CSS Course

Flavio Copes

Updated August 3, 2026

Contents

Learn the cascade, selectors, the box model, Flexbox, Grid, responsive design, and the modern CSS features used in real interfaces.

What you'll learn: Style an HTML page deliberately and build responsive layouts with Flexbox and Grid.

Add CSS to a page

Rules, declarations, stylesheets, selectors, and DevTools.

What CSS does

Understand how CSS turns structured HTML into a visual interface while keeping content, meaning, and presentation separate.

HTML gives a page structure and meaning. CSS supplies constraints for presenting that structure.

The browser already has a user-agent stylesheet. That is why headings are large and links are underlined before you write CSS. Your declarations join browser defaults, user preferences, and element state in the cascade.

```
h1 {
  color: navy;
  font-size: 3rem;
}
```

This rule selects every `h1`, but it does not directly draw pixels. The browser roughly:

1. matches applicable selectors
2. runs the cascade to find one value per property
3. computes inherited and relative values
4. performs layout using the content and available space
5. paints the result

CSS is therefore not a drawing program with fixed coordinates. It describes constraints: how wide a box may become, how siblings share space, what happens when text grows, and which declaration wins when rules disagree.

Open DevTools and inspect the heading. The Styles panel shows matched and overridden declara-

tions. Computed styles show the winning values. The Layout or box-model panel shows the size produced after layout.

Change the text, narrow the viewport, and increase the root font size. Good CSS continues to produce a usable page because its constraints respond to content instead of assuming one screenshot.

Anatomy of a CSS rule

Read selectors, declaration blocks, properties, and values, then write small rules using valid CSS syntax.

A CSS rule has a selector and a declaration block:

```
.card {
  padding: 1rem;
  border: 1px solid black;
}
```

`.card` is the selector. It chooses elements whose `class` contains `card`.

Inside the braces, each declaration has a property and value separated by a colon. A semicolon ends the declaration.

Whitespace does not usually change meaning, so this is valid too:

```
.card{padding:1rem;border:1px solid black}
```

The expanded form is easier to scan and debug.

Unknown properties and invalid values are ignored. The browser keeps processing the rest of the stylesheet, which makes CSS resilient but can also hide typing mistakes.

Add CSS to HTML

Connect an external stylesheet, use a style element when appropriate, and understand why inline styles are difficult to maintain.

Link a stylesheet from the document head:

```
<link rel="stylesheet" href="https://flaviocopes.com/styles.css">
```

External files keep presentation separate and can be cached across pages.

For a small isolated example, use `style`:

```
<style>
  body {
    font-family: system-ui, sans-serif;
  }
</style>
```

An inline style applies directly to one element:

```
<p style="color: rebeccapurple">Hello</p>
```

Inline styles are hard to reuse and override, so avoid them for normal site styling.

The order of linked stylesheets matters when otherwise equal rules compete. Put broad foundations first and more specific component or page styles later.

Inspect CSS with DevTools

Use the Elements, Styles, and Computed panels to see matched rules, overridden declarations, inherited values, and final styles.

Open developer tools and select an element in the page.

The **Styles** panel shows rules that match it. A crossed-out declaration lost to another rule or is inactive. The panel usually shows the stylesheet and line where each rule came from.

The **Computed** panel shows the final value of every property after the cascade, inheritance, and defaults have been resolved.

Try editing a value, disabling a declaration, and adding a new property. These edits are temporary. Reloading the page restores the source files, so copy useful changes back into your stylesheet.

DevTools is not only for fixing bugs. Use it while learning to ask: which rule matched, why did this value win, and what box did the browser calculate?

Start the course page

Create a small unstyled landing page and connect the stylesheet that you will improve throughout the CSS course.

Create `index.html` with a header, main section, three feature cards, and a footer. Give reusable parts classes such as `site-header`, `features`, and `card`.

Then create `styles.css` and link it from the head.

Start with a small foundation:

```
* {
  box-sizing: border-box;
}

body {
  margin: 0;
  font-family: system-ui, sans-serif;
  line-height: 1.5;
}
```

Do not design everything yet. The point is to have real content that lets you see how selectors, boxes, Flexbox, Grid, and responsive rules interact.

Keep the page open next to DevTools as you continue.

Take this interactive quiz online

The cascade

Inheritance, specificity, source order, and selector behavior.

Type, class, and ID selectors

Select elements by type, reusable class, or unique ID and choose the least specific selector that clearly expresses your intent.

A selector describes which existing elements a rule applies to. A type selector matches every element with that name:

```
p { max-width: 65ch; }
```

A class selector expresses a reusable styling role:

```
.card { padding: 1rem; }
```

An ID selector matches the element with that document ID:

```
#newsletter { border-top: 1px solid; }
```

Classes are the normal choice for components because they can repeat and remain easy to override. Type selectors are useful for broad defaults such as prose spacing. IDs are valuable for fragment links, labels, and script hooks, but their high specificity makes them awkward styling dependencies.

Selectors do not verify intent. If you reuse `.card` on an element, it receives the rule regardless of its tag or content. Name classes after a stable role rather than a visual accident such as `.blue-box`.

You can group selectors with commas:

```
h1,
h2,
h3 {
  line-height: 1.1;
}
```

Grouping does not combine specificity; each selector is evaluated separately.

In DevTools, select an element and inspect its Matched Rules. Add the same class to a second element, then remove it from the first. This makes the selector-to-element relationship visible without changing the rule.

