 8px margin-top
lg:[&:nth-child(-n+3)]:hover:underline	on large screens, the first three siblings, on hover	compiled to `:nth-child(-n+3):hover` inside the media queries
supports-[display:grid]:grid	`@supports (display:grid)`	display measured `grid`
group-[.open]:block	inside a `.group` that also has the class `open`	`hidden group-[.open]:block`: `none` without the class, `block` with it
@[500px]:p-4	a container query at an exact width	compiled to `@container (width >= 500px)`
[@media(width>=500px)]:p-1	any at-rule	compiled to `@media (width>=500px)`

The official docs describe arbitrary properties as "really like inline styles, but again with the benefit that you can use modifiers". They are for the rare one-off. If you find yourself writing the

same bracketed class in many places, it deserves a name, which is the next rung.

Your Own Utilities and Variants

`@utility` defines a class that behaves like a built-in one, including working with every variant. A static one is a name and some declarations (nested selectors are allowed):

```
@utility scrollbar-hidden {
  scrollbar-width: none;
  &::-webkit-scrollbar { display: none; }
}
```

```
.scrollbar-hidden { scrollbar-width: none; &::-webkit-scrollbar { display: none; } }
@media (hover: hover) {
  .hover\:scrollbar-hidden:hover { scrollbar-width: none; }
  .hover\:scrollbar-hidden:hover::-webkit-scrollbar { display: none; }
}

measured scrollbar-width:
scrollbar-hidden          none
hover:scrollbar-hidden    auto at rest, none while hovered
```

A **functional** utility takes a value from the class name: the `*` in the name stands for it, and `--value()` says what values are allowed. This one accepts a theme variable, a bare integer, or a bracketed integer:

```
@theme { --tab-size-github: 8; }

@utility tab-* {
  tab-size: --value(--tab-size-*, integer, [integer]);
}
```

```
tab-4          tab-size: 4                      measured 4
tab-github     tab-size: var(--tab-size-github)  measured 8
tab-[3]        tab-size: 3                      measured 3
md:tab-2       tab-size: 2 from 768px           measured 4 at 700px, 2 at 800px
```

The documentation shows each form of `--value()` on its own; I combined them in one list and it compiled and behaved as above. It also shows variants on static utilities only, so I checked a functional one: `md:tab-2` worked. For variants, `@custom-variant` (chapter 8) defines a new condition. Its companion `@variant` goes the other way: it lets *your own CSS* use a Tailwind variant. This rule compiled to the media queries shown in the comments, and I measured it:

```
.my-element {
  background: white;
  @variant dark { background: black; }   /* @media (prefers-color-scheme: dark) */
  @variant hover { background: red; }    /* &:hover inside @media (hover: hover) */
}
```

```
measured background:
system light     rgb(255, 255, 255)
system dark      rgb(0, 0, 0)
pointer over it  rgb(255, 0, 0)
```

Cascade Layers: Who Wins, Your CSS or Tailwind's

Tailwind puts its output into **cascade layers** (chapter 1): `theme`, `base`, `components`, `utilities`, in that order of strength. You can add your own CSS to `base` (element defaults, such as sizes for `h1` and `h2`) or `components` (classes like `.card`), and anything in `utilities` you define with `@utility` lands in the utilities layer. The reason to use `@layer components` is exactly that its place in the order is below the utilities, so you can still adjust a component with a class. I measured three cases:

```
@layer components {
  .card { padding: 1rem; border-radius: 0.5rem; }
}
.btn-unlayered { padding: 4px; }          /* not in any layer */

<div class="card p-8">…</div>
<div class="card rounded-none">…</div>
<div class="btn-unlayered p-8">…</div>
```

```
.card p-8              padding-top 32px      the utility beat the component class
.card rounded-none     radius 0px            the utility beat the component class
.btn-unlayered p-8     padding-top 4px       the UNLAYERED rule beat the utility
```

CSS YOU WRITE OUTSIDE @LAYER BEATS EVERY UTILITY

The third result is the surprise. In CSS, a style that is *not in any layer* outranks styles in layers, whatever the selectors' specificity. Tailwind's utilities are all in a layer, so a plain `.btn { padding: 4px }` wins against `p-8`, and utilities mysteriously "stop working" on that element. Moving the same rule into `@layer components { .btn { padding: 4px } }` measured: `btn p-8` became 32px, and `btn` alone stayed 4px. The practical rule: put your own CSS in `@layer base` or `@layer components` unless you deliberately want it to override Tailwind.

@apply and @reference

`@apply` pastes the declarations of existing utilities into your own rule, which lets you name a combination once and use it on markup you cannot give lots of classes to:

```
@layer components {
  .btn-primary { @apply rounded-md bg-blue-600 px-4 py-2 text-white hover:bg-blue-700; }
}
```

```
.btn-primary {
  border-radius: var(--radius-md);
  background-color: var(--color-blue-600);
  padding-inline: calc(var(--spacing) * 4);
  padding-block: calc(var(--spacing) * 2);
  color: var(--color-white);
  &:hover { @media (hover: hover) { background-color: var(--color-blue-700); } }
}

measured: background blue-600; blue-700 while hovered
```

Variants work inside `@apply`, and an `!` at the end of a utility name produced `!important` (`@apply bg-red-500!` compiled to `background-color: var(--color-red-500) !important`; this form is not in the documentation, I tested it). The docs' own example is the right use: styling markup produced by something else (`.select2-dropdown { @apply rounded-b-lg shadow-md; }`).

