Tailwind CSS Course

Flavio Copes

Updated August 8, 2026

Contents

Learn the utility-first approach, spacing, layout, typography, responsive states, customization, and common Tailwind CSS pitfalls.

What you'll learn: Translate deliberate CSS design decisions into maintainable Tailwind utility classes.

Why utilities

Understand utility-first composition and its tradeoffs.

Why Tailwind CSS

Understand the utility-first approach, what Tailwind generates, and the tradeoff between composing classes in HTML and writing custom CSS.

Tailwind gives you small classes that each apply one CSS decision.

For example:

```
<article class="max-w-sm rounded-lg border border-gray-200 p-6">
  <h2 class="text-lg font-semibold text-gray-950">Deploy complete</h2>
  <p class="mt-2 text-sm leading-6 text-gray-600">
    The new version is live.
  </p>
</article>
```

Each class is a small CSS decision: maximum width, border, padding, font size, weight, color, or margin. Tailwind generates CSS for the utilities it detects in your source. The browser still receives ordinary CSS.

The utility-first tradeoff is **where composition happens**. Instead of inventing `.notification-card`, moving to a stylesheet, and switching back to markup, you compose the visual rules where the element is created. That makes dependencies visible and makes deletion safer: removing the element removes its utility usage too.

Tailwind also constrains common decisions through a shared theme. Reusing `p-6`, `text-sm`, and `rounded-lg` is more consistent than choosing unrelated values for every component. Responsive and state variants keep related decisions together:

```
<button class="bg-blue-600 px-4 py-2 text-white hover:bg-blue-700 md:px-6">
  Continue
</button>
```

There are costs. Class lists can become dense, repeated patterns need component extraction, and a utility name is not a substitute for understanding layout, cascade, accessibility, or responsive design.

Do not learn Tailwind by memorizing a catalog. Start with the CSS decision—"this parent should be a grid," "this text needs a readable line height"—then find the utility that expresses it. DevTools

can always show the generated declaration.

Your action is to build a small alert with six utilities. For each class, write the CSS property it controls. Remove one class at a time and use the computed-style panel to verify your prediction.

Utilities map to CSS

Translate common Tailwind utilities back to their CSS properties so the class names become predictable instead of something to memorize blindly.

Every utility maps to CSS.

```
<div class="block w-full p-4 text-center">Hello</div>
```

The generated rules correspond to familiar declarations:

```
display: block;
width: 100%;
padding: 1rem;
text-align: center;
```

Some utilities use theme variables rather than copying a literal value. `p-4` is connected to Tailwind's spacing system, while `text-gray-700` resolves through a color token. This is why changing a theme token can update many utility uses consistently.

Variants wrap or qualify the same rule:

```
<div class="p-4 hover:bg-gray-100 md:p-8">...</div>
```

- `p-4` applies at every viewport size
- `hover:bg-gray-100` applies while the element matches the hover condition
- `md:p-8` applies from the `md` minimum width upward and replaces the base padding there

Avoid conflicting utilities for the same property and condition:

```
<div class="p-2 p-6">...</div>
```

The winning rule follows Tailwind's generated stylesheet and CSS cascade, not a promise that the last class in the HTML string wins. Express one decision per property at a given state, or use an explicit variant when the value should change.

Utilities can also collaborate on one CSS feature. `translate-x-2`, `rotate-3`, and `scale-95` contribute transform values. Removing one does not necessarily remove the others.

When a class is unfamiliar, inspect the matched rule and computed value in DevTools. Ask which property, selector, media query, or custom property it represents.

Your action is to inspect the example at widths below and above `md`. Record the matched padding rules, cross one out in DevTools, and explain the winner using CSS conditions and cascade rather than class-string order.

Try Tailwind in the browser

Use the Tailwind Play CDN for a quick experiment, while understanding why a real production project should use a build integration.

The fastest way to experiment is the Play CDN.

```
<!doctype html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
</head>
<body class="p-8">
  <h1 class="text-3xl font-bold">Hello Tailwind</h1>
</body>
</html>
```

Save this as `index.html` and open it in a browser. Change one utility, reload, and inspect the generated rule. This is useful for learning because there is no project setup between an idea and the result.

You can also try Tailwind's CSS features in the page:

```
<style type="text/tailwindcss">
  @theme {
    --color-brand: #2457d6;
  }
</style>

<button class="bg-brand px-4 py-2 text-white">Save</button>
```

The Play CDN runs Tailwind in the browser. Tailwind's documentation limits it to development and experiments; it is not a production delivery strategy. It adds runtime work, depends on a remote script, and does not give you the controlled static CSS artifact produced by a normal build.

A production integration scans known sources during development/build and emits CSS ahead of time. The finished page loads that CSS without needing Tailwind itself in the browser.

Even in a quick experiment, keep real HTML semantics. Use a `<button>` for an action, a real heading for structure, and visible keyboard focus. Fast styling should not turn into inaccessible markup.

Your action is to build a heading, paragraph, and button in one Play CDN file. Add a custom brand token, test keyboard focus, inspect one generated rule, then list what must change before shipping the page.

Install Tailwind CSS with Vite

Set up the current Tailwind CSS Vite integration and import Tailwind with the CSS-first workflow introduced in Tailwind CSS v4.

Install Tailwind and its Vite plugin:

```
npm install tailwindcss @tailwindcss/vite
```

Add the plugin to `vite.config.js`:

```
import { defineConfig } from 'vite'
import tailwindcss from '@tailwindcss/vite'

export default defineConfig({
```

```
  plugins: [tailwindcss()],
})
```

Then import Tailwind from the CSS entry used by your application:

```
@import "tailwindcss";
```

Make sure that CSS entry is actually linked from the HTML or imported by the JavaScript entry. A correct Vite plugin cannot style a page that never loads the resulting stylesheet.

Start the dev server and add one obvious utility:

```
<h1 class="text-3xl font-bold text-blue-700">Tailwind is running</h1>
```

Tailwind v4 automatically detects normal project source files, so a new basic setup does not need the v3 `content` array. Detection starts from the build's working directory and ignores sources such as `node_modules` and files excluded by `.gitignore`. Monorepos and external component libraries can need explicit source registration.

The build-time model matters: Tailwind finds complete candidates, generates the required CSS, and Vite serves or bundles that CSS. Your application does not call Tailwind at runtime.

