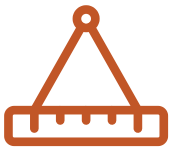

THE COMPLETE REFERENCE

Every CSS unit, explained.

All 63 units of CSS — what each one measures against, when it earns its place, and where it quietly breaks.

WRITTEN BY

Mykola Ptushchuk

WEB EDITION

cssunits.com

THIS PRINTING

August 2026

Contents

04 Viewport Units

05 Container Query Units

06 The fr Unit

07 Special Units

– Units in Math Functions

– Decision Tree

–Cheatsheet

–Glossary

CHAPTER 1 OF 7 · 7 UNITS

Absolute Units

On screens, an inch is a gentleman's agreement.

Absolute units look simple: fixed sizes, no context needed. But they hide one important nuance. On screens, they don't always match their physical counterparts. The whole system is anchored to a single equation:

1in = 96px

Every other absolute unit is defined relative to this anchor. So `1in`, `2.54cm`, `25.4mm`, `72pt` and `6pc` all compute to exactly `96px` on a screen, regardless of your monitor's physical size.

> MENTAL MODEL
>
> On screens, absolute units are pixel aliases in disguise. `1in`, `2.54cm`, `72pt` — all just creative ways to write `96px`. Only in print (where the output dimensions are physically known) do they map to real-world measurements.

```
Demo — CSS absolute units rendered at their actual size
Runs live at cssunits.com/article/absolute
```

```
Demo — Live unit converter
Runs live at cssunits.com/article/absolute
```

px

FORMULA	`1px = 1/96in`
RELATIVE TO	Reference pixel
SUPPORT	● Universal

The CSS pixel is the foundation of screen interfaces, but it is **not** a physical dot on your display.

CSS defines a *reference pixel* as the visual angle of one pixel on a 96 dpi device viewed from a distance of 28 inches (roughly arm's length). [1 That 28-inch arm is a real anthropometric estimate written into the spec: the pixel is defined as a unit of *seeing*, not of hardware.] A useful mental model is that `96px` behaves like one "logical inch", but a logical inch is not a real-world inch: on most screens `96px` won't match a ruler (see the `in` section below).

On a standard 1× display, 1 CSS px = 1 device pixel. On a HiDPI display (Retina MacBook, iPhone, high-end Android) the ratio is **2 or more per axis**: at 2×, a 1×1 px square is painted with two device pixels across and two down — four in total. [2 Worth stating plainly, because "a CSS pixel is two device pixels" is the usual shorthand and it undercounts the area by half. The ratio is linear; the pixels are a grid.]

That ratio is `window.devicePixelRatio`, and it is not a property of the screen alone. The spec defines it as the size of a CSS pixel *at the current page zoom* divided by the size of a device pixel — so zooming the page changes it on hardware that has not moved. Read it as "device pixels per CSS pixel right now", not as "what kind of display is this".

```
Demo — Border thickness in px
Runs live at cssunits.com/article/absolute
```

✓ USE WHEN

- ✓ Borders: `border: 1px solid`
- ✓ Icon sizes: `width: 24px; height: 24px`
- ✓ Precise details: outlines, shadows, hairlines
- ✓ Values that must stay constant in CSS coordinates

✕ AVOID WHEN

- ✗ `font-size` for body text — it ignores the user's default-font-size setting (use `rem`)
- ✗ Layout widths that should be fluid — `%`, `rem`, or `fr` adapt; a fixed `px` width doesn't
- ✗ Sizes that should follow the user's default font size

PITFALL

`1px` on a 2× display is drawn two device pixels wide, not one. A `100px` element is not “100 dots on screen”: it is “100 CSS pixels,” which the browser maps to device pixels via `devicePixelRatio` — 200 of them across on a 2× display, and 400 on that same display with the page zoomed to 200%. If you need a true one-device-pixel hairline, you need `1px / devicePixelRatio`; on a 2× display that’s `transform: scaleY(0.5)`, but the divisor is DPR-specific (a 3× phone needs `1/3`), so it’s usually handled with a `min-resolution` media query.

in

FORMULA	`1in = 96px = 2.54cm`
RELATIVE TO	CSS pixel (screen) / physical inch (print)
SUPPORT	● Universal

The CSS inch. On screens, `1in` is exactly `96px`, which may or may not be a real inch on your physical monitor.

Hold a ruler to your screen and measure a `1in` element. On most monitors, it won't match. The CSS inch is calibrated for the reference pixel model, not your device's physical dimensions.

```
Demo — 1in at actual CSS size
Runs live at cssunits.com/article/absolute
```

✓ USE WHEN

- ✓ Print stylesheets: `@media print`
- ✓ PDF generation and physical-output layouts
- ✓ As an anchor for understanding the unit system

✕ AVOID WHEN

- ✗ Screen UI — write `96px` instead for clarity
- ✗ Matching physical specs on screen (it won't be accurate)

PITFALL

On screens, `1in` is a *logical* inch, not a *physical* inch. Developers trying to match a physical spec (“make this 0.5 inches wide”) are often surprised that the element doesn’t match a real ruler.

`cm`

FORMULA	`1cm = 96px ÷ 2.54 ≈ 37.795px`
RELATIVE TO	CSS inch
SUPPORT	● Universal

The CSS centimeter. Since `1in = 2.54cm`, one CSS centimeter equals `≈ 37.795px`. Like `in`, it maps to a real centimeter only in print contexts.

That print context is where `cm` earns its keep: `@page { margin: 2cm }` reads exactly like the paper document it produces, and anyone who has ever set margins in a word processor can review it without converting anything.

```
Demo — 5cm with mm subdivisions
Runs live at cssunits.com/article/absolute
```

✓ USE WHEN

- ✓ Print layouts with metric measurements
- ✓ Physical templates and packaging specs

✗ AVOID WHEN

- ✗ Screen UI — on screen `1cm` is just `≈ 37.8px` with extra indirection
- ✗ Grid systems or component sizing

---

mm

FORMULA	`1mm = 1cm ÷ 10 ≈ 3.78px`
RELATIVE TO	CSS centimeter
SUPPORT	● Universal

One tenth of a CSS centimeter, or `≈ 3.78px`. Small enough to be useful for margins, gutters, and bleed areas in print layouts.

This is the unit the physical world is specced in: a European business card is `85mm × 55mm`, a typical print bleed is `3mm`. Write those numbers as they appear on the spec sheet and the CSS documents itself.

```
Demo — 20mm with 1mm ticks
Runs live at cssunits.com/article/absolute
```

✓ USE WHEN

- ✓ Precise print measurements: business cards, labels, forms
- ✓ Physical packaging templates

✗ AVOID WHEN

- ✗ Screen interfaces — not a meaningful grid unit at ~3.78px
- ✗ Component sizing

Q

FORMULA `1Q = 1mm ÷ 4 ≈ 0.945px`

RELATIVE TO CSS millimeter

SUPPORT ● Universal all engines since Mar 2020

A quarter-millimeter — the smallest absolute unit in CSS. At `0.945px`, it sits below one CSS pixel, so rendering depends on sub-pixel anti-aliasing and varies across browsers.

The `Q` unit originates from Japanese print typography, where the *kyu* (Q) is a traditional typesetting unit. [3 級 (kyū) means "grade"; one kyū is exactly 0.25mm. The romanization made `Q` the only CSS length written with a capital letter — and the only one named in Japanese.] It landed in the CSS spec to serve Japanese print workflows. It was the last absolute unit to become universal: Firefox shipped it in 2016, Chrome in 2017, and Safari only in 13.1 (March 2020). Internet Explorer never supported it, which matters only for archived stylesheets now.

Demo — Q vs px vs mm at actual size

Runs live at cssunits.com/article/absolute

✓ USE WHEN

- ✓ Specialized CJK or Japanese print typography
- ✓ Typographic systems requiring quarter-millimeter precision

✗ AVOID WHEN

- ✗ Screen UI — at `≈ 0.945px` it renders sub-pixel, so results vary by browser
- ✗ Any context where sub-pixel rendering consistency matters

PITFALL

`1Q < 1px`. The browser must round or anti-alias sub-pixel values, and results vary across engines. If you're using `Q` for print, test in multiple browsers before shipping.

---

pt

FORMULA	`1pt = 1in ÷ 72 ≈ 1.333px`
RELATIVE TO	CSS inch
SUPPORT	● Universal

The typographic point: the traditional unit of print typography, inherited from physical typesetting. There are 72 points per inch, so `1pt ≈ 1.333px` in CSS.

Points are familiar from print design tools: InDesign, Illustrator and Word all size type in `pt`. (Figma is the exception people expect to see here — it sizes text in pixels, and only converts to points on PDF export.) `12pt` body text and `24pt` headings are deeply ingrained print conventions.

Demo — Type size in points

Runs live at cssunits.com/article/absolute

✓ USE WHEN

- ✓ Print stylesheets and PDF generation
- ✓ Documents where the spec is defined in points
- ✓ Matching a print design spec exactly

✕ AVOID WHEN

- ✗ Web typography — `12pt = 16px` but won't scale with user prefs
- ✗ Pasting point values directly from Figma into browser CSS

PITFALL

`12pt` in CSS equals `16px`, coincidentally the browser's default body font size. This makes point values *look* correct on screen while silently ignoring user font preferences. Convert print specs to `rem` for the web.

---

pc

FORMULA	`1pc = 12pt = 1in ÷ 6 = 16px`
RELATIVE TO	CSS point and inch
SUPPORT	● Universal

The pica, equal to 12 points. `1pc = 16px`, which happens to equal the browser's default body font size: a coincidence worth knowing.

Picas were used to measure columns, gutters, and large typographic elements in traditional print. In web CSS, they appear almost exclusively in print stylesheets or code from developers with a print background.

Demo — `1pc` = `12pt` = `16px`

Runs live at cssunits.com/article/absolute

✓ USE WHEN

- ✓ Print layouts using a pica-based grid system
- ✓ Documents that specify column widths in picas

✗ AVOID WHEN

- ✗ General web layout — `1pc` is a fixed `16px`, so it carries the same trade-offs as `px`
- ✗ In code that mixes units — `pc` is easy to misread as `px` (it's `16px`, not `1px`)

PITFALL

`pc` looks like an abbreviation for "percent" or "pixel" at a glance. `1pc = 16px`, not `1px` — easy to misread when reviewing unfamiliar code.

Conversion reference

All absolute units mapped to CSS pixels:

UNIT	DEFINITION	CSS PIXELS
`1px`	`1in / 96`	**1 px**
`1pt`	`1in / 72`	**≈ 1.333 px**
`1Q`	`1mm / 4`	**≈ 0.945 px**
`1mm`	`1cm / 10`	**≈ 3.78 px**
`1pc`	`12pt`	**16 px**
`1cm`	`96px / 2.54`	**≈ 37.795 px**
`1in`	— anchor —	**96 px**

When to reach for absolute units

In modern web development, absolute units have a narrow role:

- `px` — the workhorse. Borders, shadows, icon sizes, anything that should stay constant regardless of font size or viewport.
- `pt`, `cm`, `mm`, `in`, `pc`, `Q` — mainly `@media print` and physical-output contexts. On screens they all resolve through the `1in = 96px` anchor, so they're pixel values with extra steps.
- **Accessibility:** here's the precise distinction. Page zoom *does* scale `px` text (the whole page magnifies). What `px` ignores is the user's **default-font-size** setting: someone who sets their browser default to 20px still sees your `font-size: 16px` as 16px. `rem`/`em` honor that preference, which is why they're the safer default for type.

Sources

- CSS Values and Units Module Level 4 — absolute lengths `(w3.org)`

- CSS Values and Units Module Level 4 — reference pixel (w3.org)
- MDN — CSS length values (developer.mozilla.org)
- CSSOM View — `window.devicePixelRatio` (drafts.csswg.org) — the algorithm that makes it follow page zoom
- Smashing Magazine — There is no such thing as a CSS absolute unit (smashingmagazine.com) — the same argument, at length

CHAPTER 2 OF 7 · 12 UNITS

Font-relative Units

Set sizes as ratios, and the page keeps its shape when the reader changes the rules.

`em` and `rem` are the two units through which CSS shows its true nature: a system of coordinates *and* a system of inheritance. In `px`, a size is a number. In `em` and `rem`, a size becomes a **ratio** — to the local typography, or to the document's root.

The short version:

```
em  = multiply by the element's font-size
rem = multiply by the root element's font-size
```

But the most important nuance is this:

```
em on font-size resolves against the INHERITED font-size;
em on other properties resolves against the COMPUTED font-size of the
element itself.
```

This is one of the most important facts in the whole article. It explains both the power of `em` and the famous cascade trap.

---

em

FORMULA	`1em = element's computed font-size (or inherited, when set on font-size itself)`
RELATIVE TO	The element's own font-size
SUPPORT	● Universal

`em` is a local font-relative unit. It measures length in terms of the element's font-size.

If an element has computed `font-size: 20px`, then `1em = 20px`, `2em = 40px`, and `0.5em = 10px`.

The mental model:

> em = multiplier of the current font-size

Historically `em` came from typography: in print, an em was the size of the current type body. On the web it became a very practical idea: don't pin a size to absolute pixels; say how much bigger or smaller it should be than the surrounding text.

For most properties:

```
.button {
  font-size: 20px;
  padding: 0.5em 1em;
}
```

resolves as:

```
font-size      = 20px
padding-block  = 0.5em = 10px
padding-inline = 1em   = 20px
```

On the `font-size` property itself, the reference is necessarily different, and this falls out of how computed values work, not from a special rule baked into `em`:

```
.child {
  font-size: 1.5em;
}
```

Here `1.5em` can't be computed from the element's *own* font-size, which is the very value being computed. So it resolves against the **inherited** font-size, i.e. the parent's computed value:

> `child font-size = parent computed font-size × 1.5`

If the parent is `16px`, the child becomes `24px`.

The cascade trap

The trouble with `em` shows up when the same multiplier repeats through nesting:

```
.item { font-size: 1.5em; }
```

```
<div class="item">
  Level 1
  <div class="item">
    Level 2
    <div class="item">
      Level 3
    </div>
  </div>
</div>
```

With a base of `16px`:

```
Level 1: 16 × 1.5  = 24px
Level 2: 24 × 1.5  = 36px
Level 3: 36 × 1.5  = 54px
```

At five levels deep: `1.5⁵ = 7.59×`, and `16 × 7.59 ≈ 121.5px`.

That's the cascade trap: a small multiplier looks innocent, but inheritance turns it into an exponent.

```
Demo — em cascade: small multipliers, exponential growth
Runs live at cssunits.com/article/font-relative
```

`em` itself isn't the problem. The trap only triggers when `em` is used for `font-size` on components that nest.

`em` shines inside a component, where sizes should follow the local type. The canonical example is a button:

```css
.button { padding: 0.625em 1em; border-radius: 0.5em; }
```

Padding and border-radius are tied to the button's text: set the button's `font-size` once, and everything inside it follows. Bigger text → bigger internal whitespace. The component scales as a single typographic body. What that one `font-size` should itself be written in is a separate question, and `em` vs `rem` answers it.

Or an inline icon next to text:

```css
.label     { display: inline-flex; align-items: center; gap: 0.5em; }
.label svg { width: 1em; height: 1em; }
```

The icon scales with the text instead of staying fixed at `16px`, which is useful when the component ships in several sizes. (A fixed `px` icon is fine too, especially for sprite assets tuned to an exact pixel size.)

✓ USE WHEN

- ✓ Padding/gap/border-radius you *want* to scale with the component's own text
- ✓ Inline icons that should match the text size: `width: 1em; height: 1em`
- ✓ Any local component where one `font-size` should drive the internal scale

✗ AVOID WHEN

- ✗ Global type scale on nestable components — `em` compounds through nesting
- ✗ Sizes that must stay stable regardless of where the component is placed
- ✗ Text that mustn't shrink because of a deep parent — `rem` is safer (note: portaled tooltips usually escape this anyway)

Pitfall: `line-height: 1.5em` vs `line-height: 1.5`

For `line-height`, a unitless number is almost always better than a length:

```
body { line-height: 1.5; }   /* good */
body { line-height: 1.5em; } /* avoid */
```

The difference is inheritance. With `line-height: 1.5`, the **number** is inherited. Every element multiplies that number by its own font-size:

```
body { font-size: 16px; line-height: 1.5; }
h1   { font-size: 32px; }
```

```
body line-height = 16 × 1.5 = 24px
h1   line-height = 32 × 1.5 = 48px
```

With `line-height: 1.5em`, the *computed value* becomes a length and is inherited as a fixed length:

```
body { font-size: 16px; line-height: 1.5em; /* computed: 24px */ }
h1   { font-size: 32px; }
```

Now `h1` inherits a `24px` line-height, which is too tight for `32px` text.

PITFALL

For `line-height`, prefer unitless numbers. They inherit as a multiplier rather than a fixed length, so every element gets a line-height proportional to its own font-size.

IN SHORT

`em` isn't just a unit of length — it's a way of saying *"this size should live with the text."* Well-suited to a component's internals; risky for global typography *on nesting components*, because the multiplier compounds.

`rem`

FORMULA	`1rem = root element's computed font-size`
RELATIVE TO	The root element (typically <html>)
SUPPORT	● Universal

`rem` stands for "root em." It's a font-relative unit that always resolves against the font-size of the root element (typically `<html>`). [1 `rem` is young: browsers only picked it up around 2010. Before that, keeping nested `em` arithmetic in your head was simply part of the

job.]

The mental model:

> `rem = multiplier of the root font-size`

If the root is `16px`, then `1rem = 16px`, `2rem = 32px`, `0.75rem = 12px`. If the user (or the site) sets the root to `20px`, then `1rem = 20px`, `2rem = 40px`, `0.75rem = 15px`.

An important accessibility fact: the typical browser default is `16px`, but users can change it in browser settings. So while `1rem` is *often* `16px`, you shouldn't treat that as guaranteed. A better phrasing:

> `1rem = the root font-size`
> `the root font-size is often 16px by default`
> `but a user may have made it larger or smaller`

Why `rem` became the standard

`rem` solves the main pain of `em`: compounding.

```
.item { font-size: 1.5rem; }
```

```
<div class="item">
  Level 1
  <div class="item">
    Level 2
    <div class="item">
      Level 3
    </div>
  </div>
</div>
```

With a root of `16px`, every level is `24px`. No `24 → 36 → 54`; nesting stops mattering.