Use `@apply` sparingly. It copies the declarations every time you use it, it moves your styling to a second file where you have to hunt for it, and it brings back the naming problem that utility classes were meant to remove. If the repetition is in *your own templates*, make a component (a

partial, a function, a Vue or React component) so the classes are written once. Reserve `@apply` for markup you do not control.

`@reference` is for CSS that lives in a different file from your Tailwind import: a Vue or Svelte `<style>` block or a CSS module. That file does not know Tailwind's theme, so `@apply` fails. I compiled the same rule three ways:

FILE STARTS WITH	RESULT	OUTPUT SIZE
(nothing)	`Error: Cannot apply unknown utility class `text-2xl`. Are you using CSS modules or similar and missing `@reference`?`	none: it fails
`@reference "tailwindcss";`	works: only the `.title` rule, with the theme values as fallbacks (`var(--text-2xl, 1.5rem)`)	672 bytes
`@import "tailwindcss";`	works, but includes Tailwind's whole theme and base layers a second time	5,305 bytes

So in a component file write `@reference "../app.css";` (the path to your main stylesheet, so your custom theme, utilities and variants are known) and never `@import "tailwindcss"` again.

CSS Functions and Variables in Your Own CSS

In ordinary CSS you can use the theme directly. Tailwind adds two functions, and the older `theme()` still works:

```
.fn-demo {
  color: --alpha(var(--color-lime-300) / 50%);
  margin: --spacing(4);
  background: var(--color-blue-500);
  border-radius: theme(borderRadius.lg);
}
```

```
--alpha(…)          color-mix(in oklab, var(--color-lime-300) 50%, transparent)   (with
--spacing(4)        calc(var(--spacing) * 4)          measured margin: 16px
var(--color-blue-500)   a plain theme variable
theme(borderRadius.lg)  0.5rem                         measured radius: 8px
```

`--alpha()` is the CSS equivalent of the `/50` opacity modifier, and `--spacing(4)` of `p-4`. `theme()` is the v3 way: it compiled in 4.3.3, but the documentation lists it as deprecated and recommends theme variables, so prefer `var(--radius-lg)`.

When to Write Plain CSS

SITUATION	USE
A one-off value that is not on the scale	an arbitrary value: `w-[372px]`
A value that changes at run time	a CSS variable: `w-(--w)` with `style="--w: …"`
A property or selector with no utility, used once	an arbitrary property or variant: `[mask-type:alpha]`, `[&>*]:p-2`
A low-level style used across the site that should work with variants	`@utility`
The same group of classes repeated in *your own* templates	a component in your framework, not CSS
Markup you cannot edit (a library's widgets, rendered Markdown)	`@layer components` with `@apply` or plain CSS
Long keyframe sequences, complex selectors, print layouts, anything that reads better as CSS	plain CSS in a layer, using theme variables

Mixing is normal and healthy: Tailwind does not ask you to give up CSS, only to reach for it last. The next chapters stay in the utility world; *Tailwind CSS Intermediate/Advanced 3* returns to custom utilities, variants and plugins in more depth.

Hands-On Exercises

EXERCISE 1

Use four off-scale techniques on one page and prove each: a width that is "the parent minus 2rem", a width taken from a CSS variable set in an inline style, a font size taken from a variable, and a variable-driven colour. Then show what Tailwind generates for an unhinted `text-(--v)` and explain why a size variable needs a hint.

View solution

EXERCISE 2

Write a static `scrollbar-hidden` utility and a functional `tab-*` utility that accepts a theme name (`tab-github` = 8), a number (`tab-4`) and a bracketed number (`tab-[3]`). Prove that both work with the `hover:` and `md:` variants, and then write a plain CSS rule that uses `@variant` for dark mode and hover and show the three resulting colours.

View solution

EXERCISE 3

A teammate wrote `.btn { padding: 4px }` in the stylesheet and now `p-8` on a button does nothing. Reproduce and measure the problem, fix it without changing the markup, and measure again. Then write a separate component CSS file that uses `@apply`, show the exact error you get without `@reference`, and compare the output sizes of the two fixes.

View solution

CHAPTER 10 QUICK REFERENCE

- Arbitrary value `w-[372px]`, `_` = space (`grid-cols-[1fr_auto]`), `calc()` gets its spaces; variable shorthand `w-(--w)`; the class must still appear whole in your files
- Ambiguous variables default to colour: use hints `text-(length:--sz)`, `bg-(image:--img)`, `bg-(length:--sz)`, `text-(color:--clr)`
- Arbitrary property `[mask-type:luminance]`; set variables per breakpoint `[--x:56px] lg:[--x:44px]`; arbitrary variants `[&>*]:p-2`, `[&_p]:mt-2`, `supports-[…]:`, `group-[.open]:`, `@[500px]:`, `[@media(…)]:`
- `@utility name { … }` (works with all variants); functional `@utility tab-* { tab-size: --value(--tab-size-*, integer, [integer]); }`; `@variant dark { … }` inside your own CSS; `@custom-variant` defines new variants
- **Unlayered CSS beats all utilities.** Put your CSS in `@layer base` / `@layer components`; then utilities can still override it (`card p-8` = 32px)
- `@apply` inlines utilities (variants and `!` work); use it for markup you cannot edit, not to avoid writing classes; in other files `@reference "../app.css"` (672 bytes vs 5,305 for a re-import)

- CSS functions: `--alpha(var(--color-x) / 50%)`, `--spacing(4)`, plain `var(--color-blue-500)`; `theme()` is deprecated
- Ladder: arbitrary value, arbitrary property/variant, `@utility`, component in your framework, `@layer components`, plain CSS

COMING NEXT

Tailwind CSS Fundamentals 11 covers motion: `transition-*` and `duration-*`, transforms (`scale`, `rotate`, `translate` and 3D), the built-in `animate-*` classes, and `starting:` for enter animations.