Use the installation guide for your actual framework when it has one. A framework can have a different CSS entry, Vite integration point, or server-rendering build. Do not copy a generic config over working framework configuration blindly.

Verify both modes:

```
npm run dev
npm run build
```

In DevTools, confirm the utility rule comes from the generated stylesheet. In the production output, confirm the CSS asset exists and the page references it.

Your action is to install Tailwind in a disposable Vite project, add one base and one hover utility, run development and production builds, and prove the production page works with no Play CDN script.

How class detection works

Understand why Tailwind scans complete class names in source files and why building class names dynamically can leave required CSS out of the build.

Tailwind scans your source as text and finds complete class names.

This works:

```
const colors = {
  success: 'bg-green-600 text-white',
  error: 'bg-red-600 text-white',
}
```

This does not work reliably:

```
const className = `bg-${color}-600`
```

Tailwind does not execute your JavaScript or understand interpolation. It sees text tokens. Since `bg-green-600` and `bg-red-600` never appear completely, their CSS may not be generated.

Map application states to complete, validated class strings:

```
const colorClasses = {
  success: 'bg-green-600 text-white hover:bg-green-700',
  error: 'bg-red-600 text-white hover:bg-red-700',
}

const className = colorClasses[state]
```

This is also a better component API. Callers choose a semantic state, while the component owns the actual design decisions.

Tailwind v4 automatically ignores common non-source inputs including `node_modules`, binary files, lockfiles, and files excluded by `.gitignore`. If a real component library is outside detection, register it explicitly in CSS:

```
@import "tailwindcss";
@source "../node_modules/@acme/ui";
```

In a monorepo, set an intentional base with `source()` or add explicit sources. Use `@source inline()` only when a generated class genuinely cannot appear in static source. A huge safelist hides design and build mistakes and increases output.

When a utility is missing, search the compiled CSS for its escaped selector. If it is absent, investigate detection. If it exists but does not apply, investigate the selector, variant condition, and cascade instead.

Your action is to reproduce a missing dynamic color class, replace it with a complete mapping, and confirm the rule appears in built CSS. Then identify one ignored path in your project and decide whether it should stay ignored or become an explicit source.

Check your understanding: utility-first Tailwind

Check class generation, composition, responsive variants, arbitrary values, and how Tailwind relates to the CSS it produces.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Spacing and sizing

Use the spacing scale, dimensions, borders, and backgrounds.

Use the spacing scale

Apply consistent padding, margin, gap, width, and height values with Tailwind spacing utilities and understand the scale behind them.

Spacing utilities share one scale.

```
<section class="p-6">
  <h2 class="mb-2">Latest articles</h2>
  <div class="grid gap-4">...</div>
```

```
</section>
```

`p-6`, `mb-2`, and `gap-4` use related design tokens. A shared scale makes spacing more consistent than unrelated one-off values.

Read the prefix as a CSS decision:

- `p-6` adds padding on every side inside the section
- `mb-2` adds margin after the heading in a horizontal writing mode
- `gap-4` adds space between Grid or Flex items without adding space outside the container

The number is a position in the spacing system, not a pixel count. Tailwind's generated rule resolves through its theme, so the same step can be shared by padding, margin, gap, width, and other utilities.

Choose the owner of the space. A stack is usually clearer when the parent owns the repeated gap:

```
<ul class="grid gap-4">
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ul>
```

Adding `mb-4` to every child leaves an unwanted margin after the last item and spreads layout responsibility across the children.

Use padding for space between a box's content and its boundary. Use margin for a relationship outside one box. Use `gap` for spacing between children participating in a Flexbox or Grid layout. These are different box-model operations even when they look similar in one screenshot.

Avoid solving every alignment problem with larger spacing. Unexpected space can come from browser heading margins, line height, Grid tracks, or margin collapse. Inspect the box model before changing a class.

Your action is to build the same three-item stack once with child margins and once with parent `gap`. Add and remove an item, inspect the outer edges, and explain which element owns every space.

Use logical spacing utilities

Use inline and block spacing utilities when a component should follow writing direction instead of assuming left and right sides.

Tailwind includes logical spacing utilities.

`ms-4` adds margin at the **inline start**. In left-to-right English that is the left side; in right-to-left Arabic it is the right side. `me-4` targets the inline end. Padding equivalents include `ps-4` and `pe-4`.

Compare a directional icon gap:

```
<a class="inline-flex items-center" href="https://flaviocopes.com/next">
  <span>Next</span>
  <svg class="ms-2 size-4" aria-hidden="true">...</svg>
</a>
```

The icon stays after the text in either writing direction. A physical `ml-2` always means left margin and can put the gap on the wrong side in RTL content.

Not every horizontal utility is logical. `px-4` deliberately applies equal padding to the physical left

and right sides, which is also equal on the inline axis in common horizontal writing modes. Use it when both sides should match. Choose start/end when the two sides have different semantic roles.

Logical spacing does not reverse the DOM or automatically mirror icons. Reading order should remain meaningful in the markup, and directional symbols may need their own RTL treatment. Test the actual component with `dir="rtl"` rather than assuming the spacing class completes localization.

Positioning has logical utilities too, such as `start-0` and `end-0`. Apply the same question: is this edge physical, or should it follow the writing direction?

Your action is to create a labeled icon control, switch an ancestor between `dir="ltr"` and `dir="rtl"`, and compare `ml-2` with `ms-2`. Record which parts change and which still require an explicit design decision.

Width, height, and maximum size

Size elements with fixed scale values, fractions, full available space, viewport values, and readable maximum widths.

Use `w-full` when a box should fill available width. Use fractions such as `w-1/2` when the relationship is deliberate.

"Full" means 100% of the containing block, not automatically the viewport. Padding, Grid tracks, Flexbox sizing, and the parent's own width all affect the final pixels.

Maximum-width utilities keep content readable:

```
<article class="mx-auto max-w-prose px-4">...</article>
```

`mx-auto` centers the constrained box. `px-4` preserves space on narrow screens.