Combinators

Target descendants, direct children, adjacent siblings, and general siblings without adding a class to every element.

Combinators describe relationships between elements.

```
.card p { }
.card > p { }
h2 + p { }
h2 ~ p { }
```

A space matches any descendant. `>` matches a direct child. `+` matches the next sibling. `~` matches later siblings with the same parent.

Use the narrowest relationship the component guarantees. `.card p` also reaches paragraphs inside a nested quote or another component. `.card > p` stops at direct children.

Sibling selectors are useful when spacing or emphasis depends on document order:

```
h2 + p {
  font-size: 1.125rem;
}
```

```
.field + .help {
  margin-block-start: 0.5rem;
}
```

The first rule matches only the paragraph immediately after an `h2`. The second adds spacing only when help text immediately follows a field. In both cases, HTML order defines the relationship.

Avoid long chains such as `main .page section .card p strong`. They bind styling to a fragile HTML shape and raise specificity. A clear component class is often easier to maintain.

Duplicate a card, then nest a second card inside it. Use DevTools to check which paragraphs match `.card p` and `.card > p`. The unexpected matches reveal whether your selector depends on more structure than you intended.

Pseudo-classes and pseudo-elements

Style element states and generated parts with hover, focus-visible, first-child, before, and after selectors.

Pseudo-classes select an existing element in a state or structural position:

```
a:hover { text-decoration-thickness: 3px; }
button:focus-visible { outline: 3px solid; }
li:first-child { font-weight: bold; }
```

`:hover` is useful feedback, but it must not be the only way to reveal an action: touch and keyboard users may never hover. Use `:focus-visible` for a clear keyboard focus indicator, and do not remove the browser outline without an equally visible replacement.

State should agree with HTML behavior. A disabled-looking button should use the `disabled` attribute so `button:disabled` reflects a real interaction state. A group can use `:focus-within` when any descendant owns focus.

Pseudo-elements select or create a part of an element:

```
.external-link::after {
  content: " ";
}
```

`::before` and `::after` are useful for decorative additions. Essential information should remain in HTML because generated content can be unavailable to assistive technology, copy operations, or reading modes.

Pseudo-classes use one colon. Modern pseudo-elements use two. Inspect the Elements panel: browsers show generated pseudo-elements even though they are absent from the HTML.

Exercise the states without a mouse. Tab to the control, activate it, disable it, and emulate touch input. If meaning exists only in `:hover` or generated `content`, move that meaning into the document.

Cascade and specificity

Predict which declaration wins by considering importance, origin, cascade layers, selector specificity, and source order.

Several declarations can target the same property on the same element. The cascade chooses one

value before layout begins.

Use this order when debugging:

1. relevance: does the selector match and does its media condition apply?
2. origin and importance: browser, user, author, animation, or transition
3. cascade layer order
4. selector specificity
5. scoping proximity, when `@scope` is involved
6. source order when everything else ties

Within ordinary author styles, specificity is a useful comparison:

- IDs outweigh classes.
- classes, attributes, and pseudo-classes outweigh type selectors.
- when specificity ties, the later declaration wins.

```
p { color: black; }
.intro { color: navy; }
```

The class wins because both declarations are normal author styles in the same layer, and `.intro` has greater specificity.

Layers can make priority explicit:

```
@layer reset, components, utilities;
```

For normal declarations, a later layer beats an earlier one regardless of selector specificity. Normal unlayered author styles beat all normal layered styles. This is useful for containing third-party CSS.

`!important` changes cascade priority and reverses layer precedence. It is not "more specificity" and should not be a routine override tool. `:where()` provides the opposite tool: its selector always contributes zero specificity.

Prefer low-specificity component classes and predictable source order. If you are fighting selectors with more selectors, simplify the structure.

In DevTools, find the crossed-out declaration and read why it lost. Change one factor at a time—layer, selector, then order—until you can predict the winner before refreshing.

Inheritance

Understand which computed values pass from parent to child and use inherit, initial, unset, and revert deliberately.

After the cascade chooses values on a parent, some computed values pass to descendants. Text color and font settings usually inherit; borders, margins, and widths usually do not.

```
body {
  color: #222;
  font-family: system-ui, sans-serif;
}
```

Most descendants use those values unless another rule overrides them.

An inherited value is weaker than any declaration that directly targets the child. A low-specificity `a { color: ... }` beats a highly specific color inherited from an ancestor because the link has its own cascaded value.

Global keywords give you control:

- `inherit` uses the parent's computed value.
- `initial` uses the property's defined initial value.
- `unset` behaves like inherit for inherited properties and initial for others.
- `revert` rolls back to an earlier cascade origin or layer.

`revert-layer` rolls back only within the current cascade layer, which can be useful when a utility should expose the value from an earlier layer.

Form controls do not always inherit typography as you expect, so projects often set `button`, `input`, `select`, and `textarea` to `font: inherit`.

Inspect a nested element's Computed panel and expand `color` or `font-family` to see the ancestor it came from. Then set `color: initial`, `inherit`, `unset`, and `revert` one at a time. The result depends on the property's inheritance behavior and cascade position, not on whether the keywords sound similar.

Take this interactive quiz online

Values and visual language

Color, units, typography, backgrounds, and custom properties.

Colors

Express colors with named values, hexadecimal, rgb, and hsl syntax while checking contrast between text and its background.