That's why `rem` became the obvious base for design systems:

```
:root {
  --space-1: 0.25rem;
  --space-2: 0.5rem;
  --space-3: 0.75rem;
  --space-4: 1rem;
  --space-6: 1.5rem;
  --space-8: 2rem;
}
```

This scale is stable anywhere in the DOM, scales with the root font-size, respects user preferences better than `px`, and is easier to read than nested `em`.

```
Demo — rem responds to root, px does not
Runs live at cssunits.com/article/font-relative
```

✓ USE WHEN

- ✓ `font-size` for body text and headings
- ✓ Global spacing scales and layout gaps
- ✓ Margins between sections, max-width on text containers
- ✓ Component sizes that must be stable across contexts
- ✓ Design tokens: `--space-*`, `--text-*`, `--radius-*`

✗ AVOID WHEN

- ✗ Internal component proportions that should follow the component's own font-size
- ✗ Anywhere you'd want `padding` to grow with a button's text — use `em` instead

Pitfall: rem is not “px, but nicer”

A common mistake is to read `rem` as if it were `px`:

```
.title {
  font-size: 2rem; /* I'm thinking "32px" */
}
```

The correct mental model is:

> `2rem = 2 × root font-size`

If the user changed their default font-size, `2rem` changes too. That’s not a bug — that’s the *point* of `rem`.

A related smell is hardcoding the root in `px`:

```
html { font-size: 16px; }
```

This makes `1rem` predictable for the designer but undermines the user’s browser default. Prefer one of:

```
html { font-size: 100%; }
```

or simply don’t set a root font-size unless you have a specific reason.

The root of this site, measured

This guide has a specific reason, so it’s fair to hold it to its own standard. Its root is fluid:

```
html { font-size: clamp(1rem, 0.9rem + 0.5vw, 1.25rem); }
```

One declaration scales the whole system with the window. The reading measure, the vertical rhythm, the header and every demo are sized in `rem`, so all of them follow, with no breakpoints involved. (The rail on this page shows what `1rem` currently resolves to on your screen.)

What that preserves is the thing the advice above is really about. Every term is in `rem`, and the floor is exactly `1rem`, so the page never renders smaller than the reader's own default, and it grows when they raise it:

BROWSER DEFAULT	THIS SITE'S ROOT
16px	20px
20px	25px
24px	28.8px

A `px` root would have pinned all three rows to one number.

The cost shows up under page zoom. Zoom scales `rem` but not `vw`, because magnifying the page shrinks the layout viewport by the same factor, so the `0.5vw` term contributes nothing to growth. Measured on a 1440px window with a 16px default:

PAGE ZOOM	ROOT, CSS PX	RENDERED	GROWTH
100%	20	20	100%
200%	18	36	**180%**
225%	17.6	39.6	198%
250%	17.3	43.2	216%

At 200% zoom the text is 180% of its original size, and a reader who wants it twice as large gets there at about 225%. Whether that trade is acceptable is a judgment call. That it exists is not.

The mechanism is the `clamp()` branch rather than the units. At 1440px the preferred term computes to 21.6px, above the ceiling, so the rendered value is a flat `1.25rem`, which would double perfectly on its own. Zoom knocks the value off that ceiling into the fluid band, where the `vw` term dilutes the growth. The `clamp()` playground shows the same effect on a headline.

The generalization is worth having, because the obvious fix isn't one:

```
/* Looks safer. Isn't. */
html { font-size: 100%; }
@media (min-width: 60rem) { html { font-size: 125%; } }
```

At 200% zoom the layout viewport drops below the breakpoint, the rule stops matching, and the root falls from 20px to 16px: 160% growth, with a visible jump on the way. **Any root that depends on the viewport pays this surcharge; only a root that ignores the viewport doubles exactly.**

Which leaves a choice rather than a rule. If you want the system to breathe with the window, accept that zoom grows text by less than its nominal factor, and measure where that lands. If you want zoom to be exact, keep the root viewport-independent and make individual elements fluid instead.

> IN SHORT
>
> `rem` is a compromise between designer control and user accessibility. It gives the designer a single shared scale, but it leaves the user the power to scale the whole system up.

`em` vs `rem` — when to use which

The core distinction:

```
em  = local scale
rem = global scale
```

`em` answers: *how big should this be relative to the text right here?* `rem` answers: *how big should this be relative to the page's base size?*

```
Demo — em compounds, rem stays locked to root
Runs live at cssunits.com/article/font-relative
```

A practical table:

TASK	USUALLY BETTER	WHY
Body text size	`rem`	Respects root / user preference; immune to nesting
Headings	`rem` or `clamp(...rem...)`	You want a shared typographic scale
Button padding	`em`	Should grow with the button's own text
Gap between icon and text	`em`	Icon and text live at one local scale
Inline SVG icon size	`em`	`width: 1em` matches the text
Page layout spacing	`rem`	Spacing should be stable across the site
Margin between sections	`rem`	This is the page's vertical rhythm
Component sized via `font-size`	`em` inside	One `font-size` controls everything internal
Tooltip / popover text	`rem`	Must not shrink because of a deep parent
Nested comments / tree UI	`rem` for text	Avoid compounding in deep nesting
`line-height`	unitless	Inherits as a multiplier, not a fixed length
Media query breakpoints	`em` or `rem`	Breakpoints can be tied to the user's text scale

The best CSS often uses both at once:

```
.button        { font-size: 1rem;     padding: 0.625em 1em; border-
radius: 0.5em; }
.button--small  { font-size: 0.875rem; }
.button--large  { font-size: 1.25rem;  }
```

> rem sets the component's size in the page's system;
> em sets the component's internal proportions.

"em vs rem" is often posed as an either/or. In practice the best answer is *both*, each doing its own job.

The 62.5% trick

A pattern you'll see in older codebases and tutorials: [2 Popularized around 2004 for `em` sizing, back when Internet Explorer could resize only text, never pages. A surprising share of '00s font-size lore is archaeology from that one limitation.]

```
html { font-size: 62.5%; }
```

If the browser default is `16px`, then `16 × 0.625 = 10px`, which makes `rem` convenient for mental arithmetic:

```
1rem   = 10px
1.4rem = 14px
1.6rem = 16px
2.4rem = 24px
```

It's not a technical necessity. It's a hack for math.

An honest assessment

`html { font-size: 62.5%; }` does **not** automatically break user preferences.

Why: `62.5%` is a percentage of the user's default. If the user raised their default from `16px` to `20px`, the root becomes `12.5px`, not `10px`. If everything downstream is in `rem`, the system continues to scale:

```
body { font-size: 1.6rem; }
```

At default `16px`: root `10px`, body `16px`. At default `20px`: root `12.5px`, body `20px`.

User preferences still flow through.

Where it breaks

The problem starts when 62.5% gets mixed with hardcoded `px`:

```
html { font-size: 62.5%; }
body { font-size: 16px; }
```

Now body is locked at `16px`. The user's preference is silently ignored for body text. Worse, when half the components use `rem` and half use `px`, the interface stops being consistent: some text scales, some doesn't.

✓ USE WHEN

- ✓ Use only if the team commits to `rem` for all sizes that should scale
- ✓ Document explicitly: `1.6rem ≈ 16px at default 16px`
- ✓ Test the site with the browser default font-size changed
- ✓ Make the rule clear: `1rem` here is **not** the browser default

✕ AVOID WHEN

- ✗ If the team reads `rem` as `px` and mixes `px` alongside it
- ✗ If you depend on an external design system that assumes `1rem = browser default`
- ✗ If component libraries expect the standard rem scale — some may render unexpectedly small

PITFALL

The real cost of the 62.5% trick isn't accessibility per se — it's that you change what `1rem` *means*. Now you have to write `1.6rem` for what most CSS would call `1rem`. Every team member must hold this extra rule in their head. That's a real cognitive tax for a small mathematical convenience.

Verdict

The 62.5% trick is acceptable, but no longer especially useful. Modern CSS has `calc()`, design tokens, editor hints, and well-established `rem` scales. The cost is a team-wide convention everyone has to remember.

A safer neutral default:

```
html { font-size: 100%; }
body { font-size: 1rem; line-height: 1.5; }
```

If your project does use 62.5%, make sure the team understands:

```
This is not a way to make rem "right."
It's a way to make px-to-rem conversion easier.
```

Accessibility and user font preferences

Users really do change the browser's default font-size. It's not a theoretical possibility.

Browsers offer font-size settings, minimum font size, page zoom, text-only zoom (in some environments), system display-scaling, and accessibility-specific overrides. They don't all work the same way, but the principle is shared: **users may need text larger than the designer planned.**

Browser default font-size

The typical default is `16px`, but it's not a law of nature.

```
default root font-size is often 16px
but the user may change it in browser settings
```

If your CSS says:

```
body { font-size: 1rem; }
```

then body follows the root.

If your CSS says:

```
body { font-size: 16px; }
```

then body is locked to `16px` as a CSS length. Page zoom will still scale the page visually, but the user's default-font-size setting will not change that pinned text.

What about zoom?

It's important to distinguish:

```
browser default font-size — changes the base text size
page zoom                 — scales the entire page
text-only zoom            — scales text without changing layout (in some
browsers)
minimum font size         — clamps text below a threshold
```

WCAG 1.4.4 (w3.org) requires that text can be resized up to 200% without losing content or functionality. It's not about *only* using `rem` — it's about the *result*: the user enlarges text, and the interface doesn't clip, overlap, or break.

Why `font-size: 16px` can still be a problem

The claim that fixed `px` font-sizes "break user preferences" needs nuance.

`font-size: 16px` does **not** break browser zoom: if the user zooms to 200%, the page scales visually.

But `font-size: 16px` *does* ignore the user's default-font-size setting. If someone set their default to `20px` because they have trouble reading at smaller sizes, your author-specified `16px` stays `16px`.

```
.a { font-size: 16px; }
.b { font-size: 1rem; }
```

At user default `16px`: both are `16px`. At user default `20px`: `.a` is still `16px`, `.b` is `20px`.

That's why `rem` is the better default for text.

How many users change font-size?

There's no single honest percentage for the whole web; it depends on audience, device, browser, age, vision, and context. Some useful reference points:

- **Evan Minto's Internet Archive analysis** (via Nicolas Hoizey's write-up `(nicolas-hoizey.com)`): ~**3.08%** of users had a changed root font-size.
- **WebAIM Survey of Users with Low Vision #2** `(webaim.org)`, among low-vision respondents:
  - **8%** had increased text size above the default. That figure is script-detected, so it counts style and browser-setting changes only, not zoom or magnification.
  - **36.7%** said they often use browser text-sizing controls.
  - **44.0%** often use browser zoom.
  - **48.4%** use screen magnification or system settings.

> Even if it's "only a few percent," it isn't an edge case.
> For a large product, 2–3% of users is a huge audience.
> For accessibility, it's exactly the audience you must not cut off.

How to test

A minimal manual test:

1. Open the page in a browser.
2. Increase the browser default font-size (e.g. from `16px` to `20px` or `24px`).
3. Reload.
4. Check whether body text, controls, labels, cards, and modals all grew.
5. Look for clipping, overlap, hidden buttons, horizontal scroll.
6. Repeat at page zoom `200%`.

What should be true:

```
Text is bigger.
Containers grew with it.
Buttons didn't clip their labels.
Modals didn't lose their action buttons.
Navigation didn't cover content.
```

You can simulate it in DevTools in one line. Temporarily set:

```
html { font-size: 125%; }
```

or:

```
html { font-size: 150%; }
```

This doesn't replace real browser-setting testing, but it instantly reveals which parts of the UI are written in `rem`/`em` versus pinned to `px`.

PITFALL

Not every accessibility mechanism shows up the same way in CSS. Browser zoom changes the CSS pixel ratio and layout viewport; minimum font size may apply at the *used value* level without changing how font-relative units resolve. Don't try to prove accessibility by formula alone — test with real settings: changed default font-size, 200% zoom, minimum font size, on real components.

IN SHORT

`rem` doesn't *guarantee* accessibility, but `px` for font-size definitely walks past the user's stated preference. Use `rem` for text; test the actual behavior with real browser settings.

Beyond `em` and `rem` — the font-metric family

`em` and `rem` resolve against the font's *size*. CSS has five more font-relative units that resolve against finer properties of the font itself:

```
ch  = advance width of the "0" glyph
ex  = x-height (height of lowercase x)
cap = cap height (height of capital letters)
ic  = advance width of the "水" CJK ideograph
lh  = computed line-height as a length
```

Each also has a **root variant** — `rch`, `rex`, `rcap`, `ric`, `rlh` — measured against the root element's font rather than the local one.

These units are quieter than `em` and `rem`. Most CSS doesn't need them. But when you need to align an icon to cap-height, limit a column to "about 65 characters," or size something to exactly one line of text — nothing else does the job.

```
Demo — Font metrics, rendered with the units that describe them
Runs live at cssunits.com/article/font-relative
```

---

ch

FORMULA	`1ch = advance width of the '0' glyph in the current font`
RELATIVE TO	The element's own font
SUPPORT	● Universal

`ch` is the advance measure, or width, of the `0` (digit zero) glyph in the current font.

The mental model:

> `1ch ≈ the width of one character`

But "approximately" is doing real work here. `ch` doesn't compute an average glyph width. It looks at one specific glyph: `U+0030 DIGIT ZERO`. In a monospaced font, `1ch` is close to the width of every character. In a proportional font, `0`, `i`, `m`, the space character, and Cyrillic letters all have different advance widths.

If the browser can't determine the `0` glyph's advance, the spec says to assume `0.5em` wide by `1em` tall. In practice `1ch` falls back to `0.5em` in normal horizontal text, and to `1em` only when the text is typeset upright — a vertical writing mode (`vertical-rl`/`vertical-lr`) with `text-orientation: upright`.

The canonical use:

```
.article {
  max-width: 65ch;
}
```

This is one of the more elegant patterns in CSS: instead of `max-width: 720px`, we ask for "roughly 65 characters per line." For prose, a range of ~45–75 characters is the classic typographic guideline, with ~66 often cited as a comfortable target.

A more defensive variant:

```
.article {
  inline-size: min(100%, 65ch);
}
```

Or with fluid scaling:

```
.article {
  max-inline-size: clamp(40ch, 70vw, 70ch);
}
```

Demo — Same `max-width: 65ch`, three fonts

Runs live at cssunits.com/article/font-relative

✓ USE WHEN

- ✓ Reading column widths: `max-inline-size: 65ch`
- ✓ Input fields with an expected character count
- ✓ Code fields and monospaced inputs
- ✓ OTP / verification fields: `inline-size: 6ch`
- ✓ Tables with character-based values; terminal-style interfaces

✕ AVOID WHEN

- ✗ General-purpose spacing — `2ch` ties the size to the font's `0` width, rarely what you mean
- ✗ When pixel-exact width must be identical across fonts
- ✗ As a stand-in for character count — different fonts give different widths

PITFALL

`65ch` doesn't mean exactly 65 characters per line. `ch` is based on the width of `0`, not on an average glyph. Proportional fonts have varying letter widths, languages have different average word lengths, and bold or variable-font axes can shift the metric. Treat `ch` as a typographic ruler graduated by zero-width — usually close enough for body text, near-perfect for monospaced UI.

IN SHORT

`ch` isn't a character counter. It's a typographic ruler whose tick is the width of `0`. For prose, it's usually close enough. For monospaced interfaces, it's almost exact.

ex

FORMULA	`1ex = x-height of the current font`
RELATIVE TO	The element's own font
SUPPORT	● Universal

`ex` is the x-height of the current font: the height of the lowercase letter `x`, i.e. the height of lowercase letters without ascenders or descenders.

X-height has a big effect on perceived size. Two fonts at the same `font-size: 16px` can look very different, because one has a taller x-height.

A common approximation:

```
1ex ≈ 0.5em
```

But it's only an approximation. A typeface with a large x-height (modern sans-serifs like Inter) gives a bigger `1ex` than a small-x-height face (some classical serifs). That's the whole point of `ex` — it reflects a real typographic property, not an arbitrary half-em.

✓ USE WHEN

- ✓ Aligning a small inline icon to the lowercase body of the text
- ✓ Decorative markers that should sit at lowercase height
- ✓ Thin elements that should match x-height (underlines, highlights)
- ✓ Typographic experiments where lowercase metrics matter

✕ AVOID WHEN

- ✗ Icons that should match the full line — use `1lh`
- ✗ Icons that should match capital letters — use `1cap`
- ✗ Layout spacing — `ex` varies unpredictably between fonts; `em` / `rem` are steadier

IN SHORT

`ex` answers *"how tall are this font's lowercase letters?"* — not *"how big is the type?"* A fine-grained unit for fine-grained typographic alignment.

---

cap

FORMULA	`1cap = cap height of the current font`
RELATIVE TO	The element's own font
SUPPORT	● Modern browsers `all engines since Dec 2023`

`cap` is cap-height: the height of capital letters in the current font. Where `ex` looks at lowercase, `cap` looks at uppercase.

It's useful because icons placed near a label often need to match the height of capital letters, not the full `1em`. Capitals fill only part of the em square (roughly `0.7em` in most fonts), which also reserves room for descenders and diacritics.

`cap` is typically smaller than `1em`. (When a font doesn't expose a cap-height metric, the spec falls back to the font's ascent.) The classic use is icons in headings:

```
.heading-with-icon {
  display: inline-flex;
  align-items: baseline;
  gap: 0.25em;
}

.heading-with-icon svg {
  inline-size: 1cap;
  block-size: 1cap;
}
```

If the heading font changes, the icon adapts to the new cap-height automatically.

