inline-size

block-size

# EVERY LAYOUT

## Relearn CSS layout by example

**3rd EDITION**

NOW WITH LOGIC(AL PROPERTIES)!

# Heydon Pickering & Andy Bell

**FREESOFF**

# Preface

This is the ebook version of every-layout.dev ↗ and contains all of the same content. We made it for you because we thought you might like to read about CSS layout in this format, and offline.

An EPUB book cannot do or show all the things a website can, and the interactive demos are replaced with links back to the every-layout.dev ↗ website. To see all of these pages and demos, you have to purchase the full version of **Every Layout**. If you are looking at this book, you have hopefully done that already. After purchase, a link to the full, unlocked site will accompany the link to this book in an email.

All external links in this book are marked with a ↗ symbol/character. If you see a link not suffixed with ↗, it points to a section within the book itself. The book has been tested to render and behave as expected in the latest versions of Apple's iBooks, and the Calibre e-book manager ↗.

## Editions

# Third edition *(current)*

This edition introduced *logical properties* for better compatibility with different languages and their writing modes. It also updated the **Frame** component to use the `aspect-ratio` property, which is now widely supported.

# Second edition

This edition converted a number of layouts to use the `gap` property which has come to be widely supported with Flexbox as well as Grid. Using `gap` simplifies many layouts and makes them easier to understand.

# First edition

We actually added a lot more content after the initial release but we hadn't started recording "editions" so all of those updates are implicitly part of the first edition.

# Ownership

When you purchase a licence for **Every Layout**, you own a license to the content that is authored and owned by Heydon Pickering and Andy Bell.

# Fair usage and redistribution

Re-publishing and re-selling of **Every Layout** is strictly forbidden and discovered instances will be pursued, legally, in accordance with United Kingdom copyright law.

We expect licence holders to use their licence in a fair manner. We put a lot of trust in our licence holders so that we can make using **Every Layout** as frictionless as possible. We believe that you, the licence holder, should be able to access the content that you paid for with little to no barriers, but this also means that the licence is easily shared.

If we suspect you are not using your license in a fair manner or sharing it irresponsibly, we reserve the right to revoke your access to **Every Layout** with no refunds, after a fair warning.
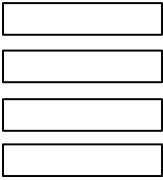
# RUDIMENTS
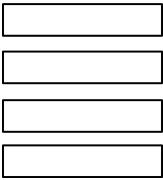
**Boxes**
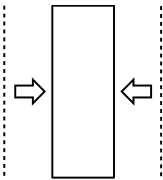
**Composition**

**Units**

**Global and local styling**

**Modular scale**

**Axioms**

# LAYOUTS

**The Stack**

**The Box**

**The Center**

**The Cluster**
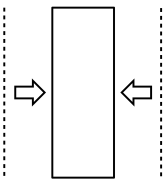
**The Sidebar**
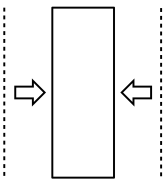
**The Switcher**
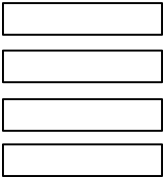
**The Cover**
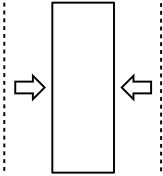
**The Grid**
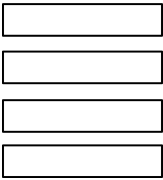
**The Frame**

**The Reel**

**The Imposter**

**The Icon**

# Boxes

As Rachel Andrew ↗ has reminded us, everything in web design is a box ↗, or the absence of a box. Not everything necessarily *looks* like a box—`border-radius`, `clip-path`, and `transforms` can be deceptive, but everything takes up a box-like space. Layout is inevitably, therefore, the arrangement of boxes.

Before one can embark on combining boxes to make composite layouts, it is important to be familiar with how boxes themselves are designed to behave as standard.

## The box model

The box model ↗ is the formula upon which layout boxes are based, and comprises content, padding, border, and margin. CSS lets us alter these values to change the overall size and shape of elements' display.
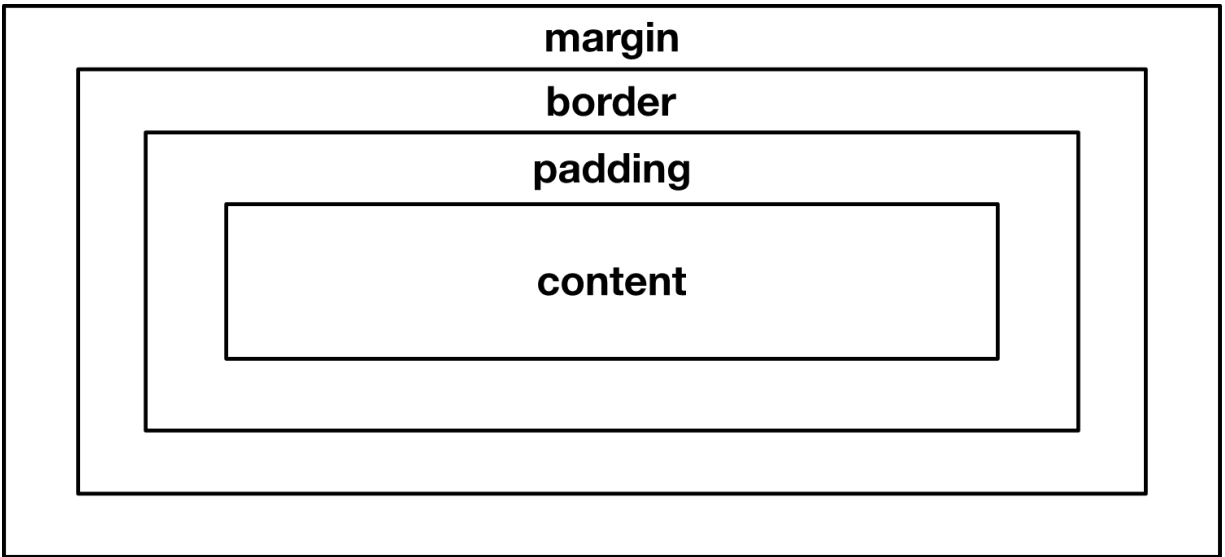
Web browsers helpfully apply default CSS styles to some elements, meaning they are laid out in a reasonably readable fashion: even where author CSS has not been applied.

In Chrome, the default user agent styles for paragraphs (`<p>`) look like…

```
p {
  display: block;
  margin-block-start: 1em;
  margin-block-end: 1em;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
}
```

… and unordered list (`<ul>`) styles look like…

```
ul {
  display: block;
  list-style-type: disc;
  margin-block-start: 1em;
  margin-block-end: 1em;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
  padding-inline-start: 40px;
}
```

# The `display` property

In both the above examples, the element's `display` property is set to `block`. Block elements assume all of the available space in one dimension. Typically, this is the horizontal dimension, because the `writing-mode` is set to `horizontal-tb` (horizontal; with a top to bottom flow direction). In some cases, and for some languages (like Mongolian ↗), `vertical-lr` is the appropriate writing mode.
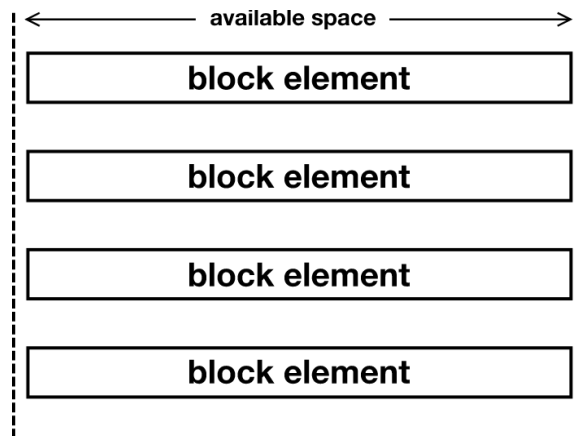
writing-mode: horizontal-tb
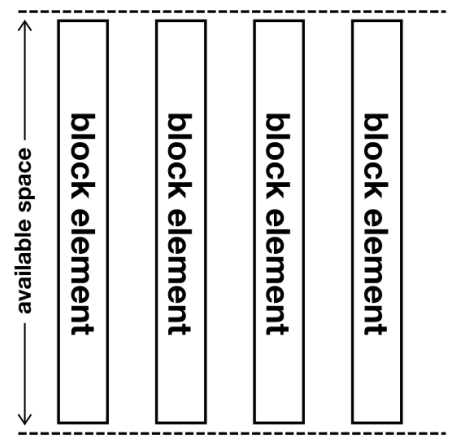
writing-mode: vertical-lr

Inline elements (with the `display` value `inline`) behave differently. They are laid out *in line* with the current context, writing mode, and

direction. They are only as wide as their content, and are placed adjacently wherever there is space to do so. Block elements follow flow direction, and inline elements follow writing direction.

writing direction ⟶

flow direction
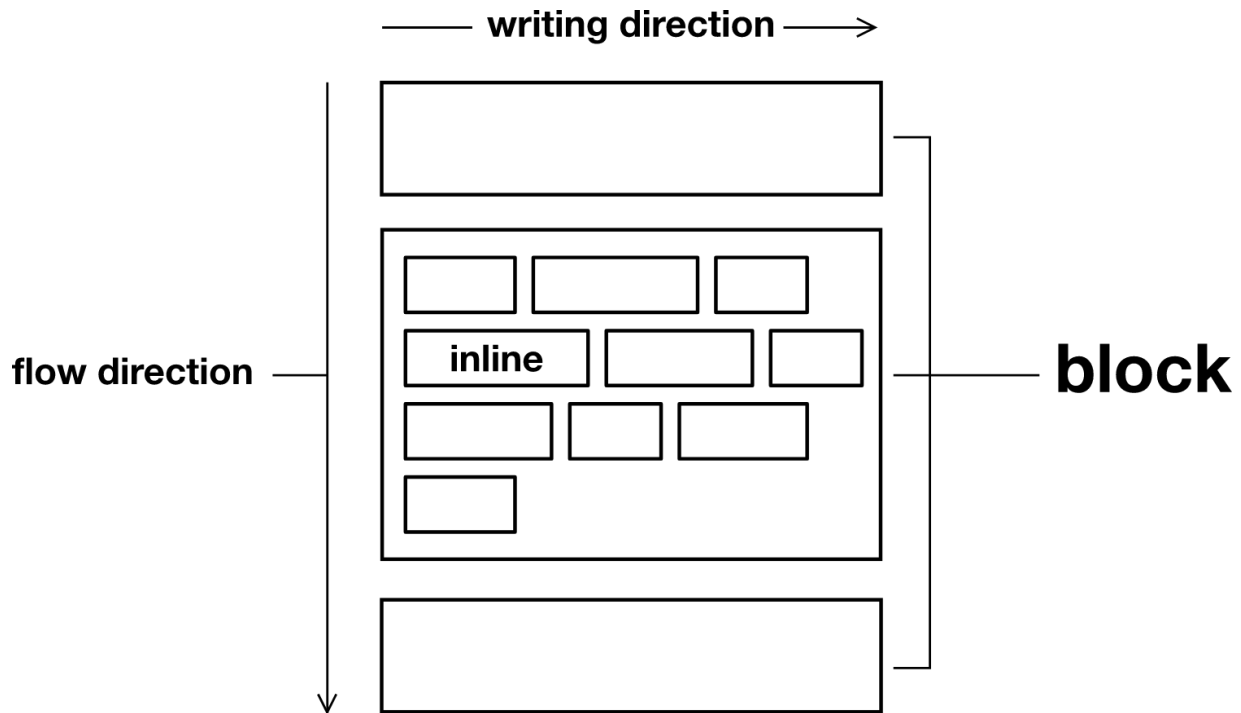
inline

block

Thinking typographically, it could be said that block elements are like paragraphs, and inline elements are like words.

Block elements (also called <u>block-level ↗</u> elements) afford you control over both the horizontal and vertical dimensions of the box. That is, you can apply width, height, margin, and padding to a block element and it will take effect. On the other hand, inline elements are sized *intrinsically* (prescribed `width` and `height` values do not take effect) and only *horizontal* margin and padding values are permitted. Inline elements are designed to conform to the flow of horizontal placement among other inline elements.