CSS accepts several color formats:

```
.examples {
  color: rebeccapurple;
  border-color: #663399;
  background: rgb(102 51 153 / 15%);
  outline-color: hsl(270 50% 40%);
}
```

Choose the format that makes the relationship easiest to maintain. Hex is compact, `rgb()` maps directly to display channels, and `hsl()` can make coordinated lightness changes easier. Modern space-separated syntax accepts alpha after `/`.

Alpha is compositing, not a lighter standalone color. `rgb(102 51 153 / 15%)` produces a different visible result over white, black, or an image. Check contrast against the final rendered background.

`currentColor` reuses the element's computed text color:

```
.notice {
  color: #2457d6;
  border: 2px solid currentColor;
}
```

Color is not only decoration. Text needs enough contrast against its background, including text placed over images or translucent layers. Do not use color as the only signal for errors, selection, or status. Add text, an icon, a border, or another visible cue.

Developer tools usually include a color picker and contrast information. Also test forced-colors mode: allow the browser to substitute the user's high-contrast palette unless a small targeted adjustment is necessary.

Create normal, hover, focus, disabled, and error states. Use the contrast tool on every state, then turn color off mentally: the control should still communicate its role and state.

Length units

Choose pixels, rems, ems, percentages, viewport units, and character units according to what a size should respond to.

CSS units express relationships.

- `px` is a CSS reference pixel, not a guaranteed physical device pixel.
- `rem` is relative to the root font size.
- `em` follows font size: on `font-size` it uses the parent; on many other properties it uses the element's computed font size.
- `%` depends on the property and its containing block.
- `vw` and `vh` relate to viewport dimensions.
- `ch` approximates the width of the `0` glyph and is useful for readable line lengths.

```
.article {
  width: min(100% - 2rem, 65ch);
  margin-inline: auto;
}
```

There is no universally best unit. Ask what the value should track: user text preferences, its component, its container, or the viewport.

Viewport height needs extra care on mobile. Browser controls can expand and collapse. `svh` represents the small viewport, `lvh` the large viewport, and `dvh` updates with the dynamic viewport. A fixed `100vh` panel can hide content behind browser UI; prefer content-driven `min-height`, then choose the viewport variant that matches the experience.

Percentages are deliberately contextual. `width: 50%` relates to the containing block's width, while percentage padding also resolves against the containing block's inline size. Inspect the used pixel value in DevTools instead of assuming every percentage uses the same reference.

Resize, zoom to 200%, and increase the default font size. Values chosen for the correct relationship should adapt without clipping text or forcing horizontal scrolling.

Typography

Set a readable font stack, size, line height, line length, weight, and spacing without fighting browser or user preferences.

Start with a reliable font stack and unitless line height:

```
body {
  font-family: system-ui, sans-serif;
  font-size: 1rem;
  line-height: 1.5;
}
```

```
h1,
h2 {
  line-height: 1.1;
}
```

A unitless line height inherits as a multiplier, so each descendant applies it to its own font size. A fixed `24px` line height inherited by a large heading can clip or crowd it. Body text usually needs more breathing room than headings.

Keep long prose to a readable measure, often around `60ch` to `70ch`. Use font weight and size to create hierarchy, but preserve semantic heading order in HTML.

If you load web fonts, provide fallbacks and avoid unnecessary weights. Request only styles the design uses. The fallback's metrics affect line breaks and layout while the font loads, so test the transition rather than treating the fallback as invisible.

Do not make text containers rigid. Let paragraphs grow when translation is longer or users enlarge text. Avoid clipping with fixed heights, and do not reduce line height merely to make a screenshot fit.

In DevTools, disable the preferred font to see the fallback. Then emulate a slow connection, zoom to 200%, and replace a short heading with a long one. The content should remain readable before, during, and after font loading.

Backgrounds and borders

Combine background colors, images, sizing, position, and borders to create visual separation without changing document structure.

A background paints an element without contributing intrinsic size. By default it extends underneath the border and covers the content and padding areas.

```
.hero {
  background-color: #eef4ff;
  background-image: url('/images/pattern.svg');
  background-repeat: no-repeat;
  background-position: right top;
  background-size: contain;
}
```

Use `cover` when an image should fill the box and cropping is acceptable. Use `contain` when the whole image must remain visible.

Background layers are painted with the first listed image on top. Always provide a usable `background-color` beneath an image so text remains readable while it loads or if it fails.

Borders occupy space in the box model:

```
.card {
  border: 1px solid currentColor;
}
```

`currentColor` uses the element's computed `color`, which keeps borders coordinated with themes. Decorative background images need no alt text, but meaningful images belong in HTML where they can have accessible alternatives.

Borders participate in box sizing and can move surrounding layout when added on hover. Reserve the border's space from the initial state, or use an outline when the effect should not affect dimensions. Outlines are also appropriate for focus indicators because they do not change layout and should not be clipped.

Disable the background image in DevTools and resize the box. Verify the color fallback, text contrast, crop, and border-box dimensions rather than checking only the ideal image state.

Custom properties and calculations

Define reusable CSS values, inherit component tokens, provide fallbacks, and combine different units with calc, min, max, and clamp.

Custom properties store token streams in the cascade:

```
:root {
  --space: 1rem;
  --accent: #2457d6;
}

.card {
  padding: var(--space);
  border-color: var(--card-accent, var(--accent));
}
```

They inherit, so a component can redefine a token for its descendants.