✓ USE WHEN

- ✓ Icons placed next to a heading or uppercase label
- ✓ Icons inside button labels (when buttons use uppercase or title-case)
- ✓ Badges aligned with capitalized text
- ✓ Decorative typographic bars sized to cap-height

✕ AVOID WHEN

- ✗ Body-text alignment — `cap` matches capitals, so it mismatches lowercase-dominant text (`ex` / `lh` fit better)
- ✗ Icons that should just be "roughly text-sized" — `1em` needs no font-metric support
- ✗ Layouts targeting older browsers without an `@supports` fallback

PITFALL

`cap` arrived much later than `em`, `rem`, and `ch`: Firefox shipped it in 2022, Chrome in late 2023, Safari in 17.2 (December 2023). Every current engine supports it, so the `@supports` dance below is now only about *older* installed browsers — worth keeping if your analytics still show pre-2024 Safari:

```
.icon { inline-size: 1em; block-size: 1em; }
@supports (height: 1cap) {
  .icon { inline-size: 1cap; block-size: 1cap; }
}
```

IN SHORT

`cap` is the unit for the moments when `1em` looks *almost right, but a touch too tall*. It pins graphics to real capital height instead of to the abstract em-square.

---

ic

FORMULA	`1ic = advance width of the '水' (water) CJK ideograph`
RELATIVE TO	The element's own font
SUPPORT	● Modern browsers `all engines since Sep 2022`

`ic` is a font-relative unit built for CJK typography. It equals the advance measure of `水`, the CJK water ideograph (`U+6C34`). [3 水 is the test glyph because ideographs are square by design: one 水 is one cell of the grid. A paragraph of CJK text is, typographically, graph paper.]

> `1ic ≈ the width of one full-width CJK character`

For Latin-script interfaces this is exotic. For Chinese, Japanese, and Korean text, it's a much more meaningful unit than `ch` (which is anchored to a Latin digit).

```
.cjk-article {
  max-inline-size: 32ic;
  line-height: 1.7;
}
```

Reads as: *"limit the column to about 32 CJK characters."*

✓ USE WHEN

- ✓ Column widths in CJK layouts
- ✓ Input fields sized for an expected number of CJK characters
- ✓ Vertical writing-mode CJK typography
- ✓ Where `ch` would give a Latin proxy but you need a CJK proxy

✕ AVOID WHEN

- ✗ Latin or Cyrillic interfaces — `1ic` silently falls back to another font
- ✗ Any context without CJK characters

PITFALL

`ic` depends on the rendered font actually containing a `水` glyph. If your text font doesn't, the browser uses a fallback font for the metric. So `1ic` may secretly be measured against a different font than the one rendering your text — which is why `ic` looks random in Latin-only interfaces. (If the `水` advance can't be determined at all, the spec assumes `1em`.)

IN SHORT

`ic` exists not as a niche curiosity but because `ch` is a Latin perspective on text width. CJK text needs its own measuring stick, and `水` is it.

`lh`

FORMULA	`1lh = the element's computed line-height as a length`
RELATIVE TO	The element's own line-height
SUPPORT	● Modern browsers `all engines since Nov 2023`

`lh` resolves to the computed `line-height` of the element, expressed as a length.

```
1lh = the height of one line
```

If `font-size: 16px` and `line-height: 1.5`, then `1lh = 24px`. If `font-size: 20px` and `line-height: 1.4`, then `1lh = 28px`. Critically, `lh` depends on both font-size *and* line-height, not just the type size.

The killer use case is icons inside a single line of text:

```
.label svg {
  inline-size: 1lh;
  block-size: 1lh;
}
```

When line-height changes, the icon follows. Strictly, `1lh` is the computed `line-height` — the size of an ideal empty line — so a line box whose content is taller can still exceed it.

Demo — Icon sized at `1lh × 1lh`

Runs live at cssunits.com/article/font-relative

✓ USE WHEN

- ✓ Inline icons that should fill a line: `width: 1lh; height: 1lh`
- ✓ Line-height-aware controls and indicators
- ✓ Skeleton placeholders sized to one line
- ✓ Vertical rhythm where measurements should be multiples of a line
- ✓ `padding-block: 0.5lh` for line-aware spacing

✕ AVOID WHEN

- ✗ When the element should match the type size only — use `1em`
- ✗ When it should match capital letters — use `1cap`
- ✗ As a replacement for unitless `line-height` — they're different things

`lh` vs unitless `line-height`

It's easy to confuse these two:

```
line-height: 1.5;  /* unitless multiplier — for the line-height property */
height: 1lh;       /* length — for any other property */
```

`line-height: 1.5` inherits as a number; descendants multiply by their own font-size. `1lh` is the resolved length once computed, usable as a length in other properties. They're complements, not alternatives.

The well-formed pattern:

```
.text { line-height: 1.5; }
.text .icon { block-size: 1lh; }
```

Body text uses the multiplier (so nested elements get proportional line-heights). The icon uses the resolved length (so it fits one line of the parent).

> IN SHORT
>
> `lh` makes `line-height` a real unit of measure. Not just *space between lines*, but a building block for anything that should live in the rhythm of the line.

Root variants — `rch`, `rex`, `rcap`, `ric`, `rlh`

FORMULA	`r* = the same metric, measured on the root element's font`
RELATIVE TO	The root element (typically <html>)
SUPPORT	● Modern browsers `all engines since Jan 2026`

Each font-metric unit has a root variant. They follow the same idea as `rem`:

```
rch  = ch of the root font
rex  = ex of the root font
rcap = cap of the root font
ric  = ic of the root font
rlh  = computed line-height of the root element
```

Where `ch`, `ex`, `cap`, `ic`, `lh` are local — they shift if the element changes `font-family` or `font-size` — the root variants stay locked to whatever `<html>` is using.

```
html {
  font-family: Inter, sans-serif;
  font-size: 16px;
  line-height: 1.5;
}

.card {
  font-family: Georgia, serif;
  max-inline-size: 65rch;
}
```

`65rch` is measured against Inter's `0`, not Georgia's, even inside a Georgia component.

✓ USE WHEN

- ✓ A global text measure that ignores any local font changes
- ✓ Vertical rhythm via `rlh`: `margin-block: 2rlh`
- ✓ Layout tokens tied specifically to root font metrics
- ✓ Documentation / editor interfaces where the root defines the typographic grid

✗ AVOID WHEN

- ✗ Internal component proportions that should follow the component's own font
- ✗ Default spacing scale — a plain `rem` multiple is enough; the root metric adds no benefit here
- ✗ Iconography inside a heading — use the local `1cap` so it adapts to that heading's font

In practice, `rem` handles 90% of "I want something tied to the root" cases. Root metric units are reserved for the times when you specifically need *that root metric* — root character width, root cap-height, root line-height — rather than just a multiple of root font-size.

The support story finally closed

This family took the longest of any unit group in this guide to become usable everywhere. `rlh` had all three engines by November 2023, but `rch`, `rex`, `rcap`, and `ric` sat in a two-engine limbo for years: Chromium shipped them in 2023 (Chrome 111, then 118 for `rcap`), Safari in 17.2 (December 2023), and Firefox only in **version 147, January 2026**. [4 Which makes these the newest CSS units in this guide — younger than container queries, and by some margin the last to arrive.]

So the practical guidance flipped recently:

```
before 2026: rch/rex/rcap/ric needed an @supports fallback
now:         all three engines ship them — only older installs need one
```

If you support browsers released before 2026, keep a fallback in `rem`:

```
.measure { max-inline-size: 34rem; }          /* fallback */
@supports (width: 1rch) {
  .measure { max-inline-size: 70rch; }        /* the metric you actually
meant */
}
```

SUPPORT CHECKED AGAINST MDN BROWSER-COMPAT DATA · 3 AUG 2026

PITFALL

Root variants and local variants are easy to confuse in code:

```
65ch   /* depends on this element's font */
65rch  /* depends on the root element's font */
```

When a component changes `font-family`, that one-letter difference becomes visible. Sometimes that's a feature; sometimes it's a bug.

IN SHORT

Root variants answer one question: *"what if I want a font metric, but not from this component — from the system's root?"* Usually the answer is `rem`. Sometimes, it really is `rch`, `rcap`, or `rlh`.

Compact reference

UNIT	FORMULA	BEST FOR	WATCH OUT
`em`	element's computed font-size (inherited, when set on `font-size`)	component-internal proportions	compounds through nesting
`rem`	root element's font-size	type scale, spacing tokens	it's not a fancy `px` alias
`ch`	advance of `0` in the current font	`max-inline-size: 65ch`	font-dependent; ≠ exactly N characters
`ex`	x-height of the current font	lowercase-aware alignment	varies strongly between fonts
`cap`	cap height of the current font	icons next to headings	all engines only since Dec 2023
`ic`	advance of `水` in the current font	CJK layouts	rarely useful in Latin / Cyrillic UI
`lh`	computed line-height, as a length	icons sized to one line	not a substitute for unitless `line-height`
`rch` `rex` `rcap` `ric` `rlh`	same metrics, on the root	system measures that ignore local fonts	`rem` is simpler in 90% of cases

Two rules worth keeping next to this table: for `line-height`, prefer the unitless form (`1.5`) so it inherits as a multiplier, not a frozen length. And the 62.5% trick is acceptable only if every size downstream uses `rem`: mixing `px` alongside doesn't change what `1rem` means, but it does leave half the interface scaling with the reader's settings and half of it ignoring them.

Sources

- CSS Values and Units Module Level 4 — font-relative lengths (w3.org)
- MDN — `font-size` (developer.mozilla.org)
- MDN — `line-height` (developer.mozilla.org)
- MDN — `<length>` values (developer.mozilla.org)
- WCAG 1.4.4 — Resize Text (w3.org)
- WebAIM Survey of Users with Low Vision #2 (webaim.org)
- Nicolas Hoizey — "Users DO change font size" (nicolas-hoizey.com)
- web.dev — Typography fundamentals (line length) (web.dev)

CHAPTER 3 OF 7 · 1 UNIT

Percentage (%)

Fifty percent of what? — the whole subject, in one question.

`%` is not a single unit with a single rule. It's a **context-dependent value**, and the meaning is set by the specific CSS property where it appears.

The same `50%` can mean:

```
50% of the parent's width
50% of the parent's height
50% of the element's own size
50% of (container size - image size)
50% of the inherited font-size
50% of the box dimensions, per axis (for border-radius)
```

That's why `%` is the most dangerous "simple" unit in CSS.

There is no single formula. The general shape is:

```
resolved value = reference value × percentage
```

FORMULA	`resolved value = reference value × percentage`
RELATIVE TO	Defined by the property where % appears
SUPPORT	● Universal

But what counts as the *reference value* depends on the property:

```
width: 50%;                    /* containing block inline-size */
height: 50%;                   /* containing block block-size — if defined
*/
padding-top: 10%;              /* containing block inline-size  (yes —
inline!) */
transform: translateY(50%);    /* the element's own height */
font-size: 120%;               /* inherited font-size */
```

Demo — What `%` resolves to, per property

Runs live at cssunits.com/article/percentage

Here is the reference table worth keeping at hand:

CSS	`%` IS RELATIVE TO
`width: 50%`	the containing block's **width**
`inline-size: 50%`	the containing block's **inline size** (its width only in horizontal writing)
`height: 50%`	the containing block's **height**, if that height is definite
`block-size: 50%`	the containing block's **block size**, if that size is definite
`padding: 10%` (all sides)	inline-size of the containing block
`padding-top: 10%`	inline-size of the containing block — **not** height
`margin: 10%` (all sides)	inline-size of the containing block
`margin-top: 10%`	inline-size of the containing block — **not** height
`transform: translateX(50%)`	the element's own width
`transform: translateY(50%)`	the element's own height
`background-position: 50% 50%`	(container size – image size) on each axis
`border-radius: 50%`	width for horizontal radii, height for vertical
`font-size: 120%`	inherited font-size (parent's computed)
`line-height: 150%`	element's own font-size, but inherits as a *length*
`top: 50%` / `left: 50%`	containing block size on the matching axis

`width: 50%`

The most intuitive case:

```
.child {
  width: 50%;
}
```

If the containing block is `800px` wide, then `width = 400px`.

In logical-property terms it's clearer to think:

> `width (in horizontal writing) = inline-size of the containing block`

For writing-mode neutrality, use:

```
.child {
  inline-size: 50%;
}
```

`height: 50%` and the famous `height: 100%` trap

`height: 50%` is measured against the containing block's height, but only if that height is *defined*.

```
.parent {
  /* no height set */
}

.child {
  height: 100%;
}
```

In ordinary flow, this often doesn't work the way authors expect. If the parent's height depends on its content, then a percentage height on the child has no *definite* reference and resolves to `auto`.

Working example:

```
.parent { height: 400px; }
.child  { height: 50%; }  /* = 200px */
```

Broken example:

```
.parent { height: auto; }
.child  { height: 100%; }  /* often resolves to auto, not "fill" */
```

That's the source of the eternal question:

> *Why doesn't `height: 100%` work?*

Because `100%` of `auto` doesn't give a definite height.

For "fill the visible viewport" layouts, you usually want `min-height: 100dvh`, `100svh` or `100vh` (or flex/grid stretch), not a chain of `height: 100%` without explicit heights.

`padding: 10%` and `margin: 10%`

This is the most counter-intuitive part of `%`:

```
.box {
  padding-top: 10%;
}
```

`padding-top` is **not** measured against the parent's height. It's measured against the parent's **inline-size** (width in horizontal writing). [1 Why width? Vertical padding resolving against a possibly-unknown height would be circular — content height depends on

padding depends on height. Width was the one measure the engine already knew. The oddity is load-bearing.]

In horizontal writing:

> `padding-top: 10% = 10% of the containing block's width`

The same is true for `padding-bottom`, `padding-left`, `padding-right`, and for every side of `margin`.

This is the basis of the classic aspect-ratio hack:

```
.ratio {
  height: 0;
  padding-top: 56.25%;  /* 56.25 = 9 / 16 × 100 */
}
```

The `56.25%` is computed from the *width*, producing a 16:9 aspect ratio.

Today the modern alternative is:

```
.ratio {
  aspect-ratio: 16 / 9;
}
```

But the hack worked specifically because vertical padding is measured against horizontal size.

> PITFALL
>
> `padding-top: 10%` and `margin-top: 10%` both resolve against the parent's **inline-size**, never against its height. If you write `padding-top: 10%` expecting "10% of the height," your layout will quietly diverge from your intent.

`transform: translate(50%)`

Percentages inside `transform: translate()` are measured against the **element's own** reference box — not its parent.

```
.box {
  transform: translateX(50%);
}
```

If `.box` is `200px` wide, `translateX(50%) = 100px`.

For vertical:

```
.box {
  transform: translateY(50%);
}
```

If `.box` is `80px` tall, `translateY(50%) = 40px`.

The classic absolute-centering pattern combines both reference systems at once:

```
.modal {
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%, -50%);
}
```

Two different percentages, two different reference boxes:

```
left: 50%              = 50% of the containing block's width
top: 50%               = 50% of the containing block's height
translateX(-50%)       = -50% of the modal's own width
translateY(-50%)       = -50% of the modal's own height
```

The same number `50%`, two different systems of measurement — percentage logic in a single snippet.

`background-position: 50% 50%`

`background-position` is another case where "percent of the container" is incomplete.

The correct model:

```
offset = (container size - image size) × percentage
```

On each axis:

```
x offset = (container width - image width) × x%
y offset = (container height - image height) × y%
```

So:

```
.hero {
  background-position: 50% 50%;
}
```

means:

```
align the image's 50% point with the container's 50% point
```

— i.e., the *center of the image* meets the *center of the container*.

If the image is the same size as the container, the difference is zero, and percentages produce no visible movement.

`border-radius: 50%`

`border-radius: 50%` doesn't mean "50% of one dimension." It means **50% on each axis**, independently:

```
.pill-or-ellipse {
  width: 200px;
  height: 100px;
  border-radius: 50%;
}
```

Horizontal radii resolve against the width, vertical radii against the height. A rectangle becomes an ellipse. A square becomes a circle:

```
.circle {
  width: 100px;
  height: 100px;
  border-radius: 50%;
}
```

> `border-radius: 50%  →  ellipse from a rectangle, circle from a square`

`font-size: 120%`

Percentage `font-size` behaves like `em`:

```
.child {
  font-size: 120%;
}
```

means:

> `child font-size = inherited font-size × 1.2`

— the same as `font-size: 1.2em`.

That means percentage `font-size` carries the same cascade trap as `em`. Repeat it on nested elements and the size compounds:

```
.nested {
  font-size: 120%;
}
```

> `1.2 × 1.2 × 1.2 ...`

For global typography, `rem` is usually safer.

`line-height: 150%`

Percentage `line-height` is computed relative to the element's own `font-size`, but it inherits as a length:

```
body {
  font-size: 16px;
  line-height: 150%;
}
```

Computed line-height becomes a length: `24px`. Descendants may inherit `24px`, not the multiplier `1.5`, which is the same bug pattern as `line-height: 1.5em`.

For base typography, prefer the unitless form:

```
body {
  line-height: 1.5;
}
```

And use `lh` when you need the *resolved length* in another property:

```
.icon {
  block-size: 1lh;
}
```

When to use `%`

Use `%` when the size should depend on **the property's specific reference value**.

✓ USE WHEN

- ✓ Flexible layout columns: `width: 50%`
- ✓ Responsive media: `inline-size: 100%`
- ✓ Absolute positioning relative to a containing block
- ✓ Transform-based centering: `translate(-50%, -50%)`
- ✓ Circles and ellipses: `border-radius: 50%`
- ✓ Background placement: `background-position: 50% 50%`
- ✓ Legacy aspect ratios via `padding-top`
- ✓ Font scaling relative to a parent component

✕ AVOID WHEN

- ✗ When the reference isn't obvious to the reader — `padding-top: 10%` can surprise
- ✗ `height: 100%` without a defined chain of parent heights
- ✗ `%` on `font-size` across deeply nested components — prefer `rem`
- ✗ Whenever you'd have to mentally simulate "% of what?"

PITFALL

Percentages compute and inherit differently in different properties. You can’t memorize one rule like “% means percent of the parent”: it’s too coarse and often wrong. The accurate version: `%` **resolves against the reference value defined by the specific property.**

> IN SHORT
>
> `%` isn't a unit. It's a question — *"percent of what?"* — and the right answer is always in the spec for the property where you wrote it.

