

CSS interview questions

25 questions · 9 easy · 11 medium · 5 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1 Explain the CSS box model.

Easy



#box model #layout

Every element is a rectangular box made of four layers, from the inside out:

- **content:** the text or image itself
- **padding:** space inside the border, takes the background colour
- **border**
- **margin:** transparent space outside the border, separating it from other boxes

By default (`box-sizing: content-box`), `width` and `height` apply to the **content only**, so padding and border are added on top. A `width: 200px` box with `20px` padding and a `1px` border is actually 242px wide.

DevTools shows this as the coloured box diagram in the Computed panel.

? Likely follow-up: What is margin collapsing?

2 What does `box-sizing: border-box` do, and why do most resets use it?

Easy



#box model

With `border-box`, the `width` and `height` you set **include padding and border**. Adding padding shrinks the content area instead of making the box bigger.

That makes layouts predictable: two `width: 50%` columns with padding still fit side by side, which they would not with the default `content-box`.

That is why almost every reset (and Tailwind's preflight) applies it to everything, including pseudo-elements. Margin is never included in either model.

Example

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

.card {
  width: 300px;
  padding: 20px;
  border: 1px solid #ddd; /* still exactly 300px wide */
}
```

3

How does CSS specificity work?

Medium 🔥🔥 #specificity #cascade

When several rules target the same element, the one with the highest specificity wins. It is compared as three numbers (**IDs, classes, elements**):

- **IDs:** `#nav` is (1,0,0)
- **classes, attributes, pseudo-classes:** `.btn`, `[type="text"]`, `:hover` are (0,1,0)
- **elements and pseudo-elements:** `p`, `::before` are (0,0,1)

The columns are compared left to right, so **one ID beats any number of classes**. Inline `style` beats all of them, and `!important` beats inline. With equal specificity, the rule that comes last wins.

`*` and combinators add nothing; `:where()` counts as zero, and `:is()` / `:not()` take their most specific argument.

Example

```
p { color: black; } /* 0,0,1 */
.intro { color: blue; } /* 0,1,0 */
article p.intro { color: green; } /* 0,1,2 wins over .intro */
#hero p { color: red; } /* 1,0,1 wins over all above */
:where(#hero) p { color: gray; } /* 0,0,1 :where adds zero */
```

🔗 **Likely follow-up:** Why is overusing `!important` a problem, and what are cascade layers (`@layer`)?

4

What is the cascade, and which properties are inherited?

Medium



#cascade

The **cascade** is the algorithm that decides which declaration wins when several apply. In order of priority:

1. origin and importance (`!important` , author vs browser styles)
2. cascade layers (`@layer`)
3. specificity
4. source order: later wins

Inheritance is separate: some properties pass from parent to child when the child has no value of its own. Mostly **text-related properties inherit** (`color` , `font-*` , `line-height` , `visibility`), while box properties (`margin` , `padding` , `border` , `background` , `width`) do not.

You can force it with `inherit` , reset with `initial` , or use `unset` (inherit if inheritable, else initial).

5

Explain `position` : static, relative, absolute, fixed and sticky.

Easy



#positioning #layout

- `static` (default): normal flow; `top` / `left` and `z-index` have no effect
- `relative` : stays in the flow, but can be nudged with offsets. Its main job is being the **containing block** for absolute children
- `absolute` : removed from the flow, positioned relative to the **nearest positioned ancestor** (any non-static position), or the page if there is none
- `fixed` : removed from the flow, positioned relative to the viewport, so it stays put on scroll
- `sticky` : behaves like relative until a scroll threshold (`top: 0`), then sticks within its parent

A common gotcha: a parent with `transform` or `filter` becomes the containing block even for `fixed` children.

Example

```
.card { position: relative; }

.card .badge {
  position: absolute;
  top: 8px;
  right: 8px;          /* corner of .card, not the page */
}