The fallback is used when the referenced custom property is missing or invalid as a custom property. It does not validate the final property value:

```
.card {
  --card-width: red;
  width: var(--card-width, 20rem);
}
```

Because `--card-width` exists, the fallback is not used. `red` is invalid for `width`, so the declaration becomes invalid at computed-value time. DevTools exposes this clearly.

CSS math functions make values responsive:

```
h1 {
  font-size: clamp(2rem, 6vw, 4rem);
}

.layout {
  width: min(100% - 2rem, 70rem);
}
```

`clamp()` sets a minimum, preferred value, and maximum. `min()` chooses the smallest expression; `max()` chooses the largest. `calc()` combines compatible values and requires spaces around `+` and `-`.

Use custom properties for relationships that should vary through the cascade: themes, component density, and shared spacing. Do not turn every literal into a global token; keep values local until several consumers need the same decision.

Override `--space` on one component ancestor and inspect which descendants inherit it. Then misspell the property name and supply an invalid value. The two failures exercise different fallback behavior.

Take this interactive quiz online

Boxes and positioning

The box model, normal flow, positioning, and stacking.

The box model

Visualize every element as content surrounded by padding, border, and margin, and see how those layers determine occupied space.

Layout works with boxes generated by elements and text. One element can generate more than one box, as inline content wraps across lines, but the block box model is the useful starting point.

From inside to outside, a box has:

1. content
2. padding
3. border
4. margin

```
.card {
  width: 20rem;
  padding: 1rem;
  border: 2px solid;
  margin: 1rem;
}
```

With the default `content-box`, `width: 20rem` sizes only the content. Horizontal padding and borders are added to produce the border box. Margins sit outside that border box and influence spacing, but are not painted by the element's background.

Do not assume `height` includes all visible content. Text wrapping, replaced elements, minimum content sizes, and overflow can make the used result more complicated than the declaration.

Open DevTools and inspect the box-model diagram. It shows the calculated size of each layer and is often the fastest way to understand why a box is larger or farther away than expected.

Change padding, border, and margin one at a time. Predict the border-box width before reading the diagram. Then add a long unbreakable word and watch how content constraints can override your mental arithmetic.

Box sizing

Compare content-box with border-box and make declared widths include padding and borders for predictable component sizing.

With the default `content-box`, a declared width applies only to content. Padding and border are added outside it.

A `200px` box with `20px` padding and `5px` borders occupies `250px` horizontally.

`border-box` includes padding and border inside the declared dimensions:

```
html {
  box-sizing: border-box;
}


*,
*::before,
*::after {
  box-sizing: inherit;
}
```

This common rule makes component dimensions easier to reason about and lets a component deliberately choose another model that descendants inherit. Margin remains outside the declared width.

With `border-box`, the browser subtracts padding and borders to find the content box. If those layers consume more than the declared size, the content box cannot become negative; the element still needs enough border and padding space.

`box-sizing` does not stop content from overflowing a box that is too small. You still need flexible dimensions, wrapping, or overflow handling.

Create two otherwise identical inputs at `width: 100%`, give them padding and borders, and toggle `content-box` versus `border-box` in DevTools. Measure the containing block and border box instead of relying on visual alignment.

Margin and padding

Create internal and external spacing, use logical properties, and recognize when vertical block margins collapse.

Padding creates space inside a border. Margin separates boxes.

```
.card {
  padding: 1rem 1.5rem;
  margin-block: 2rem;
}
```

Logical properties such as `margin-block`, `margin-inline`, `padding-block`, and `padding-inline` follow the writing direction instead of assuming top, right, bottom, and left.

Block-axis margins in normal flow can collapse. Adjacent sibling margins may combine into one, usually the larger positive value. A first or last child's margin can also collapse through a parent when no border, padding, inline content, height, or new formatting context separates them. Flex and Grid item margins do not collapse.

Do not “fix” an unexplained collapse by adding arbitrary padding. Inspect the boxes, decide whether the spacing belongs to the parent or siblings, and create a new formatting context only when that layout boundary is intentional.

Use `gap` when spacing items inside a Flexbox or Grid layout. It expresses the relationship on the container and avoids first-child and last-child margin exceptions.

Set two paragraphs to `margin-block: 2rem` and measure the space between them. Then place

them in a flex column with `gap: 2rem`. The first demonstrates collapsing; the second expresses one explicit inter-item gap.

Display and normal flow

Compare block, inline, inline-block, none, flex, and grid boxes while preserving the document's natural reading order.

Normal flow lays block boxes one after another in the block direction and inline content into line boxes. It gives the document a usable reading order before specialized layout begins.

- `block` starts a new block and usually fills available inline space.
- `inline` participates in text flow and does not accept width and height normally.
- `inline-block` flows inline but keeps box dimensions.
- `flex` and `grid` create layout containers for their children.
- `none` removes the element's box from layout.

`display` has an outer role and an inner role. `display: flex` makes the element behave as a block externally while laying out its direct children with Flexbox internally. `inline-flex` changes the outer participation but keeps the same flex formatting context inside.

`visibility: hidden` hides a box but preserves its space. `display: none` creates no box and normally removes that subtree from the accessibility tree. Neither should be used when content must remain available to assistive technology but merely look visually hidden.

Start with normal flow. Add a layout mode only when the content needs it. A page built on natural source order stays more resilient and accessible when CSS fails or the viewport changes.

Disable `display: flex` or `grid` in DevTools. If the source order becomes confusing, repair the HTML rather than depending on visual rearrangement.

Positioning

Use relative, absolute, fixed, and sticky positioning while understanding containing blocks and removal from normal flow.