Compact reference

PROPERTY	`%` RESOLVES AGAINST
`width`	containing block width (its inline size only in horizontal writing)
`inline-size`	containing block inline size
`height` / `block-size`	the matching containing-block dimension, if definite
`padding-*` / `margin-*`	containing block inline-size — always
`transform: translate(X, Y)`	the element's own width / height
`background-position: X% Y%`	(container – image), per axis
`border-radius`	border-box width / height, per axis
`font-size`	inherited font-size
`line-height`	own font-size — but inherits as a length
`top` / `left` / `right` / `bottom`	containing block, matching axis

Sources

- CSS Values and Units Module Level 4 — `<percentage>` (w3.org)

- MDN — `<percentage>` (developer.mozilla.org)
- MDN — `padding` (developer.mozilla.org)
- MDN — `margin` (developer.mozilla.org)
- MDN — `height` (developer.mozilla.org)
- MDN — `transform: translate()` (developer.mozilla.org)
- MDN — `background-position` (developer.mozilla.org)
- MDN — `border-radius` (developer.mozilla.org)

CHAPTER 4 OF 7 · 24 UNITS

Viewport Units

The screen is not a rectangle. It's a negotiation.

Viewport units measure the page against the viewport — the visible area where the document is rendered. They are easy to start using, but they hide one of the most important layout problems in modern CSS: **the viewport is not always one stable rectangle.**

On desktop, the mental model is simple: the browser window has a width and a height, and `1vw` or `1vh` is one percent of those dimensions. On mobile, the browser itself takes up space — address bars, bottom toolbars, tab controls, pull-to-refresh areas, safe areas, and sometimes a virtual keyboard. Some of that browser UI appears and disappears while the user scrolls. That is why viewport units now come in several families: **default, small, large, and dynamic**.

The short version:

```
vw / vh      old default viewport units
vmin / vmax  smaller / larger side of the viewport
vi / vb      logical viewport units, aware of writing direction
sv*          small viewport: sized as if chrome were expanded
lv*          large viewport: sized as if chrome were retracted
dv*          dynamic viewport: current visible state
```

The big lesson:

> ***`100vh` does not always mean "the visible screen right now".*** *On modern mobile browsers it often behaves like `100lvh` — the height of the large viewport, as if dynamic browser chrome were hidden. That one detail explains years of broken mobile full-screen layouts.*

The simulator below shows the difference. Toggle the browser toolbars and watch each unit respond — both on the screen and in the numbers beside it.

```
Demo — Viewport units on a simulated mobile screen
Runs live at cssunits.com/article/viewport
```

`vw` / `vh`

FORMULA	`1vw = 1% of default viewport width; 1vh = 1% of default viewport height`
RELATIVE TO	Default viewport size
SUPPORT	● Universal

`vw` and `vh` are the classic viewport units. They let an element size itself as a percentage of the viewport width or height.

```
1vw  = 1% of the viewport width
1vh  = 1% of the viewport height
```

If the viewport is `1200px` wide and `800px` tall, then `1vw = 12px`, `1vh = 8px`, `100vw = 1200px`, and `100vh = 800px`.

These felt magical when they became widely supported. You could create full-screen panels without JavaScript:

```css
.hero {
  min-height: 100vh;
}
```

Or make typography respond to the window size:

```
.headline {
  font-size: 8vw;
}
```

But hold that thought — **pure viewport units on `font-size` are an accessibility hazard**, covered just below. The safe form caps them with `clamp()` and a `rem` term.

The *default* family is defined against the **large** viewport. That is worth stating precisely, because it changed: earlier editions of CSS Values 4 let each browser pick between the small size, the large size or something in between, and the current text pins `v*` to the large viewport size because that is what engines had already shipped and what web compatibility now requires. So **`vw`/`vh` are `lvw`/`lvh`**, not "usually" but by definition. (On desktop there's no retractable chrome, so small, large and default all coincide.) The distinction only shows up on mobile.

`vw` and `vh` are relative to the **initial containing block**, which is based on the viewport in continuous media such as screens. They are not relative to a parent, not relative to font size, and not relative to the element itself. That makes them useful for page-level composition.

```
Demo — A box sized at 30vw × 20vh
Runs live at cssunits.com/article/viewport
```

✓ USE WHEN

- ✓ Full-bleed visual sections: `width: 100vw` on a deliberate hero band
- ✓ Editorial headlines — via `clamp(2rem, 5vw + 1rem, 4rem)`, where the `rem` term keeps it zoom-accessible
- ✓ Hero sections, slide-like layouts, app shells where main region fills vertical space
- ✓ Map or editor canvases that should fill the window

✗ AVOID WHEN

- ✗ **Bare** `font-size: 8vw` for text — viewport units don't enlarge on zoom (accessibility risk)
- ✗ Replacing `width: 100%` blindly — `100vw` can include the scrollbar gutter
- ✗ Critical mobile full-screen UI — bottom buttons can hide behind chrome
- ✗ Anything where overflow or clipping would be unacceptable

Pitfall: viewport units and the zoom accessibility trap

There's a catch with viewport-based typography that bites real users. Pure viewport-relative text is tied to the viewport geometry, not to the user's chosen text scale. Depending on the browser's zoom and reflow behavior, a headline sized only in `vw` can fail to become meaningfully larger for a low-vision user. That can fail WCAG 1.4.4 Resize Text (w3.org), which requires text to be resizable up to 200% without loss of content or functionality.

```
/* Risky: may fail to reach 200% effective text enlargement */
.headline { font-size: 8vw; }

/* Safer: rem terms still respond to zoom, vw adds fluidity */
.headline { font-size: clamp(2rem, 5vw + 1rem, 4rem); }
```

Two further nuances from accessibility research:

- **Capping with `clamp()` / `max()` can *itself* fail 1.4.4** if the cap stops text from reaching 200%. Maxwell Barvian's analysis for Smashing Magazine `(smashingmagazine.com)` offers a rule of thumb: keep the maximum at or under 2.5× the minimum. Treat it as a smell test rather than a proof — how much zoom actually reaches the text also depends on how the preferred value splits between its `vw` and `rem` terms, so verify at 200% instead of trusting the ratio.
- **`rem` is preferable to `px` in the fluid formula, but `rem` alone isn't a guarantee** — always test at 200% zoom and with an enlarged browser default font-size.

> PITFALL
>
> Pure viewport units on `font-size` are a common resizability failure. Blend a `rem` term inside `clamp()` (e.g. `clamp(2rem, 5vw + 1rem, 4rem)`) and verify the text still reaches 200% — both the floor *and* the cap can break resizability.

Pitfall: the `100vh` mobile bug

The famous `100vh` problem is not that browsers cannot calculate height. The problem is that mobile viewport height has more than one reasonable answer. `[1` Mobile Safari has collapsed its address bar on scroll since 2014 — the same year it shipped, then withdrew, a `minimal-ui` viewport flag aimed at this exact problem. `dvh`, the real fix, arrived in 2022. Eight years is the going rate for a viewport bug to become a unit.]

Imagine an iPhone browser in two states:

```
State A: address bar visible
Visible content height: 700px

State B: address bar hidden after scroll
Visible content height: 780px
```

What should `100vh` be?

If `100vh = 700px`, the section fits while the address bar is visible, but leaves a gap when it hides. If `100vh = 780px`, the section fills the larger screen after chrome retracts, but part of it is hidden behind the address bar when chrome is visible.

Browsers historically chose stability. Instead of changing `vh` constantly while the user scrolls, many mobile browsers made `100vh` behave like the *larger* height. That avoids jumpy layouts but creates the classic bug:

> ***`100vh` is taller than the visible screen when mobile browser chrome is visible.***

That is why a `height: 100vh` modal can have its bottom buttons hidden behind Safari's bottom toolbar or Google Chrome's address controls.

Pitfall: `100vw` and scrollbars

`100vw` means the full viewport width. On desktop systems with classic scrollbars, the viewport width can include the scrollbar area. The document content area is then slightly narrower than `100vw`, which causes a horizontal scrollbar to appear.

PITFALL

`100vw` does **not** subtract scrollbar width. On systems with persistent vertical scrollbars, `width: 100vw` overflows the content area and triggers a horizontal scrollbar. For ordinary full-width sections, prefer `width: 100%`.

> IN SHORT
>
> `vw` and `vh` look like the most obvious units in CSS — one percent of the screen. But the word *screen* is doing too much work. On desktop, it usually means one stable rectangle. On mobile, the rectangle changes depending on browser chrome, scrolling, and UI state.

`vmin` / `vmax`

FORMULA	`1vmin = 1% of the smaller default viewport dimension; 1vmax = 1% of the larger`
RELATIVE TO	Default viewport width/height comparison
SUPPORT	● Universal

`vmin` and `vmax` compare the viewport's two dimensions:

```
1vmin = 1% of the smaller viewport dimension
1vmax = 1% of the larger viewport dimension
```

On a `1200×800` landscape viewport, `1vmin = 8px` and `1vmax = 12px`. On a `390×844` portrait viewport, `1vmin = 3.9px` and `1vmax = 8.44px`. The key idea: **`vmin` and `vmax` are orientation-aware without needing media queries.** [2 `vmin` is the unit of board games, clock faces, and camera viewfinders — anything that must stay square no matter the shape of the room it's shown in.]

```
1vmin = min(1vw, 1vh)
1vmax = max(1vw, 1vh)
```

They do not care about the parent element. They do not care about writing mode. They only ask: *which side of the viewport is smaller, and which side is larger?*

The most common real use for `vmin` is a square or circular object that should fit inside the viewport in both portrait and landscape:

```
.orb {
  width: 80vmin;
  height: 80vmin;
  border-radius: 50%;
}
```

Or a viewport-sized chessboard:

```
.board {
  width: 90vmin;
  height: 90vmin;
}
```

This works because `vmin` always follows the limiting side. On a phone, width is often the limit. On a laptop, height may be the limit.

`vmax` is useful when something should scale with the *larger* dimension — typically oversized background shapes, full-bleed masks, or large visual treatments that should feel huge in any orientation.

```
Demo — vmin vs vmax across orientations
Runs live at cssunits.com/article/viewport
```

✓ USE WHEN

- ✓ Square or circular hero artwork: `width/height: 80vmin`
- ✓ Game boards, diagrams, centered visual demos that must stay fully visible
- ✓ `vmax` for oversized background shapes and full-bleed masks
- ✓ Anywhere you want orientation-aware sizing without media queries

✕ AVOID WHEN

- ✗ Plain text headings via bare `8vmin` — it can land too small on phones and skips zoom-safety (wrap it in `clamp()` with a `rem` term)
- ✗ `100vmax` for layout dimensions — it exceeds one axis by definition, so it overflows
- ✗ Anywhere overflow or clipping would be unacceptable

`vmin` and `vmax` inherit the same mobile viewport ambiguity as `vh` and `vw`. Plain `vmin`/`vmax` belong to the default viewport family — equivalent to `lvmin`/`lvmax`. That means mobile browser chrome can still affect whether the value matches what the user currently sees. For most `vmin` cases it is less dramatic, because phone width is usually the smaller dimension and chrome usually changes height. But it can matter in landscape, on foldables, on tablets, or in unusual browser UI.

> IN SHORT
>
> `vmin` is the *fit inside* unit. `vmax` is the *cover everything* unit. If `vw` and `vh` ask for a specific axis, `vmin` and `vmax` ask for a strategy.

`vi` / `vb`

FORMULA	`1vi = 1% of viewport inline size; 1vb = 1% of viewport block size`
RELATIVE TO	Default viewport size on the logical inline/block axes
SUPPORT	● Modern browsers `all engines since Nov 2022`

`vi` and `vb` are *logical* viewport units. They are like `vw` and `vh`, but instead of measuring physical width and height, they measure the viewport along the document's **logical text axes**.

```
vi = viewport inline axis
vb = viewport block axis
```

In a typical English or Russian page (`writing-mode: horizontal-tb`), the inline axis is horizontal and the block axis is vertical, so `1vi = 1vw` and `1vb = 1vh`. But in vertical writing modes this flips:

```
html {
  writing-mode: vertical-rl;
}
```

In vertical writing, the inline axis runs vertically and the block axis runs horizontally — `1vi` behaves like `1vh`, and `1vb` like `1vw`. One precision that matters: the mapping is decided by the **root element's** writing mode. An element's own `writing-mode` doesn't change what `1vi` resolves to — that per-element sensitivity belongs to container units like `cqi`.

`vi` and `vb` are part of the same mental model as logical CSS properties: `margin-inline`, `padding-block`, `inset-inline`. The point is not "left/right/top/bottom"; the point is *inline direction* and *block direction*.

Use them when your layout should work across writing modes — CJK typography (especially vertical Japanese or Chinese), multilingual templates, e-book or reading interfaces, publishing systems:

```
.reader-page {
  max-inline-size: 80vi;
  min-block-size: 100vb;
  padding-inline: 5vi;
}
```

In horizontal writing this behaves like a viewport-width-aware reading page. In vertical writing, the same CSS follows the text flow.

```
Demo — Writing mode and logical viewport units
Runs live at cssunits.com/article/viewport
```

✓ USE WHEN

- ✓ CJK typography, especially vertical Japanese or Chinese layouts
- ✓ Multilingual article templates that must support horizontal and vertical writing
- ✓ Codebases that already use logical properties everywhere
- ✓ Components that need to support RTL/LTR without rewriting CSS

✗ AVOID WHEN

- ✗ Single-language horizontal sites where writing mode never changes
- ✗ When the team isn't familiar with logical CSS — adds cognitive overhead
- ✗ As decorative complexity — only use when logical axes are a real product requirement

Pitfall: `vi` ≠ `vw`

`vi` and `vb` are not just new names for `vw` and `vh`. In horizontal writing, `vi = vw` and `vb = vh`, but that equivalence is a coincidence of the writing mode. Once the root's writing mode changes, the mapping changes.

Another nuance: `direction: rtl` changes the inline *direction*, but not which physical *dimension* is inline. For any horizontal text — LTR or RTL — inline runs horizontally, so `vi` still maps to width. The dramatic difference comes only from vertical writing modes.

> IN SHORT
>
> `vw` and `vh` are *physical*: width and height. `vi` and `vb` are *typographic*: inline flow and block flow. They exist because not every page thinks "left to right, top to bottom". CSS is a writing system as much as it is a layout system.

The Disappearing Toolbar Problem

Retractable browser toolbars are the reason the new viewport unit families exist.

Chrome is the old interface word for the frame around the content: the address bar, the bottom toolbar, the tab strip, the navigation controls. Every browser has chrome, and this chapter is about all of them — the word is decades older than the browser that took it for a name. On a phone that frame is **dynamic**: it is there when the page loads, then shrinks or slides away as the reader scrolls. The visible content area changes while the layout underneath still needs to hold still. That creates a conflict:

> ***Should viewport units follow the current visible area? Or should they stay stable while the user scrolls?*** *Both answers are useful. Neither answer is always correct.*

Why `100vh` can be bigger than the iPhone screen

Imagine a simplified phone:

```
Physical screen height:                  844px
Visible web content with browser chrome: 720px
Visible web content after chrome hides:  800px
```

(The large viewport is 800, not the full 844 — non-retractable UI like the status bar and home indicator still reserves space.)

If CSS says:

```css
.modal {
  height: 100vh;
}
```

the browser has to choose what `100vh` means. Historically, many mobile browsers made `vh` match the **larger** viewport: the size available after dynamic browser UI retracts. That makes the layout stable and prevents the page from resizing every time the address bar animates.

So the browser effectively says `100vh = 800px` — but the user currently sees only `720px`. Result: 80px of the element are behind browser chrome or below the visible area.

Why browsers did this

At first glance, it sounds wrong. If the visible area is 720px, why not make `100vh = 720px`?

Because then `vh` would change while scrolling. Consider:

```
.hero {
  height: 100vh;
}
```

If browser chrome hides during scroll, `100vh` would animate from `720px` to `800px`. That means the layout changes while the user is in the middle of reading or scrolling. Content jumps. Scroll positions become weird. Animations and sticky elements feel unstable.

So browsers picked a stable value. That solved one class of problems and created another.

The old workaround era

Before `svh`, `lvh`, and `dvh`, developers used JavaScript:

```
document.documentElement.style.setProperty(
  '--vh',
  `${window.innerHeight * 0.01}px`
);
```

Then:

```
.screen {
  height: calc(var(--vh) * 100);
}
```

This measured the current inner height and stored it in a CSS custom property. It worked, but it had costs: JavaScript had to run before layout was correct; resize and orientation changes needed handling; mobile browser behavior still varied; scroll-driven chrome changes could cause jank; every project reinvented the same workaround.

The new viewport units are the CSS-native answer to this problem.

The conceptual fix

CSS now gives names to the different possible viewport heights:

```
small viewport:   sized as if browser chrome were expanded
large viewport:   sized as if browser chrome were retracted
dynamic viewport: current visible state
```

Authors can finally choose the behavior they actually want:

```
.safe-screen     { min-height: 100svh; }
.immersive-hero  { min-height: 100lvh; }
.modal           { height: 100dvh; }
```

PITFALL

Retracting toolbars are not only a Safari problem, and not only about iOS. Different mobile browsers and operating systems have made different choices over time. Also: the **virtual keyboard is a separate problem**. The on-screen keyboard isn't always treated as browser chrome for viewport unit calculations. A layout that behaves well with `100dvh` may still need special handling for input focus and keyboard overlap.

IN SHORT

Mobile browsers are trying to do two good things at once: give the website more space when the user scrolls, and keep the layout from jumping while that happens. Old `vh` forced browsers to pick one compromise. New viewport units let authors choose the compromise per component.

`svh` / `lvh` / `dvh` — and the full S/L/D family

The new viewport units split viewport sizing into three explicit families:

```
sv* = small viewport
lv* = large viewport
dv* = dynamic viewport
```

Each family has six units. Together with the default family, this gives the complete viewport unit map:

AXIS / STRATEGY	DEFAULT	SMALL	LARGE	DYNAMIC
Width	`vw`	`svw`	`lvw`	`dvw`
Height	`vh`	`svh`	`lvh`	`dvh`
Inline axis	`vi`	`svi`	`lvi`	`dvi`
Block axis	`vb`	`svb`	`lvb`	`dvb`
Smaller of width / height	`vmin`	`svmin`	`lvmin`	`dvmin`
Larger of width / height	`vmax`	`svmax`	`lvmax`	`dvmax`

The default units are defined against the **large** family, so each pair is the same unit under two names:

```
vw = lvw   vh = lvh   vi = lvi   vb = lvb   vmin = lvmin   vmax = lvmax
```

That is the reason `100vh` behaves like the large viewport on mobile — and since the spec now says so normatively, it is not a browser quirk you can hope to see fixed.