A relatively new display property, `inline-block`, is a hybrid of `block` and `inline`. You *can* set vertical properties on `inline-block` elements, although this is not always desirable—as the proceeding illustration demonstrates.

**inline**

**inline-block**
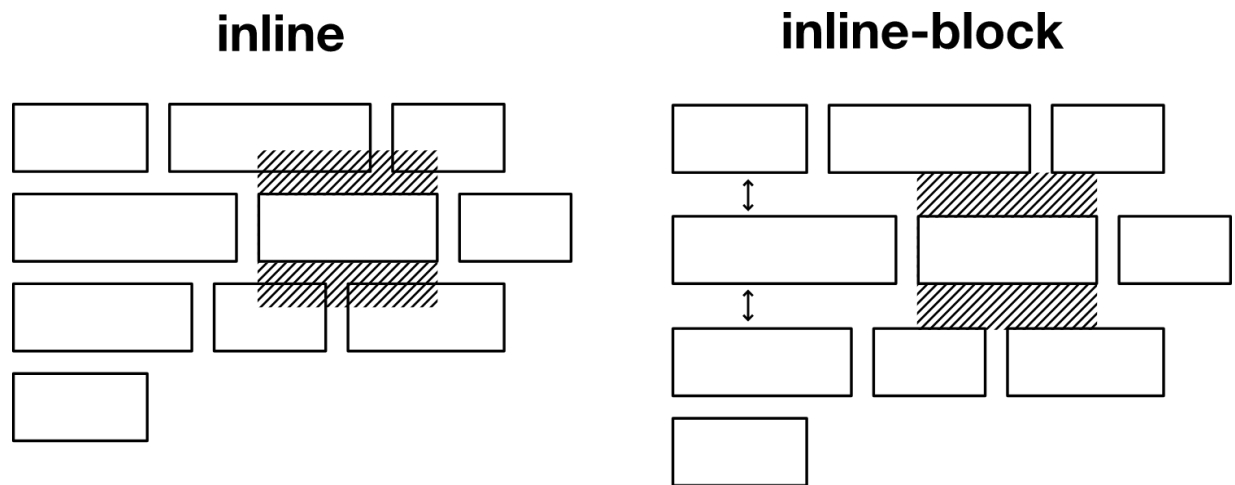
Of the basic `display` types, only `none` remains. This value removes the element from the layout entirely. It has no visual presence, and no impact on the layout of surrounding elements. It is as if the element itself has been removed from the HTML. Accordingly, browsers do not communicate the presence or content of `display: none` elements to assistive technologies like underline screen reader software ↗.

# Logical properties

What are logical properties ↗ and does their existence imply the existence of *illogical* properties? English speakers accustomed to reading left to right (`direction: ltr`) and top to bottom (`writing-mode:`

`horizontal-tb`) find it logical to use properties that include the words "left", "right", "top", and "bottom" when applying styles like margin and padding.

```css
.icon {
  margin-right: 0.5em;
}
```

It's when the direction or writing mode changes that this becomes illogical, because left and right (and/or top and bottom) are flipped. Now the margin you put on the right you really need on the left.



Logical properties eschew terminology like "left" and "right" because we know they can be reversed, making the terms a nonsense. Instead, we apply styles like margin, padding, and border according to the block and inline direction.

```css
.icon {
  margin-inline-end: 0.5em;
}
```

In a `ltr` direction, `margin-inline-end` applies margin to the right. In a `rtl` direction, `margin-inline-end` applies margin to the left. In both cases, it is applied where it is needed: at the end of the inline dimension.

direction: ltr

direction: rtl

# Formatting contexts

When you apply `display: flex` or `display: grid` to a `<div>`, it continues to behave like a block element, using `display: block`. However, it changes the way its *child* elements behave. For example, with just `display: flex` (and no other Flexbox-related properties) applied to the parent, its children will distribute themselves horizontally. Or, to put it another way, the *flow direction* is switched from vertical to horizontal.

Formatting contexts are the basis of many of the layouts documented in this project. They turn elements into layout components. In **Composition**, we'll explore how different formatting contexts can be nested, to create *composite* layouts.

# Content in boxes

The web is a conduit for primarily textual information supplemented by media such as images and videos, often referred to collectively as *content*. Browsers incorporate line wrapping and scrolling algorithms to make sure content is transmitted to the user in its entirety, irrespective of their screen sizes and dimensions, and settings such as zoom level. The web is *responsive* ↗ largely by default.

Without intervention, it is the contents of an element that determines its size and shape. Content makes `inline` elements grow horizontally, and `block` elements grow vertically. Left to its own devices, the *area* of a box is determined by the area of the content it contains. Because web content is *dynamic* (subject to change), static representations of web layouts are extremely misleading. Working directly with CSS and its flexibility from the outset, as we are here, is highly recommended.
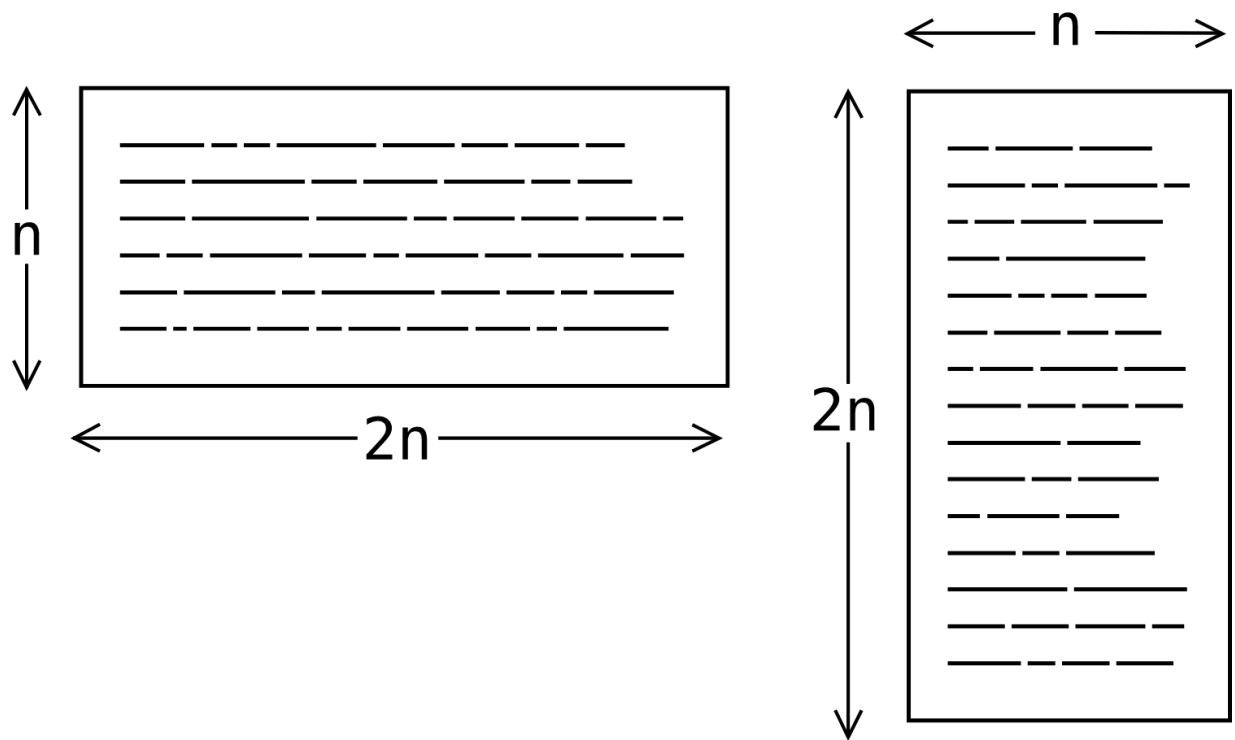
*If you halve the width of an element, it will have to be twice as tall to contain the same amount of content*

## The `box-sizing` property

By default, the dimensions of a box are the dimensions of the box's content *plus* its padding and border values (implicitly: `box-sizing: content-box`). That is, if you set an element to be `10rem` wide, then add padding on both sides of `1rem`, it will be `12rem` wide: `10rem` plus `1rem` of left padding and `1rem` of right padding. If you opt for `box-sizing: border-box`, the content area is reduced to accommodate the padding and the total width equals the prescribed `width` of `10rem`.

```
box-sizing: content-box          box-sizing: border-box
```

Generally, it is considered preferable to use the `border-box` model for all boxes. It makes calculating/anticipating box dimensions easier.

Any styles, like `box-sizing: border-box`, that are applicable to all elements are best applied using the `∗` ("universal" or "wildcard") selector. As covered in detail in **Global and local styling**, being able to affect the layout of multiple elements (in this case, *all* elements) simultaneously is how CSS brings efficiency to layout design.

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

# Exceptions

There are exceptions to the `border-box` rule-of-thumb, such as in the **Center** layout where measurement of the *content* is critical. CSS's cascade ↗ is designed to accommodate exceptions to general rules.

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

center-l {
  box-sizing: content-box;
}
```

Only where the height or width of a box is constrained does the difference between `content-box` and `border-box` come into play. For illustration, consider a block element placed inside another block element. Using the `content-box` model and a padding of `1rem`, the child element will overflow by `2rem` when `inline-size: 100%` (equivalent to `width: 100%` in a `horizontal-tb` writing mode) is applied.
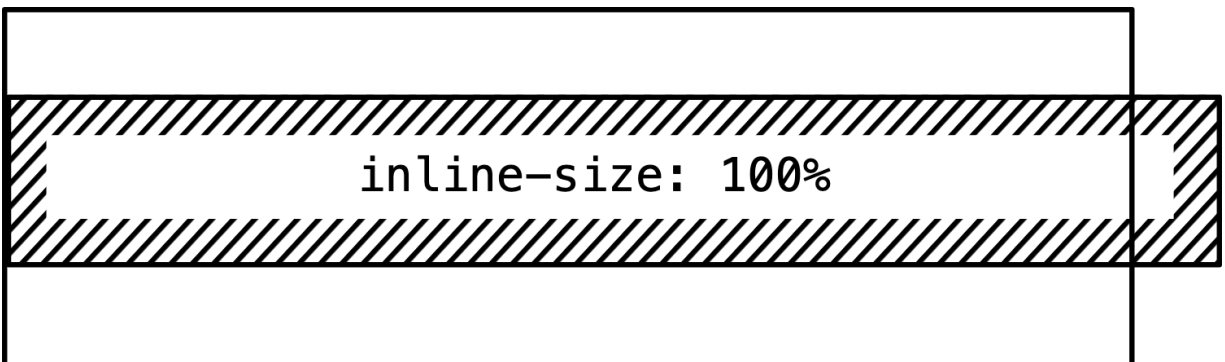
inline-size: 100%

Why? Because `inline-size: 100%` means *"make the width of this element the same as the parent element"*. Since we are using the `content-box` model, the *content* is made `100%` wide, then the padding is added on to this value.

But if we use `inline-size: auto` (we can just remove `inline-size: 100%`, since `auto` is the default value) the child box fits within the parent box perfectly. And that's *regardless* of the `box-sizing` value.