This pattern combines three decisions:

- `w-full` can let a control consume the available inline space
- `max-w-prose` prevents long lines from becoming uncomfortable
- `mx-auto` distributes leftover horizontal margin when the box is narrower than its container

Height needs a definite reference. `h-full` can only resolve usefully when the parent has a defined height. For a page that should reach the visible screen, `min-h-dvh` often expresses the intention better than forcing every ancestor to `h-full`; dynamic viewport units account for mobile browser UI more accurately than older fixed viewport assumptions.

Flex and Grid items have automatic minimum sizes. A long filename can overflow even inside a seemingly flexible row. `min-w-0` tells a flexible item it may shrink below its content's intrinsic width:

```
<div class="flex max-w-md gap-3">
  <div class="min-w-0 flex-1">
    <p class="truncate">a-very-long-file-name-that-must-fit.txt</p>
  </div>
</div>
```

Do not use a fixed height just to make cards look equal when content can wrap, zoom, or translate. Let layout tracks align items and use minimum sizes when the content may legitimately grow.

Your action is to place a long unbroken label inside a flexible row, observe the overflow, then add `min-w-0`. Test the readable article at narrow and wide widths and explain which box constrains it.

Borders, radius, and focus rings

Add borders and rounded corners while using ring and outline utilities to keep keyboard focus visible.

Borders use familiar pieces:

```
<div class="rounded-md border border-gray-300 p-4">...</div>
```

A border contributes to the box model; a radius clips the visual corner. Choose radius from the design language rather than adding `rounded-*` to every object. Adjacent controls may need coordinated corners instead of each looking like a separate pill.

Rings and outlines are especially useful for state that should sit outside the control without changing its layout. For interactive controls, keep a visible keyboard focus state:

```
<button class="rounded-md bg-blue-600 px-4 py-2 text-white
               focus-visible:outline-2 focus-visible:outline-offset-2
               focus-visible:outline-blue-600">
  Save
</button>
```

`focus-visible` normally avoids showing the custom indicator for pointer interactions while preserving it for keyboard-like focus. Do not remove the browser outline unless an equally clear replacement is present.

Check the indicator against every surface, including dark mode and error backgrounds. A two-pixel blue outline can disappear on a blue panel. Offset can create separation, while a contrasting outer ring or different color can make the state visible.

State cannot rely on color alone. An invalid input should have text or an icon connected with `aria-describedby`, not only a red border. Disabled controls need both a real `disabled` attribute and styling; opacity by itself does not prevent interaction.

Use DevTools to distinguish border, outline, and box-shadow-based ring rules. If a focus style is absent, first verify the element can receive focus and that the expected state variant matches.

Your action is to create a button and invalid input. Navigate with Tab in light and dark themes, zoom to 200%, and verify that focus and error states remain visible without relying only on color.

Use arbitrary values carefully

Apply a one-off CSS value with bracket notation and decide when a repeated value should become a theme token instead.

Bracket notation accepts a one-off value:

```
<div class="grid-cols-[16rem_1fr]">...</div>
```

Underscores represent spaces where a class token cannot contain them, so this becomes `grid-template-columns: 16rem 1fr`. Arbitrary values still use Tailwind's variant and utility system:

```
<aside class="w-[18rem] lg:w-[22rem]">...</aside>
```

You can also write a property Tailwind does not expose directly:

```
<div class="[content-visibility:auto]">...</div>
```

Use these forms for a real exception, a value supplied by an external design, or a quick experiment. They keep the declaration local but bypass the shared theme scale.

If the same value appears repeatedly or represents a design concept, give it a theme name. `w-[18rem]` copied across ten files is still a magic value. A named container or spacing token communicates why the number exists and makes later changes safer.

For an existing CSS custom property, current Tailwind supports concise custom-property forms in many utilities, such as `w-(--panel-width)`. This preserves a runtime value without constructing a class name dynamically.

Do not insert untrusted text into an arbitrary class. A class string is source code for styling, not a safe place for user-supplied CSS. Map allowed application values to complete classes.

Arbitrary values can also hide a wrong layout model. Before adding `left-[137px]`, ask whether Grid alignment, gap, or normal flow should own the placement.

Your action is to use one arbitrary track definition, inspect its generated CSS, then use it in a second component. Decide whether repetition revealed a theme token or whether the value is still a legitimate exception.

Flexbox and Grid

Translate familiar CSS layouts into utilities.

Build a Flexbox layout

Translate the main Flexbox container and item properties into Tailwind utilities for rows, columns, alignment, wrapping, and flexible sizing.

Put Flexbox utilities on the parent:

```
<nav class="flex flex-wrap items-center justify-between gap-4">...</nav>
```

Flexbox is a one-dimensional layout model. The parent distributes its direct children along a **main axis** and aligns them on the **cross axis**.

With the default `flex-row`, the main axis is horizontal:

- `justify-between` distributes free space along the row
- `items-center` aligns items vertically on the cross axis
- `gap-4` creates consistent space between children
- `flex-wrap` allows items to form another line when the row is too narrow

Change to `flex-col` and the axes rotate. `justify-*` now acts vertically, while `items-*` acts horizontally. Memorizing "justify means horizontal" produces bugs as soon as direction changes.

Items also negotiate size:

```
<div class="flex gap-4">
  <aside class="w-64 shrink-0">Filters</aside>
  <main class="min-w-0 flex-1">Results</main>
</div>
```

`flex-1` lets the main area grow and shrink. `shrink-0` protects the fixed sidebar. `min-w-0` lets long content inside the main area shrink instead of overflowing due to the automatic minimum size.

Use `justify-between` only when the free space belongs between the items. For a navigation bar with many links, a flexible spacer or grouped regions can preserve more predictable relationships than spreading every child across the full width.

Inspect the Flexbox overlay in DevTools. It shows axes, gaps, free space, and item sizes more clearly than changing classes at random.

Your action is to build a toolbar that becomes `flex-col` on narrow screens and `sm:flex-row` when wider. Predict both axes before changing alignment, then test one long label and explain every shrink decision.

Build a Grid layout

Create explicit and responsive Grid tracks with Tailwind utilities, fractional columns, gaps, spans, and arbitrary track definitions.

Create three equal columns like this:

```
<div class="grid grid-cols-3 gap-6">...</div>
```

Grid is a two-dimensional layout model. The parent defines rows and columns called **tracks**; direct children are placed into the resulting cells.