`svh` (small viewport)

FORMULA	`1svh = small viewport height ÷ 100`
RELATIVE TO	Viewport with dynamic browser UI assumed to be expanded
SUPPORT	● Modern browsers `all engines since Nov 2022`

The small viewport is the viewport size when dynamic browser UI is **expanded**: address bar visible, toolbar visible, maximum browser chrome taking space.

> `100svh = the height that safely fits when browser chrome is visible`

The full small family is `svw`, `svh`, `svi`, `svb`, `svmin`, `svmax` — each measured against the conservative viewport size.

```
.onboarding {
  min-height: 100svh;
}
```

This says: fit within the conservative mobile viewport.

✓ USE WHEN

- ✓ First-screen content with important controls at the bottom
- ✓ Onboarding screens, payment / checkout steps
- ✓ Compact app screens where safety matters more than immersion
- ✓ Fallback sizing when leaving extra space is better than hiding content

✕ AVOID WHEN

- ✗ Immersive hero sections — `svh` can leave a visible gap after chrome retracts
- ✗ Layouts that should always cover the full available visual area

`svh` is **stable, not dynamic**. It does not grow just because more space becomes available. That is its strength and its weakness:

- Pro: content remains safely visible.
- Con: it may not use all available space.

RULE OF THUMB

Use `svh` when clipping would be worse than empty space.

`lvh` (large viewport)

FORMULA	`1lvh = large viewport height ÷ 100`
RELATIVE TO	Viewport with dynamic browser UI assumed to be retracted
SUPPORT	● Modern browsers `all engines since Nov 2022`

The large viewport is the viewport size when dynamic browser UI is **retracted**: address bar hidden, toolbar minimized, maximum content area.

> `100lvh = the height of the screen when browser chrome is out of the way`

The full large family is `lvw`, `lvh`, `lvi`, `lvb`, `lvmin`, `lvmax`.

```
.landing-hero {
  min-height: 100lvh;
}
```

This says: size the hero to the largest mobile viewport state.

✓ USE WHEN

- ✓ Immersive hero sections, full-bleed background media
- ✓ Large editorial visuals where being partially behind chrome at load is acceptable
- ✓ Decorative content that can safely be cropped
- ✓ Layouts where the user is expected to scroll past the first screen

✕ AVOID WHEN

- ✗ Critical controls — bottom buttons can hide behind chrome
- ✗ Modals, drawers, or any flow where the bottom must always be reachable

`lvh` is essentially the named version of the old `vh` behavior. That is useful because it is now **explicit**. But it also means it carries the classic `100vh` mobile pitfall:

PITFALL

`100lvh` can be taller than the currently visible viewport. If you put critical UI inside a `100lvh` container, those controls may sit behind browser chrome until the user scrolls.

RULE OF THUMB

Use `lvh` when cropping is acceptable and visual fullness matters.

`dvh` (dynamic viewport)

FORMULA	`1dvh = dynamic viewport height ÷ 100`
RELATIVE TO	Currently available viewport, taking dynamic browser UI into account
SUPPORT	● Modern browsers `all engines since Nov 2022`

The dynamic viewport tracks the **current viewport state** as browser UI expands or retracts.

> `100dvh = the currently available visible viewport height`

When chrome is fully expanded, `100dvh` equals `100svh`; when fully retracted, it equals `100lvh`; mid-transition it lies between the two.

The full dynamic family is `dvw`, `dvh`, `dvi`, `dvb`, `dvmin`, `dvmax`.

```
.dialog {
  height: 100dvh;
}
```

This is usually the best first choice for a true mobile full-screen modal. For app shells:

```
.app {
  min-height: 100dvh;
  display: grid;
  grid-template-rows: auto 1fr auto;
}
```

✓ USE WHEN

- ✓ Mobile modals, full-screen dialogs, drawers, bottom sheets
- ✓ App shells: chat, maps, dashboards, editors
- ✓ Camera / scanner UIs, layouts where bottom controls must remain visible
- ✓ Anywhere *fits the visible screen right now* beats layout stability

✕ AVOID WHEN

- ✗ As a default replacement for every `vh`
- ✗ Long article pages — chrome animations cause layout shifts mid-scroll
- ✗ Performance-sensitive layouts — `dvh` may cause layout work as chrome animates

`dvh` is tempting because it sounds like the correct modern answer, but dynamic sizing can make layouts move while the user scrolls — scroll position shifts, animation jank, expensive reflow, content that feels unstable.

PITFALL

Dynamic does not mean perfectly animated at 60 fps. Browsers may throttle or debounce `dvh` updates while browser chrome changes. `dvh` is a layout unit, not a precision animation API. The on-screen keyboard may also not behave exactly like dynamic browser chrome for viewport unit calculations in every browser.

RULE OF THUMB

Use `dvh` when *fits the visible screen right now* is more important than layout stability.

Which S/L/D unit should you never use?

There is no unit that should literally never be used. But there is one dangerous habit:

> ***Do not replace every `vh` with `dvh` automatically.*** *Use it for surfaces that need the current visible height. Do not use it blindly for long scrolling pages.*

The cleaner mental model:

```
svh → safety
lvh → visual fullness
dvh → current fit
```

Safe-area insets — a separate problem

The small / large / dynamic families solve one thing: *retractable browser chrome*. They do **not** account for the *non-rectangular* parts of a display — the notch, rounded corners, and the home indicator. `100dvh` fills the visible viewport, but its bottom edge can still sit under the home indicator, so controls pinned to the bottom of a `100dvh` shell may be partly covered.

That's what the `env()` safe-area insets are for:

```
.app         { min-block-size: 100dvh; }
.app__footer {
  /* keep the action bar clear of the home indicator */
  padding-block-end: calc(0.75rem + env(safe-area-inset-bottom));
}
```

`env(safe-area-inset-top | right | bottom | left)` resolve to `0` by default — they become positive only once the page opts into the full display area with the viewport meta tag:

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

Together the four insets describe a rectangle of always-safe space. Pair them with viewport units rather than replacing them: `dvh` decides *how tall* the surface is, `env()` decides *how much edge to keep clear*.

Decision tree

Full-screen mobile modal

```
.modal {
  height: 100dvh;
}
```

A modal is an active surface — buttons, form fields, and close controls must remain visible in the current viewport. If stability matters more than using all space, use `min-height: 100svh` instead. **Avoid** `height: 100vh` here, because `100vh` can behave like `100lvh` and hide bottom content behind browser chrome.

Landing-page hero

```
.hero {
  min-height: 100svh;
}
```

Use `100svh` when the first viewport contains important text and buttons that must be visible immediately. Use `100lvh` if the hero is immersive and slight initial cropping is acceptable. Use `100dvh` only if the hero genuinely needs to track browser chrome changes.

A practical pattern combines both:

```
.hero {
  min-height: 100svh;
}

@supports (height: 100dvh) {
  .hero.is-interactive {
    min-height: 100dvh;
  }
}
```

Full-bleed background image / video

```
.visual {
  min-height: 100lvh;
}
```

Background media can usually tolerate being partially covered or cropped. The large viewport gives the intended immersive size. If important content sits at the bottom, combine with safe padding or move content into a safer layout region.

App shell

```
.app {
  min-height: 100dvh;
}
```

App shells need to fit the current visible viewport. This is especially true for chat, maps, dashboards, and editors.

Bottom sheet

```
.sheet      { max-height: 100dvh; }
.sheet__body { overflow: auto; }
```

A bottom sheet must stay reachable when browser chrome is visible. Add internal scrolling so content can grow without the sheet escaping the viewport.

Square visual that should fit on screen

```
.demo {
  width: 90vmin;
  height: 90vmin;
}
```

Use `vmin` for simple stable demos. Use `dvmin` only if changing browser chrome should affect the size.

Huge visual that must cover the viewport

```
.cover {
  width: 120vmax;
  height: 120vmax;
}
```

Or, for explicit large-viewport behavior, use `120lvmax`.

Multilingual reading layout

```
.reader {
  max-inline-size: 80vi;
  min-block-size: 100vb;
}
```

Logical viewport units follow writing mode. This matters for vertical CJK layouts and systems that already use logical CSS properties.

Ordinary full-width section

```
.section {
  width: 100%;
}
```

Prefer `100%` here — it matches the container's content width. Reserve `100vw` for deliberate edge-to-edge bleed, and note that on systems with classic scrollbars `100vw` includes the scrollbar gutter and can trigger horizontal overflow.

Compact reference

AXIS	DEFAULT	SMALL	LARGE	DYNAMIC	1 UNIT =
Width	`vw`	`svw`	`lvw`	`dvw`	1% of that viewport's width
Height	`vh`	`svh`	`lvh`	`dvh`	1% of that viewport's height
Inline axis	`vi`	`svi`	`lvi`	`dvi`	1% along the inline axis
Block axis	`vb`	`svb`	`lvb`	`dvb`	1% along the block axis
Smaller side	`vmin`	`svmin`	`lvmin`	`dvmin`	1% of the smaller dimension
Larger side	`vmax`	`svmax`	`lvmax`	`dvmax`	1% of the larger dimension

Small = sized as if chrome were expanded · Large = as if retracted · Dynamic = current state · Default is defined as Large.

VIEWPORT-FAMILY BEHAVIOR CHECKED AGAINST MDN AND CSS VALUES 4 · 3 AUG 2026

Sources

- CSS Values and Units Module Level 4 (w3.org) — viewport-percentage lengths
- MDN: viewport-percentage units (developer.mozilla.org)
- web.dev: The large, small, and dynamic viewport units (web.dev)
- MDN: `env()` — safe-area insets (developer.mozilla.org)
- Smashing Magazine — Addressing accessibility concerns with fluid type (smashingmagazine.com) — where the ~2.5× cap guidance comes from

CHAPTER 5 OF 7 · 6 UNITS

Container Query Units

A component never sees the world — only the room it's given.

Viewport units ask: *how big is the window?* Container query units ask a better question for component design: *how big is the box I'm actually in?*

This is the conceptual leap. A card in a 300px sidebar and the same card in a 700px main column live in the same viewport, so viewport units can't tell them apart. Container query units can. They measure against the nearest **query container** ancestor, which means a component can finally be responsive to its own context rather than the whole page. [1 Authors had been asking for "element queries" since the early 2010s, and container queries topped the most-wanted-feature list in the State of CSS survey year after year until they shipped. The blocker was circularity — size depends on content depends on size — and `container-type` is the truce that finally broke it.]

```
cqw   = 1% of the query container's width
cqh   = 1% of the query container's height
cqi   = 1% of the query container's inline size
cqb   = 1% of the query container's block size
cqmin = the smaller of cqi and cqb
cqmax = the larger of cqi and cqb
```

The demo below is a single component. Drag its container wider and narrower. It reflows and rescales entirely on its own.

```
Demo — Resize the container: drag its right edge, or use the slider
Runs live at cssunits.com/article/container
```

Setting up a query container

Container query units don't work in a vacuum. They need an ancestor that has been declared a **query container**. Otherwise there's nothing to measure against.

You opt in with `container-type`:

```
.sidebar {
  container-type: inline-size;
}
```

`container-type` has three values:

- `inline-size` — the element is a query container on its **inline axis** only (width, in horizontal writing). This is by far the most common choice, because querying block size requires the container to have a definite height, which most flow content doesn't.
- `size` — query container on **both** axes. It contains the block axis, so it needs a definite height to be useful, which is why it's usually only worth reaching for when you actually control the element's height.
- `normal` — not a size query container, but still a container for *style* queries and a reference for `container-name`.

You can also name containers, so a descendant can target a specific ancestor:

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

/* shorthand */
.sidebar {
  container: sidebar / inline-size;
}
```

One refinement worth carrying: eligibility is decided **per axis**. An ancestor declared `container-type: inline-size` can answer `cqi` and `cqw` but not `cqb`, so in deep nesting `cqi` and `cqb` may end up measuring two different ancestors.

> MENTAL MODEL
>
> `container-type` is the "I am measurable" switch. Until some ancestor flips it on, `cqi` and friends have no container to read, and they fall back to the small viewport (more on that below).

`container-type` isn't entirely free, though. It establishes an **independent formatting context** (so floats, margins, and clearance stay contained), and with `size` it additionally contains the *block* axis, meaning the element's height stops being derived from its content. One historical gotcha has since been fixed: early implementations also made the container a *containing block* for `position: absolute`/`fixed` descendants — because `container-type` applied layout containment — which silently trapped positioned children. A 2024 CSSWG resolution removed that requirement, so in current browsers `container-type` no longer anchors positioned descendants. If you need the container to *be* the positioning context, set `position: relative` (or `contain: layout`) on it explicitly.

> PITFALL
>
> `container-type: size` contains the block axis, so the element's height no longer grows from its content. Without an explicit height it typically **collapses toward 0**, taking `cqh`/`cqb` (and the visible box) with it. That collapse, not merely "queries don't resolve," is the real reason `size` is a footgun. For the common "respond to available width" case, use `container-type: inline-size`, the axis-limited form intended for exactly that.

`cqi` / `cqw` — inline and width

FORMULA	`1cqi = 1% of the query container's inline size`
RELATIVE TO	Nearest query container ancestor (inline axis)
SUPPORT	● Modern browsers `all engines since Feb 2023`

`cqi` (container query inline) is the workhorse. In horizontal writing it's the container's width; in vertical writing it follows the inline axis, exactly like the logical-vs-physical relationship between `vi` and `vw`.

`cqw` is the physical-width sibling. Most of the time you want `cqi`: it's writing-mode aware, and inline-size is what `container-type: inline-size` establishes.

```
.card-title {
  font-size: clamp(1rem, 4cqi, 1.5rem);
}
```

This title is 4% of its container's inline size, clamped to a sane range. Put the card anywhere — the title sizes itself to fit.

✓ USE WHEN

- ✓ Component typography that should scale with its container: `clamp(1rem, 4cqi, 1.5rem)`
- ✓ Internal spacing/padding proportional to component width
- ✓ Reusable cards, widgets, and embeds that live in many slot sizes
- ✓ Prefer `cqi` over `cqw` for writing-mode safety

✕ AVOID WHEN

- ✗ Without an ancestor `container-type` — the unit silently falls back to `sv*`
- ✗ Page-level sizing where the viewport really is the reference — `vw` / `vi` say that directly
- ✗ Inside an `@container` condition on the same element — a query can't test the container it lives on

Demo — Typography that scales with the container, not the viewport

Runs live at cssunits.com/article/container

`cqb` / `cqh` — block and height

FORMULA	`1cqb = 1% of the query container's block size`
RELATIVE TO	Nearest query container ancestor (block axis)
SUPPORT	● Modern browsers `all engines since Feb 2023`

`cqb` (container query block) measures the container's block size; `cqh` is the physical-height sibling. These are far less common than `cqi` / `cqw`, for one reason: they require the query container to have a **definite block size**, which means

`container-type: size` and a known height.

Most layouts don't have a definite height (content determines it), so `cqb` / `cqh` only make sense in fixed-height regions: a full-height panel, a card with a locked aspect ratio, a slide.

✓ USE WHEN

- ✓ Fixed-height regions: full-height panels, slides, locked-ratio cards
- ✓ Sizing something to a fraction of a known container height
- ✓ When you've deliberately set `container-type: size`

✗ AVOID WHEN

- ✗ Auto-height containers — with no definite block size these don't resolve meaningfully
- ✗ Typical component work — `cqi` is usually the more practical starting point
- ✗ Anywhere the container's height isn't something you control

---

`cqmin` / `cqmax`

FORMULA `1cqmin = 1% of the smaller container dimension; 1cqmax = 1% of the larger`

RELATIVE TO Nearest query container ancestor (both axes)

SUPPORT ● Modern browsers `all engines since Feb 2023`

These mirror `vmin` / `vmax`, but for the container instead of the viewport:

```
cqmin = 1% of the smaller container dimension
cqmax = 1% of the larger container dimension
```

`cqmin` is useful for things that should fit inside the container regardless of orientation: a square media area, an icon that must never overflow either axis. Like `cqb`/`cqh`, `cqmin` and `cqmax` are most meaningful when the container has both dimensions defined (`container-type: size`).

> IN SHORT
>
> `cqmin` / `cqmax` are the "fit inside the box" / "cover the box" strategies — the container-scoped versions of `vmin` / `vmax`.

What happens with no container?

This is the rule that saves you from mysterious zero-sized layouts:

> *A `cq*` unit with no query container to measure doesn't fail: it falls back to the **small viewport**, matching axis for axis.*

So `cqi` with no container behaves like `svi`, `cqb` like `svb`, and so on. Nothing throws; the value just stops being container-relative and quietly becomes small-viewport-relative, which can still look wrong (see the pitfall below).

> PITFALL
>
> The fallback is silent. If you forget `container-type` on the ancestor, `font-size: 4cqi` doesn't throw. It resolves against the small viewport, which can look almost right at some widths and very wrong at others. When container-query sizing "isn't responding," the first thing to check is whether an ancestor actually declares `container-type`.

The killer use case: truly portable components

Before container queries, “responsive” meant “responds to the viewport.” But a component doesn’t control the viewport — all it ever sees is the box it’s dropped into. Container query units close that gap.

A single card definition can now:

- stack on narrow, go side-by-side on wide, based on its slot, not the screen;
- scale its own headline with `clamp(1rem, 5cqi, 2rem)`;
- be dropped into a sidebar, a modal, a grid cell, or a full-width hero **without variant classes or viewport breakpoints**.

That’s the real win: components become genuinely context-independent. The same markup is correct everywhere because it measures the only thing that matters to its own layout — its container.

```
.card-slot { container-type: inline-size; }   /* the container… */
.card      { /* …and the component inside it */ }

.card__title { font-size: clamp(1rem, 5cqi, 2rem); }
.card__body  { padding: 4cqi; }

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

Note the wrapper. The `@container` rule styles *descendants* of the query container, never the container itself, so the element that carries `container-type` and the element that reacts to the query have to be two different boxes.