Tailwind CSS Fundamentals

Chapter 11 · Transitions, Transforms and Animation

Motion in CSS comes in three kinds, and Tailwind has a small set of classes for each. A **transition** smooths a change between two states (hover, focus, a class being toggled). A **transform** moves, scales, rotates or skews an element without disturbing the layout around it. An **animation** runs on its own, from keyframes, with no state change needed. The last part of the chapter is the one that matters most for real users: some people ask their system to reduce motion, and Tailwind has two variants for respecting that.

> **EVERY EXAMPLE HERE WAS RUN**
>
> Every class was compiled with Tailwind CSS 4.3.3 and the generated CSS read. The behaviours were measured in headless Chrome, by hovering with real mouse input, sampling the computed value part-way through a transition, inserting elements while the page was running, and emulating the system's reduced-motion setting. Values sampled mid-transition depend on exact timing, so yours will differ by a few units; the point is that they sit between the start and end values.

Transitions

A transition needs two things: a property list (*what* is allowed to animate) and a duration. Tailwind's `transition-*` classes set the property list and, apart from `transition-none`, also set a default duration of `150ms` and the easing `cubic-bezier(0.4, 0, 0.2, 1)`. Measured from the computed style:

CLASS	PROPERTIES IT TRANSITIONS
transition	colours (`color`, `background-color`, borders, outline, text-decoration, fill, stroke, gradient stops), `opacity`, `box-shadow`, `transform`, `translate`, `scale`, `rotate`, `filter`, `backdrop-filter`, `display`, `content-visibility`, `overlay`, `pointer-events`
transition-colors	the colour properties and gradient stops only
transition-opacity	`opacity`

CLASS	PROPERTIES IT TRANSITIONS
transition-shadow	`box-shadow`
transition-transform	`transform`, `translate`, `scale`, `rotate`
transition-all	`all` (every property that changes)
transition-none	`none`, with no duration set
transition-[height]	whatever you write (measured: `height`)
transition-(--my)	a CSS variable holding the list (measured with `--my: width`: `width`)

PLAIN `TRANSITION` DOES NOT ANIMATE WIDTH OR HEIGHT

The list above has no `width`, `height`, `margin` or `padding` in it. I hovered three boxes that grow from 40px to 160px, each with a 1000ms duration, and read the width 300ms in:

- `transition duration-1000`: `160px` at 300ms. It jumped; there was no animation at all.
- `transition-all duration-1000`: `124.5px` at 300ms, `160px` at the end. It animated.
- `transition-[width] duration-1000`: `121.156px` at 300ms, `160px` at the end. It animated.

Use `transition-[width]` (or `transition-all` for quick work) when the size changes. Animating layout properties is also the most expensive kind, which is why transforms below are usually the better choice.

Duration, Delay and Easing

`duration-300` is `transition-duration: 300ms`, and any whole number works (`duration-[1s,15s]` gives a separate time per property). `delay-200` waits before starting. The easing classes come from theme variables: `ease-in`, `ease-out` and `ease-in-out` are the three curves, and `ease-linear` is `linear`. For a custom curve write it in brackets: `ease-[cubic-bezier(.2,.8,.2,1)]`. One measured example:

```
<button class="bg-blue-500 text-white p-2
               transition-colors duration-300 ease-out
```

```
                hover:bg-red-500">Save</button>
```

```
transition-property:        color, background-color, border-color, … (the colour list)
transition-duration:        0.3s
transition-timing-function: cubic-bezier(0, 0, 0.2, 1)        (ease-out)

background-color, sampled while hovering:
  start          oklch(0.623 0.214 259.815)     blue-500
  after 100ms    oklab(0.63661 0.207186 0.0927007)   an in-between colour
  after 600ms    oklch(0.637 0.237 25.331)      red-500
```

The in-between colour in the middle sample is the proof that the browser is blending, not jumping. Delay and duration work together: `transition-opacity delay-200 duration-700 ease-in-out` measured `opacity`, `0.2s` delay, `0.7s` duration.

Transitions are written on the element that changes, and the *end* state is whatever variant applies: `hover:rotate-0` on an element that starts at `-rotate-6` (with `duration-1000`) measured `-4.80862deg` at 100ms, then `0deg`.