.table-header { position: sticky; top: 0; }
```

🔗 Likely follow-up: Why might `position: sticky` not work? (overflow on an ancestor, no `top` set)

6

What are the common `display` values?

Easy



#display #layout

- `block`: new line, full available width, respects width/height
- `inline`: flows in text, ignores width/height and vertical margins
- `inline-block`: in the text flow but respects width, height and margins
- `flex` / `inline-flex`: one-dimensional layout of its children
- `grid` / `inline-grid`: two-dimensional layout of its children
- `none`: removed completely, takes no space and is hidden from screen readers
- `contents`: the box disappears and its children act as children of the parent

Note that `display` has an outer part (how the box sits among siblings) and an **inner part** (how its children are laid out): `inline-flex` is inline outside, flex inside.

7

`display: none` vs `visibility: hidden` vs `opacity: 0`?

Easy



#display #accessibility

- `display: none`: removed from layout, **takes no space**, not clickable, hidden from screen readers. Cannot be transitioned (until recent `transition-behavior` support).
- `visibility: hidden`: invisible but **keeps its space**, not clickable, hidden from screen readers. Children can override it with `visibility: visible`.

- `opacity: 0`: invisible but keeps its space, **still clickable and focusable**, still read by screen readers. Can be animated smoothly.

To hide something visually but keep it for screen readers, use a `.sr-only` (visually-hidden) class instead.

8

Flexbox vs Grid: when do you use each?

Medium



#flexbox #grid #layout

Flexbox is **one-dimensional**: it lays items out in a row **or** a column, and sizes them based on their content. Great for nav bars, toolbars, button groups, centering, and lists that wrap.

Grid is two-dimensional: you define rows **and** columns, and place items into that structure. Great for page layouts, card galleries, dashboards and anything that must line up in both directions.

A rule of thumb: flexbox is **content-out** (items decide their size), grid is **layout-in** (the container decides). They combine well: a grid page layout with flex inside each card.

Example

```
/* flex: a row of items, spacing between them */
.toolbar { display: flex; align-items: center; gap: 8px; }
.toolbar .spacer { flex: 1; }

/* grid: responsive cards, no media query */
.cards {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(220px, 1fr));
  gap: 16px;
}
```

Likely follow-up: What is the difference between `auto-fill` and `auto-fit`?

9

What does `flex: 1` mean? Explain grow, shrink and basis.

Medium



#flexbox

`flex` is shorthand for three properties:

- `flex-grow`: how much of the **leftover space** the item takes, relative to siblings

- `flex-shrink`: how much it shrinks when there is **not enough** space
- `flex-basis`: its starting size before growing or shrinking (`auto` means use its width or content)

`flex: 1` means `1 1 0%`: start from zero and share the space equally, so **items get equal widths** regardless of content. `flex: auto` (`1 1 auto`) starts from content size, so bigger content gets a bigger box. `flex: none` makes an item rigid.

Gotcha: flex items have `min-width: auto`, so long text can overflow. Fix with `min-width: 0`.

10

How do you center a div horizontally and vertically?

Easy



#layout #flexbox #grid

The modern answers:

- **Flexbox** on the parent: `display: flex; justify-content: center; align-items: center`
- **Grid** on the parent: `display: grid; place-items: center` (the **shortest option**)
- **Absolute** positioning: `position: absolute; inset: 0; margin: auto` with a set size, or `top: 50%; left: 50%; transform: translate(-50%, -50%)` for unknown sizes

For horizontal-only centering of a block with a width, `margin: 0 auto` still works. For inline content and text, `text-align: center`.

Remember the parent needs a height for vertical centering to be visible.

Example