Compact reference

UNIT	1 UNIT =
`cqi`	1% of the query container's inline size — the workhorse
`cqw`	1% of the query container's width
`cqb`	1% of the query container's block size
`cqh`	1% of the query container's height
`cqmin`	1% of the smaller container dimension
`cqmax`	1% of the larger container dimension

SETUP	EFFECT
`container-type: inline-size`	query the inline axis — the common case
`container-type: size`	query both axes — needs a definite height
`container-type: normal`	style queries / naming only (the default)
`container-name: sidebar`	let descendants target a specific ancestor

With no eligible query container, `cq*` units fall back to the small viewport (`sv*`).

SUPPORT AND CONTAINMENT BEHAVIOR CHECKED AGAINST MDN AND CSS CONTAIN 3 · 3 AUG 2026

Sources

- CSS Containment Module Level 3 — container query length units (w3.org)
- MDN — container query length units (developer.mozilla.org)
- MDN — `container-type` (developer.mozilla.org)
- MDN — CSS container queries (developer.mozilla.org)

CHAPTER 6 OF 7 · 1 UNIT

The fr Unit

Not a length at all: a share of whatever is left.

`fr` is the odd one out. Every other unit in this guide resolves to a length: pixels, eventually. `fr` doesn't. It's a **flexible length** that only means anything inside CSS Grid, and it represents a *share of the leftover space* rather than a fixed measurement. [1 Its spec type is `<flex>`, and `fr` is the only unit in it. Angles, times and resolutions have their own types too — but they measure quantities. `<flex>` exists to express a proportion, which is why it can't mix with lengths.]

> `1fr = one fraction of the free space in the grid container`

The word that matters is **leftover**. `fr` is computed last, after fixed tracks, gaps, and content-based minimums have already taken their space. Whatever remains is divided among the `fr` tracks in proportion to their values.

Demo — How `fr` distributes leftover space

Runs live at cssunits.com/article/fr

`fr`

FORMULA	`1fr = one share of the grid's remaining free space`
RELATIVE TO	Leftover space after fixed tracks, gaps, and content minimums
SUPPORT	● Universal

Consider three equal columns:

```
.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

The container width is split into three equal shares — each track is one-third.

Change the proportions and the shares re-weight:

```
grid-template-columns: 1fr 2fr 1fr;
```

Now there are `1 + 2 + 1 = 4` shares. The middle track gets `2/4` (half the width); the outer tracks get `1/4` each.

Mix fixed and flexible tracks, and the fixed ones are removed *first*:

```
grid-template-columns: 200px 1fr;
```

The fixed `200px` (plus any gap) is subtracted, and the single `1fr` track takes **all** the remaining space. This is the canonical sidebar-plus-content layout.

✓ USE WHEN

- ✓ Proportional column/row layouts: `1fr 2fr`
- ✓ Sidebar + content: `200px 1fr` or `240px 1fr`
- ✓ Equal-width tracks: `repeat(3, 1fr)`
- ✓ Filling remaining space after fixed tracks are placed
- ✓ Combining with `minmax()` and `repeat(auto-fit, …)` for fluid grids

✗ AVOID WHEN

- ✗ Outside Grid — `fr` is invalid in `width`, `flex-basis`, etc.
- ✗ When you actually want content-based sizing — use `auto` or `min-content`
- ✗ Assuming bare `1fr` can shrink below its content — it has an implicit `auto` minimum

---

The `minmax(0, 1fr)` trap

This is the single most common `fr` gotcha, and it causes mysterious overflow.

A bare `1fr` is not `minmax(0, 1fr)`. It's effectively `minmax(auto, 1fr)`. That `auto` minimum means **the track will not shrink below the size of its content**. So if a grid item contains something that can't shrink (a long unbroken string, an image with intrinsic width, a `white-space: nowrap` element), the track refuses to go below that minimum, and the grid blows out of its container.

```
/* Can overflow: the fr track won't shrink below its content */
.grid {
  grid-template-columns: 1fr 1fr;
}
```

The fix is to give the track a real zero minimum:

```
/* Allows the track to shrink below content size */
.grid {
  grid-template-columns: minmax(0, 1fr) minmax(0, 1fr);
}
```

In the demo above, compare the `1fr 1fr 1fr` template (long content forces the track wide) with `minmax(0, 1fr) 1fr` (the minmax track shrinks and ellipsizes).

> PITFALL
>
> `1fr` means `minmax(auto, 1fr)`, not `minmax(0, 1fr)`. The `auto` minimum prevents shrinking below content size, the usual cause of "my grid overflows its container." When a track holds potentially-long or non-shrinkable content, write `minmax(0, 1fr)` (and often `min-width: 0` on the item too).

`fr` can't go inside calc()

`fr` is a *flexible* length, not a regular one, so it can't be mixed with ordinary lengths in `calc()`:

```
/* Invalid — fr is not allowed in calc() */
grid-template-columns: calc(1fr - 2em);
```

If you need "flexible track, minus some fixed space," that's what the track list itself is for: put the fixed amount in its own track or use `minmax()`:

```
/* Do this instead */
grid-template-columns: 2em 1fr;          /* fixed gutter + flexible track
*/
grid-template-columns: minmax(0, 1fr);   /* flexible, allowed to shrink
*/
```

`fr` *can* sit alongside fixed lengths in the same track list, though: `250px repeat(12, 1fr)` is perfectly valid; the flexible tracks just divide whatever the fixed `250px` leaves behind.

`fr` is not `flex-grow`

`fr` and `flex-grow` feel similar (both distribute extra space proportionally), but they resolve differently, and conflating them leads to surprises.

- `flex-grow` distributes only the *extra* space, and it does so **on top of each item's** `flex-basis` (which defaults to the content size). Two items with `flex-grow: 1` but different content can end up *different* widths: each keeps its content-based starting size and only the leftover space is shared equally, so unequal content yields unequal totals.
- `fr` distributes the *track* space itself. Two `1fr` tracks are equal regardless of content (assuming the `auto` minimum doesn't intervene), because `fr` sizes the track, not the growth-beyond-basis.

```
/* Flexbox: items can end up unequal — growth adds to each item's content basis */
.flex { display: flex; }
.flex > * { flex-grow: 1; }   /* basis stays auto → wider content keeps a wider item */

/* Grid: tracks are genuinely equal shares of the container */
.grid { display: grid; grid-template-columns: 1fr 1fr; }
```

(Note that `flex: 1` is the *equalizing* case: its shorthand sets `flex-basis: 0%`, which is exactly what removes the content-based difference. Plain `flex-grow: 1` is what exposes the inequality.)

A useful way to hold it:

> *`flex-grow` answers "how should the extra space be shared?" `fr` answers "how should the whole space be divided into tracks?" That's why `fr` lives in Grid and not in Flexbox — Grid defines explicit tracks; Flexbox grows items.*

Why `fr` only exists in Grid

`fr` needs a defined set of tracks to divide space among. Grid provides exactly that — an explicit track list in `grid-template-columns` / `grid-template-rows`. Flexbox has no track list; it has items that flex along a single axis, so it uses `flex-grow` / `flex-shrink` / `flex-basis` instead. That's why `fr` is valid across Grid's track-sizing properties — `grid-template-columns` / `-rows`, the `grid-auto-columns` / `-rows` that size implicit tracks, and the `grid` / `grid-template` shorthands — and nowhere outside Grid.

Compact reference

`FR`	
Meaning	one share of the grid's leftover free space
Computed	after fixed tracks, gaps, and content minimums are subtracted
Valid in	`grid-template-columns` / `-rows`, `grid-auto-columns` / `-rows`, and the `grid` / `grid-template` shorthands (incl. `repeat()` and `minmax()`)
Invalid in	`width`, `flex-basis`, `gap`, `calc()` — anything expecting a length

PATTERN	RESULT
`1fr`	= `minmax(auto, 1fr)` — implicit auto minimum, can overflow
`minmax(0, 1fr)`	lets the track shrink below its content
`1fr 2fr 1fr`	shares 1 : 2 : 1 → 25% / 50% / 25%
`200px 1fr`	fixed track is removed first; `fr` takes the rest

And the sibling concept: `flex-grow` distributes the *extra* space on top of each item's `flex-basis` (content-aware), while `fr` divides the *whole* track space into shares (track-based).

Sources

- CSS Grid Layout Module Level 1 — the fr unit (w3.org)
- MDN — `<flex>` (the fr value) (developer.mozilla.org)
- MDN — `minmax()` (developer.mozilla.org)
- MDN — `grid-template-columns` (developer.mozilla.org)

CHAPTER 7 OF 7 · 12 UNITS

Special Units

Not everything CSS measures is a distance.

Everything so far has measured *distance*: lengths that ultimately resolve to pixels. But CSS has four more families of units that measure other dimensions entirely: **angles, time, frequency, and resolution**.

```
angle       deg, grad, rad, turn   — rotations, gradients, hue
time        s, ms                  — transitions and animations
frequency   Hz, kHz                — speech CSS (essentially unused)
resolution  dpi, dpcm, dppx, x     — density media queries, image-set()
```

They're not `<length>` values, so you can't put them in `width` or `padding`. In day-to-day work, angles and time come up constantly, resolution occasionally, and frequency almost never (its properties aren't implemented). Here's the honest tour.

Angles — `deg`, `grad`, `rad`, `turn`

FORMULA `360deg = 400grad = 2π rad = 1turn`

RELATIVE TO A full circle

SUPPORT ● Universal

Four units, one circle. They all describe rotation; they just divide the circle differently:

```
deg   degrees      — a full circle is 360deg
grad  gradians     — a full circle is 400grad
rad   radians      — a full circle is 2π ≈ 6.283rad
turn  turns        — a full circle is 1turn
```

`deg` is the everyday choice. `turn` reads well for simple fractions (`0.5turn` = half a rotation). `rad` shows up when values come from JavaScript math (`Math.PI`). `grad` is the odd one out — an eighteenth-century French attempt to decimalize the circle that survives in surveying and on European calculators; valid and unambiguous in CSS, just unfamiliar to most readers.

```
Demo — One angle, four units
Runs live at cssunits.com/article/special
```

Angles appear in more places than people expect: [1 Hue is an angle, which makes every color picker secretly a protractor — `oklch(0.58 0.15 42)` points at this site's terracotta the way a compass points north-east.]

```
.spin    { transform: rotate(45deg); }
.pie     { background: conic-gradient(red 0turn, blue 1turn); }
.diagonal{ background: linear-gradient(135deg, #f00, #00f); }
.swatch  { background: hsl(200deg 80% 50%); }   /* hue is an angle */
```

✓ USE WHEN

- ✓ Rotations: `transform: rotate(45deg)`
- ✓ Gradients: `linear-gradient(135deg, …)`, `conic-gradient()`
- ✓ Hue channel in `hsl()` / `oklch()`: `hsl(200deg …)`
- ✓ `turn` for clean fractions: `0.5turn`, `0.25turn`
- ✓ `rad` when the value comes straight from JS trig

✕ AVOID WHEN

- ✗ Defaulting to `grad` for shared code — prefer `deg`, which more readers parse at a glance
- ✗ Mixing angle units within one codebase without reason — pick one and stay consistent

IN SHORT

`deg` for almost everything, `turn` when fractions of a circle read more clearly, `rad` when JavaScript hands you radians. `grad` is valid but rarely the most readable choice for web work.

Time — `s` and `ms`

FORMULA	`1s = 1000ms`
RELATIVE TO	Seconds
SUPPORT	● Universal

Two units, one quantity. Time drives `transition` and `animation` durations and delays:

```
.fade {
  transition: opacity 0.3s ease;
}

.slide {
  animation: in 600ms ease-out;
}
```

`1s = 1000ms`, so they're fully interchangeable. Teams usually pick one for consistency; `ms` reads naturally for short UI timings (`150ms`, `300ms`), `s` for longer ones (`1.5s`).

The numbers themselves matter more than the unit. The rough bands most interface work settles on: around `100ms` reads as instant, `200-300ms` as smooth motion, and past half a second an interface starts to feel like it's ignoring you. These are conventions rather than findings, and worth re-testing on your own product, but they explain why most UI transitions live in a narrow `150-300ms` band — and why `ms` is usually the more natural notation.

```
Demo — Transition duration in s and ms
Runs live at cssunits.com/article/special
```

PITFALL

Time always needs a unit. `transition-duration: 1` is **invalid** and silently ignored — there is no "unitless seconds." It must be `1s` or `1000ms`. (This trips people up because `line-height` and `z-index` *do* take unitless numbers; time does not.)

✓ USE WHEN

- ✓ Transition and animation durations: `0.2s`, `200ms`
- ✓ Delays: `transition-delay`, `animation-delay`
- ✓ `ms` for short UI timings; `s` for longer sequences
- ✓ Negative time on `animation-delay` to start mid-animation

✕ AVOID WHEN

- ✗ Unitless time — `1` is invalid; always write `1s` / `1000ms`
- ✗ Time anywhere outside duration/delay properties — it's not a length

IN SHORT

`1s = 1000ms`, always with a unit. The only real gotcha is that a bare number is invalid.

Frequency — `Hz` and `kHz`

FORMULA	`1kHz = 1000Hz`
RELATIVE TO	Cycles per second
SUPPORT	● Not implemented

Frequency units exist in the spec for **aural / speech CSS**: properties like `pitch` that were meant to control synthesized speech. `1kHz = 1000Hz`.

In practice they have essentially no browser support: the speech CSS properties they were designed for were never broadly implemented, and the aural module was deprecated. Voice characteristics are set by the reader in their screen reader or OS instead, which is the honest answer — the CSS Speech module that replaced the aural one is barely implemented either, and ARIA describes semantics, not delivery.

✓ USE WHEN

- ✓ Know they exist so the spec's unit list isn't a mystery
- ✓ Recognize them if you encounter very old aural stylesheets

✕ AVOID WHEN

- ✗ Relying on them — the consuming properties aren't implemented in shipping browsers
- ✗ Confusing CSS frequency units with the Web Audio API (unrelated)

IN SHORT

`Hz` and `kHz` are there for spec completeness; with no implemented properties to use them, recognizing them is about all they ask of you.

Resolution — `dpi`, `dpcm`, `dppx`, `x`

FORMULA	`1x = 1dppx = 96dpi`
RELATIVE TO	Pixel density of the output device
SUPPORT	● Universal

Resolution units describe **pixel density** — how many device dots fit in a unit of length:

```
dpi   dots per inch
dpcm  dots per centimeter
dppx  dots per px unit (1dppx = one device pixel per CSS pixel)
x     shorthand for dppx
```

The key equivalence ties them to the absolute-units anchor:

```
1x = 1dppx = 96dpi
```

Because `1in = 96px`, “96 dots per inch” is the same as “1 dot per CSS pixel.” A 2× Retina display reports `2dppx` / `2x` — equivalently `192dpi` in media-query terms (not the panel’s physical ppi, which is far higher).

```
Demo — Your display's resolution, in CSS units
Runs live at cssunits.com/article/special
```

You won’t write resolution units in `width`; they’re for two specific places: density media queries and resolution-aware image selection.

```
/* Serve a sharper background to HiDPI screens */
@media (min-resolution: 2dppx) {
  .logo { background-image: url(logo@2x.png); }
}

/* Let the browser pick the right asset by density */
.avatar {
  background-image: image-set(
    "avatar.png"    1x,
    "avatar@2x.png" 2x
  );
}
```

✓ USE WHEN

- ✓ Density media queries: `@media (min-resolution: 2dppx)`
- ✓ Resolution-aware images: `image-set()` with `1x` / `2x`
- ✓ Prefer `dppx` / `x` — they map directly to devicePixelRatio
- ✓ Print stylesheets that genuinely target a known DPI

✗ AVOID WHEN

- ✗ As a length — resolution units can't size boxes
- ✗ `dpi` for screens — it works, but screens are described by device-pixel ratio, which `dppx` / `x` map to directly
- ✗ `dpcm` — the metric sibling of `dpi`, with the same screen caveat and little practical use
- ✗ Assuming `min-resolution` alone covers all HiDPI cases — test on real devices

IN SHORT

`1x = 1dppx = 96dpi`. Use `dppx` / `x` for density media queries and `image-set()`; they're the screen-density mirror of the `px` / `in` relationship.

Compact reference

FAMILY	UNITS	ANCHOR	USED FOR
Angles	`deg` `grad` `rad` `turn`	360deg = 400grad = 2π rad = 1turn	`rotate()`, gradients, `hsl()` hue
Time	`s` `ms`	1s = 1000ms — unitless is invalid	transition / animation duration & delay
Frequency	`Hz` `kHz`	1kHz = 1000Hz	speech CSS — not implemented
Resolution	`dpi` `dpcm` `dppx` `x`	1x = 1dppx = 96dpi	resolution media queries, `image-set()`

Sources

- CSS Values and Units Module Level 4 — angles (w3.org)
- CSS Values and Units Module Level 4 — time (w3.org)
- CSS Values and Units Module Level 4 — resolution (w3.org)
- MDN — `<angle>` (developer.mozilla.org)
- MDN — `<time>` (developer.mozilla.org)
- MDN — `<resolution>` (developer.mozilla.org)

- MDN — `image-set()` (developer.mozilla.org)

REFERENCE

Units in Math Functions

Where units stop being numbers and start being sentences.

Knowing every unit is half the story. The other half is **combining** them. A modern layout rarely uses a unit in isolation; it mixes them: *"this big, but never smaller than that, and no larger than this."* That's what CSS math functions are for, and they're where units stop being a glossary and start being a system.