Transforms

In Tailwind v4 the transform utilities use the modern *individual* CSS properties, so they combine freely without one class overriding another, and each is transitioned separately. From the compiled output and the computed styles:

CLASS	CSS PROPERTY IT SETS	MEASURED
scale-110	`scale`	`scale: 1.1`; a 40px box measured 44 × 44px
scale-x-150	`scale`	`scale: 1.5 1`
rotate-45	`rotate`	`rotate: 45deg`; the bounding box of a 40px square grew to 56.57 × 56.57px
translate-x-4 - translate-y-2	`translate`	`translate: 16px -8px` (spacing scale)
translate-x-1/2	`translate`	`50%` of the element's own width
skew-x-12	`transform`	`matrix(1, 0, 0.212557, 1, 0, 0)`

CLASS	CSS PROPERTY IT SETS	MEASURED
origin-top-left	`transform-origin`	`0px 0px` instead of the default centre `20px 20px`

Two properties of transforms explain why they are preferred for motion. They do not change the element's place in the layout (its neighbours stay where they were), and browsers can animate them without recomputing the page layout. Add `transition-transform` (or plain `transition`) and a `hover:` variant and you have the standard lift, grow or slide effect. Variants combine: `hover:-translate-y-0.5`, `md:rotate-0`, `group-hover:scale-105`.

3D Transforms

Rotating around the X or Y axis needs depth, which in CSS comes from `perspective` on the *parent* (the distance from the viewer), and children only share a 3D space if `transform-style: preserve-3d` is set (`transform-3d`). The built-in perspectives are named distances; I read them from the compiled theme: `perspective-midrange` is `800px` and `perspective-distant` is `1200px`.

```
<div class="perspective-distant">
  <div class="size-10 bg-blue-500 transform-3d
              rotate-x-45 rotate-y-12 translate-z-4"></div>
</div>

<div class="size-10 bg-blue-500 backface-hidden rotate-y-180"></div>
```

```
parent perspective:       1200px
child transform-style:    preserve-3d
child translate:          0px 0px 16px
child transform:          matrix3d(0.978148, 0.147016, -0.147016, 0, 0, 0.707107, 0.7071
backface-hidden:          backface-visibility: hidden
perspective-origin-top:   perspective-origin: top   (measured 20px 0px)
```

`backface-hidden` is what makes a "flip card" work: put the front and back faces back to back, each with `backface-hidden`, and rotate the card 180 degrees on the Y axis. The matrix above shows the 3D rotation is really being applied; I did not build a full flip card for this chapter.

Animations

The four built-in animations are classes that set `animation: var(--animate-…)`; the theme variables hold the real shorthand. I measured each element's `animation-name`, duration, easing and loop count, and asked the browser how many animations it was running (`getAnimations().length`):

CLASS	MEASURED	USE FOR
animate-spin	`spin`, 1s, linear, infinite	loading spinners
animate-ping	`ping`, 1s, `cubic-bezier(0, 0, 0.2, 1)`, infinite	a notification dot that radiates outwards
animate-pulse	`pulse`, 2s, `cubic-bezier(0.4, 0, 0.6, 1)`, infinite	skeleton placeholders while content loads
animate-bounce	`bounce`, 1s, infinite	a "scroll down" arrow
animate-none	`animation: none`	turning one off at a breakpoint: `animate-none md:animate-spin`

Each of the four ran (`getAnimations().length` was 1). The responsive form works as expected: `animate-none md:animate-spin` measured `none` at a 700px viewport and `spin` at 900px.

For your own animation, define it in `@theme` together with its keyframes (the technique from chapter 9: `--animate-wiggle` plus `@keyframes wiggle`) and use `animate-wiggle`. You can also write the shorthand in brackets, `animate-[wiggle_1s_ease-in-out_infinite]`, and it reads the animation name and settings correctly (measured `wiggle`, 1s, ease-in-out, infinite) — but the bracket form *only names* the animation. On a test page that had no `@keyframes wiggle` anywhere, the browser reported **0** running animations. If an arbitrary animation does nothing, check that the keyframes exist.

Enter Animations with `starting:`

A transition needs a before and after, and a brand-new element has no "before". The `starting:` variant wraps its styles in CSS `@starting-style`, which supplies the first-frame values for an element's first render (or when it goes from `display: none` to visible). I compiled `starting:opacity-0` and the output is exactly `@starting-style { .starting\:opacity-0 {`

`opacity: 0%; } }`. Then I inserted an element into the running page with JavaScript and read its opacity:

```
<div class="opacity-100 starting:opacity-0 transition-opacity duration-500">…</div>
```

```
inserted with starting:opacity-0     opacity at 0ms: 0     at 150ms: 0.367318     at 700
inserted WITHOUT it                   opacity at 0ms: 1     at 150ms: 1            (it j
```

The official docs show it on a popover: `class="opacity-0 starting:open:opacity-0 …"`, combined with `open:`. I did not run that popover example; what I tested is the insertion case above. Because the default `transition` list includes `display`, and `transition-discrete` sets `transition-behavior: allow-discrete` (measured), the pieces for fading something in from `hidden` without JavaScript timers are in place; I did not test a `display` transition in this chapter. The docs' page for `starting` gives no browser-support notes, so check a support table before depending on it for something important.

will-change

`will-change-transform`, `will-change-scroll` (`scroll-position`), `will-change-contents`, `will-change-auto`, and the bracket form `will-change-[top,left]` (measured `top, left`) tell the browser in advance what is about to change so it can prepare. The official guidance is to use them as a last resort for a performance problem you have actually measured, not everywhere "just in case"; applied broadly they can make a page slower. If you need one, add it just before the change and go back to `will-change-auto` afterwards.