`position` changes how offsets such as `top` and `inset-inline-end` work.

- `relative` keeps the original space and offsets the box visually.
- `absolute` removes the box from normal flow and positions it against a containing block.
- `fixed` normally positions against the viewport.
- `sticky` behaves normally until it reaches an offset, then sticks within its scroll container.

For an absolute badge inside a card:

```
.card { position: relative; }
.badge {
  position: absolute;
  inset-block-start: 0.5rem;
  inset-inline-end: 0.5rem;
}
```

The card becomes the badge's containing block because it is the nearest positioned ancestor. Without it, the badge may position against a more distant ancestor or the initial containing block.

Sticky positioning requires a non-`auto` inset on the axis that should stick. It follows the nearest

ancestor with a scrolling mechanism, including one created by `overflow: hidden`, `auto`, or `scroll`. An unexpected overflow ancestor is a common reason a sticky header appears not to work.

Transforms and some containment properties can also establish containing blocks, including for elements that would otherwise be fixed to the viewport.

Use positioning for overlays and anchored details, not as a substitute for page layout. Absolute and fixed content can cover text when users zoom or content grows.

In DevTools, inspect the containing block and scroll containers. Remove ancestors' `position`, `transform`, and `overflow` one at a time until you can explain the reference box.

Stacking and overflow

Diagnose clipped content and overlapping layers by understanding overflow behavior, z-index, and independent stacking contexts.

Content can overflow when a box is smaller than what it contains.

```
.code-sample {
  overflow-x: auto;
}
```

Prefer scrolling or wrapping to clipping important content. `overflow: hidden` clips and creates a scrolling mechanism even when no scrollbar is shown; it can also change sticky behavior. Avoid clipping focus outlines and controls that keyboard users must reach.

`z-index` controls stacking within a stacking context. A large number does not escape an ancestor's context. Properties such as positioned `z-index`, `opacity`, and transforms can create new contexts.

Other context creators include fixed or sticky positioning and `container-type: inline-size`. Once an ancestor forms a stacking context, all of its descendants are painted as one unit relative to sibling contexts.

When a menu appears behind another panel, inspect both ancestors. The problem is often not that the number is too small; the elements are competing in different contexts.

Keep a small, named layer scale for normal interfaces instead of escalating to arbitrary values.

Native dialogs and popovers can enter the browser's top layer, which sits above ordinary stacking contexts. Prefer that platform behavior for true modal UI instead of trying to outbid every `z-index`.

Use the DevTools stacking-context view when available. Toggle ancestor `opacity`, `transform`, positioning, and overflow—not just the child's number—until the layer tree explains the result.

Take this interactive quiz online

Flexbox

Build and align one-dimensional layouts.

Start a Flexbox layout

Turn a parent into a flex container and identify the main axis, cross axis, container properties, and item properties.

Flexbox lays direct children out along one main axis. The cross axis runs perpendicular to it.

```
.navigation {
  display: flex;
  gap: 1rem;
}
```

The direct children become flex items. In a left-to-right horizontal writing mode, the default `row` main axis runs left to right. Do not equate "row" permanently with "horizontal": writing mode and direction influence the logical axes.

Container properties such as `flex-direction`, `justify-content`, `align-items`, `flex-wrap`, and `gap` organize the group. Item properties such as `flex`, `align-self`, and `order` affect individual children.

By default, items do not wrap, can shrink, and stretch on the cross axis when their cross size is automatic. Their automatic minimum size can still prevent them from becoming narrower than unbreakable content.

Flexbox does not change the HTML order. Keep the source order logical even if a visual property can rearrange items.

Enable the Flexbox overlay in DevTools. Switch `flex-direction` between `row` and `column`, then identify the main and cross axes before changing any alignment property. This prevents the common habit of guessing between `justify-content` and `align-items`.

Align flex items

Position items along the main and cross axes with justify-content, align-items, align-self, and gap.

`justify-content` distributes free space on the main axis. `align-items` aligns items on the cross axis.

```
.hero {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: 2rem;
}
```

With `flex-direction: row`, this distributes items horizontally and aligns them vertically. Switch to `column` and the axes switch too.

`justify-content` has visible work only when free space remains after item sizing. If one item grows to consume all available space, changing `space-between` may appear to do nothing.

`align-items` defaults to `stretch` for auto-sized items. `align-self` overrides cross-axis alignment for one item. Auto margins absorb available space on their axis before distribution, which makes `margin-inline-start: auto` useful for pushing one navigation action away from its siblings.

`gap` creates consistent spacing between items without adding space outside the group.

To center one child in both directions:

```
.container {
  display: flex;
  justify-content: center;
  align-items: center;
```

```
}
```

Give the container a visible outline and a definite minimum height, then enable the Flexbox overlay. Toggle direction and alignment values. Describe the free space and axes first; the resulting position should then be predictable.

Grow, shrink, and basis

Control how flex items claim free space or surrender space with flex-grow, flex-shrink, flex-basis, and the flex shorthand.

Flex sizing begins from each item's flex base size, compares the items with the container, then distributes positive or negative free space.

```
.sidebar { flex: 0 0 16rem; }
.main { flex: 1 1 30rem; }
```

The shorthand order is grow, shrink, basis.

The sidebar neither grows nor shrinks from `16rem`. The main area can grow and shrink from a starting basis of `30rem`.