```
calc()                        mix units and do arithmetic
min() max()                   take the smallest / largest value
clamp()                       lock a value between a floor and a ceiling
round() mod() rem()           snap a value to a step; take a remainder
sin() cos() tan() atan2()     turn angles into numbers, and back
pow() sqrt() hypot() log()    exponents, roots, distances
abs() sign()                  magnitude and direction
pi  e  infinity  NaN          constants you'd otherwise type out
```

The first four are the ones that mix units — `px` with `rem`, `%` with `vw`, `em` with `cqi` — resolving them at used-value time. That's the bridge between every unit family in this guide. The rest of the list is fussier about types, and each section below says how.

Everything above the typed-arithmetic section is cross-browser: the four classics have been universal for a decade, and the rest of the list reached all three engines between 2023 and mid-2025. Each section below states when.

`calc()`

`calc()` performs arithmetic, and crucially it can **mix units** that would otherwise be incompatible:

```
.sidebar { width: calc(100% - 240px); }
.title   { font-size: calc(1rem + 1vw); }
.box     { margin: calc(1.5em + 4px); }
```

The browser resolves the mixed terms once it knows the concrete values during layout. A few rules that trip people up:

- **Operators need whitespace.** `calc(100% - 240px)` is valid; `calc(100%-240px)` is not. The `+` and `-` operators *must* be surrounded by spaces (because `-240px` is otherwise a single negative number).
- **Multiplying by a number always works; multiplying two lengths produces nothing usable.** `calc(2 * 1rem)` is fine. `calc(1rem * 1rem)` would be a length², and CSS has no `<area>` type for any property to consume, so it's useless rather than helpful. Dividing two lengths, by contrast, cancels the units to a plain `<number>`, which *is* usable (see typed arithmetic below).
- **It nests.** `calc()` can contain `min()`, `max()`, `clamp()`, and other `calc()`s.

✓ USE WHEN

✓ Mixing units: `calc(100% - 240px)`, `calc(1rem + 1vw)`

✓ Subtracting a fixed gutter from a flexible width

✓ Composing with `min()` / `max()` / `clamp()`

✗ AVOID WHEN

✗ Forgetting spaces around `+` / `-` — it silently invalidates the value

✗ Using `fr` inside `calc()` — flexible lengths aren't allowed there

✗ Reaching for it where `clamp()` or a single relative unit already expresses the intent

`min()` and `max()`

`min()` returns the smallest of its comma-separated arguments; `max()` the largest. Both accept mixed units.

```
/* Never wider than 65ch, but shrink on small screens */
.article { inline-size: min(100%, 65ch); }

/* At least 44px tall for touch targets, larger if the content wants */
.btn { min-block-size: max(44px, 2.5em); }
```

The intuition flips for the responsive case: **`min()` sets a ceiling** (the value can't exceed the smallest argument), while **`max()` sets a floor**. `min(100%, 65ch)` means "65ch, but cap at the available width."

✓ USE WHEN

- ✓ Responsive caps: `min(100%, 65ch)` for a fluid-but-bounded column
- ✓ Accessibility floors: `max(44px, 2.5em)` for minimum touch size
- ✓ Mixing relative + absolute to express intent in one line

✕ AVOID WHEN

- ✗ Confusing the directions — `min()` caps, `max()` floors
- ✗ Stacking a floor and a ceiling by hand where `clamp(min, pref, max)` expresses exactly that

---

`clamp()`

`clamp(MIN, PREFERRED, MAX)` is `max(MIN, min(PREFERRED, MAX))` in one readable call: a value that scales with `PREFERRED` but never drops below `MIN` or rises above `MAX`. It's the backbone of fluid typography: `[1` `clamp()` reads like a sentence: *never smaller than, ideally, never larger than.* The most literary function in CSS.]

```
.headline {
  font-size: clamp(2rem, 5vw + 1rem, 4rem);
}
```

This is the pattern that appears throughout this guide — in the viewport and container sections especially. The `PREFERRED` term is where you put your fluid unit (`vw`, `cqi`), while the `rem` floor and ceiling keep it accessible and sane.

Build one below: drag the four terms and the simulated viewport, and watch which of the three arguments is actually deciding the size.

Demo — Build a `clamp()` and see which term wins

Runs live at cssunits.com/article/math-functions

PITFALL

Zoom scales `rem` but not `vw`: magnifying the page shrinks the layout viewport by the same factor, so a `vw` term's rendered size doesn't budge. A `font-size` whose `PREFERRED` term is **pure viewport units** is WCAG 1.4.4 (w3.org)'s F94 failure (w3.org) in miniature. `rem` bounds do rescue it — under enough zoom the floor takes over — but they spend the reader's zoom budget doing it. The playground above computes how much: the canonical `clamp(2rem, 5vw + 1rem, 4rem)` reaches only 131% at 200% zoom and needs 330% to double. The same arithmetic applies to a fluid *root* — including this site's own, which is measured there rather than defended.

✓ USE WHEN

- ✓ Fluid type: `clamp(2rem, 5vw + 1rem, 4rem)`
- ✓ Fluid spacing: `clamp(1rem, 4cqi, 2rem)` tied to a container
- ✓ Always include a `rem` term in the preferred value for zoom safety

✗ AVOID WHEN

- ✗ Pure viewport preferred value: `clamp(2rem, 8vw, 4rem)` still risks zoom failure
- ✗ A cap written in `px` — it pins the ceiling where the reader's zoom can't lift it
- ✗ Using it for fixed values that never need to scale

`round()`, `mod()`, `rem()`

SUPPORT

All three engines since **May 2024**: Safari 15.4 (2022), Firefox 118 (2023), Chrome 125 (2024).

The stepped-value functions quantize a value: they take a length (or angle, or time) and snap it to a multiple of a step you choose.

```
round(strategy?, value, step)   snap value to the nearest multiple of step
mod(a, b)                       remainder, sign follows the DIVISOR
rem(a, b)                       remainder, sign follows the DIVIDEND
```

`round()` takes four strategies: `nearest` (the default), `up`, `down`, and `to-zero`:

```css
/* A three-column track that never lands on a fractional pixel */
.col { inline-size: round(down, 100% / 3, 1px); }

/* Snap fluid spacing to the vertical rhythm of one line */
.stack { margin-block-end: round(3.4vh, 1lh); }
```

That first example is the practical one: fluid math routinely produces `213.333px`, and sub-pixel edges are where hairline borders go soft. `round(down, …, 1px)` gives you fluid *and* crisp.

`mod()` and `rem()` differ only in whose sign survives, which matters exactly once — when one operand is negative:

```
mod(5px, 3px)   =  2px        rem(5px, 3px)   =  2px
mod(-5px, 3px)  =  1px        rem(-5px, 3px)  = -2px
```

PITFALL

A zero step makes `round()` return `NaN`, and a `NaN` that reaches the top level of a property computes to **zero** — so the box quietly collapses instead of falling back to something sane. Guard any step that arrives from a custom property: `round(down, 100%, max(1px, var(--step)))`. (A *unitless* `0` step is a different failure: `round(down, 100px, 0)` is a type error, so the declaration is dropped rather than zeroed.)

Trigonometry — `sin()`, `cos()`, `tan()`, and friends

> SUPPORT
>
> All three engines since **March 2023**: Safari 15.4 (2022), Firefox 108 (2022), Chrome 111 (2023).

This is where the angle units stop being decoration and start doing arithmetic. The trig functions are the bridge between angles and plain numbers:

```
sin(<angle>) cos(<angle>) tan(<angle>)   → <number>
asin() acos() atan() atan2(y, x)         → <angle>
```

Since a `<number>` can be multiplied back into any type, “angle in, length out” becomes a one-liner. The canonical use is placing things on a circle without JavaScript:

```css
.dot {
  --angle: 30deg;
  --radius: 8rem;
  translate:
    calc(cos(var(--angle)) * var(--radius))
    calc(sin(var(--angle)) * var(--radius));
}
```

Combined with `sibling-index()` (below) this distributes any number of children around a dial with no per-item CSS. [2 The same identity underneath a clock face, a radial menu, and a pie chart — CSS can now express all three in the units it already had.]

PITFALL

A bare number inside `sin()` / `cos()` / `tan()` is **radians**, not degrees. `sin(45)` is `0.851`; `sin(45deg)` is `0.707`. If a value arrives as a unitless custom property, convert it explicitly: `sin(calc(var(--deg) * 1deg))`.

Exponentials — `pow()`, `sqrt()`, `hypot()`, `log()`, `exp()`

SUPPORT

All three engines since **December 2023**: Safari 15.4 (2022), Firefox 118 (2023), Chrome 120 (2023).

`pow()`, `sqrt()`, `log()`, and `exp()` are strictly numeric: numbers in, numbers out, no units allowed. `hypot()` is the exception, and the reason this family belongs in a guide about units: it accepts *dimensioned* values and returns one:

```
/* The diagonal of a 3×4 box, as a length */
--diagonal: hypot(3rem, 4rem);   /* = 5rem */
```

The numeric ones are how you build a modular type scale in CSS itself, instead of pasting values from a generator:

```
:root {
  --ratio: 1.25;
  --step-1: calc(1rem * pow(var(--ratio), 1));   /* 1.25rem */
  --step-2: calc(1rem * pow(var(--ratio), 2));   /* 1.5625rem */
  --step-3: calc(1rem * pow(var(--ratio), 3));   /* 1.953rem */
}
```

Change `--ratio` and the whole scale re-derives — the unit stays `rem`, only the multiplier moves.

`abs()` and `sign()`

> SUPPORT
>
> All three engines since **June 2025**: Safari 15.4 (2022), Firefox 118 (2023), Chrome 138 (2025).

```
abs(x)    magnitude, keeping the type:  abs(-2rem) = 2rem
sign(x)   direction, as a plain number: sign(-2rem) = -1
```

`abs()` is for offsets that must be positive whichever way a variable points; `sign()` turns a signed length into a `-1 / 0 / 1` flag you can multiply by anything:

```
.tooltip {
  --offset: -12px;                          /* may be either sign */
  padding-block: abs(var(--offset));        /* always positive spacing */
  translate: calc(sign(var(--offset)) * 4px);
}
```

Chrome was five years behind Safari here, so this pair only became safe to use without a fallback in mid-2025.

Constants — `pi`, `e`, `infinity`, `NaN`

> SUPPORT
>
> `pi` and `e` in all engines since **early 2023**; `infinity` / `NaN` since **June 2023**.

Inside any math function you can write these instead of typing digits:

```
--half-turn:  calc(1rad * pi);   /* π radians — the long way to say
0.5turn */
--e-squared:  pow(e, 2);         /* 7.389… */
```

(`log()` is already natural: `log(x)` is base *e*, and `log(x, 10)` when you want decades.)

`infinity` has one genuinely useful trick — the "pill" radius that can't be beaten by content size:

```
.pill { border-radius: calc(infinity * 1px); }
```

It reads better than `9999px` and says what it means: *as round as this box can be.*

PITFALL

`NaN` is easy to misdiagnose, so it's worth knowing both halves of the rule. Dividing by zero is *not* how you get it: `calc(1px / 0)` is `infinity`, which the browser clamps to the largest value the property accepts (`width` lands somewhere around 3.4×10^7px). `NaN` comes from the indeterminate cases — `0 / 0`, `infinity - infinity`, `0 * infinity` — and from a zero-step `round()`. And a `NaN` that reaches the top level of a property **computes to zero** of that type. So the symptom isn't a fallback and isn't a wild number: it's a length that silently became `0`, which in a layout usually looks like a box that vanished.

Typed arithmetic

For most of CSS's history, `calc()` let you *multiply* a length by a number but not *divide one length by another*. **Typed arithmetic** in CSS Values 4 changes that model: compatible typed values can now participate in unit algebra, so dividing one length by another can cancel the units and produce a plain `<number>`.

```
/* Divide a length by a length → unitless ratio */
--columns: calc(100cqi / 20rem);

/* Multiply a unitless value back into a type */
--angle: calc(var(--ratio) * 1turn);
```

The rules:

- **Division of same-type values cancels units** → the result is a `<number>` you can reuse anywhere a number is allowed.

- **Multiplying a number by a typed value** (`* 1deg`, `* 1rem`) **converts it back** into that type.
- It works across the numeric types: `<length>`, `<percentage>`, `<angle>`, and so on.

The round-trip this unlocks — divide to a number, compute, then multiply back into a type — is the whole point:

```
/* turn the viewport width into a unitless number, then into an angle */
--vw-count: calc(100vw / 1px);              /* e.g. 1280 on a 1280px-wide viewport */
--spin:     calc(var(--vw-count) * 0.1deg); /* now usable wherever an <angle> is */
```

PITFALL

Typed arithmetic is the newest thing in this chapter, and the only feature here that is **not** cross-browser. Length-by-length division shipped in Chrome 140 and Safari 26, both in September 2025, and Firefox has not implemented it yet (bug 1264520 (bugzil.la)). Treat it as an enhancement, not a foundation:

```
.thing {
  --spin: 12deg;                                /* fallback everyone gets */
}
@supports (width: calc(100vw / 1px * 1px)) {
  .thing {
    --vw-count: calc(100vw / 1px);
    --spin: calc(var(--vw-count) * 0.01deg);  /* only where it resolves */
  }
}
```

SUPPORT CHECKED AGAINST MDN BROWSER-COMPAT DATA · 3 AUG 2026

IN SHORT

Units rarely work alone. `calc()` mixes them, `min()` / `max()` bound them, `clamp()` does both at once, `round()` snaps them to a step, trigonometry converts angles to numbers, and typed arithmetic lets compatible units divide into ratios where supported. This is the layer that turns the unit catalog into a sizing system.

The new frontier

Everything above is either cross-browser or, in the case of typed arithmetic, two engines in. This last group is younger still — worth knowing, worth using behind `@supports`, not worth depending on.

FUNCTION	WHAT IT GIVES YOU	SUPPORT
`progress(v from a to b)`	a unitless 0–1 position of `v` between two typed values	Chrome 138, Safari 26 — no Firefox
`sibling-index()` / `sibling-count()`	the element's position among its siblings, as an `<integer>`	Chrome 138, Safari 26.2 — Firefox in preview
`calc-size()`	lets intrinsic sizes (`auto`, `max-content`) take part in math	Chrome 129 only
`if()`	inline conditionals over `style()` / `media()` / `supports()` tests	Chrome 137 only
`random()`	a random value in a range, resolved once at computed-value time	Safari 26.2 only

SUPPORT CHECKED AGAINST MDN BROWSER-COMPAT DATA · 3 AUG 2026

Two of them matter for units specifically.

`progress()` does what typed division does, without the arithmetic: it maps a value onto a 0–1 scale, and both ends can carry units.

```
/* 0 at a 20rem viewport, 1 at 80rem — then use it anywhere a number fits */
--fluid: progress(100vw from 20rem to 80rem);
opacity: var(--fluid);
font-size: calc(1rem + var(--fluid) * 1.5rem);
```

`sibling-index()` finally gives CSS a counter usable inside `calc()`, so a stagger no longer needs one rule per child:

```
li {
  animation-delay: calc(sibling-index() * 60ms);
  rotate: calc(sibling-index() * (360deg / sibling-count()));
}
```

Multiply the integer by a unit and you have the delay, angle, or offset you wanted: the same “number × unit” move that typed arithmetic makes explicit.

PITFALL

None of these have three-engine support yet, and their fallback behavior differs: an unsupported *function* makes the declaration invalid at parse time, so the previous cascade value survives — but only if you wrote one. Always ship the plain declaration first, then upgrade:

```
li { animation-delay: 0ms; }
@supports (animation-delay: calc(sibling-index() * 1ms)) {
  li { animation-delay: calc(sibling-index() * 60ms); }
}
```

Compact reference

FUNCTION	WHAT IT DOES	TAKES UNITS?
`calc(expr)`	mixes units and does arithmetic; `+` and `-` need surrounding spaces	yes
`min(a, b, …)`	takes the smallest argument — acts as a **ceiling**	yes
`max(a, b, …)`	takes the largest argument — acts as a **floor**	yes
`clamp(min, pref, max)`	= `max(min, min(pref, max))`	yes
`round(strategy?, v, step)`	snaps `v` to a multiple of `step` (`nearest`/`up`/`down`/`to-zero`)	yes
`mod(a, b)` / `rem(a, b)`	remainder; sign follows the divisor / the dividend	yes
`abs(x)` / `sign(x)`	magnitude, keeping the type / direction as `-1`, `0`, `1`	yes / returns number
`hypot(a, b, …)`	√(a² + b² …) — the one exponential that keeps units	yes
`pow` `sqrt` `log` `exp`	exponents, roots, logarithms	numbers only
`sin` `cos` `tan`	`<angle>` (or radians) → `<number>`	angle in
`asin` `acos` `atan` `atan2`	`<number>` → `<angle>`	angle out
`pi` `e` `infinity` `NaN`	constants inside any math function	—

TYPED ARITHMETIC	RESULT
length ÷ length	`<number>` — the units cancel
number × `1deg`	`<angle>` — converts a number back into a type

The fluid-type pattern to remember: `clamp(2rem, 5vw + 1rem, 4rem)` — the `rem` term keeps it zoom-accessible. And the recurring gotchas: `fr` is not allowed inside `calc()`, a pure-`vw` preferred value can fail WCAG 1.4.4, a bare number in `sin()` is radians, and a zero step in `round()` yields `NaN`.

Sources

- MDN — `calc()` (developer.mozilla.org)
- MDN — `min()` (developer.mozilla.org), `max()` (developer.mozilla.org), `clamp()` (developer.mozilla.org)
- MDN — `round()` (developer.mozilla.org), `mod()` (developer.mozilla.org), `rem()` (developer.mozilla.org)
- MDN — `sin()` (developer.mozilla.org), `atan2()` (developer.mozilla.org)
- MDN — `pow()` (developer.mozilla.org), `hypot()` (developer.mozilla.org)
- MDN — `abs()` (developer.mozilla.org), `sign()` (developer.mozilla.org)
- MDN — `<calc-keyword>`: `pi`, `e`, `infinity`, `NaN` (developer.mozilla.org)
- MDN — Using CSS typed arithmetic (developer.mozilla.org)
- MDN — `progress()` (developer.mozilla.org), `sibling-index()` (developer.mozilla.org), `calc-size()` (developer.mozilla.org), `if()` (developer.mozilla.org), `random()` (developer.mozilla.org)
- Smashing Magazine — Addressing accessibility concerns with fluid type (smashingmagazine.com)
- CSS Values and Units Module Level 4 — math functions (w3.org)
- CSS Values and Units Module Level 5 — the newer functions (drafts.csswg.org)

REFERENCE

Decision Tree

The right unit is the one that fails gracefully.

Forget the categories for a moment. In real work you don't think *"I need a viewport unit"* — you think *"I need this modal to fill the screen."* This page is organized by **task**, not by unit family. Find your scenario, get the answer and the reasoning.

> ***The one question that solves most cases:*** *relative to what should this size be? The element's own text? Use* `em`*. The page baseline?* `rem`*. The window?* `vw`*/* `vh`*/* `dvh`*. The component's box?* `cq*`*. The parent?* `%`*. Pin it exactly?* `px`*. Almost every decision below is a variation of that question.*

Runs live at cssunits.com/article/decision-tree

Prefer to browse? Every scenario the wizard covers is written out below, with the reasoning and the caveats.

Typography

Body text and headings → `rem`