`grid-cols-3` creates three `minmax(0, 1fr)` columns, so each receives an equal share of available width. Three columns are not automatically responsive. On a narrow screen, use a base layout and add tracks only when space exists:

```
<div class="grid gap-6 sm:grid-cols-2 lg:grid-cols-3">...</div>
```

Items can span tracks when the content hierarchy calls for it:

```
<article class="sm:col-span-2">Featured story</article>
```

Do not use spans to recreate pixel coordinates. Let source order remain meaningful for keyboard and screen-reader navigation; visual placement should not turn the DOM into a puzzle.

For a component grid that should add columns based on available card width, an intrinsic track definition can remove viewport-specific guesses:

```
<div class="grid grid-cols-[repeat(auto-fit,minmax(16rem,1fr))] gap-6">
  ...
</div>
```

Here each card needs roughly `16rem`; `auto-fit` creates as many columns as fit, and `1fr` shares leftover space. Test the minimum against real translated and zoomed content.

Use Flexbox when one axis drives the layout. Use Grid when rows and columns must coordinate or when the container should define tracks. Both can be nested.

Your action is to build the same card list with responsive explicit tracks and with `auto-fit/minmax`. Resize the container, inspect the Grid overlay, and explain which behavior better matches the content requirement.

Position elements

Use relative, absolute, fixed, sticky, and inset utilities while keeping normal document flow as the default layout tool.

Anchor an absolute badge to a card:

```
<article class="relative">
  <span class="absolute end-2 top-2">New</span>
</article>
```

`relative` leaves the article in normal flow and establishes the containing block for its positioned descendant. `absolute` removes the badge from normal flow, while `end-2` and `top-2` set logical-inline and physical-block offsets.

The other position modes solve different problems:

- `static` uses normal positioning
- `relative` keeps the element's original space and can anchor descendants
- `absolute` positions against a containing block and does not reserve space
- `fixed` positions against the viewport
- `sticky` behaves in flow until its inset threshold is reached inside the relevant scroll container

A sticky header needs an inset such as `top-0`. It can appear broken when an ancestor creates the scrolling area, clips overflow, or does not provide enough scroll distance.

Positioned elements can overlap content. Add space for a badge if it could cover a heading, and test text zoom and long translations. If "New" carries meaning, keep it in the accessibility tree; decorative overlays should be hidden appropriately.

`z-10` changes stacking order only within the applicable stacking context. Transforms, opacity, isolation, and positioned ancestors can create new contexts, so an enormous z-index does not always escape an ancestor. Inspect stacking contexts before escalating numbers.

Use normal flow, Flexbox, or Grid for primary page structure. Absolute coordinates couple the layout to one content size and viewport.

Your action is to add an absolute badge and sticky section heading. Increase text to 200%, add a long title, and place the component inside an overflow container. Record which containing block and scroll container control each result.

Center content deliberately

Choose the correct Flexbox, Grid, margin, or text-alignment utility based on what you are centering and along which axis.

Center one child on both axes with Grid:

```
<div class="grid min-h-screen place-items-center">...</div>
```

`place-items-center` aligns Grid items on both axes. `min-h-screen` gives the grid at least viewport height, although `min-h-dvh` can better track changing mobile browser chrome in modern layouts.

"Center this" is not one CSS operation. First identify the box and axis:

- center a constrained block horizontally: `mx-auto max-w-xl`
- center inline text inside its box: `text-center`
- center Flex children on both axes: `flex items-center justify-center`
- center a Grid item in its cell: `place-self-center`
- center all Grid items in their cells: `place-items-center`

In the default Flexbox row direction, `items-center` centers vertically and `justify-center` cen-

ters horizontally. `flex-col` swaps those axes: `justify-center` then centers vertically and `items-center` horizontally.

`mx-auto` needs leftover inline space. A `w-full` block already occupies all available width, so its automatic margins have nothing to distribute. `text-center` changes inline content alignment; it does not center the paragraph box itself.

Vertical centering can conflict with overflow. A sign-in card centered in a fixed-height viewport may push content offscreen when text grows. Prefer a minimum height plus page padding so the layout can expand and scroll:

```
<main class="grid min-h-dvh place-items-center p-6">
  <form class="w-full max-w-md">...</form>
</main>
```

Do not center everything by reflex. Long prose is usually easier to scan left-aligned even when its container is centered.

Your action is to center a form box while keeping its labels left-aligned. Test a short and very tall form at 200% zoom, then explain why `min-h-dvh` plus padding is safer than a fixed height.

Build a responsive card layout

Combine Grid, gap, width, padding, borders, and content alignment to build a reusable card list without custom component CSS.

Build the smallest layout first:

```
<ul class="grid gap-6 md:grid-cols-2 lg:grid-cols-3">
  <li>
    <article class="flex h-full flex-col rounded-lg border border-gray-200 p-6">
      <h2 class="text-lg font-semibold">CSS layout</h2>
      <p class="mt-2 leading-6 text-gray-600">Build resilient interfaces.</p>
      <a class="mt-auto pt-6 font-medium text-blue-700 focus-visible:outline-2"
         href="https://flaviocopes.com/css/">
        Read the course
      </a>
    </article>
  </li>
</ul>
```

The base layout is one column. More tracks appear when the viewport reaches each breakpoint.

The list semantics describe a collection. Grid belongs on the list because it controls the cards. Each card uses a vertical Flexbox: `h-full` fills the Grid item's track, `flex-col` stacks content, and `mt-auto` pushes the action to the end without fixed card heights.

Equal card bottoms are a layout choice, not a reason to truncate useful descriptions. Test a card with a two-line title, long translated text, and no image. The track should grow without covering the action.

Make the real interactive element obvious. A visible link gives keyboard focus and a meaningful accessible name. If the entire card must be clickable, avoid nested links and buttons; restructure the markup rather than adding click handlers to the `<article>`.

Responsive variants are minimum-width conditions. Do not choose `md` and `lg` from habit. Resize with real content and add a column where the cards still meet their useful minimum width. A container query or `auto-fit` grid can be better when the card list lives in several page regions.

Use one spacing system: outer `gap-6`, inner `p-6`, and small content relationships such as `mt-2`. This creates rhythm without one-off margins on every child.

Your action is to render six cards with deliberately uneven content. Test keyboard focus, 200% zoom, and narrow containers. Remove every fixed height and explain which Grid and Flex rules produce the final alignment.