```
┌─────────────────────────────────────────────┐
│  ///////////////////////////////////////////│//
│  ///                                      ///│//
│        inline-size: 100%                     │
│  ///                                      ///│//
│  ///////////////////////////////////////////│//
│                                              │
└─────────────────────────────────────────────┘
```
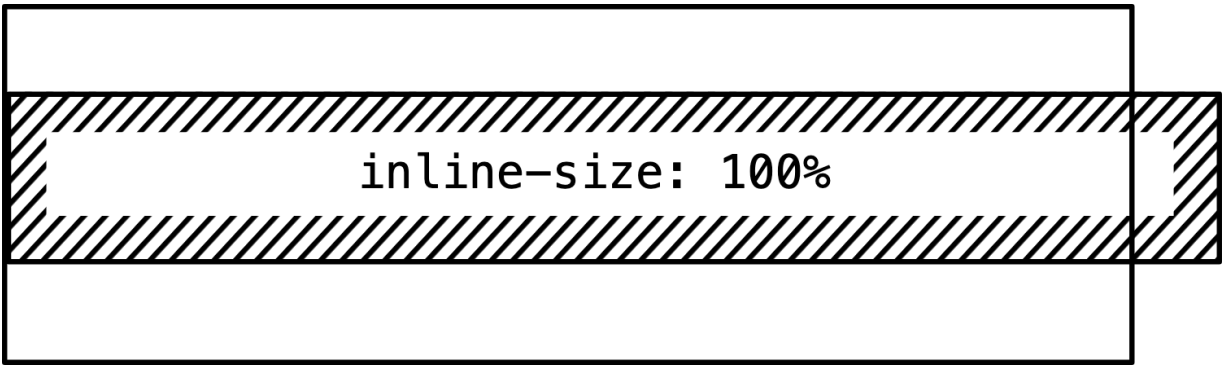
Implicitly, the `height` is also set to `auto`, meaning it is derived from the content. Again, `box-sizing` has no effect.

The lesson here is the dimensions of our elements should be largely *derived* from their inner content and outer context. When we try to *prescribe* dimensions, things tend to go amiss. All we should be doing as visual designers is making *suggestions* as to how the layout should take shape. We might, for instance, apply a `min-height` (as in the **Cover** layout) or proffer a `flex-basis` (as in the **Sidebar**).

The CSS of suggestion is at the heart of algorithmic layout design. Instead of telling browsers what to do, we allow browsers to make their own calculations, and draw their own conclusions, to best suit the user, their screen, and device. Nobody should experience obscured content under any circumstances.

# Composition

If you are a programmer, you may have heard of the composition over inheritance ↗ principle. The idea is that combining simple independent parts (objects; classes; functions) gives you more flexibility, and leads to more efficiency, than connecting everything—through inheritance—to a shared origin.

*Composition over inheritance* does not have to apply to "business logic". It is also beneficial to favor composition in front-end architecture and visual design (the React documentation even has a dedicated page about it ↗).

## Composition and layout

To understand how composition benefits a *layout system*, let's consider an example component. Let's say that this component is a dialog box, because the interface (for reasons we won't get into right now) *requires* a dialog box. Here is what it looks like:

## Are you sure you want to include a dialog in your pattern library?

**Okay**    **Cancel**
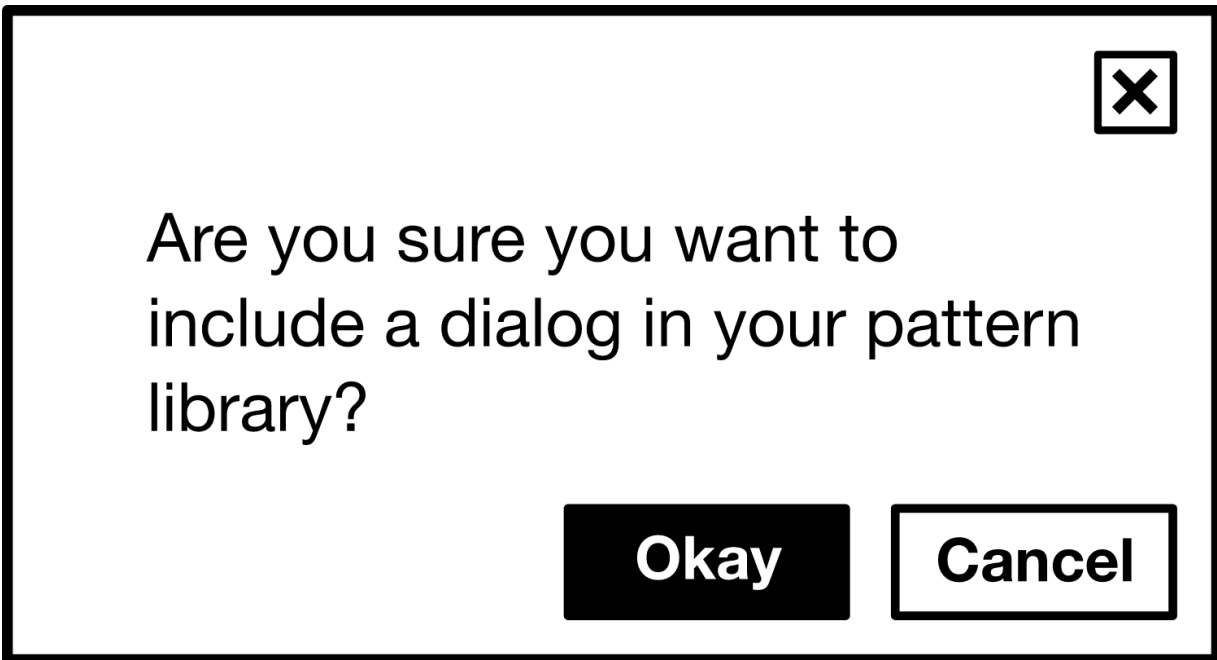
But how does it get to look like that? One way is to write some dedicated dialog CSS. You might give the dialog box a "block" identifier (`.dialog` in CSS, and `class="dialog"` in HTML) and use this as your namespace to attach style declarations.