```
.parent {
  display: grid;
  place-items: center;
  min-height: 100vh;
}

/* or */
.parent {
  display: flex;
  justify-content: center; /* main axis */
  align-items: center;    /* cross axis */
}
```

11

What is the difference between `px`, `em`, `rem`, `%`, `vh` and `vw`?

Easy



#units #responsive

- `px`: fixed (a CSS pixel, not a device pixel)
- `em`: relative to the **current element's font size** (for `font-size` itself, the parent's), so it compounds when nested
- `rem`: relative to the **root font size** (on `html`), usually 16px, so it is predictable and respects the user's browser font setting
- `%`: relative to the parent (width for widths and padding, font size for font size)
- `vh` / `vw`: 1% of the viewport height / width

Common practice: `rem` for font sizes and spacing, `em` for things that should scale with their own text (button padding), `px` for borders. On mobile, `100vh` can be taller than the visible area; use `100dvh`.

🔗 Likely follow-up: Why is `rem` better for accessibility than `px` for font sizes?

12

How do media queries work, and what is mobile-first?

Easy



#responsive #media queries

Media queries apply styles only when a condition is true, usually the viewport width:

```
@media (min-width: 768px) { ... }
```

Mobile-first means writing the base styles for small screens and adding `min-width` queries to enhance for larger ones. Benefits:

- the simplest layout is the default, so phones load the least CSS overrides
- it forces you to prioritise content
- **styles build up instead of being undone**

Desktop-first uses `max-width` queries and tends to pile up overrides. Beyond width, you can query `prefers-color-scheme`, `prefers-reduced-motion`, `hover` and `orientation`. Pick breakpoints where your content breaks, not for specific devices.

Example

```
.grid { display: grid; gap: 16px; } /* phone: 1 column */

@media (min-width: 640px) {
  .grid { grid-template-columns: 1fr 1fr; } /* tablet */
}

@media (min-width: 1024px) {
  .grid { grid-template-columns: repeat(4, 1fr); }
}

@media (prefers-reduced-motion: reduce) {
  * { animation: none !important; }
}
```

13

Why is my `z-index: 9999` not on top? Explain stacking contexts.

Hard 🔥🔥 #z-index #stacking

`z-index` only compares elements **within the same stacking context**. A stacking context is a group that is layered as a single unit: **children can never escape their parent's context**, however high their `z-index`.

New stacking contexts are created by, among others:

- `position` relative/absolute with a `z-index` other than `auto`, and all `fixed` / `sticky` elements
- `opacity` below 1, `transform`, `filter`
- flex or grid children with a `z-index`
- `isolation: isolate`

So a modal with `z-index: 9999` inside a parent with `transform` and `z-index: 1` stays below a sibling with `z-index: 2`. Fixes: move the modal to the end of `<body>` (a React portal) or use `<dialog>`, which renders in the top layer.

14

What is the difference between pseudo-classes and pseudo-elements?

Easy 🔥🔥 #selectors

A **pseudo-class** (single colon) selects an element in a certain **state or position**: `:hover`, `:focus-visible`, `:checked`, `:disabled`, `:first-child`, `:nth-child(2n)`,

`:not(.active)`.

A **pseudo-element** (double colon) targets a **part** of an element, or creates a virtual one:

`::before`, `::after`, `::first-line`, `::placeholder`, `::selection`, `::marker`.

`::before` and `::after` need a `content` property (even `content: ''`) to render. They are great for decorations and icons, but their text is not reliably selectable or accessible, so do not put real content there.

Example

```
a:hover { text-decoration: underline; }           /* state */
li:nth-child(odd) { background: #f6f6f6; }        /* position */

.required::after {                               /* generated part */
  content: ' *';
  color: crimson;
}
```

15

What is the difference between transitions and animations?

Medium



#animation #performance

A **transition** animates a property between two states when it **changes**, for example on `:hover` or when a class is added. It needs a trigger and only has a start and an end.

An **animation** uses `@keyframes` with as many steps as you want. It can **run on its own**, loop (`infinite`), alternate direction, pause and be delayed.

Use transitions for simple interactive feedback, animations for loaders, attention effects and multi-step sequences.

For smooth 60fps, animate only `transform` and `opacity`, which the compositor can handle without layout or paint. Respect `prefers-reduced-motion`.

Example