`flex: 1` is commonly treated as `1 1 0`, so equal siblings share free space from a zero basis. `flex: auto` means `1 1 auto` and begins from the item's width or content size. Those are different layout decisions.

Grow factors divide positive free space. Shrink factors act on negative free space and are weighted by the base sizes, so two items with `flex-shrink: 1` do not necessarily lose the same number of pixels.

Long unbreakable content can prevent an item from shrinking because a flex item's automatic minimum often follows its min-content size. `min-inline-size: 0` allows it to shrink below that content-based minimum; the content still needs wrapping or intentional overflow handling.

In DevTools, inspect the Flexbox sizing information. Compare `flex: 1`, `flex: auto`, and `flex: none`, then insert a long URL. Do not add `overflow: hidden` merely to make the symptom disappear—identify which minimum size is constraining the item.

Wrap flex items

Let items form multiple lines, control line spacing, and decide when a one-dimensional wrapping layout should become a Grid instead.

Flex items stay on one line by default. Allow wrapping on the container:

```
.tags {
  display: flex;
  flex-wrap: wrap;
  gap: 0.5rem;
}
```

Give items a flexible basis when they need a useful minimum:

```
.tags > * {
  flex: 1 1 12rem;
}
```

The browser forms lines from available space. Each line is then sized independently. Items in different rows do not share column tracks, so wrapped cards may not align vertically across rows.

Use `align-content` to distribute multiple lines when the container has extra cross-axis space. It has no effect on a single line.

If you need consistent rows and columns together, Grid is usually a better model. Flexbox remains ideal for toolbars, navigation, button groups, and content that flows in one primary direction.

Resize slowly and zoom to 200%. Watch where content—not a named device—forces a new line. Keep source order meaningful because wrapping changes only visual lines, not reading or keyboard order.

Build a Flexbox header

Apply Flexbox to the course project header, keep navigation usable on narrow screens, and test flexible sizing with long content.

Turn the project header into a flexible row:

```
.site-header {
  display: flex;
  align-items: center;
  justify-content: space-between;
  gap: 1rem;
  flex-wrap: wrap;
  padding: 1rem;
}

.site-nav {
  display: flex;
  gap: 1rem;
  flex-wrap: wrap;
}
```

Test the header with a long site name, several links, browser zoom at 200%, and a narrow viewport. Wrapping is better than clipping or making targets overlap.

Do not use `order` to create a visual sequence that differs from keyboard and reading order. Change the HTML if the logical order is wrong.

Take this interactive quiz online

Grid

Build two-dimensional layouts with explicit tracks.

Start a Grid layout

Turn a parent into a grid container and distinguish explicit tracks, grid lines, cells, areas, and automatically placed items.

Grid defines a two-dimensional coordinate system of rows and columns.

```
.features {
```

```
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  gap: 1rem;
}
```

The declaration creates three explicit column tracks. Four numbered grid lines surround them, and cells appear where row and column tracks cross.

Direct children become grid items. Without explicit placement, the auto-placement algorithm fills available cells in source order and creates implicit rows when needed.

`fr` represents a fraction of leftover space after non-flexible tracks and gaps are resolved. Intrinsic content minimums can still make a track wider than an equal fraction, which is why a long unbreakable value can upset apparently equal columns.

Grid controls the items, not all descendants. A nested child participates in its grid item's layout unless you deliberately create another grid.

Enable the Grid overlay in DevTools. Show line numbers and track sizes, then add a fourth item. The overlay makes the explicit columns, implicit row, gaps, and auto-placement order visible.

Define flexible tracks

Build fixed and flexible tracks with fr, repeat, minmax, auto-fit, and auto-fill for layouts that adapt to available space.

Track sizes can mix fixed and flexible values:

```
.layout {
  display: grid;
  grid-template-columns: minmax(12rem, 1fr) 3fr;
}
```

`repeat()` avoids repeated values:

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

For a responsive card grid:

```
grid-template-columns: repeat(auto-fit, minmax(min(16rem, 100%), 1fr));
```

The grid creates as many tracks as fit. `auto-fit` collapses empty tracks so existing items expand. `auto-fill` keeps the empty tracks.

The inner `min()` prevents a `16rem` minimum from overflowing a container narrower than `16rem`.

Track minimums matter. A plain `1fr` track has an automatic minimum that can honor min-content sizing. When a track must be allowed to shrink below a long child's intrinsic minimum, use `minmax(0, 1fr)` and handle wrapping or overflow on the child.

Use a definite minimum only when the component truly cannot be useful below it. `repeat(auto-fit, minmax(16rem, 1fr))` will overflow a container narrower than `16rem`; the `min(16rem, 100%)` safeguard makes that constraint responsive.

In the Grid inspector, compare `auto-fit` and `auto-fill` with fewer items than available columns. Then add a long URL and compare `1fr` with `minmax(0, 1fr)`. The track overlay shows whether the constraint comes from the grid or its content.

Place grid items

Position and span items using numbered or named grid lines while preserving a sensible source order for automatic placement.

Place an item between grid lines:

```
.featured {
  grid-column: 1 / 3;
  grid-row: 1;
}
```

Or express the span directly:

```
.featured {
  grid-column: span 2;
}
```

Grid lines start at 1. Negative numbers count backward from the end, so `grid-column: 1 / -1` spans all explicit columns.