```
.dialog {
  /* ... */
}

.dialog__header {
  /* ... */
}

.dialog__body {
  /* ... */
}

.dialog__foot {
  /* ... */
}
```

Alternatively, these dialog styles might be imported from a third-party CSS library/framework. In either case, a lot of the CSS used to make the dialog *look* like a dialog, could be used to make other, similar layouts. But since everything here is namespaced under `.dialog`, when we come to make the next component, we'll end up duplicating would-be shared styles. This is where most CSS bloat comes from.

The `namespacing` part is key here. The inheritance mindset encourages us to think about what finalized parts of UI should be called before we've even decided what they *do*, or what other, smaller parts can *do for them*. That's where composition comes in.

# Layout primitives

The mistake in the last example was to think of everything about the dialog's form as isolated and unique when, really, it's just a composition of simpler layouts. The purpose of **Every Layout** is to identify and document what each of these smaller layouts are. Together, we call them *primitives*.

The term primitive has linguistic, mathematical, and computing connotations. In each case, a primitive is something without its own meaning or purpose as such, but which can be used *in composition* to make something meaningful, or *lexical*. In language it might be a word or phrase, in mathematics an equation, in design a pattern, or in development a component.

In JavaScript, the Boolean data type is a primitive. Just looking at the value `true` (or `false`) out of context tells you very little about the

larger JavaScript application. The object data type, on the other hand, is *not* primitive. You cannot write an object without designating your own properties. Objects are therefore meaningful; they necessarily tell you something of the author's intent.

The dialog is meaningful, as a piece of UI, but its constituent parts are not. Here's how we might compose the dialog box using **Every Layout's** layout primitives:

**Center**

**Stack** — Are you sure you want to include a dialog in your pattern library?

**Cluster**

⊠

**Okay** **Cancel**

**Box**
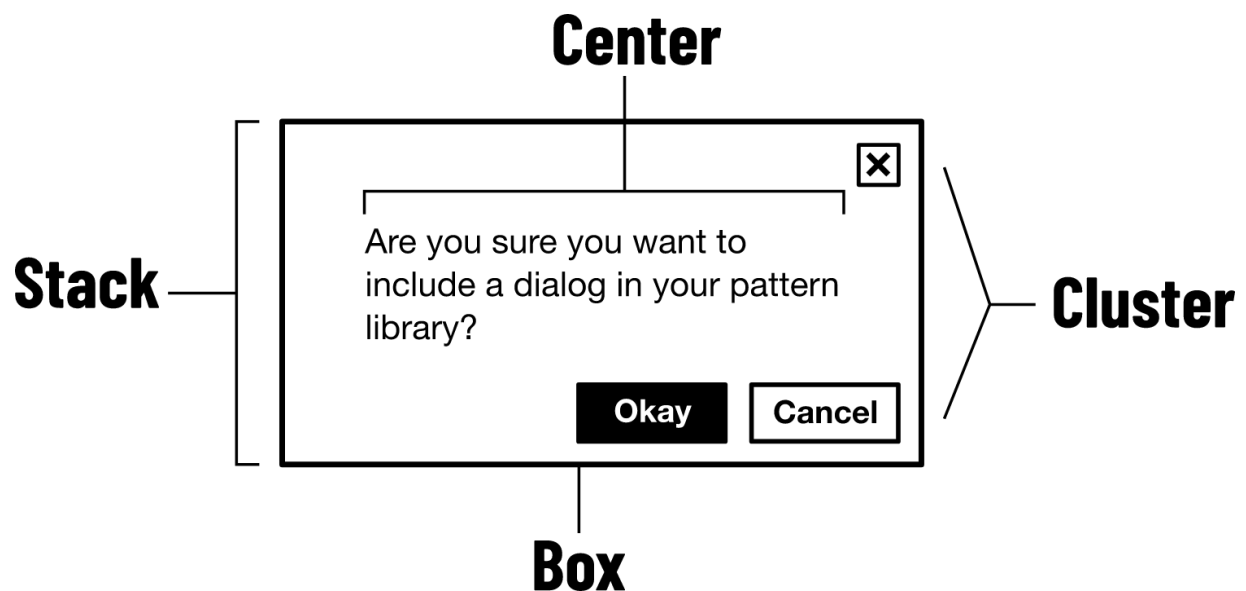
Using many of the same primitives, we can create a registration form…

Email

Password

Confirm password

Submit

(nested)
**Stack**

**Stack**

**Cluster**

**Box**
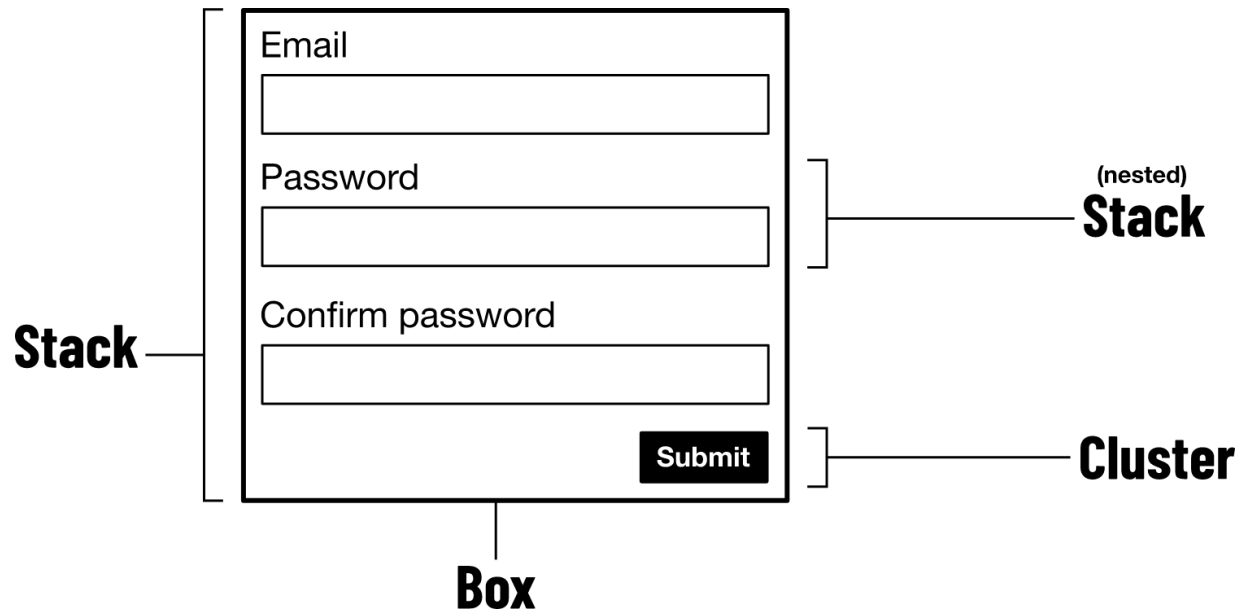
… or a slide layout for a conference talk:



**Box**

**Stack**

Edit

Delete

Share

# Creating custom dialogs

< >

**Cover**

**Sidebar**
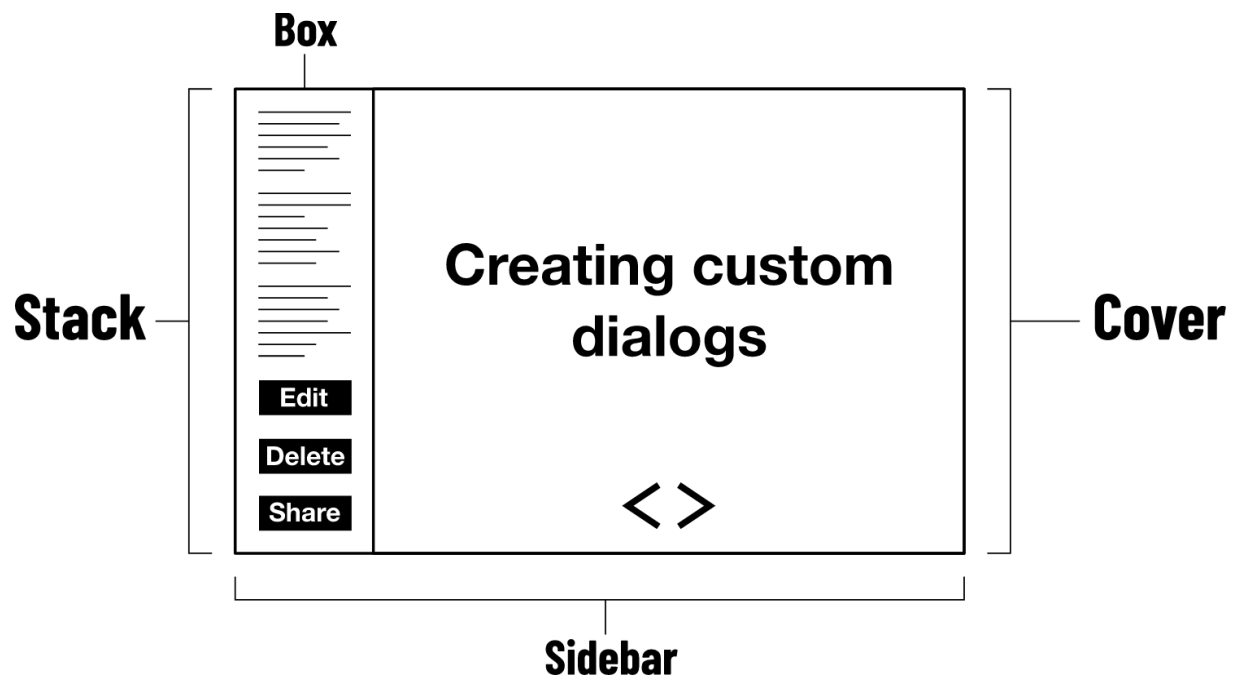
# Intrinsically responsive

Each layout in **Every Layout** is intrinsically responsive. That is, it will wrap and reconfigure internally to make sure the content is visible (and well-spaced) to fit any context/screen.

You may feel compelled to add `@media` query breakpoints, but these are considered "manual overrides" and **Every Layout** primitives do not depend on them.

Without primitive data types, you would have to be constantly teaching your programming language how to do basic operations. You would quickly lose sight of the specific, *meaningful* task you set out to accomplish with the language in the first place. A design system that does not leverage primitives is similarly problematic. If every component in your pattern library follows its own rules for layout, inefficiencies and inconsistencies will abound.

The primitives each have a simple responsibility: "*space elements vertically*", "*pad elements evenly*", "*separate elements horizontally*", etc. They are designed to be used in composition, as parents, children, or siblings of one another.

You probably cannot create *literally* every layout using **Every Layout's** primitives alone. But you can certainly make most, if not all, common web layouts, and achieve many of your own unique conceptions.

In any case, you should walk away with an understanding and appreciation for the benefits of composition, and the power to create all sorts of interfaces with just a little reusable code. The English alphabet is only 26 bytes, and think of all the great works created with that!

# Units

Everything you see on the web is composed out of the little dots of light that make up your device's screen: *pixels*. So, when measuring out the artefacts that make up our interfaces, thinking in terms of pixels, and using the CSS `px` unit, makes sense. Or does it?

Screens' pixel geometries vary wildly ↗, and most modern displays employ sub-pixel rendering, which is the manipulation of the color components of individual pixels, to smooth jagged edges and produce a higher *perceived* resolution. The notion of `1px` is fuzzier than how it's often portrayed.
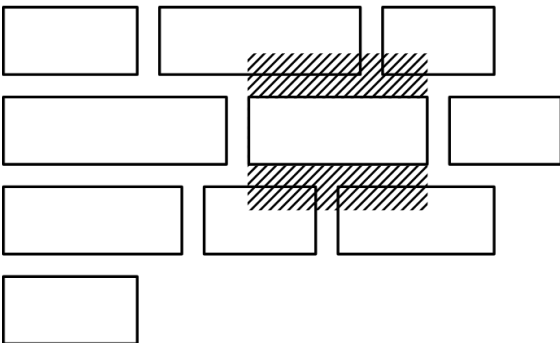
**inline**                    **inline-block**
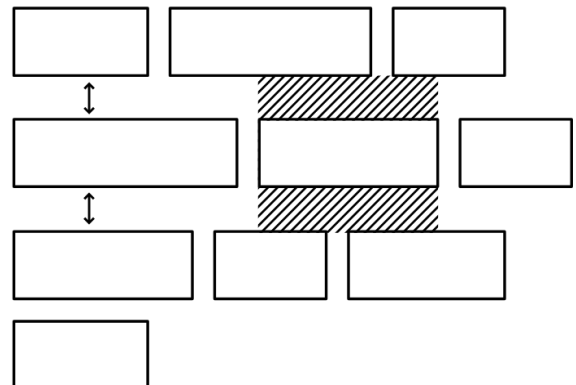
*The Samsung Galaxy Tab S 10.5 alternates the arrangement of subpixels between pixels. Every other pixel is composed differently.*

Screen resolutions—how many pixels screens pack—also differ. Consequently, while one "CSS pixel" (`1px` in CSS) may approximate one "device" or "hardware" pixel on a lower resolution screen, a high resolution screen may proffer *multiple* device pixels for each `1px` of CSS. So there are pixels, and then there are pixels *of* pixels.

**Pixel** ⌐ [ grid ] ● ⎯ **Pixel**
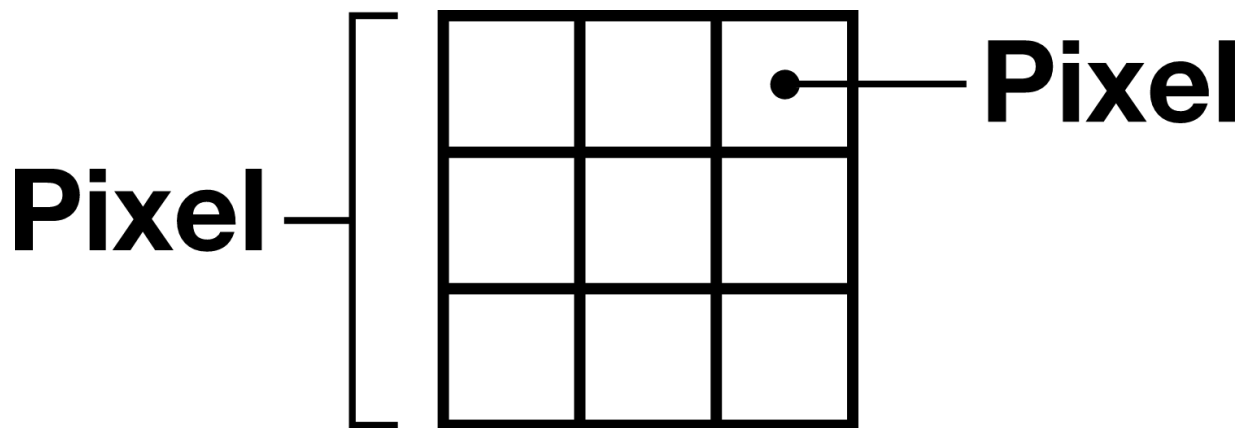
Suffice it to say that, while screens are indeed made up of pixels, pixels are not regular, immutable, or constant. A `400px` box viewed by a user browsing *zoomed in* is simply not `400px` in CSS pixels. It may not have been `400px` in *device pixels* even before they activated zoom.

Working with the `px` unit in CSS is not *incorrect* as such; you won't see any error messages. But it encourages us to labour under a false premise: that pixel perfection ↗ is both attainable and desirable.

# Scaling and accessibility

Designing using the `px` unit doesn't only encourage us to adopt the wrong mindset: there are manifest limitations as well. For one, when you set your fonts using `px`, browsers assume you want to *fix* the fonts at that size. Accordingly, the font size chosen by the user in their browser settings is disregarded.

With modern browsers now supporting <u>full page zoom ↗</u> (where everything, including text is zoomed proportionately), this is often blown off as a solved problem. However, as <u>Evan Minto discovered ↗</u>, there are more users who adjust their default font size in browser settings than there are users of the browsers Edge or Internet Explorer. That is: disregarding users who adjust their default font size is as impactful as disregarding whole browsers.

The units `em`, `rem`, `ch`, and `ex` present no such problem because they are all units *relative* to the user's default font size, as set in their operating system and/or browser. Browsers translate values using these units into pixels, of course, but in such a way that's sensitive to context and configuration. Relative units are arbitrators.

# Relativity

Browsers and operating systems typically only afford users the ability to adapt the *base* or *body* font size. This can be expressed as `1rem`: exactly one times the root font size. Your paragraph elements

should always be `1rem`, because they represent body text. You don't need to set `1rem` explicitly, because it's the default value.

```css
:root {
  /* ↓ redundant */
  font-size: 1rem;
}

p {
  /* ↓ also redundant */
  font-size: 1rem;
}
```

Elements, like headings, should be set *relatively* larger — otherwise hierarchy will be lost. My `<h2>` might be `2.5rem`, for example.

```css
h2 {
  /* ↓ 2.5 × the root font-size */
  font-size: 2.5rem;
}
```

While the units `em`, `rem`, `ch`, and `ex` are all measurements of text, they can of course be applied to the `margin`, `padding`, and `border` properties (among others). It's just that text is the basis of the web medium, and these units are a convenient and constant reminder of this fact. Learn to extrapolate your layouts from your text's intrinsic dimensions and your designs will be beautiful.

## ⚠ Needless conversion

A lot of folks busy themselves converting between `rem` and `px`, making sure each `rem` value they use equates to a whole pixel value. For example, if the base size is `16px`, `2.4375rem` would be `39px`, but `2.43rem` would be `38.88px`.

There's no need to do precise conversion, since browsers employ sub-pixel rendering and/or rounding to even things out automatically. It's less verbose to use simple fractions like `1.25rem`, `1.5rem`, `1.75rem`, etc — or to let `calc()` do the heavy lifting in your **Modular scale**.

## Proportionality and maintainability

My `<h2>` is 2.5 times the root/base size. If I enlarge the root size, my `<h2>`—and all the other dimensions set in `rem`-based multiples—will by enlarged *proportionately*. The upshot is that scaling the entire interface is trivial:

```
@media (min-width: 960px) {
  :root {
    /* ↓ Upscale by 25% at 960px */
    font-size: 125%;
  }
}
```

If I had instead adopted `px`, the implications for maintenance would be clear: the lack of relative and proportional sizing would require adjusting individual elements case-by-case.