Respecting Reduced Motion

Some users get dizzy or sick from large movement and turn on "reduce motion" in their operating system. Browsers report this through `prefers-reduced-motion`, and Tailwind gives you the variants `motion-safe:` (`no-preference`) and `motion-reduce:` (`reduce`). I emulated the setting in Chrome and read the results:

CLASSES	NORMAL	REDUCED MOTION
motion-safe:animate-spin	`animation-name: spin`	`none`

CLASSES	NORMAL	REDUCED MOTION
animate-spin motion-reduce:animate-none	`spin`	`none`
transition … motion-reduce:transition-none	the full property list	`transition-property: none`
transition ease-linear motion-safe:hover:translate-x-10	mid-hover `translate: 17.3358px` (animating)	hover does nothing: `translate: none`
transition ease-linear hover:translate-x-10 motion-reduce:transition-none	animates	jumps straight to `40px`: the move happens, the animation does not

The last two rows are two different design decisions. With `motion-safe:` in front of the effect itself, reduced-motion users never get the movement. With `motion-reduce:transition-none`, they get the end state instantly, which is usually what you want when the change carries information (an item that moved, a panel that opened). Decorative loops such as `animate-pulse` and `animate-bounce` are the strongest candidates for `motion-safe:`. The docs' own pairing is `duration-300 motion-reduce:duration-0`.

A SENSIBLE DEFAULT

Prefer `transition-colors`, `transition-opacity` and `transition-transform` to `transition-all`: they say what moves, avoid accidentally animating something you did not intend, and keep the browser's work small. Keep durations short (150–300ms for interface feedback), put decorative looping animations behind `motion-safe:`, and never rely on motion alone to convey something.

Hands-On Exercises

EXERCISE 1

Build a button that turns from blue to red on hover over 300ms with an ease-out curve and show, by sampling the colour part-way, that it is blending. Then make a box grow from 40px to 160px wide on hover with three different transition classes (`transition`, `transition-all`, `transition-[width]`) and show which one fails to animate, and why.

View solution

EXERCISE 2

Make seven small squares showing scale, rotate, translate, skew and a changed origin, and read the computed property for each. Show that rotating a 40px square increases its bounding box. Then build the 3D case: a parent with perspective, a child with `transform-3d` rotated on X and Y and moved on Z, and a `backface-hidden` element rotated 180 degrees.

View solution

EXERCISE 3

Show all four built-in animations are running, make one respect reduced motion in two different ways (with `motion-safe:` and with `motion-reduce:`) and prove both by emulating the setting, then fade in an element inserted by JavaScript using `starting:` and compare it with the same element without. Finally write an arbitrary `animate-[…]` with no keyframes and show the browser runs nothing.

View solution

CHAPTER 11 QUICK REFERENCE

- Transitions: `transition` (colours, opacity, shadow, transforms, filters, display…), `transition-colors` / `-opacity` / `-shadow` / `-transform` / `-all` / `-none`, `transition-[prop]`, `transition-(--var)`; default 150ms and `cubic-bezier(0.4, 0, 0.2, 1)`
- **Plain `transition` does not animate width/height**: use `transition-[width]` or `transition-all`
- `duration-300`, `delay-200`, `ease-in` / `ease-out` / `ease-in-out` / `ease-linear`, `ease-[cubic-bezier(…)]`
- Transforms are the individual properties: `scale-110`, `rotate-45`, `translate-x-4`, `skew-x-12`, `origin-top-left`; they do not move neighbours in the layout
- 3D: `perspective-midrange` (800px) / `perspective-distant` (1200px) on the parent, `transform-3d`, `rotate-x-45`, `rotate-y-12`, `translate-z-4`, `backface-hidden`
- Animations: `animate-spin` / `-ping` / `-pulse` / `-bounce` / `-none`; custom ones need `@keyframes` (in `@theme`); `animate-[…]` without keyframes does nothing
- `starting:opacity-0` = `@starting-style`: the first-frame value so a newly inserted element can transition in; `transition-discrete` allows discrete properties like `display`
- `will-change-transform` etc.: a last resort for a measured problem

- Reduced motion: `motion-safe:` (no-preference) and `motion-reduce:` (reduce); `motion-reduce:transition-none` gives the end state instantly, `motion-safe:` removes the effect

COMING NEXT

Tailwind CSS Fundamentals 12 is the capstone, Building Real Components: buttons, cards, a form with validation states, a responsive navbar and a modal, why you reuse with components rather than `@apply`, and a landing page built from scratch.

Tailwind CSS Fundamentals

Chapter 12 · Building Real Components (Capstone)

The usual first complaint about utility classes is the long `class` attribute you have to repeat on every button. This chapter answers it properly and then puts the whole course to work: you will build a small landing page from components (a button with variants, a card), a responsive navbar, a form with validation states, and a modal dialog, and check what each does in a real browser. Along the way you meet the three small libraries almost every Tailwind project uses for composing class names: `clsx`, `tailwind-merge` and `class-variance-authority` (`cva`).

> **EVERY EXAMPLE HERE WAS RUN**
>
> The page in this chapter was generated by a Node script, compiled with Tailwind CSS 4.3.3 and opened in headless Chrome. The library behaviour shown is the real output of `clsx` 2.1.1, `tailwind-merge` 3.7.0 and `class-variance-authority` 0.7.1. Clicks, typing and keys were real input events. Two things were not tested: other browsers, and a screen reader (the accessibility checklist is a checklist, not a test result).