Check your understanding: layout and spacing

Check spacing utilities, dimensions, Flexbox, Grid, alignment, gaps, positioning, and common layout patterns.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Typography and color

Style readable text and accessible color systems.

Set text size and weight

Create typographic hierarchy with text-size, font-weight, line-height, letter-spacing, and readable-width utilities.

A heading can combine several typography decisions:

```
<h1 class="text-4xl font-bold leading-tight tracking-tight">Build better pages</h1>
```

These utilities control separate CSS properties:

- `text-4xl` chooses a font-size token
- `font-bold` chooses a font-weight value
- `leading-tight` controls line height
- `tracking-tight` controls letter spacing

Use only the decisions the text needs. Tight line height can work for a short display heading but makes a multi-line paragraph difficult to read. Small text often needs more line height, not simply a smaller version of heading styles.

Tailwind also supports paired size and line-height forms such as `text-base/7`:

```
<p class="text-base/7 text-gray-700">A readable paragraph...</p>
```

Build a small type system instead of choosing sizes independently on each page. A title, section heading, body, and supporting label should have clear roles and restrained differences in size, weight, and color.

HTML structure and visual hierarchy are related but not interchangeable. Keep heading levels in document order. A heading can be styled smaller without changing its semantic level, and a large price is not automatically a heading.

Responsive typography should respond to a real need. Large display type may need `text-4xl md:text-6xl`, but body text rarely benefits from dramatic viewport jumps. Test long words, translated strings, zoom, and user font settings.

Not every font provides every declared weight. The browser can synthesize a weight, which may look different from the intended face. Load the weights you use and inspect the rendered font in DevTools.

Your action is to define four text roles using no more than four sizes and three weights. Test a long heading at narrow width and 200% zoom, then explain every line-height choice.

Use font families

Choose built-in font stacks and add a project font with Tailwind CSS v4 theme variables when the design genuinely needs one.

Built-in classes include `font-sans`, `font-serif`, and `font-mono`.

Each resolves to a font-family stack, not one guaranteed installed font. A stack gives the browser fallbacks when the preferred face is unavailable.

In Tailwind v4, add a project font to the theme in CSS:

```
@theme {
  --font-display: "Geist", ui-sans-serif, system-ui, sans-serif;
}
```

You can then use `font-display`:

```
<h1 class="font-display text-4xl font-semibold">Products</h1>
```

The theme variable creates a utility because its name uses Tailwind's `--font-*` namespace. A regular `:root` variable is enough when you need a CSS variable but do not want to expand the utility API.

Defining the stack does not load the font. Add a correct `@font-face`, framework font integration, or hosted stylesheet separately. Match the declared weight and style to the actual files. Asking for `font-bold` when only a regular file is loaded can trigger synthetic bolding.

Font loading affects performance and layout. A fallback with very different metrics can cause text to reflow when the custom font arrives. Prefer a small number of necessary files, suitable caching, and a `font-display` strategy that keeps content readable. Consider privacy and availability before depending on a third-party font host.

Use fonts by role. A display face can distinguish headings while body copy keeps a familiar system stack. Monospace communicates code or aligned technical data; it is not automatically more readable for every label.

Your action is to add one local display font with a system fallback. Disable the font request in DevTools, compare layout shift and readability, and verify that every requested weight has a real font file.

Apply colors and opacity

Use palette utilities for text, backgrounds, borders, and opacity modifiers while preserving sufficient visual contrast.

Color utilities combine a property, color name, and shade:

```
<p class="border-blue-200 bg-blue-50 text-blue-950">Saved</p>
```

The prefixes target different CSS properties: `text-*`, `bg-*`, `border-*`, `fill-*`, and `stroke-*`. Reusing the same shade number across them does not guarantee the same visual importance because area, neighboring colors, and display conditions change perception.

Slash modifiers add alpha to that color:

```
<div class="bg-black/50 text-white">Overlay label</div>
```

This makes the background color translucent while leaving the text opaque. `opacity-50` would reduce the opacity of the entire element, including its children, and can destroy text contrast.

Contrast must be checked against the **composited** background. `text-white` over `bg-blue-600/70` can land on different page colors. Test normal, hover, focus, disabled, selected, and dark states—not just the default screenshot.

Color should not be the only signal. Pair error red with text or an icon, selected color with shape or label, and focus color with a visible outline. Check forced-colors and high-contrast modes when the interface is critical.

Raw palette utilities are useful while exploring. When a color represents a stable semantic role across many components, define a project theme token such as brand, surface, or danger. That makes a redesign a token decision rather than a search for every `blue-600`.

Use browser contrast tooling, but inspect the actual rendered state and text size. Transparency, gradients, images, and antialiasing can make a nominal ratio misleading.

Your action is to create success and error messages with text, border, and background colors. Test both over two parent backgrounds, remove color in DevTools, and confirm that the meaning remains available.

Add a dark color scheme

Use dark variants to define deliberate alternate colors instead of mechanically inverting the interface.

Add a dark variant beside the default:

```
<div class="bg-white text-gray-950 dark:bg-gray-950 dark:text-white">...</div>
```

By default, Tailwind's `dark:` variant follows the browser's `prefers-color-scheme` media feature. The base utility remains the light/default decision; the variant overrides the properties that genuinely change.

For a manual theme switch in Tailwind v4, define the variant against a class or data attribute:

```
@import "tailwindcss";
@custom-variant dark (&:where([data-theme=dark], [data-theme=dark] *));
```

Now an ancestor activates it:

```
<html data-theme="dark">...</html>
```

A three-way control—light, dark, system—must store the explicit choice and observe the system preference when “system” is selected. Apply the initial attribute before first paint when possible so the page does not flash the wrong theme.

Review every semantic color: page and elevated surfaces, normal and muted text, borders, links, controls, focus indicators, success, warnings, errors, code, shadows, and images. Dark mode is not a mechanical inversion. Pure black/white everywhere can create glare, while weak gray-on-gray combinations can fail contrast.

Set an appropriate CSS `color-scheme` as well so browser-provided controls, form widgets, and scrollbars can match. Tailwind includes color-scheme utilities, but choose them to match the theme actually active.

Theme tokens reduce duplication. If every component repeats the same light/dark palette pair, semantic surface and text variables may express the system more clearly.