```
h3 {
  font-size: 32px;
}
h2 {
  font-size: 40px;
}

@media (min-width: 960px) {
  h3 {
    font-size: 40px;
  }
  h2 {
    font-size: 48px;
  }
  /* etc etc ad nauseum */
}
```

# Viewport units

In **Every Layout**, we eschew width-based `@media` queries. They represent the hard coding of layout reconfigurations, and are not sensitive to the immediate available space actually afforded the element or component in question. Scaling the interface at a discrete *breakpoint*, as in the last example, is arbitrary. What's so special about `960px`? Can we really say the smaller size is acceptable at `959px`?
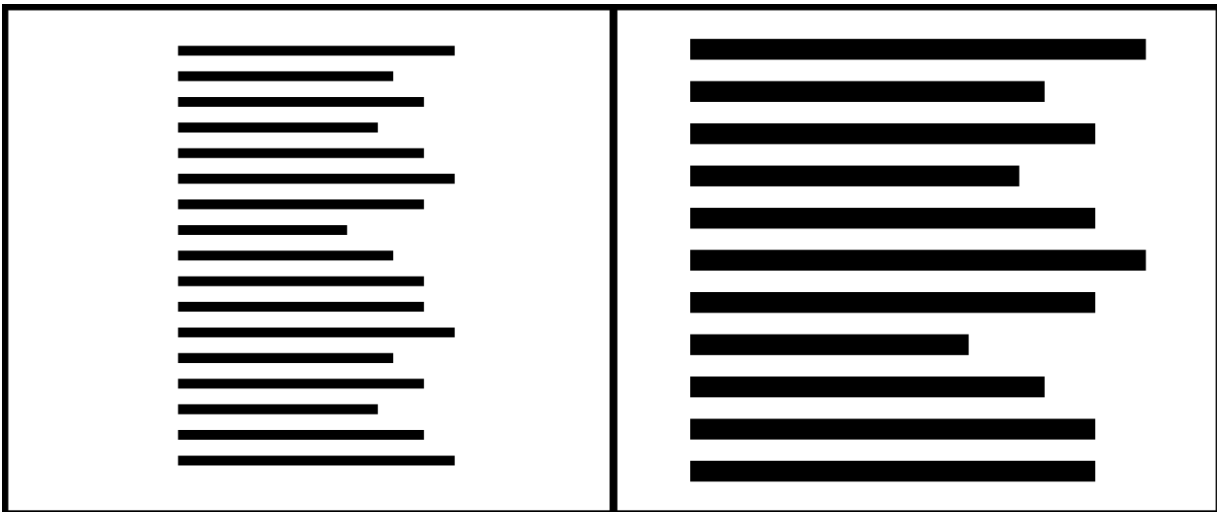
**959px**   **960px**

*A `1px` disparity represents a significant jump when using a breakpoint.*

Viewport units ↗ are relative to the browser viewport's size. For example, `1vw` is equal to 1% of the screen's width, and `1vh` is equal to 1% of the screen's height. Using viewport units and `calc()` we can create an algorithm whereby dimensions are scaled proportionately, but from a *minimum* value.

```
:root {
  font-size: calc(1rem + 0.5vw);
}
```

The `1rem` part of the equation ensures the `font-size` never drops *below* the default (system/browser/user defined) value. That is, `1rem + 0vw` is `1rem`.

# The em unit

The `em` unit is to the `rem` unit what a <u>container query ↗</u> is to a `@media` query. It pertains to the immediate context rather than the outer document. If I wanted to slightly enlarge a `<strong>` element's `font-size` within my `<h2>`, I could use `em` units:

```
h2 {
  font-size: 2.5rem;
}

h2 strong {
  font-size: 1.125em;
}
```

The `<strong>`'s `font-size` is now `1.125 × 2.5rem`, or `2.53125rem`. If I set a `rem` value for the `<strong>` instead, it wouldn't scale with its parent `<h2>`: if I changed the `h2` value, I would have to change the `h2 strong` CSS value as well.

As a rule of thumb, `em` units are better for sizing inline elements, and `rem` units are better for block elements. <u>SVG icons are perfect candidates ↗</u> for `em`-based sizing, since they either accompany or supplant text.

0.75em ⎯⎯[ ⬇ **Download Stack.zip**

*The actual value, in `ems`, of the icon height/width must be adapted to the accompanying font's own metrics, in some cases. The Barlow Condensed font used on this site has a lot*

## The `ch` and `ex` units

The `ch` and `ex` units pertain to the (approximate) width and height of one character respectively. `1ch` is based on the width of a `0`, and `1ex` is equal to the *height* of your font's `x` character—also known as the x height or corpus size ↗.

0 X 1ex
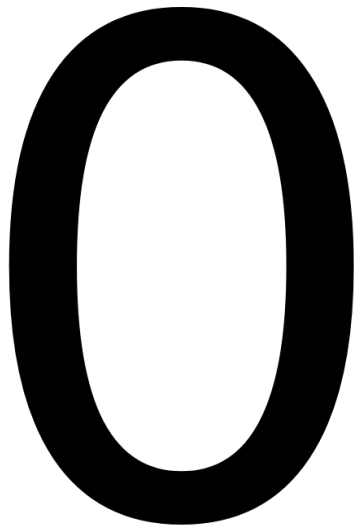
←1ch→
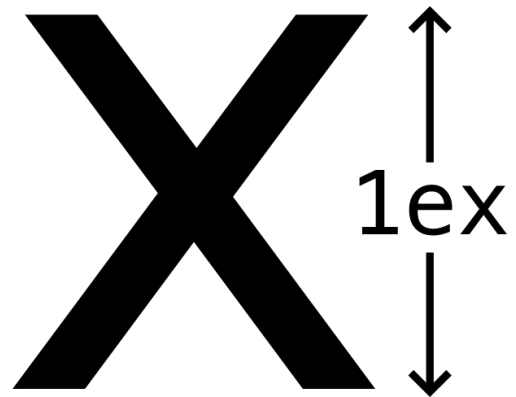
In the **Axioms** section, the `ch` unit is used to restrict elements' measure ↗ for readability. Since measure is a question of characters per line, `ch` (short for *character*) is the only appropriate unit for this styling task.

An `<h2>` and an `<h3>` can have different `font-size` values, but the same (maximum) measure.

```
h2,
h3 {
  max-inline-size: 60ch;
}

h3 {
  font-size: 2rem;
}
h2 {
  font-size: 2.5rem;
}
```

The width, in pixels, of one full line of text is *extrapolated* from the relationship between the `rem`-based `font-size` and `ch`-based `max-width`. By delegating an algorithm to determine this value—rather than hard coding it as a `px`-based `width`—we avoid frequent and serious error. In CSS layout terms, an error is malformed or obscured content: *data loss for human beings*.

# Global and local styling

In the **Composition** section we covered how small, *nonlexical* components for layout can be used to create larger composites, but not all styles within an efficient and consistent CSS-based design system should be strictly component based. This section will contextualize layout components in a larger system that includes global styles.

## What are global styles?

When people talk about the *global* nature of CSS, they can mean one of a few different things. They may be referring to rules on the `:root` or `<body>` elements that are *inherited* globally (with just a few exceptions).

```css
:root {
  /* ↓ Now (almost) all elements display a sans-serif font */
  font-family: sans-serif;
}
```

Alternatively, they may mean using the unqualified ∗ selector ↗ to style all elements *directly*.

```
* {
  /* ↓ Now literally all elements display a sans-serif font */
  font-family: sans-serif;
}
```

Element selectors are more specific, and only target the elements they name. But they are still "global" because they can *reach* those elements wherever they are situated.

```
p {
  /* ↓ Wherever you put a paragraph, it'll be sans-serif */
  font-family: sans-serif;
}
```

A liberal use of element selectors is the hallmark of a comprehensive design system. Element selectors take care of generic atoms ↗ such as headings, paragraphs, links, and buttons. Unlike when using classes (see below), element selectors can target the arbitrary, unattributed content produced by WYSIWYG editors ↗ and markdown ↗.

The layouts of **Every Layout** do not explore or prescribe styles for simple elements; that is for you to decide. It is the imbrication of simple elements into composite layouts that we are interested in here.

# layout



```
┌─────────────────────────────────────────────────┐
│  ┌───────────┐  ┌───────────┐  ┌───────────┐   │
│  │           │  │           │  │           │   │
│  │ element or│  │ element or│  │ element or│   │
│  │  layout   │  │  layout   │  │  layout   │   │
│  │           │  │           │  │           │   │
│  └───────────┘  └───────────┘  └───────────┘   │
└─────────────────────────────────────────────────┘
```
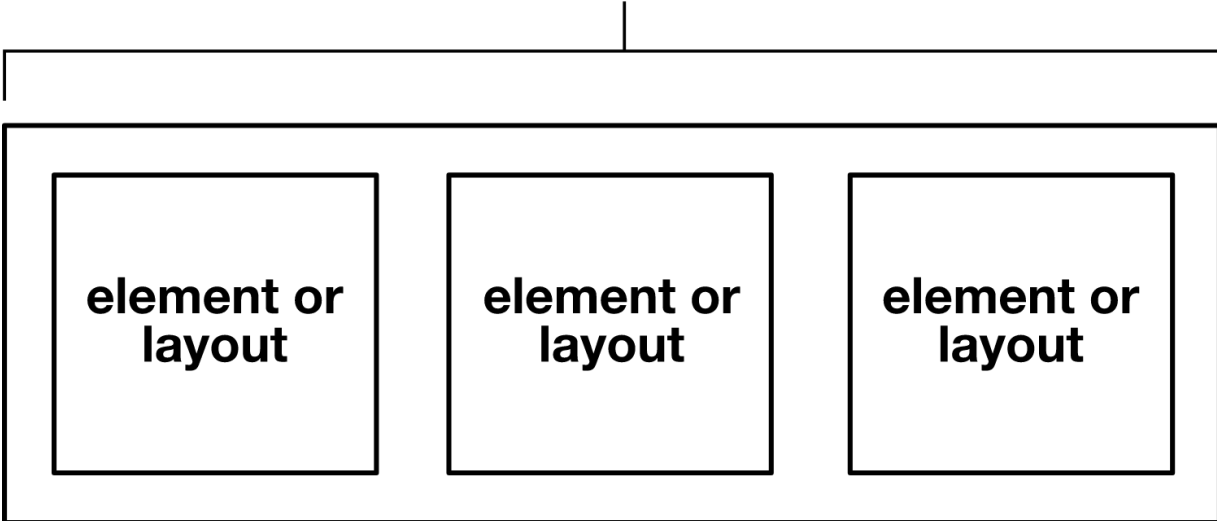
*Each layout requires a container element which establishes a formatting context for its children. Simple elements, without children for which they establish a context, can be thought of as 'end nodes' in the layout hierarchy.*

Finally, class-based styles, once defined, can be adhered to any HTML element, anywhere in a document. These are more portable and composable than element styles, but require the author to affect the markup directly.

```css
.sans-serif {
  font-family: sans-serif;
}
```

```html
<div class="sans-serif">...</div>

<small class="sans-serif">...</small>

<h2 class="sans-serif">...</h2>
```

It should be appreciated how important it is to leverage the global reach of CSS rules. CSS itself *exists* to enable the styling of HTML globally, and by category, rather than element-by-element. When used as intended, it is the most efficient way to create any kind of layout or aesthetic on the web. Where global styling techniques (such as the ones above) are used appropriately, it's much easier to separate branding/aesthetic from layout, and treat the two as separate concerns ↗.



**fonts, colors, shadows, etc**

**arrangement of boxes**
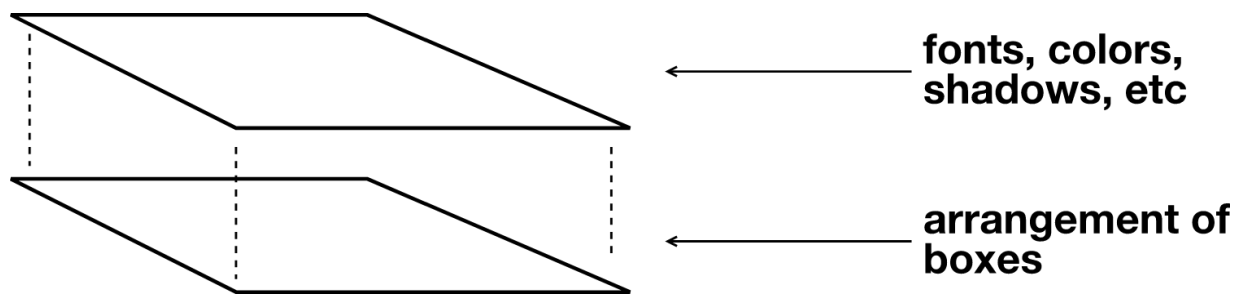
# Utility classes

As we already stated, classes differ from the other global styling methods in terms of their portability: you can use classes between different HTML elements and their types. This allows us to *diverge* from inherited, universal, and element styles *globally*.

For example, all of our `<h2>` elements may be styled, by default, with a `2.25rem font-size`:

```
h2 {
  font-size: 2.25rem;
}

h3 {
  font-size: 1.75rem;
}
```

However, there may be a specific cases where we want that `font-size` to be diminished slightly (perhaps horizontal space is at a premium, or the heading is somewhere where it should have less visual affordance). If we were to switch to an `<h3>` element to affect this visual change, we would make a nonsense of the document structure ↗.

Instead, we could build a more complex selector pertaining to the smaller `<h2>`'s context:

```
.sidebar h2 {
  font-size: 1.75rem;
}
```

While this is better than messing up the document structure, I've made the mistake of not taking the *whole* emerging system into consideration: We've solved the problem for a specific element, in a specific context, when we should be solving the general problem (needing to adjust `font-size`) for *any* element in *any* context. This is where utility classes come in.