The Repeated-Markup Problem

A button needs about a dozen utilities, and a page has many buttons. Is that a problem? It depends where the repetition is, and the answer decides how you fix it:

SITUATION	IS IT A PROBLEM?	WHAT TO DO
A class list repeated across a loop (a list of cards, rows of a table)	No: it is written once in the template and the loop repeats it	nothing
The same element in several files or places (buttons, inputs, cards)	Yes: a change means editing everywhere	make it a **component** (a function, partial, include or framework component)
A piece of markup you do not control (rendered Markdown, a third-party widget)	Yes: you cannot add classes	`@layer components` and `@apply` (chapter 10)

SITUATION	IS IT A PROBLEM?	WHAT TO DO
One value repeated inside one file	Rarely	multi-cursor editing, or a theme variable (chapter 9)

So the answer to "I have repeated classes" is the same as for any other repeated code: **extract the thing, not the CSS**. A component keeps the markup, the classes and any behaviour together in one place, and works with every variant (`hover:`, `md:`) because those are still just classes. That is why the earlier recommendation was to use `@apply` sparingly: a component solves the problem without moving your styling into a second file.

Composing Class Names

Components need to build a class string from conditions and options. Three tiny libraries cover it, and each does one job. I ran all the outputs below in Node.

clsx: Conditional Classes

`clsx` joins strings, skipping anything false, null or empty, and turns an object of `{ class: condition }` into the classes whose condition is true:

```
import { clsx } from 'clsx';

clsx('px-4', false && 'x', { 'font-bold': true, hidden: false }, ['py-2', null, 'text-sm
clsx('a', undefined, null, 0, '', false, 'b');
```

```
"px-4 font-bold py-2 text-sm"
"a b"
```

tailwind-merge: Resolving Conflicts

Here is a fact that surprises people: **the order of classes in the `class` attribute does not decide which one wins.** The order of the rules in the generated stylesheet does. I measured it:

```
<div class="p-2 p-8">   <div class="p-8 p-2">
<div class="bg-red-500 bg-blue-500">   <div class="bg-blue-500 bg-red-500">
```

```
p-2 p-8                 padding 32px
p-8 p-2                 padding 32px            same, whichever order
bg-red-500 bg-blue-500  oklch(0.637 0.237 25.331)   red-500
bg-blue-500 bg-red-500  oklch(0.637 0.237 25.331)   red-500: same
```

That matters the moment a component accepts extra classes from outside (`class="px-8"` on a button that already has `px-4`): the caller's class is not guaranteed to win. `twMerge` fixes it by understanding Tailwind's classes, removing the earlier ones that conflict with a later one:

```
import { twMerge } from 'tailwind-merge';

twMerge('px-2 py-1 p-3');
twMerge('bg-red-500 bg-blue-500');
twMerge('p-3 px-2');
twMerge('text-sm text-red-500 text-lg');
twMerge('hover:bg-red-500 hover:bg-blue-500 bg-green-500');
```

```
"p-3"                       (p-3 covers px and py, so they go)
"bg-blue-500"
"p-3 px-2"                  (a later, more specific class does not remove the shorthand)
"text-red-500 text-lg"      (size and colour are different things: only text-sm is dropp
"hover:bg-blue-500 bg-green-500"   (variants are tracked separately)
```

The usual helper joins the two: `const cn = (...inputs) => twMerge(clsx(inputs));`. Use tailwind-merge 3.x with Tailwind v4 (its README says 3.7.0 supports Tailwind 4.0 up to 4.3, and v2.6.0 is for Tailwind v3).

cva: Variants as Data

`cva` turns a button's options into a function: a base string, named variant groups, defaults, and optional compound rules (`compoundVariants`):

```
import { cva } from 'class-variance-authority';

const btn = cva('inline-flex items-center rounded-md font-medium', {
  variants: {
    intent: { primary: 'bg-blue-600 text-white', danger: 'bg-red-600 text-white' },
    size:   { sm: 'px-2 py-1 text-sm', md: 'px-4 py-2' },
  },
```

```
  compoundVariants: [{ intent: 'danger', size: 'sm', class: 'uppercase' }],
  defaultVariants: { intent: 'primary', size: 'md' },
});

btn();
btn({ intent: 'danger', size: 'sm' });
btn({ class: 'px-8' });
twMerge(btn({ class: 'px-8' }));
```

```
"inline-flex items-center rounded-md font-medium bg-blue-600 text-white px-4 py-2"
"inline-flex items-center rounded-md font-medium bg-red-600 text-white px-2 py-1 text-sm
"inline-flex items-center rounded-md font-medium bg-blue-600 text-white px-4 py-2 px-8"
"inline-flex items-center rounded-md font-medium bg-blue-600 text-white py-2 px-8"
```

The third line is the reason the libraries are used together: `cva` chooses the classes, `twMerge` settles any conflicts. The full treatment of these libraries and of design systems built on them belongs to the next course; this is enough to build with.