Named lines make placement more resilient than unexplained numbers in a large layout:

```
.layout {
  grid-template-columns:
    [sidebar-start] minmax(12rem, 1fr)
    [content-start] minmax(0, 3fr) [content-end];
}
```

Items without placement continue through auto-placement. Keep HTML in logical reading order, especially when placed items overlap or appear in a different visual position.

Avoid `grid-auto-flow: dense` when its visual backfilling would make reading or keyboard order confusing. CSS placement changes appearance, not DOM order.

Grid can deliberately layer items in the same cell. Control painting carefully, preserve contrast, and make sure overlapping interactive controls do not cover one another.

Turn on line numbers in DevTools and place one item with numbers, one with a span, and one with named lines. Then tab through the page to compare keyboard order with visual placement.

Named grid areas

Describe a page layout with readable named regions and assign header, navigation, main, aside, and footer elements to them.

Named areas make page-level layouts easy to read:

```
.page {
  display: grid;
  grid-template-columns: 16rem 1fr;
  grid-template-areas:
    "header header"
    "sidebar main"
    "footer footer";
}
```

```
.site-header { grid-area: header; }
.sidebar { grid-area: sidebar; }
.content { grid-area: main; }
.site-footer { grid-area: footer; }
```

Every quoted row must have the same number of cells. Repeating a name creates one rectangular area. A dot marks an empty cell.

Start from a one-column template that matches the document order, then redefine only the visual map when space permits:

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

@media (min-width: 48rem) {
  .page {
    grid-template-columns: minmax(12rem, 1fr) 3fr;
    grid-template-areas:
      "header header"
      "sidebar main"
      "footer footer";
  }
}
```

Areas change visual placement but not reading, focus, or accessibility-tree order. Preserve HTML that makes sense without Grid, even if a wide layout displays the sidebar first.

Use the Grid overlay to display area names. Intentionally make a non-rectangular area or misspell one assignment and inspect how the browser rejects or auto-places the result.

Build the feature grid

Turn the project features into a responsive card grid and compare the two-dimensional result with a wrapping Flexbox layout.

Style the project features as a responsive grid:

```
.features {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(min(16rem, 100%), 1fr));
  gap: 1.5rem;
}

.card {
  padding: 1.5rem;
  border: 1px solid currentColor;
```

```
}
```

Add cards with short and long content. The shared tracks keep columns aligned in a way independently wrapped flex lines would not.

Resize slowly and watch tracks appear or disappear based on available space. This layout may not need a breakpoint because the component responds through its track definition.

Take this interactive quiz online

Responsive and modern CSS

Adapt layouts and motion to screens and user preferences.

Responsive foundations

Build flexible pages that respond to content, container space, zoom, text settings, and input methods instead of a list of device models.

Responsive design is not only about phone widths. A page must handle long text, browser zoom, split-screen windows, large fonts, touch input, and unknown content.

Start with intrinsic, flexible foundations:

```
img {
  max-width: 100%;
  height: auto;
}

.page {
  width: min(100% - 2rem, 70rem);
  margin-inline: auto;
}
```

The image can shrink without exceeding its intrinsic proportions. The page follows its container until reaching a readable maximum. Neither rule needs to know a device name.

Let normal flow, wrapping, `min()`, `max()`, `clamp()`, Flexbox, and Grid solve as much as possible before adding a breakpoint. Prefer `max-width` or a flexible basis to a fixed width, and avoid fixed heights around text.

Do not disable zoom or assume a fixed viewport. Test by resizing, zooming, increasing text size, and replacing short sample content with realistic extremes.

Also test input differences. A coarse pointer needs generous targets, while a keyboard needs visible focus and logical order. Responsive design includes capabilities and preferences, not only geometry.

Use DevTools responsive mode as one source of evidence, then resize a real browser window and zoom to 200%. Emulation presets are examples, not the boundaries of the web.

Media queries

Apply styles when viewport or user media features match and choose content-driven breakpoints instead of device labels.

A media query conditionally applies a block of CSS:

```
.hero {
  display: block;
}

@media (min-width: 48rem) {
  .hero {
    display: grid;
    grid-template-columns: 1fr 1fr;
  }
}
```

This is mobile-first: the simple layout is the default, and the enhanced layout begins when enough space exists.

Choose the breakpoint where the content needs to change, not because a device list calls it “tablet.” A `rem` breakpoint follows the browser's initial font-size reference, including user defaults, rather than a font size you later set on `html`.

Media queries can also detect preferences and capabilities:

```
@media (prefers-reduced-motion: reduce) { }
@media (prefers-color-scheme: dark) { }
@media (hover: hover) and (pointer: fine) { }
@media print { }
```

Do not infer a whole device from one capability. A laptop may have both touch and a trackpad. Build the essential interaction for every input, then use a query for a genuine enhancement such as hover feedback.

In DevTools, drag the viewport until the layout—not the ruler—looks constrained. Record that width, test just below and above it, then emulate reduced motion, dark color scheme, and print. A query is correct only if both branches remain complete and usable.

Container queries

Adapt a reusable component to the space provided by its own container instead of coupling it to the full viewport width.

A component may appear in a wide main area or a narrow sidebar on the same viewport. Container queries let it respond to that local space.