```
/* ↓ Backslash to escape the colon */
.font-size\:base {
  font-size: 1rem;
}

.font-size\:biggish {
  font-size: 1.75rem;
}

.font-size\:big {
  font-size: 2.25rem;
}
```

We use a very *on the nose* naming convention, which emulates CSS
declaration structure: `property-name:value`. This helps with
recollection of utility class names, especially where the value echos
the actual value, like `.text-align:center`.

Sharing values between elements and utilities is a job for custom
properties ↗. Note that we've made the custom properties
themselves globally available by attaching them to the `:root` (`<html>`)
element:

```
:root {
  --font-size-base: 1rem;
  --font-size-biggish: 1.75rem;
  --font-size-big: 2.25rem;
}

/* elements */

h3 {
  font-size: var(--font-size-biggish);
}
h2 {
  font-size: var(--font-size-big);
}

/* utilities */

.font-size\:base {
  font-size: var(--font-size-base) !important;
}

.font-size\:biggish {
  font-size: var(--font-size-biggish) !important;
}

.font-size\:big {
  font-size: var(--font-size-big) !important;
}
```
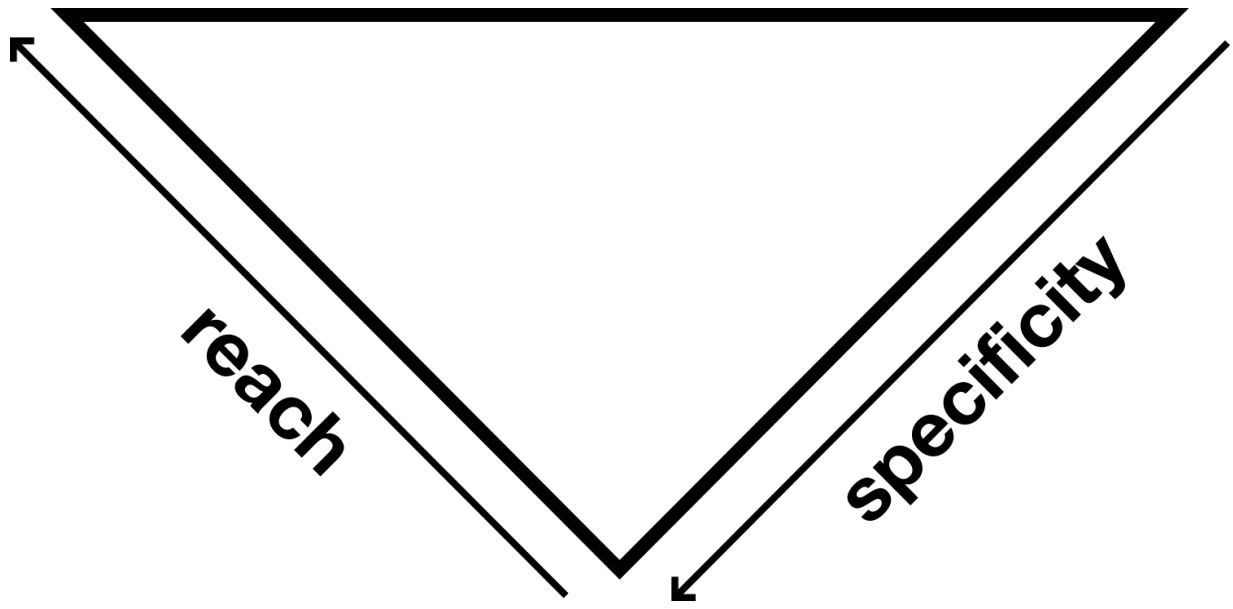
Each utility class has an `!important` suffix to max out its specificity. Utility classes are for final adjustments, and should not be overridden by anything that comes before them.

*Sensible CSS architecture has "reach" (how many elements are affected) inversely proportional to specificity (how complex the selectors are). This was formalized by Harry Roberts as ITCSS, with IT standing for Inverted Triangle.*

The values in the previous example are just for illustration. For consistency across the design, your sizes should probably be derived from a modular scale. See **Modular scale** for more.

## ⚠ Too many utility classes

One thing we highly recommend is *not* including utility classes until you need them. You don't want to send users any unused or redundant data. For this project, we maintain a `helpers.css` file and add utilities as we find ourselves reaching for them. If we find the `text-align:center` class isn't taking effect, we must not have added it in the CSS yet — so we put it in our `helpers.css` file for present and future use.

In a utility first ↗ approach to CSS, inherited, universal, and element styles are not really leveraged at all. Instead, combinations of individual styles are applied on a case-by-case basis to individual and independent elements. Using the Tailwind utility-first framework ↗ this might result—as exemplified by Tailwind's own documentation—in class values like the following:

```
class="rounded-lg px-4 md:px-5 xl:px-4 py-3 md:py-4 xl:py-3 bg-teal-500 hover:bg-
teal-600 md:text-lg xl:text-base text-white font-semibold leading-tight shadow-
md"
```

There may be reasons, under specific circumstances, why one might want to go this way. Perhaps there is a great deal of detail and disparity in the visual design that benefits from having this kind of granular control, or perhaps you want to prototype something quickly without context switching between CSS and HTML. **Every Layout**'s approach assumes you want to create robustness and consistency with the minimum of manual intervention. Hence, the concepts and techniques laid out here leverage axioms, primitives, and the styling algorithms that extrapolate from them instead.

## Local or 'scoped' styles

Notably, the `id` attribute/property (for reasons of accessibility ↗, most importantly) can only be used on one HTML element per document. Styling via the `id` selector is therefore limited to one instance.

```
#unique {
  /* ↓ Only styles id="unique" */
  font-family: sans-serif;
}
```

The `id` selector has a very high specificity ↗ because it's assumed unique styles are intended to override competing generic styles in all cases.

Of course, there's nothing more "local" or *instance specific* than applying styles directly to elements using the `style` attribute/property:

```
<p style="font-family: sans-serif">...</p>
```

The only remaining *standard* for localizing styles is within Shadow DOM. By making an element a `shadowRoot`, one can use low-specificity selectors that only affect elements inside that parent.

```
const elem = document.querySelector('div');
const shadowRoot = elem.attachShadow({mode: 'open'});
shadowRoot.innerHTML = `
  <style>
    p {
      /* ↓ Only styles <p>s inside the element's Shadow DOM */
      font-family: sans-serif;
    }
  </style>
  <p>A sans-serif paragraph</p>
`;
```

# Drawbacks

The `id` selector, inline styles, and Shadow DOM all have drawbacks:

- **`id` selectors:** Many find the high specificity to cause systemic issues. There's also the necessity of coming up with the `id`'s name in each case. Dynamically generating a unique string is often preferable.
- **Inline styles:** A maintenance nightmare, which is the reason CSS was antidotally conceived in the first place.
- **Shadow DOM:** Not only are styles prevented from "leaking out" of the Shadow DOM root, but (most) styles are not permitted to get *in* either — meaning you can no longer leverage global styling.
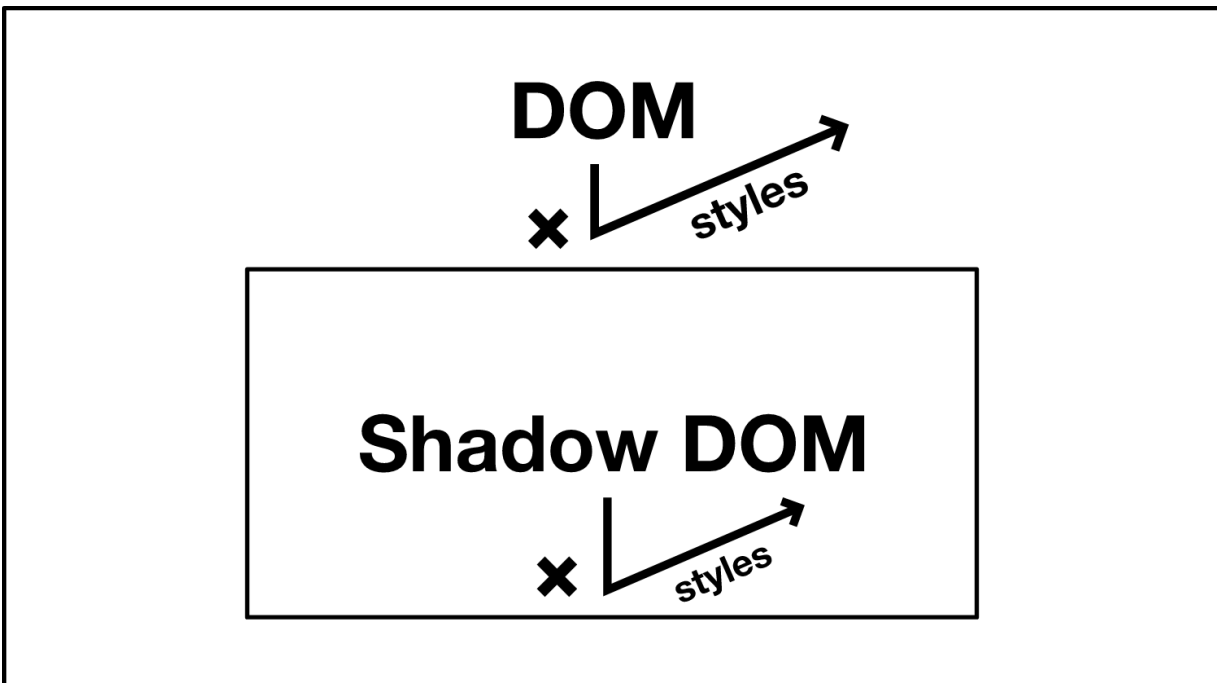
What we need is a way to simultaneously leverage global styling, but apply local, *instance-specific* styles in a controlled fashion.

# Primitives and props

As set out in **Composition**, the main focus of **Every Layout** is on the simple layout primitives that help to arrange elements/boxes together. These are the primary tools for eliciting layout and take their place between generic global styles and utilities.

1. Universal (including inherited) styles
2. Layout primitives
3. Utility classes

Manifested as reusable components, using the custom elements specification ↗, these layouts can be used globally. But unique *configurations* of these layouts are possible using props (properties).

# Interoperability

Custom elements are used in place of React, Preact, or Vue components (which all also use props) in **Every Layout** because they are native, and can be used *across* different web application frameworks. Each layout also comes with a code generator to produce just the CSS code needed to achieve the layout. You can use this to create a Vue-specific layout primitive (for example) instead.
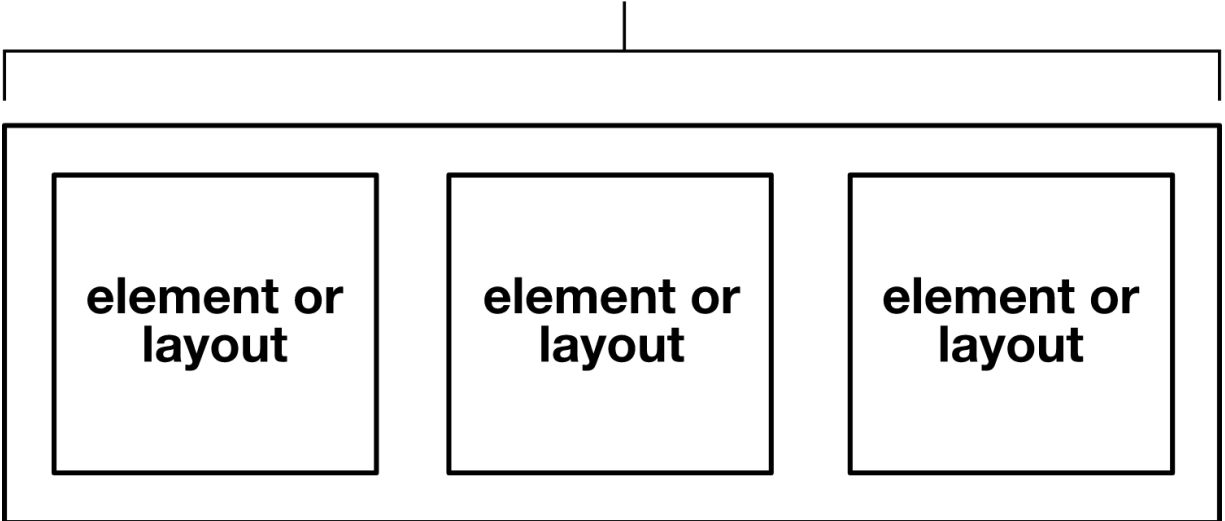
# Defaults

Each layout component has an accompanying stylesheet that defines its basic and default styles. For example, the **Stack** stylesheet (`Stack.css`) looks like the following.