Your action is to implement light, dark, and system choices for one card. Reload before and after changing the system theme, watch for flash, and audit every interactive state in both schemes.

Style readable prose

Combine width, spacing, text color, line height, and link states to make long-form content comfortable to read.

Start with the reading measure:

```
<article class="mx-auto max-w-prose px-4 text-lg leading-8 text-gray-800">...</article>
```

`max-w-prose` limits line length, `mx-auto` centers the article box, and `px-4` protects its edges on narrow screens. `text-lg leading-8` creates a comfortable body rhythm, but the correct size depends on the chosen font and audience.

Long-form content is a system of relationships:

- headings need clear hierarchy and more space before than after
- paragraphs need consistent vertical rhythm
- lists need visible markers and indentation
- links need a non-color affordance and clear focus
- code needs wrapping or intentional horizontal scrolling
- figures and tables need captions and overflow handling

Do not center long paragraphs simply because the article box is centered. Left-aligned text is usually easier to scan in left-to-right prose.

When content comes from Markdown or a CMS, styling every descendant in the template becomes noisy. A semantic prose component, ordinary descendant CSS, or Tailwind's optional Typography plugin can be a better boundary. Utilities are still useful for the article container and exceptional elements.

Preserve semantic HTML. A visually large `<div>` does not become a heading, and adding margin to fake separate paragraphs does not give assistive technology paragraph structure.

Test actual content: a short paragraph, a long URL, nested list, code sample, blockquote, image, and table. Zoom to 200%, narrow the viewport, and verify that horizontal overflow is confined to the element that needs it.

Your action is to style a real 500-word article, not placeholder sentences. Measure line length, navigate headings and links by keyboard, and fix one overflow case without shrinking all text.

Check your understanding: typography and colors

Check text utilities, font loading, line height, palette shades, opacity, backgrounds, gradients, and contrast-aware choices.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Responsive and interactive states

Compose mobile-first, focus, hover, group, and peer states.

Use mobile-first responsive variants

Apply base utilities for small screens and layer changes at min-width breakpoints with sm, md, lg, xl, and 2xl variants.

Unprefixed utilities apply everywhere. A breakpoint variant applies from that minimum width upward.

```
<img class="w-16 md:w-32 lg:w-48" src="photo.jpg" alt="">
```

The image is `w-16` below `md`, `w-32` from `md` until `lg`, and `w-48` from `lg` upward. Each larger condition overrides the same width property when it becomes active.

Do not use `sm:` to mean "on phones." It means the `sm` minimum width and wider. Put the constrained layout in the base classes, then add complexity when space is available:

```
<section class="grid gap-6 md:grid-cols-2 lg:grid-cols-3">...</section>
```

Choose breakpoints from the content. Add a second column when two useful cards fit, not because a device list calls something a tablet. Browser windows, split-screen views, zoom, sidebars, and embedded components make device labels unreliable.

Variants can target bounded ranges with a minimum and maximum condition when a design genuinely needs one. Prefer a few cumulative rules first; many narrow ranges make the cascade difficult to understand.

Responsive changes can affect access. `hidden md:block` removes content below `md`, including from keyboard and accessibility navigation. Use it only for duplicate or genuinely optional content, not to hide the only navigation without providing an equivalent control.

Test by resizing slowly rather than checking three preset screenshots. Watch for the exact width where text wraps, controls collide, or the reading order stops making sense. Also test zoom, which reduces the effective CSS viewport width and should activate the compact layout.

Your action is to build a one-column list that gains columns only when cards remain readable. Record the content failure width that justified each variant and verify that no action disappears at 200% zoom.

Style interaction states

Use hover, focus-visible, active, disabled, and other state variants without hiding essential feedback from keyboard or touch users.

Variants prefix a utility:

```
<button class="bg-blue-600 hover:bg-blue-700 focus-visible:outline-2 active:translate-y-px di
```

Each prefix adds a condition to the generated CSS. The base background is always present, `hover:` responds when hover is available and active, `focus-visible:` exposes keyboard-like focus, `active:` covers activation, and `disabled:` matches a disabled control.

The HTML state must be real:

```
<button disabled class="disabled:cursor-not-allowed disabled:opacity-50">
  Saving…
</button>
```

Styling something to look disabled does not prevent clicks or communicate the state to assistive technology. Likewise, a `<div>` with hover styles does not become a button; use the native interactive element first.

Hover is an enhancement. Touch users may never hover, and some devices emulate hover in surprising ways. Essential instructions and actions must remain visible or available through focus and activation.

Use `focus-visible` for a clear keyboard indicator without necessarily showing it after every pointer click. Never remove focus just because the hover style looks cleaner. Test the outline against every background and at high zoom.

State combinations matter. A disabled button should not appear to lift on hover. A pressed button may need `aria-pressed` and an `aria-pressed:` variant so the visual state follows the accessibility state rather than separate JavaScript styling.

Keep transitions short and optional. A state must be understandable even when animation is disabled.

Your action is to build a button with default, hover, focus-visible, active, disabled, and pressed states. Test with keyboard, mouse, and touch emulation, then verify DOM attributes and visual classes never disagree.

Coordinate state with group and peer

Style one element based on the state of a parent or sibling using group and peer variants while keeping the HTML relationship clear.

Mark the state owner with `group`:

```
<a class="group" href="https://flaviocopes.com/docs">
  Docs <span class="motion-safe:group-hover:translate-x-1">→</span>
</a>
```

The descendant's variant responds to the ancestor's state. The ancestor remains the real link; `group` only exposes that existing relationship to descendant styling.

`peer` lets a later sibling respond to a control:

```
<label>
  <span>Email</span>
  <input class="peer" type="email" required>
  <span class="hidden text-red-700 peer-invalid:block">
```

```
    Enter a valid email address.
  </span>
</label>
```

Peer selectors rely on CSS sibling relationships, so the styled target must come after the marked peer in the DOM. Do not reorder meaningful content just to satisfy a selector; change the markup boundary or use another state mechanism.

Native state is the best source when available: `checked`, `invalid`, `disabled`, and focus already communicate with the platform. For application state, use attributes such as `aria-expanded` and style from that attribute so accessibility and appearance share one source of truth.