```
.card-wrapper {
  container-name: card;
  container-type: inline-size;
}

@container card (min-width: 30rem) {
  .card {
    display: grid;
    grid-template-columns: 10rem 1fr;
  }
}
```

The query evaluates a named eligible ancestor, not the viewport. The rules apply to descendants of

that container, not the container itself. A wrapper is therefore useful when the component's outer box establishes the query and its child changes layout.

Use media queries for page-wide environment decisions and container queries for reusable component layout. They complement each other.

Size containment prevents the queried dimension from depending normally on descendant sizes, avoiding a loop where the child changes the container that changes the child. Check that adding `container-type` does not alter a wrapper that previously relied on its contents for sizing.

Container query units such as `cqi` represent a percentage of the query container's inline size. Use them with `clamp()` when a value should vary locally without becoming unbounded.

Place the same card in a sidebar and main column at one viewport width. Use the Container Queries panel to inspect which ancestor and threshold matched. The component should adapt differently without a viewport breakpoint.

Transitions, animations, and motion

Add purposeful state transitions and keyframe animation while respecting reduced-motion preferences and avoiding costly layout changes.

A transition animates a property change:

```
.button {
  transition: transform 150ms ease, background-color 150ms ease;
}

.button:hover {
  transform: translateY(-2px);
}
```

List the properties deliberately. `transition: all` can animate a later layout or visibility change that was never designed for motion.

Keyframes describe multi-step or repeating motion:

```
@keyframes spin {
  to { transform: rotate(1turn); }
}
```

`transform` and `opacity` often avoid layout work and are good first choices, though the browser decides how they are composited. Animating dimensions or position can cause layout and paint on every frame. Confirm with the Performance panel when motion is substantial.

Motion should explain state, not delay interaction. Large scaling, panning, parallax, and continuous animation can trigger discomfort. Never make animation the only signal that something changed.

Respect the user's preference:

```
@media (prefers-reduced-motion: reduce) {
  .button {
    transition: background-color 0s;
  }

  .progress-animation {
```

```
    animation: none;
  }
}
```

`reduce` asks you to remove, reduce, or replace non-essential motion. A targeted alternative is safer than a universal rule that can break functional animation events or component assumptions.

Emulate reduced motion in DevTools and navigate entirely by keyboard. Verify state changes remain clear with motion removed, then record a performance trace to see whether the normal version triggers layout or paint on each frame.

Finish the responsive page

Combine intrinsic Grid, one content-driven breakpoint, flexible media, focus styles, and reduced motion in the course project.

Review the course page from the smallest usable width upward.

Keep the card grid intrinsic with `auto-fit` and `minmax()`. Add a media query only where the hero or navigation needs a deliberate layout change. Make images flexible, links visibly focused, and controls large enough to use.

Then test:

- narrow and wide windows
- 200% browser zoom
- long headings and navigation labels
- keyboard-only navigation
- reduced motion
- light and dark browser preferences if your design supports them

Responsive work is finished when content remains usable across these conditions, not when it matches a handful of screenshots.

Take this interactive quiz online

Debugging and review

Inspect layout problems and review the complete course.

Debug the cascade

Trace a wrong visual value back through matched rules, specificity, inheritance, custom properties, layers, and source order.

When a value is wrong, inspect the element instead of adding a stronger rule blindly.

1. Find the property in the Computed panel.
2. Expand it to see which declaration supplied the value.
3. Inspect crossed-out competing declarations.
4. Check inherited values and custom-property definitions.
5. Confirm the selector matches the element you intended.

If the rule is absent, check whether the stylesheet loaded and whether an earlier syntax error affected parsing.

Fix the cause. A duplicate selector with higher specificity may hide the symptom, but it makes the next override harder.

Debug layout and overflow

Find unexpected width, spacing, alignment, clipping, and horizontal scrolling by inspecting boxes and simplifying layout constraints.

Start at the element that looks wrong and inspect its box model and layout badge.

For overflow, temporarily outline everything:

```
* {
  outline: 1px solid rgb(255 0 0 / 25%);
}
```

Look for fixed widths, long unbreakable strings, images without flexible sizing, flex items that need `min-width: 0`, and positioned elements outside their container.

For Grid and Flexbox, use the DevTools overlay to see tracks, gaps, axes, and free space.

Do not hide horizontal overflow on the whole page before finding its source. That removes the scrollbar but can leave content unreachable.

Bisect a CSS problem

Isolate a difficult styling bug by disabling half the possible causes, repeating the test, and reducing the page to a minimal reproduction.

When many rules could cause a bug, bisect them.

Disable half the relevant stylesheet or component rules. If the problem remains, the cause is in the enabled half. If it disappears, the cause is in the disabled half. Repeat until one small group remains.

Then create a minimal reproduction with only the necessary HTML and CSS. Remove wrappers, properties, and content one at a time while keeping the bug.

This process reveals interactions that are hard to spot in a full application: a containing block created by a transform, a flex item minimum, a stacking context, or an inherited custom property.

Keep the reduced example as a regression test or note when the bug could return.

Final CSS review

Review the completed page for cascade clarity, resilient layout, responsive behavior, accessibility, and maintainable component boundaries.

Review the finished project without looking only at its ideal desktop screenshot.

Check that:

- selectors are short and reusable
- the cascade works without escalating specificity
- widths include sensible minimums and maximums
- source order remains meaningful
- focus indicators are visible

- text contrast and line length remain readable
- content wraps instead of clipping
- layout survives zoom, long content, and narrow containers
- motion respects user preferences

Remove rules that no longer affect the page. Group related component styles and keep shared tokens in custom properties.

Finally, select one element in DevTools and explain how its color, type, size, spacing, and position were calculated. If you can trace those values, you can debug the page later.

Take this interactive quiz online

Take this interactive quiz online

Take this interactive quiz online

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