```
.btn { transition: transform 150ms ease-out; }  
.btn:hover { transform: translateY(-2px); }  
  
@keyframes spin {  
  to { transform: rotate(360deg); }  
}  
  
.loader { animation: spin 1s linear infinite; }
```

16

What is the difference between reflow (layout) and repaint?

Hard



#performance #rendering

- **Reflow / layout:** the browser recalculates the size and position of elements. Triggered by changing geometry (`width`, `margin`, `top`, font size, adding DOM nodes, resizing the window). It is the **most expensive step** and can cascade to many elements.
- **Repaint:** redrawing pixels without changing layout, for example a new `color` or `background`.
- **Composite only:** `transform` and `opacity` can often skip both.

Layout thrashing happens when JavaScript alternates writes and reads (`el.style.width = ...` then `el.offsetHeight`), forcing a synchronous layout each time. Batch reads, then writes, or use `requestAnimationFrame`.

? Likely follow-up: What does `will-change` do, and when should you avoid it?

17

What are CSS custom properties (variables), and how are they different from Sass variables?

Medium



#variables #theming

Custom properties are declared with `--name` and read with `var(--name, fallback)`.

Unlike Sass variables, which are replaced at **build time**, CSS variables **live in the browser at runtime**:

- they **cascade and inherit**, so you can override them in a subtree (`.dark { --bg: #111 }`)
- they can change in media queries and be updated from JavaScript with `style.setProperty`

- they are visible and editable in DevTools

This makes them the standard way to do theming (dark mode, brand colours) and design tokens. Sass variables are still handy for build-time things like loops and breakpoints in media queries, where `var()` cannot be used.

Example

```
:root {
  --bg: #fff;
  --text: #111;
  --space: 1rem;
}

@media (prefers-color-scheme: dark) {
  :root { --bg: #111; --text: #eee; }
}

body { background: var(--bg); color: var(--text); padding: var(--space);
}
```

18

What is margin collapsing?

Medium



#box model #layout

When **vertical** margins of block elements touch, they **combine into one margin** equal to the larger of the two, instead of adding up. A `margin-bottom: 20px` followed by `margin-top: 30px` gives a 30px gap, not 50px.

It happens between adjacent siblings, between a parent and its first or last child (the child's margin "leaks" out of the parent), and in empty blocks.

It does **not** happen with horizontal margins, in flex or grid containers, or when the parent has padding, a border or creates a new block formatting context (for example `display: flow-root`).

Many teams avoid the issue by using margin in one direction only, or `gap` in flex and grid layouts.

19

What is BEM, and why use a CSS naming convention?

Medium



#architecture #naming

BEM stands for Block, Element, Modifier:

- **block**: a standalone component, `.card`
- **element**: a part of it, joined with two underscores, `.card__title`
- **modifier**: a variant, joined with two dashes, `.card--featured`

The goal is **flat, low-specificity selectors** (one class each) that say exactly where a style belongs. That avoids specificity wars and makes CSS safe to change in a big codebase.

Today, scoping often comes from tooling instead: CSS Modules generate unique class names, Tailwind uses utility classes, and CSS-in-JS ties styles to components. BEM is still common in plain CSS and Sass projects.

Example