```
stack-l {
  display: block;
}

stack-l > * + * {
  margin-top: var(--s1);
}
```

A few things to note:

- The `display: block` declaration is necessary since custom elements render as inline elements by default. See **Boxes** for more information on block and inline element behavior.
- The `margin-top` value is what makes the **Stack** a *stack*: it inserts margin between a vertical stack of elements. By default, that margin value matches the first point on our modular scale: `--s1`.
- The `*` selector applies to any element, but our use of `*` is qualified to match any *successive* child element of a `<stack-l>` (note the adjacent sibling combinator ↗). The layout primitive is a composition in abstract, and should not prescribe the content, so I use `*` to match any child element types given to it.

# layout



In the custom element definition itself, we apply the default value to the `space` property:

```
get space() {
   return this.getAttribute('space') || 'var(--s1)';
}
```

Each **Every Layout** custom element builds an embedded stylesheet based on the instance's property values. That is, this…

```
<stack-l space="var(--s3)">
  <div>...</div>
  <div>...</div>
  <div>...</div>
</stack-l>
```

…would become this…

```
<stack-l data-i="Stack-var(--s3)" space="var(--s3)">
  <div>...</div>
  <div>...</div>
  <div>...</div>
</stack-l>
```

… and would generate this:

```
<style id="Stack-var(--s3)">
  [data-i='Stack-var(--s3)'] > * + * {
    margin-top: var(--s3);
  }
</style>
```

However—and this part is important—the `Stack-var(--s3)` string only represents a unique *configuration* for a layout, not a unique instance. One `id="Stack-var(--s3)"` `<style>` element is used to serve all instances of `<stack-l>` sharing the configuration represented by the `Stack-var(--s3)` string. Between instances of the same configuration, the only thing that *really* differentiates them is their *content*.

While each item of content within a web page should generally offer unique information, the *look and feel* should be consistent and regular, using familiar and repeated patterns, motifs, and arrangements. Leveraging global styles along with controlled layout configurations results in consistency and cohesion, and with minimal code.

# Modular scale

Music is fundamentally a mathematical exposition, and when we talk about <u>the musicality of typesetting ↗</u> it is because typesetting and music share a mathematical basis.

We're sure you will have heard of concepts like frequency, pitch, and harmony. These are all mathematically determinable, but were you aware that perceived pitch can be formed of multiple frequencies?

A single musical note, such as one produced by plucking a guitar string, is in itself a composition. The different frequencies (or harmonics) together belong to a harmonic series. A harmonic series is a sequence of fractions based on the arithmetic series of incrementation by 1.

```
1,2,3,4,5,6 // arithmetic series
1,½,⅓,¼,⅕,⅙ // harmonic series
```

1st harmonic

2nd harmonic

3rd harmonic

4th harmonic
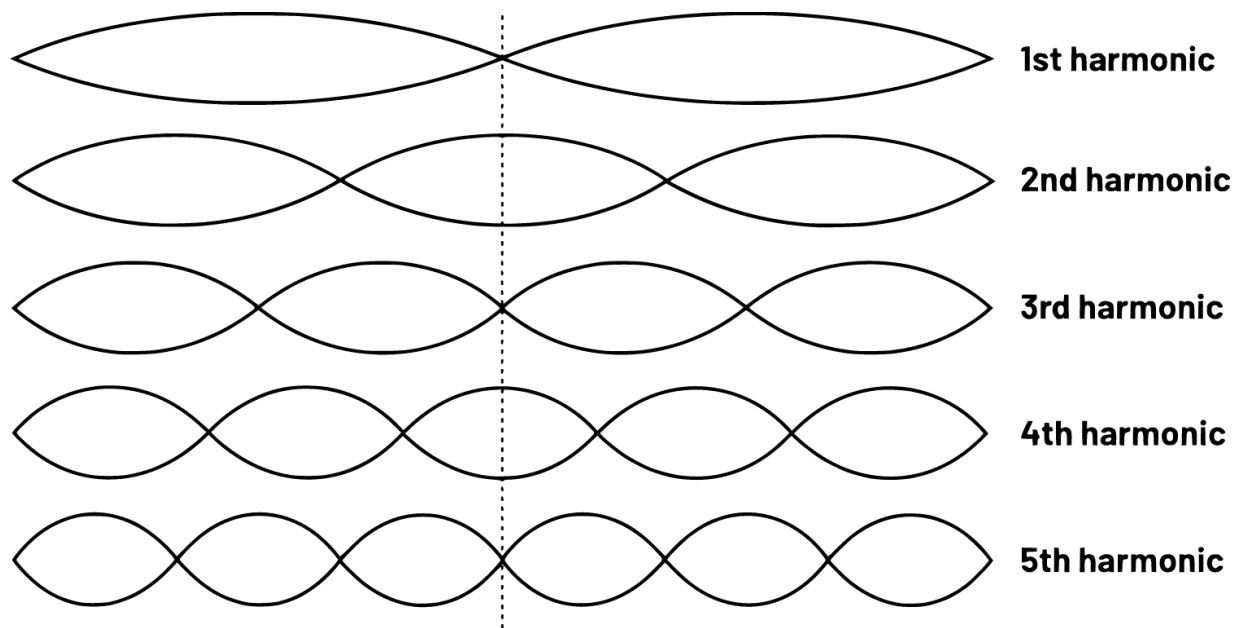
5th harmonic

The resulting sound is harmonious because of its regularity. The fundamental frequency is divisible by each of the harmonic frequencies, and each harmonic frequency is the mean of the frequencies either side of it.

# Visual harmony

We should aim for harmony in our visual layout too. Like the sound of a plucked string, it should be cohesive. Given we're working predominantly with text, it's sensible to treat the `line-height` as a basis for extrapolating values for white space. A font-size of (implicitly) `1rem`, and a `line-height` of 1.5 creates a default value of `1.5rem`. A harmoniously larger space might be `3rem` (2 × 1.5) or `4.5rem` (3 × 1.5).

Creating a sequence by adding 1.5 at each step results in large intervals. Instead, we can multiply by 1.5. The result is still regular; the increments just smaller.

```
1 * 1.5; // 1.5
1.5 * 1.5; // 2.25
1.5 * 1.5 * 1.5; // 3.375
```

This algorithm is called a modular scale, and like a musical scale is intended for producing harmony. How you employ it in your design depends on what technology you are using.

# Custom properties

In CSS, you can describe a modular scale using custom properties and the `calc()` function, which supports simple arithmetic.

In the following example, we divide or multiply by the set `--ratio` custom property (variable) to create the points on our scale. We can make use of already set points to generate new ones. That is, `var(--s2) * var(--ratio)` is equivalent to `var(--ratio) * var(--ratio) * var(--ratio)`.

```
:root {
  --ratio: 1.5;
  --s-5: calc(var(--s-4) / var(--ratio));
  --s-4: calc(var(--s-3) / var(--ratio));
  --s-3: calc(var(--s-2) / var(--ratio));
  --s-2: calc(var(--s-1) / var(--ratio));
  --s-1: calc(var(--s0) / var(--ratio));
  --s0: 1rem;
  --s1: calc(var(--s0) * var(--ratio));
  --s2: calc(var(--s1) * var(--ratio));
  --s3: calc(var(--s2) * var(--ratio));
  --s4: calc(var(--s3) * var(--ratio));
  --s5: calc(var(--s4) * var(--ratio));
}
```
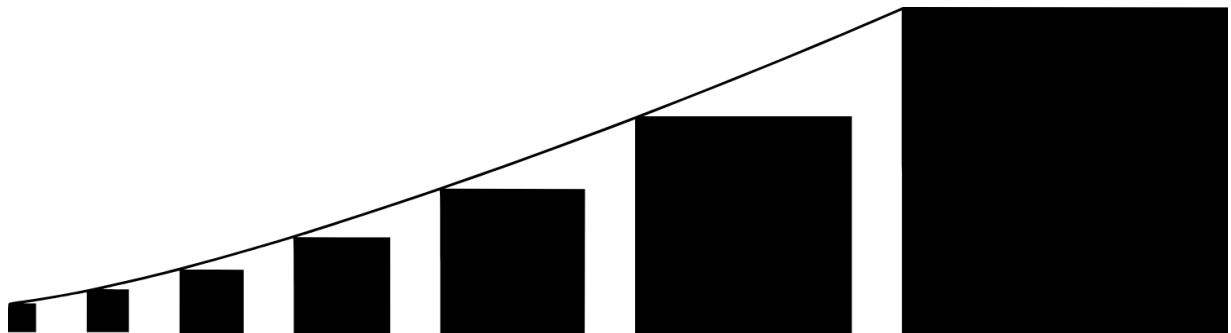
*Note the curved incline observable when connecting the top left corners of squares representing points on the scale*

# The `pow()` function

At the time of writing, browsers only support basic arithmetic in `calc()` operations. However, a new suite of mathematical functions/expressions ↗ are coming to CSS. Crucially, this includes the `pow()` function, with which accessing and creating modular scale points becomes much easier.

```
:root {
  --ratio: 1.5rem;
}

.my-element {
  /* ↓ 1.5 * 1.5 * 1.5 is equal to 1.5³ */
  font-size: pow(var(--ratio), 3);
}
```

# JavaScript access

Our scale variables are placed on the `:root` element, making them globally available. And by global, we mean *truly* global. Custom properties are available to JavaScript and also "pierce" Shadow DOM boundaries to affect the CSS of a `shadowRoot` stylesheet.

JavaScript consumes CSS custom properties like JSON properties. You can think of global custom properties as configurations shared by CSS and JavaScript. Here's how you would get the `--s3` point on the scale using JavaScript (`document.documentElement` represents the `:root`, or `<html>` element):

```
const rootStyles = getComputedStyle(document.documentElement);
const scale3 = rootStyles.getPropertyValue('--s3');
```

# Shadow DOM support

The same `--s3` property is successfully applied when invoked in Shadow DOM, as in the following example. The `:host` selector refers

to the hypothetical custom element itself.

```
this.shadowRoot.innerHTML = `
  <style>
    :host {
      padding: var(--s3);
    }
  </style>
  <slot></slot>
`;
```

## Passing via props

Sometimes we might want our custom element to consume certain styles from properties (*props*) — in this case a `padding` prop.

```
<my-element padding="var(--s3)">
  <!-- Light DOM contents -->
</my-element>
```

The `var(--s3)` string can be interpolated into the custom element instance's CSS using a <u>template literal ↗</u>:

```
this.shadowRoot.innerHTML = `
  <style>
    :host {
      padding: ${this.padding};
    }
  </style>
  <slot></slot>
`;
```

But first we need to write a getter and a setter for our `padding` prop. The `|| var(--s1)` suffix in the getter's `return` line is the *default* value. Use of sensible defaults makes working with layout components less laborious; we're aiming for convention over configuration ↗.