Nested components can name groups or peers, for example `group/card` with `group-hover/card:*`, so an inner group does not accidentally respond to an outer one. Names clarify which state owner controls a variant.

Showing validation text with CSS is not the complete validation experience. Connect the message to the input, choose when validation should appear, and make dynamic updates understandable to assistive technology.

Your action is to build one grouped link and one peer-validated field. Add a nested group, test keyboard focus and invalid submission, and draw the exact ancestor or sibling selector each variant represents.

Use container query variants

Adapt a component to the width of its own container with built-in container query utilities and named containers.

Mark a container and respond inside it:

```
<div class="@container">
  <article class="flex flex-col @md:flex-row">...</article>
</div>
```

`@container` establishes an inline-size query container. The `@md:` variant becomes active when that container—not the viewport—reaches Tailwind's `md` container size. Like viewport breakpoints, these variants are mobile-first minimum conditions.

This makes the component portable. At the same browser width, a card can stay stacked in a narrow sidebar and become a row in the main content area.

Nested containers can be named:

```
<section class="@container/main">
  <article class="grid @md/main:grid-cols-[10rem_1fr]">...</article>
</section>
```

The name makes the dependency explicit when the nearest unnamed container is not the intended one. Keep names about layout regions rather than one page instance.

Container queries do not replace viewport queries. Page-wide navigation and global gutters often depend on the viewport. A reusable card, toolbar, or widget usually cares about its allocated space. Choose the boundary that owns the constraint.

The parent must actually establish a container. If `@md:*` never matches, inspect `container-type`

in DevTools and measure the container's content box. Do not compare the breakpoint against the window width.

Use arbitrary container thresholds only for real exceptions. If several components share a threshold, define a `--container-*` theme value so the variant has a name and consistent meaning.

Your action is to render the same card in a 20rem sidebar and a flexible main column. Keep the viewport unchanged, inspect both container sizes, and prove that only the wide instance activates `@md:`.

Respect reduced motion

Use motion-safe and motion-reduce variants so transitions and animations respect the visitor's operating-system preference.

Apply optional motion only when it is welcome:

```
<div class="motion-safe:transition motion-safe:hover:scale-105 motion-reduce:transform-none">
```

`motion-safe:` applies only when the visitor has not requested reduced motion. This is often simpler than defining animation by default and trying to undo every transform, duration, and transition later.

For an existing animation that must retain a reduced alternative, use `motion-reduce:` deliberately:

```
<svg class="animate-spin motion-reduce:hidden" aria-hidden="true">...</svg>
<span>Saving</span>
```

The visible text preserves the status when the decorative spinner disappears.

Reduced motion does not always mean "remove every transition." Eliminate non-essential movement, parallax, large zooms, continuous animation, and vestibular triggers. A short opacity change may remain acceptable, but the interface must not require animation to explain where content went.

Avoid motion that delays input or completion. A `duration-700` class is not proof that a transition improves comprehension. State should update immediately and remain clear if animations fail to run.

Test the real media feature through the operating system or browser emulation. Also test keyboard focus during transitions: an element that moves away while focused can be disorienting.

Motion preferences belong with other accessibility conditions, not in a separate afterthought. Review hover transforms, loading indicators, page transitions, accordions, auto-scrolling, and animated charts together.

Your action is to audit one component with at least two animations. Enable reduced motion, preserve every piece of information and functionality, and explain why each remaining transition is necessary.

Check your understanding: responsive design and states

Check mobile-first breakpoints, hover and focus variants, dark mode, group and peer behavior, motion preferences, and container queries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Customization and debugging

Work with theme tokens and diagnose generated-class problems.

Customize with theme variables

Define project design tokens in Tailwind CSS v4 with the theme directive and understand how those tokens create utility classes.

Tailwind v4 uses CSS-first theme variables:

```
@import "tailwindcss";

@theme {
  --color-brand: #2457d6;
  --font-display: "Geist", sans-serif;
  --breakpoint-3xl: 120rem;
}
```

The variable namespace tells Tailwind which API to extend:

- `--color-brand` creates color utilities such as `bg-brand`, `text-brand`, and `border-brand`
- `--font-display` creates `font-display`
- `--breakpoint-3xl` creates the `3xl:` responsive variant

Theme variables are also emitted as ordinary CSS custom properties, so custom CSS or JavaScript-driven styles can reference the same value:

```
.chart-line {
  stroke: var(--color-brand);
}
```

Use `@theme` when a value should participate in Tailwind's utility API. Use `:root` for a custom property that should exist at runtime but should not generate utility families.

Choose tokens from repeated design decisions. A one-off marketing illustration color does not automatically deserve a global name. A brand color used for links, buttons, and focus might—but test whether one value can satisfy every contrast context.

Names can be raw (`--color-brand-600`) or semantic (`--color-action`). Raw scales expose palette choices; semantic names communicate purpose and can make theming easier. Many projects use a small semantic layer over a restrained palette.

Custom breakpoints affect every component that uses them. Define them in compatible units and sorted order, then document the content problem they solve. A new device nickname is not enough justification.

Changing a token has a large blast radius. Use visual regression checks and inspect light, dark, hover, focus, and disabled states before treating a theme edit as a harmless color replacement.

Your action is to inventory repeated colors, fonts, and breakpoints in a small page. Promote only three justified values into `@theme`, replace the repeated classes, and list every state that must be reviewed when each token changes.

Add a custom utility

Create a project-specific utility when a reusable behavior does not belong in the built-in scale or a semantic component class.

Tailwind v4 lets you define a utility in CSS:

```
@utility content-auto {
  content-visibility: auto;
}
```

You can now use `content-auto` in markup, including with variants:

```
<section class="content-auto lg:content-visible">...</section>
```

Tailwind inserts a registered utility into the utility layer, so it participates in variants and cascade ordering like built-in utilities. This is different from writing an unrelated class in an arbitrary stylesheet position.

Create a custom utility when all of these are true:

- the CSS behavior is useful in several places
- Tailwind does not already express it clearly
- the class has a narrow, property-focused meaning
- variants such as `hover:` or `lg:` should compose with it

Use an arbitrary property for a genuine one-off:

```
<div class="[content-visibility:auto]">...</div>
```

Promote it to `@utility` when repetition reveals a project-level vocabulary. If the value belongs to an existing family such as color, font, spacing, or breakpoint, a theme variable is usually the better extension point.