```
<article class="card card--featured">
  <h2 class="card__title">Title</h2>
  <button class="card__button card__button--primary">Read</button>
</article>
```

Likely follow-up: How do CSS Modules or Tailwind solve the same problem?

20

CSS Modules vs Tailwind vs CSS-in-JS: what are the trade-offs?

Medium



#architecture #tooling

- **CSS Modules**: plain CSS files whose class names are scoped automatically at build time. Zero runtime, familiar syntax; styles live in a separate file.
- **Tailwind**: utility classes in the markup (`flex gap-4 p-2`). Very fast to build with, consistent design tokens, tiny final CSS. Markup gets verbose, and there is a learning curve.
- **CSS-in-JS** (styled-components, Emotion): styles written in JavaScript, easy dynamic styling from props. Runtime libraries add bundle size and cost, and they **do not fit well with React Server Components**. Zero-runtime options (vanilla-extract, Panda) avoid that.

There is no single right answer: interviewers want to hear that you know the trade-offs and what your team chose and why.

21

How do you make images responsive in CSS? What does `object-fit` do?

Medium



#images #responsive

The baseline is `img { max-width: 100%; height: auto; }`, so images never overflow their container and keep their ratio.

When an image must fill a box of a fixed shape (a card thumbnail, an avatar), use:

- `object-fit: cover`: fills the box and **croops the overflow**, keeping the ratio
- `object-fit: contain`: fits entirely inside, leaving empty space
- `object-position` to choose which part stays visible

`aspect-ratio: 16 / 9` reserves the right height before the image loads, preventing layout shift. For background images, `background-size: cover` does the same as `object-fit: cover`.

Serving the right file size is HTML's job (`srcset`), not CSS's.

Example

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

.thumb {
  width: 100%;
  aspect-ratio: 16 / 9;
  object-fit: cover;
  object-position: center top;
}
```

22

What is a block formatting context, and how did we clear floats?

Hard



#layout #floats

A **block formatting context (BFC)** is an independent layout region: floats inside it are contained, and margins do not collapse across its edge.

Floats were the old way to build columns. A floated child is taken out of normal flow, so its parent **collapses to zero height**. Fixes, oldest to newest:

- the **clearfix** hack: a `::after` with `content: ''; display: table; clear: both`
- `overflow: hidden` on the parent (creates a BFC, but clips content)
- `display: flow-root`, which creates a BFC with no side effects

Today, use flex or grid for layout. Floats are only for wrapping text around an image.

23

What does the `:has()` selector do?

Medium



#selectors #modern css

`:has()` is the long-awaited **parent selector**: it matches an element based on what it contains, or what comes after it.

- `.card:has(img)`: cards that contain an image
- `form:has(:invalid)`: a form with any invalid field
- `label:has(+ input:checked)`: a label followed by a checked input
- `body:has(dialog[open])`: lock scrolling while a modal is open

It removes a lot of JavaScript that only existed to toggle a class on a parent. It is supported in all modern browsers. Its specificity is that of its most specific argument, and very broad uses (like on `body`) can cost performance.

Example

```
.field:has(input:invalid) label { color: crimson; }

.card:has(img) { grid-template-rows: 200px auto; }

body:has(dialog[open]) { overflow: hidden; }
```

24

What are container queries, and how do they differ from media queries?

Hard



#responsive #modern css

Media queries respond to the **viewport**. Container queries respond to the **size of a parent container**.

That matters for reusable components: the same card might sit in a wide main column or a narrow sidebar on the same screen. With media queries it cannot know; with container queries it adapts to the space it actually has.

How it works:

- mark the parent with `container-type: inline-size` (optionally `container-name`)
- write `@container (min-width: 400px) { ... }` for the children
- units like `cqi` are relative to the container size

Use media queries for page-level layout, container queries for component-level layout.

Example

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

.card { display: grid; gap: 8px; }

@container (min-width: 400px) {
  .card { grid-template-columns: 120px 1fr; } /* side-by-side when roomy
*/
}
```

25

How do you keep CSS fast and maintainable in a large app?

Hard



#performance #architecture

Performance:

- ship less CSS: remove unused styles (Tailwind and CSS Modules help), split per route
- inline the critical CSS for above-the-fold content, load the rest without blocking
- animate `transform` and `opacity`, avoid layout thrashing
- `content-visibility: auto` to skip rendering off-screen sections
- reserve space for images and fonts to avoid layout shift (CLS)

Maintainability:

- low, flat specificity and a naming convention or scoped styles
- design tokens as CSS variables for colours, spacing and typography
- `@layer` to control the order of resets, libraries and components
- a linter such as Stylelint

Selector matching speed rarely matters in practice; the size of the CSS and layout work do.