```
body { font-size: 1rem; line-height: 1.5; }
h1   { font-size: clamp(2rem, 3vw + 1rem, 3rem); }
```

`rem` respects the user's default font-size setting and never compounds through nesting. Keep `line-height` unitless so it inherits as a *factor* (not a fixed length) and stays proportional to each element's own size.

A component's internal proportions (button padding, gap to its icon) → `em`

```
.button { font-size: 1rem; padding: 0.625em 1em; border-radius: 0.5em; }
```

`em` makes padding scale with the button's own text. Resize the button via `font-size` and every `em`-based dimension follows (a `px` border won't).

Headline that scales with its container, not the window → `cqi`

```
.card { container-type: inline-size; }
.card__title { font-size: clamp(1rem, 5cqi, 2rem); }
```

Reading column width → `ch`

```
.article { max-inline-size: 65ch; }
```

Aim for roughly 45–75 characters per line. `ch` is the advance of `0`, so `65ch` *approximates* that range rather than guaranteeing exactly 65 characters — proportional fonts vary (see the ch caveat).

PITFALL

Prefer `rem` over `px` for `font-size`: `px` ignores the user's **default-font-size** setting (someone who sets 20px still sees your `16px` as 16px). Page zoom *does* still scale `px` text — so this is about honoring that one preference, not about zoom being broken.

Full-screen and viewport layouts

Mobile full-screen modal → `dvh`

```
.modal { height: 100dvh; }
```

Tracks the *currently visible* viewport so bottom controls stay reachable as browser chrome moves. Use `100svh` if you prefer stability over filling every pixel. Prefer `dvh`/`svh` over `100vh` here: on current mobile browsers `100vh` resolves like `100lvh`, so it can hide content behind the chrome.

Landing hero with important content above the fold → `svh`

```
.hero { min-height: 100svh; }
```

The conservative viewport helps ensure the call-to-action is visible even when chrome is expanded (provided the content itself fits).

Immersive / full-bleed background → `lvh`

```
.cover { min-height: 100lvh; }
```

Cropping is acceptable here; you want maximum coverage.

App shell (chat, map, dashboard) → `dvh`

```
.app { min-height: 100dvh; display: grid; grid-template-rows: auto 1fr
auto; }
```

A square that always fits the screen → `vmin`

```
.board { width: 90vmin; height: 90vmin; }
```

PITFALL

For a section that should match its container's width, use `width: 100%`. Reserve `100vw` for deliberate edge-to-edge bleed — and know that on classic-scrollbar systems it includes the scrollbar gutter and can cause horizontal overflow.

Layout and spacing

Page-level spacing scale → `rem`

```
:root { --space-4: 1rem; --space-6: 1.5rem; }
.stack { display: grid; gap: var(--space-6); }
```

Stable everywhere, scales with the root, immune to nesting.

Proportional grid columns → `fr`

```
.layout { display: grid; grid-template-columns: 240px 1fr; }
```

Fixed sidebar, flexible content. Use `minmax(0, 1fr)` when a track holds long content that might overflow.

Fluid width relative to the parent → `%`

```
.col { width: 50%; }
```

Aspect-ratio box (modern) → `aspect-ratio`; legacy → `%` padding

```
.ratio { aspect-ratio: 16 / 9; }            /* modern */
.ratio-legacy { padding-top: 56.25%; }       /* % of width, not height */
```

Borders, icons, and fine details

Hairline borders and crisp details → `px`

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

A clear case for `px` — values that should stay constant regardless of font or viewport (borders, shadows, hairline offsets, fixed breakpoints).

Icon the size of the text → `1em` Icon the height of capital letters → `1cap` Icon that fills one line → `1lh`

```
.icon-text { width: 1em;  height: 1em;  }   /* matches type size */
.icon-cap  { width: 1cap; height: 1cap; }   /* matches capitals */
.icon-line { width: 1lh;  height: 1lh;  }   /* matches the line box */
```

Motion, rotation, and media

Transition / animation timing → `s` or `ms`

```
.fade { transition: opacity 200ms ease; }
```

`1s = 1000ms`. Remember: a bare number is invalid.

Rotation and gradients → `deg` (or `turn`)

```
.spin { transform: rotate(45deg); }
.pie  { background: conic-gradient(red 0turn, blue 1turn); }
```

Serve sharper images to HiDPI screens → `dppx` / `x`

```
@media (min-resolution: 2dppx) { /* … */ }
```

The quick table

YOU WANT…	USE	WHY
Body text / headings	`rem`	Respects user prefs, no compounding
Component-internal proportions	`em`	Scales with the component's text
Container-aware sizing	`cqi`	Responds to the box, not the window
Reading column width	`ch`	Tracks the font's character width
Mobile full-screen modal	`dvh`	Fits the currently visible viewport
Hero with content above fold	`svh`	Conservative height — fits with chrome expanded
Immersive background	`lvh`	Maximum coverage, cropping ok
Square that fits the screen	`vmin`	Follows the limiting axis
Full-width section	`%`	Avoids `100vw` scrollbar overflow
Grid columns	`fr`	Shares leftover space
Borders, hairlines	`px`	Must stay constant
Icon = text size	`1em`	Matches the type
Icon = one line	`1lh`	Matches the line height
Animation timing	`s` / `ms`	The only time units
Rotation / gradient angle	`deg` / `turn`	The angle units
HiDPI image switch	`dppx` / `x`	Density media queries

When you've matched a scenario, jump to the full unit article for the caveats. Or skim the Cheatsheet for every unit at a glance.

Sources

- CSS Values and Units Module Level 4 (w3.org)
- CSS Containment Module Level 3 — container query length units (w3.org)
- WCAG 2.2 Understanding Success Criterion 1.4.4: Resize Text (w3.org)

REFERENCE

Cheatsheet

The whole system on one page.

Every CSS unit at a glance: formula, reference, and common use. For the nuance behind each, follow through to its full article.

Absolute lengths

Fixed on screen via the anchor `1in = 96px`. → Full article

UNIT	EQUALS	IN PX	COMMON USE
`px`	`1/96in`	1	Borders, icons, crisp details
`pt`	`1/72in`	≈ 1.333	Print typography
`pc`	`12pt`	16	Print (picas)
`in`	anchor	96	Print / physical output
`cm`	`96px/2.54`	≈ 37.8	Print (metric)
`mm`	`1cm/10`	≈ 3.78	Print (small)
`Q`	`1mm/4`	≈ 0.945	Japanese print typography

Font-relative — local

Resolve against **the element's own font**. → Full article

UNIT	FORMULA	RELATIVE TO	COMMON USE	ALL ENGINES SINCE
`em`	× computed font-size	Element font-size (inherited on `font-size`)	Component-internal proportions	forever
`ch`	advance of `0`	Element font	Reading column width (`65ch`)	forever
`ex`	x-height	Element font	Lowercase-aware alignment	forever
`cap`	cap height	Element font	Icons next to headings	Dec 2023
`ic`	advance of `水`	Element font	CJK layout	Sep 2022
`lh`	computed line-height	Element line-height	Icon = one line	Nov 2023

Font-relative — root

Same metrics, measured on the **root element**. → Full article

UNIT	FORMULA	RELATIVE TO	COMMON USE	ALL ENGINES SINCE
`rem`	× root font-size	Root font-size	Type scale, spacing tokens	forever
`rch`	root advance of `0`	Root font	System-wide text measure	**Jan 2026**
`rex`	root x-height	Root font	Root-locked alignment	**Jan 2026**
`rcap`	root cap height	Root font	Root-locked cap sizing	**Jan 2026**
`ric`	root advance of `水`	Root font	Root-locked CJK measure	**Jan 2026**
`rlh`	root line-height	Root element	Vertical rhythm (`2rlh`)	Nov 2023

The four bold rows are the newest units in CSS: Chromium and Safari had them from 2023, Firefox only from version 147 (January 2026). See the support story.

Percentage

Reference is set **by the property**. → Full article

PROPERTY	% RELATIVE TO
`width` / `inline-size`	Containing block inline-size
`height` / `block-size`	Containing block block-size (if defined)
`padding-*` / `margin-*`	Containing block inline-size (always)
`transform: translate()`	The element's own size
`background-position`	(container – image) per axis
`border-radius`	Border-box width (horizontal radii) / height (vertical)
`font-size`	Inherited font-size
`top` / `left` / `right` / `bottom`	Containing block, matching axis

Viewport

`1×unit = 1%` of the chosen viewport dimension. The default family is defined against the large one. → Full article

AXIS / STRATEGY	DEFAULT	SMALL	LARGE	DYNAMIC
Width	`vw`	`svw`	`lvw`	`dvw`
Height	`vh`	`svh`	`lvh`	`dvh`
Inline	`vi`	`svi`	`lvi`	`dvi`
Block	`vb`	`svb`	`lvb`	`dvb`
Smaller axis	`vmin`	`svmin`	`lvmin`	`dvmin`
Larger axis	`vmax`	`svmax`	`lvmax`	`dvmax`

```
svh → safe (chrome expanded)   lvh → full (chrome retracted)
dvh → current (tracks chrome)  vh  = lvh (by definition)
```

`vw` / `vh` / `vmin` / `vmax` are as old as CSS3; the logical (`vi` / `vb`) and small/large/dynamic families reached all engines in **Nov 2022**.

Container query

`1×unit = 1%` of the nearest query container. Needs `container-type`. → Full article

UNIT	RELATIVE TO
`cqw`	Container width
`cqh`	Container height
`cqi`	Container inline size (most used)
`cqb`	Container block size
`cqmin`	Smaller container dimension
`cqmax`	Larger container dimension

> `No eligible container → falls back to the small viewport (sv*)`

All engines since **Feb 2023** (Chrome 105, Safari 16, Firefox 110).

Flexible length (Grid)

→ Full article

UNIT	MEANING	NOTES
`fr`	A share of the grid's leftover space	**CSS Grid only** (not flexbox). `1fr = minmax(auto, 1fr)`; use `minmax(0, 1fr)` to allow shrinking

---

Special — not lengths

→ Full article

Angles: `360deg = 400grad = 2π rad = 1turn`

UNIT	FULL CIRCLE	USE
`deg`	360	Rotations, gradients, hue
`grad`	400	Rare (surveying)
`rad`	2π ≈ 6.283	From JS trig
`turn`	1	Clean fractions

Time: `1s = 1000ms` (unitless is invalid)

UNIT	USE
`s`	Longer durations
`ms`	Short UI timings

Frequency: `1kHz = 1000Hz`

UNIT	USE
`Hz`, `kHz`	Speech CSS — effectively unused

Resolution: `1x = 1dppx = 96dpi`

UNIT	USE
`dppx` / `x`	Density media queries, `image-set()`
`dpi`	Dots per inch (print)
`dpcm`	Dots per cm (rare)

Math functions

Where units get combined. → Full article

FUNCTION	DOES	UNITS
`calc()`	arithmetic across types; `+` / `-` need spaces	mixes them
`min()` / `max()`	smallest / largest argument — a ceiling / a floor	mixes them
`clamp(a, b, c)`	`max(a, min(b, c))` — the fluid-type workhorse	mixes them
`round(strategy?, v, step)`	snap to a step; `nearest` / `up` / `down` / `to-zero`	keeps them
`mod()` / `rem()`	remainder — sign follows divisor / dividend	keeps them
`abs()` / `sign()`	magnitude / direction as `-1`, `0`, `1`	keeps / drops
`hypot()`	√(a²+b²…) — diagonals as a length	keeps them
`pow` `sqrt` `log` `exp`	exponents, roots, logs — type scales	numbers only
`sin` `cos` `tan`	angle → number	angle in
`asin` `acos` `atan` `atan2`	number → angle	angle out
`pi` `e` `infinity` `NaN`	constants; `calc(infinity * 1px)` = pill radius	—

```
clamp(2rem, 5vw + 1rem, 4rem)   ← keep a rem term, or zoom won't grow the text
round(down, 100% / 3, 1px)      ← fluid columns that still land on whole pixels
```

Not yet cross-browser: typed arithmetic (Chrome 140, Safari 26), `progress()`, `sibling-index()`, `calc-size()`, `if()`, `random()`.

Need the *reasoning*, not just the value? Start from the Decision Tree, or open any unit's full article above. New to a term? See the Glossary.

Sources

- CSS Values and Units Module Level 4 (w3.org)
- CSS Containment Module Level 3 (w3.org)
- CSS Grid Layout Module Level 1 (w3.org)

REFERENCE

Glossary

The words behind the numbers.

CSS units lean on a handful of underlying concepts. When a definition says "relative to the containing block" or "the advance measure of the glyph," these are the terms doing the work. Here they are, in plain language.

Containing block

The rectangle a box's percentages and offsets are resolved against. It is **not** simply "the parent" — it depends on the box's `position`:

- `static` / `relative` → the content box of the nearest **block-container** ancestor.
- `absolute` → the padding box of the nearest **positioned** ancestor.
- `fixed` → the viewport (or initial containing block) — *unless* an ancestor establishes one via `transform`, `perspective`, `filter`, `backdrop-filter`, `contain: layout|paint|strict|content`, `will-change` of a transform-like property, or `content-visibility`, in which case that ancestor's padding box is used instead. The same ancestor rule also applies to `absolute` boxes.

When you read "*`width: 50%` is relative to the containing block,*" this is the rectangle in question. See Percentage.

Initial containing block (ICB)

The root-level containing block, derived from the viewport in continuous media (the size of the viewport) and from the page area in paged media. Viewport units are ultimately defined against the ICB. See Viewport units.

Definite size

A size the browser can resolve **without first laying out the contents** — e.g. a pixel height, or a percentage whose reference is itself definite. Its opposite is an *indefinite* (often `auto`) size. This is the crux of the `height: 100%` trap: a percentage height needs a *definite* reference, so `height: 100%` inside an auto-height parent is treated as `auto`. See Percentage.

Inline axis / block axis

Writing-mode-aware directions. The **inline axis** is the direction text flows (horizontal in English); the **block axis** is the direction lines stack (vertical in English). Logical units and properties (`vi`/`vb`, `cqi`/`cqb`, `inline-size`, `margin-block`) are defined along these axes rather than physical width/height, so they adapt to vertical and RTL writing. See Viewport units.

Advance measure

The space a glyph occupies along the direction text runs, *including its side bearings* — i.e. how far the cursor advances after drawing it, not the visible ink width. It follows the writing direction, so in a vertical writing mode the advance measure is vertical. `ch` is the advance measure of `0`; `ic` is the advance measure of `水`. See Font-relative units.

x-height

The height of lowercase letters without ascenders or descenders — literally the height of a lowercase `x`. Drives the `ex` unit and strongly affects how large a typeface *looks* at a given `font-size`. See Font-relative units.

Cap height

The height of capital letters from the baseline. Drives the `cap` unit; useful for aligning icons to the top of uppercase text. Typically smaller than `1em`. See Font-relative units.

Reference pixel

The CSS definition of `px`: the visual angle of one pixel on a 96 DPI device viewed at arm's length. It decouples the CSS pixel from the physical device pixel, which is why `1px` can map to 2+ device pixels on HiDPI screens. See Absolute units.

Device pixel ratio (DPR)

The number of device pixels per CSS pixel, along one axis (`window.devicePixelRatio`). A "2× / Retina" display has a DPR of 2, so `1px` of CSS is drawn with 4 device pixels (2 × 2). It tracks page zoom as well as hardware — the spec measures the CSS pixel *at the current page zoom* — so the same screen reports 2 at 100% and 4 at 200%. Mirrored by the resolution units: `2dppx = 2x = 192dpi`. See Special units.

Query container

An ancestor element opted in with `container-type: inline-size` or `size` (the default `normal` is the opt-*out*), against which container query units (`cq*`) and `@container` rules resolve. Without an eligible query container, `cq*` units fall back

to the small viewport. See Container query units.

Computed value vs used value

The **computed value** is what the cascade resolves a property to before layout (e.g. `font-size: 1.5em` → a pixel length). The **used value** is the final value after layout, when any remaining dependencies (like percentage widths needing a definite container) are known. Font-relative units resolve against computed metrics; a user-agent feature like an enforced minimum font size can still adjust the size that's actually *used*. See Font-relative units.

Browser chrome

The browser's own UI around the page — address bar, toolbars, tab strip. On mobile it expands and retracts as you scroll, changing the visible area. This is the whole reason the small / large / dynamic viewport unit families exist. See Viewport units.

Flexible length

A length that represents a *proportion of available space* rather than a fixed measurement. `fr` is the only one in CSS; it resolves to leftover grid space, so it can't be used outside a track list. See The fr unit.

Compounding (cascade trap)

When a relative `font-size` like `em` (or `%`) is applied to nested elements, each level multiplies the previous one — `1.5em` five levels deep is $1.5^5 \approx 7.6\times$. `rem` avoids this by always referring to the root. See Font-relative units.

You'll meet these terms in context throughout the guide — start from any unit article, the Decision Tree, or the Cheatsheet.

Sources

- CSS Values and Units Module Level 4 (w3.org)
- CSS Display Module Level 3 — containing block (w3.org)
- CSS Containment Module Level 3 (w3.org)
- CSS Box Sizing Module Level 3 — definite and indefinite sizes (w3.org)

Colophon

Every CSS unit, explained — written and built by Mykola Ptushchuk. The web edition lives at **cssunits.com**, where every demo in this book runs live and the numbers are measured in your own browser.

Set in Newsreader and JetBrains Mono. Support claims are dated facts checked against browser compatibility data, not adjectives — where a claim decays quickly it carries the date it was last reviewed.

This printing: August 2026. The interactive demos are the one thing this edition cannot carry — where a chapter asks you to drag something or toggle it, that is what the web edition is for.