A component class is a different abstraction. `.button-primary` may combine layout, color, typography, and state because it represents a reusable interface concept. `content-auto` represents one CSS behavior. Do not hide a complete component inside a "utility" name.

Functional utilities can accept validated values, but they create a public styling API that the team must understand and maintain. Start with a simple fixed utility unless the value family is genuinely useful.

Your action is to use one arbitrary property twice, promote it to `@utility`, and apply a responsive variant. Then explain why it is a utility rather than a theme token or component class.

Know when to write CSS

Decide when utilities keep a component clear and when a semantic class, custom property, or ordinary stylesheet is easier to maintain.

Utilities are not a rule that forbids CSS.

Keep utilities inline when the element owns a small, explicit set of responsive and state decisions. Extract a framework component when markup and behavior repeat; the component can still contain utilities and expose semantic props such as `tone="danger"`.

Write ordinary CSS when CSS itself is the clearer abstraction:

- rich prose descendants generated by Markdown or a CMS

- third-party markup you cannot add classes to
- complex selectors or pseudo-elements
- keyframes and coordinated animation
- a semantic component style shared outside the template system
- browser behavior that has no useful built-in utility

For example:

```
.article-content h2 {
  margin-block-start: 2.5rem;
  font-size: var(--text-2xl);
}
```

This can be easier to audit than repeating descendant arbitrary variants on every article container.

Use `@utility` for a small reusable CSS behavior that should support Tailwind variants. Use `@theme` for values that should expand Tailwind's token-driven API. Use ordinary custom properties when you need runtime values without generating utilities.

Avoid extracting a class merely because a class list is long. A card with many unique visual decisions may be clearer in one component file than split between markup and a one-off `.card` selector. Extract when there is stable reuse or a semantic boundary.

Also avoid copying the same long class string. In React, Vue, Astro, or templates, a reusable component is often a better abstraction than a CSS class because it preserves markup, accessibility, and behavior together.

Use the simplest layer that makes the design clear. Tailwind and ordinary CSS work together.

Your action is to classify five styles in your project as inline utility, component, custom utility, theme token, or ordinary CSS. Write one sentence for each boundary and remove one abstraction that adds indirection without reuse.

Debug missing classes

Diagnose class detection, generated output, incorrect variants, conflicting utilities, and CSS cascade problems with DevTools.

If a class has no effect, inspect the element.

Use a fixed sequence instead of changing random utilities.

First, confirm the class token in the DOM is exactly what you expect. Template conditionals, merge helpers, and typos can produce a different string.

Second, search the generated stylesheet:

- **Rule missing:** make sure the complete class appears in a detected source file. Dynamic fragments such as `bg-${color}-600` cannot be understood by the text scanner. Check ignored directories, monorepo base paths, and external libraries; add an intentional `@source` only when needed.
- **Rule present:** inspect whether its selector and variant condition match. `md:*` needs the minimum viewport width, `@md:*` needs a query container, `peer-*` needs the correct sibling order, and `dark:*` needs the configured dark condition.

Third, inspect the Styles and Computed panels. A crossed-out declaration lost the cascade. The

order of class names in the HTML is not a reliable conflict-resolution tool; remove contradictory utilities or make the condition explicit.

Fourth, verify the property can affect this box. `justify-center` needs a suitable Flex or Grid container. `h-full` needs a definite parent height. `z-50` cannot escape every stacking context. A generated, winning rule can still appear ineffective because the layout assumption is wrong.

Finally, check browser support, invalid arbitrary syntax, inherited values, CSS variables, and stale build output. Restarting the build can confirm a watcher problem, but it should not be the first or only diagnosis.

Record evidence at each layer: source token, generated selector, matched condition, winning computed value, and layout context.

Your action is to create four controlled failures: dynamic class, inactive variant, conflicting utility, and wrong layout context. Diagnose each without guessing and save the decisive DevTools evidence.

Build a complete Tailwind page

Translate the CSS course landing page into Tailwind utilities while keeping its semantic HTML, accessibility, and responsive behavior intact.

Rebuild the CSS course project with Tailwind.

Start from the behavior, not from a list of classes. Write down the page requirements:

- semantic header, navigation, main content, sections, and footer
- readable type and content width
- one-column base layout with wider card tracks when content fits
- visible hover, focus, active, and disabled states
- complete light and dark palettes
- reduced-motion behavior
- no horizontal overflow at 200% zoom

Build the structural skeleton first:

```
<body class="bg-white text-gray-950 dark:bg-gray-950 dark:text-white">
  <header class="border-b border-gray-200 dark:border-gray-800">...</header>

  <main>
    <section class="mx-auto max-w-6xl px-4 py-16 sm:px-6">
      <h1 class="max-w-3xl text-4xl font-bold tracking-tight md:text-6xl">
        Learn CSS by building
      </h1>

      <ul class="mt-10 grid gap-6 md:grid-cols-2 lg:grid-cols-3">
        ...
      </ul>
    </section>
  </main>
</body>
```

Then work one system at a time: page/container sizing, layout, spacing rhythm, typography, colors, interactive states, responsive changes, and motion. This makes conflicts easier to locate than styling

each element to completion in isolation.

Keep repeated values in the theme only when they represent project decisions. Extract repeated card markup into a component so semantics and focus behavior stay consistent. Keep one-off composition local.

Use real content early. Add the longest title, missing image, validation error, translated navigation item, and several cards. Keyboard-test every action. Emulate dark scheme and reduced motion. Resize continuously and zoom instead of checking only named breakpoints.

Compare the two versions in DevTools. The generated CSS should express the same deliberate layout, not a different design chosen by class-name convenience.

For each important element, compare computed display, track/flex sizing, width constraints, spacing, typography, color, focus, and media-query conditions. Differences are acceptable only when intentional and documented.

Your action is to complete the page and produce a short verification table covering semantic structure, keyboard order, focus, contrast, responsive layout, zoom, dark mode, reduced motion, class detection, and production CSS output. Fix every unexplained difference before calling the translation complete.

Check your understanding: customization and gotchas

Check theme variables, arbitrary values, custom utilities, source detection, class conflicts, important modifiers, and when custom CSS is clearer.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/tailwind/.