The Components

The components are plain functions returning HTML, so they work in any build (a static generator, a server template). In React, Vue or Svelte they become framework components with the same `cn` and `cva` code inside. The button:

```
import { clsx } from 'clsx';
import { twMerge } from 'tailwind-merge';
import { cva } from 'class-variance-authority';

export const cn = (...inputs) => twMerge(clsx(inputs));

const buttonStyles = cva(
  'inline-flex items-center justify-center rounded-md font-medium transition-colors ' +
  'focus-visible:outline-2 focus-visible:outline-offset-2 ' +
  'disabled:pointer-events-none disabled:opacity-50',
  {
    variants: {
      intent: {
        primary:   'bg-blue-600 text-white hover:bg-blue-700 focus-visible:outline-blue-
        secondary: 'bg-gray-100 text-gray-900 hover:bg-gray-200 focus-visible:outline-gr
        danger:    'bg-red-600 text-white hover:bg-red-700 focus-visible:outline-red-600
```

```
      },
      size: { sm: 'px-3 py-1.5 text-sm', md: 'px-4 py-2 text-base' },
    },
    defaultVariants: { intent: 'primary', size: 'md' },
  },
);

export function Button({ intent, size, class: extra, label, ...attrs }) {
  const a = Object.entries(attrs).map(([k, v]) => (v === true ? k : `${k}="${v}"`)).join
  return `<button ${a} class="${cn(buttonStyles({ intent, size }), extra)}">${label}</bu
}

export function Card({ title, body }) {
  return `<article class="rounded-xl border border-gray-200 bg-white p-6 shadow-sm">
  <h3 class="text-lg font-semibold text-gray-900">${title}</h3>
  <p class="mt-2 text-gray-600">${body}</p>
</article>`;
}
```

```
Button({ type: 'button', label: 'Get started' })      background oklch(0.546 0.245 262.8
hover over it                                          background oklch(0.488 0.243 264.
Button({ size: 'sm', class: 'px-8', label: 'Override' })   padding-left 32px, padding-to
```

The last line shows the pattern working: the small size would give `px-3`, the caller asked for `px-8`, `cn` dropped the conflict and the caller won, while the small size's `py-1.5` and `text-sm` were kept (6px and 14px). Note also that Tailwind finds the class names in the generated page because they appear in full in the file it scans; the components never build a class name out of pieces.

The Page

Three parts of the page deserve a closer look, because each combines several ideas from the course.

A Responsive Navbar

```
<nav class="mx-auto flex max-w-5xl items-center justify-between p-4" aria-label="Main">
  <a href="#" class="text-xl font-bold">Acme</a>
  <ul id="links" class="hidden items-center gap-6 md:flex"> … </ul>
```

```
    <button id="menuBtn" type="button" aria-expanded="false" aria-controls="mobileMenu"
            class="rounded-md p-2 hover:bg-gray-100 md:hidden">Menu</button>
  </nav>
  <ul id="mobileMenu" class="hidden border-t border-gray-200 p-4 md:hidden"> … </ul>
```

```
  390px wide:   links display none, menu button block, mobile menu none, aria-expanded "fa
  click Menu:    mobile menu block, aria-expanded "true"
  click Menu:    mobile menu none, aria-expanded "false"
  900px wide:    links display flex, menu button none, mobile menu none
```

It is mobile-first (chapter 7): the desktop links are `hidden md:flex` and the button `md:hidden`. The only JavaScript flips `aria-expanded` and toggles the `hidden` class; the attribute is what tells assistive technology whether the menu is open. The feature-card grid below it uses `grid gap-6 sm:grid-cols-2 lg:grid-cols-3`: measured at 390px it is one column (`343px`), at 700px two, at 1100px three.

A Form with Validation States

```
  <input id="email" type="email" required aria-describedby="emailErr"
    class="peer block w-full rounded-md border border-gray-300 px-3 py-2
           invalid:border-yellow-500 user-invalid:border-red-500 user-invalid:bg-red-50">
  <p id="emailErr" class="hidden text-sm text-red-600 peer-user-invalid:block">
    Enter a valid email address.
  </p>
```

```
  page loaded, field empty
    :invalid true     :user-invalid false     border yellow-500   error message display no
  typed "abc", clicked elsewhere
    :user-invalid true   border red-500   background red-50   error message display block
  replaced with "me@example.com", clicked elsewhere
    :user-invalid false  border gray (the normal colour)   error message display none
```

This is the important difference between the two variants. `invalid:` matches as soon as the browser considers the field invalid, which for a `required` field is *before the user has typed anything*, so a red border there would greet people with an error on arrival. `user-invalid:` only matches after the person has interacted with the field. In my test the field was yellow on load and turned red only after typing a bad value and leaving it. The `peer` class on the input lets the

sibling message react to it (`peer-user-invalid:block`), and `aria-describedby` ties the message to the field for screen readers. I used `novalidate` on the form so the browser's own bubble does not compete with the page's message.

A Modal Dialog

```
<dialog id="dlg" aria-labelledby="dlgTitle"
        class="m-auto w-full max-w-sm rounded-xl p-6 backdrop:bg-black/50">
  <h2 id="dlgTitle" class="text-lg font-semibold">Delete account?</h2>
  …
  <button id="cancel" type="button" autofocus … >Cancel</button>
  <button id="confirm" type="button" … >Delete</button>
</dialog>

document.getElementById('open').addEventListener('click', () => dlg.showModal());
document.getElementById('cancel').addEventListener('click', () => dlg.close());
```

```
before:                 dlg.open false
click "Delete account": dlg.open true, focus moved to #cancel (the autofocus button)
backdrop:               oklab(0 0 0 / 0.5)   (backdrop:bg-black/50)
Tab, Tab, Tab:          confirm, then focus left the page's elements, then cancel
                        (it never landed on a button behind the dialog)
Escape:                 dlg.open false, focus returned to #open
```

The key decision is to use the native `<dialog>` element with `showModal()` rather than a `div` you show and hide. The browser provides the focus handling you would otherwise write by hand: focus moves into the dialog, the rest of the page cannot be reached, Escape closes it, and focus goes back to the button that opened it. Tailwind's `backdrop:` variant styles the dim layer behind it. Putting `autofocus` on the *safe* action (Cancel) means a keyboard user's first Enter does not delete anything. In the Tab test the middle step was the browser's own interface, which is normal for a modal dialog; I did not test with a screen reader.

An Accessibility Checklist

Tailwind styles things; it does not make them accessible. Before you ship a page, go through these (I built the example page to satisfy them):

CHECK	IN THIS CHAPTER
Use the right element	`button` for actions, `a` for navigation, `dialog` for modals, `nav` with `aria-label`
Visible keyboard focus	`focus-visible:outline-2 focus-visible:outline-offset-2` on the button, on the input too; never remove the outline without a replacement
State exposed to assistive technology	`aria-expanded` on the menu button, updated by the script
Labels and messages tied to fields	`label for`, `aria-describedby` for the error
Do not rely on colour alone	the error is a text message as well as a red border
Contrast	check text colours against their backgrounds with a tool; I did not measure contrast ratios here
Respect user preferences	`motion-safe:` / `motion-reduce:` (chapter 11), dark mode (chapter 8)
Responsive and zoomable	the viewport meta tag (chapter 7) and layouts that work at 390px

WHAT THIS PAGE PROVES, AND WHAT IT DOES NOT

The components, the responsive behaviour, the validation timing and the dialog's focus handling were all observed. Cross-browser behaviour (Safari and Firefox), screen-reader output and colour-contrast ratios were not tested, so run those checks before using this as a template for a real site.

Hands-On Exercises

EXERCISE 1

Prove that class order in the attribute does not decide the winner, then build a `cn()` helper from `clsx` and `tailwind-merge` and run the five conflict examples from this chapter. Show what happens to `px-4 px-8` before and after merging.

View solution

EXERCISE 2

Write a `Badge` component with `cva`: three colours, two sizes, a default of each, and a compound rule for one combination. Print the class string for four different calls, including one that passes an extra class that conflicts, and merge it.

View solution

EXERCISE 3

Build a sign-up form whose email field shows a yellow border while empty-and-required, a red border and an error message only after the user has typed a bad value and left the field, and returns to normal when fixed. Show the `:invalid` and `:user-invalid` state at each of the three moments, and explain why `invalid:` alone would be wrong for the error colour.

View solution

CHAPTER 12 QUICK REFERENCE

- Repeated markup: extract a **component** (function, partial, framework component), not CSS; `@apply` only for markup you cannot edit
- `clsx` joins and filters; `twMerge` removes conflicting Tailwind classes (`px-2 py-1 p-3` becomes `p-3`); `cn = (...a) => twMerge(clsx(a))`; use tailwind-merge 3.x for Tailwind v4
- Class order in the attribute does **not** decide the winner; stylesheet order does, so merge conflicts before they reach the page
- `cva(base, { variants, compoundVariants, defaultVariants })` returns a function from options to a class string
- Never build class names from fragments; the full names must appear in files Tailwind scans
- Forms: `invalid:` matches immediately, `user-invalid:` after interaction; `peer` + `peer-user-invalid:` shows a message; add `aria-describedby`
- Modal: native `<dialog>` + `showModal()` gives focus handling, Escape and focus return; style the dim layer with `backdrop:`; `autofocus` the safe button
- Navbar: `hidden md:flex` + `md:hidden` button with `aria-expanded` and `aria-controls`

Course Complete: What You Can Now Do

CHAPTER	YOU CAN NOW…
1 · Why Utility-First, and Getting Set Up	explain the approach and set up Tailwind v4 with the CLI or Vite
2 · The Scale: Spacing, Sizing and Typography	use the spacing scale, sizing and type utilities fluently
3 · Colour, Backgrounds and Borders	apply colour, gradients, borders, rings and shadows
4 · Layout: Display, Flexbox and Grid	build layouts with flex and grid
5 · Positioning, Stacking and Overflow	position, stack and clip elements
6 · State Variants: Hover, Focus and Friends	style states, groups, peers and form states
7 · Responsive Design and Container Queries	design mobile-first with breakpoints and container queries
8 · Dark Mode and Preference Variants	support dark mode and user preferences
9 · Your Theme: @theme, Variables and Fonts	customise the design system in CSS
10 · Going Beyond the Utilities	use arbitrary values, custom utilities and layers, and know when to write CSS
11 · Transitions, Transforms and Animation	add motion that respects reduced-motion settings
12 · Building Real Components	compose classes, build components, and assemble an accessible landing page

COMING NEXT

The follow-on course is *Tailwind CSS Intermediate/Advanced*. It opens with how the v4 engine works, then covers design tokens and multiple themes, custom utilities and plugins, advanced variants, framework integration, component patterns, migrating from v3, and a themed component kit. Start with *Tailwind CSS Intermediate/Advanced 1*.