```
get padding() {
  return this.getAttribute('padding') || 'var(--s1)';
}

set padding(val) {
  return this.setAttribute('padding', val);
}
```

# ⚠ Eschewing Shadow DOM

The custom elements used to implement **Every Layout's** layouts do not use Shadow DOM because they are designed to more fully leverage 'global' styles. See **Global and local styling** for more information.

Not using Shadow DOM also makes it easier to server-side render the embedded styles. The *initial* styling of any one layout is embedded into the document as part of the build process, meaning **Every Layout's** custom elements are not dependent on JavaScript, *except* for the dynamic processing of their values in developer tools, or via your own custom scripting.

# Enforcing consistency

This `padding` prop is currently permissive; the author can supply a custom property, or a simple length value like `1.25rem`. If we wanted to enforce the use of our modular scale, we would accept only numbers (`2`, `3`, `-1`) and interpolate them like `var(--${this.padding})`.

We could check that an integer value is being passed using a regular expression. HTML attribute values are implicitly strings. We are looking for a single digit string containing a number.

```
if (!/(?<!\S)\d(?!\S)/.test(this.padding)) {
  console.error('<my-component>'s padding value should be a number representing a
point on the modular scale');
  return;
}
```

The modular scale is predicated on a single number, in this case `1.5`. Through extrapolation—as a multiplier and divisor—the number's presence can be felt throughout the visual design. Consistent, balanced design is seeded by simple <u>axioms</u> like the modular scale ratio.

Some believe the specific ratio used for one's modular scale is important, with many adhering to the <u>golden ratio ↗</u> of `1.61803398875`. But it is in the strict adherence to *whichever* ratio you choose that harmony is created.

# Axioms

As <u>the mathematician Euclid was aware ↗</u>, even the most complex geometries are founded on simple, irreducible axioms (or *postulates*). Unless your design is founded on axioms, your output will be inconsistent and malformed. The subject of this section is how to honor a design axiom system-wide, using *typographic measure* as an exemplar.

## Measure

The width of a line of text, in characters, is known as its <u>measure ↗</u>. Choosing a *reasonable* measure is critical for the comfortable scanning of successive lines. <u>The Elements Of Typographic Style ↗</u> considers any value between 45 and 75 reasonable.

Setting a measure for print media is relatively straightforward. It is simply the width of the paper artefact divided by the number of text columns placed within it — minus <u>margins and gutters ↗</u>, of course.

**measure**

← →
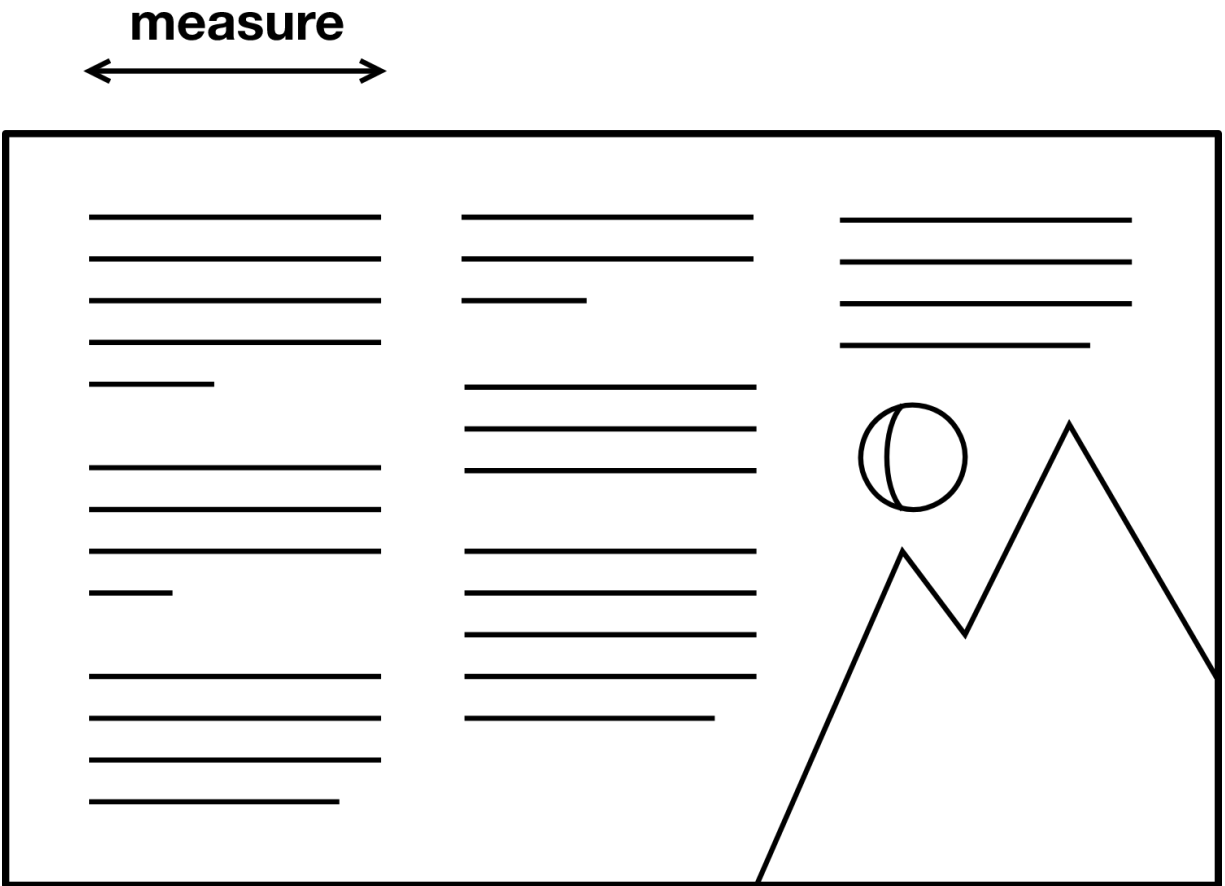
The web is not static or predictable like print. Each word is separated by a breaking space (unicode point U+0020 ↗), freeing runs of text to wrap dynamically according to available space. The amount of available space is determined by a number of interrelated factors including device size and orientation, text size, and zoom level.

As designers, we seek to control the users' experience. But, as John Allsopp wrote in 2000's The Dao Of Web Design ↗, attempting *direct* control over the way users consume content on the web is foolhardy. Enforcing a specific measure would mean setting a fixed width. Many users would experience horizontal scrolling and broken zoom functionality.
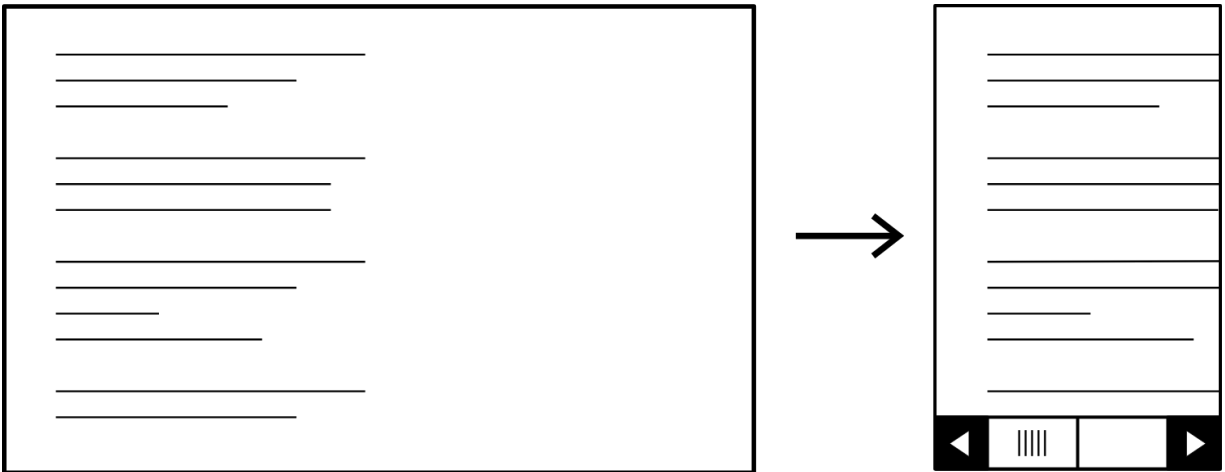
To design "adaptable pages" (Allsopp's term), we must relinquish control to the algorithms (like text wrapping) browsers use to lay out web pages automatically. But that's not to say there's no place for *influencing* layout. Think of yourself as the browser's mentor, rather than its micro-manager.

# The measure axiom

It's good practice to try and set out a design axiom in a short phrase or sentence. In this case that statement might be, *"the measure should never exceed 60ch"*.

The measure of what? And where? There's no reason why *any* line of text should become too lengthy. This axiom, like all axioms, should pervade the design without qualifications or exceptions. The real question is: how? In **Global and local styling** we set out three main tiers of styling:

1. Universal (including inherited) styles
2. Layout primitives
3. Utility classes

The measure axiom should be seeded as pervasively as possible in the universal styles, but also made available to layout primitives (see **Composition**) and <u>utility classes</u>. But first, which property and which value should inscribe the rule?

# The declaration

Fixed widths (and heights!) are anathema to responsive design, as we established in **Boxes** and again here. Instead, we should deal in *tolerances*. The `max-inline-size` property, for example, tolerates any length of text, in any writing mode, *up to* a certain value.

```
p {
  max-inline-size: 700px;
}
```

That's the property covered. However, the `px` unit is problematic. We may be able to judge, by eye, that `700px` creates a reasonable measure for the given `font-size`. But the given `font-size` is really just the `font-size` our screen happens to be displaying at the time — it's our parochial view of our own design.

Changing `font-size` for paragraphs, or adjusting the default system font size, will create a different (maximum) measure. Because there is no *relationship* between character length and pixel width, we do not have an algorithm that can guarantee the correct maximum measure value.
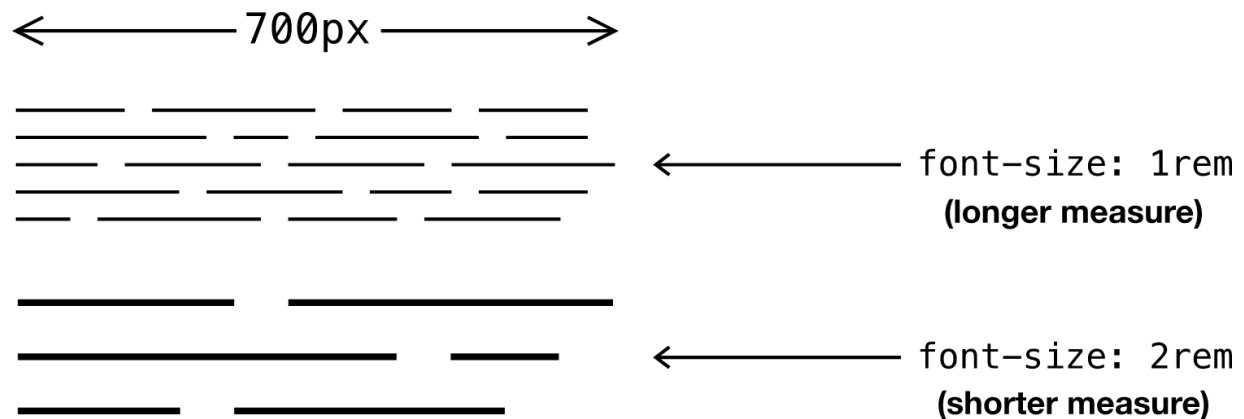


Fortunately, CSS includes the `ch` unit. The value of `1ch` is based on the width of the font's `0` character. Importantly, this means changing the `font-size` changes the value of `1ch`, thereby adapting the measure. Using `ch` units is an innately algorithmic approach to measure, because the outcome is predicated on a calculation you permit the browser to make for you.

Using `ch` enables us to enforce the axiom independent of `font-size`, allowing it to be highly pervasive and in no danger of "going wrong". Where "*the measure should never exceed 60ch*" might have been a note in some documentation, it can instead be a quality directly coded into the design's character.

# Designing without seeing

Designing by axiom requires something of a mental shift. Axioms do not directly create visual artefacts, only the characteristics of artefacts that might emerge.

Sometimes the resulting artefacts look and behave in ways that you might not have foreseen. For example, in a container which is wider than the agreed measure as applied to the base font size, elements with different font sizes will take up different proportions of that container's width. This is because `1ch` is wider for a larger font size.
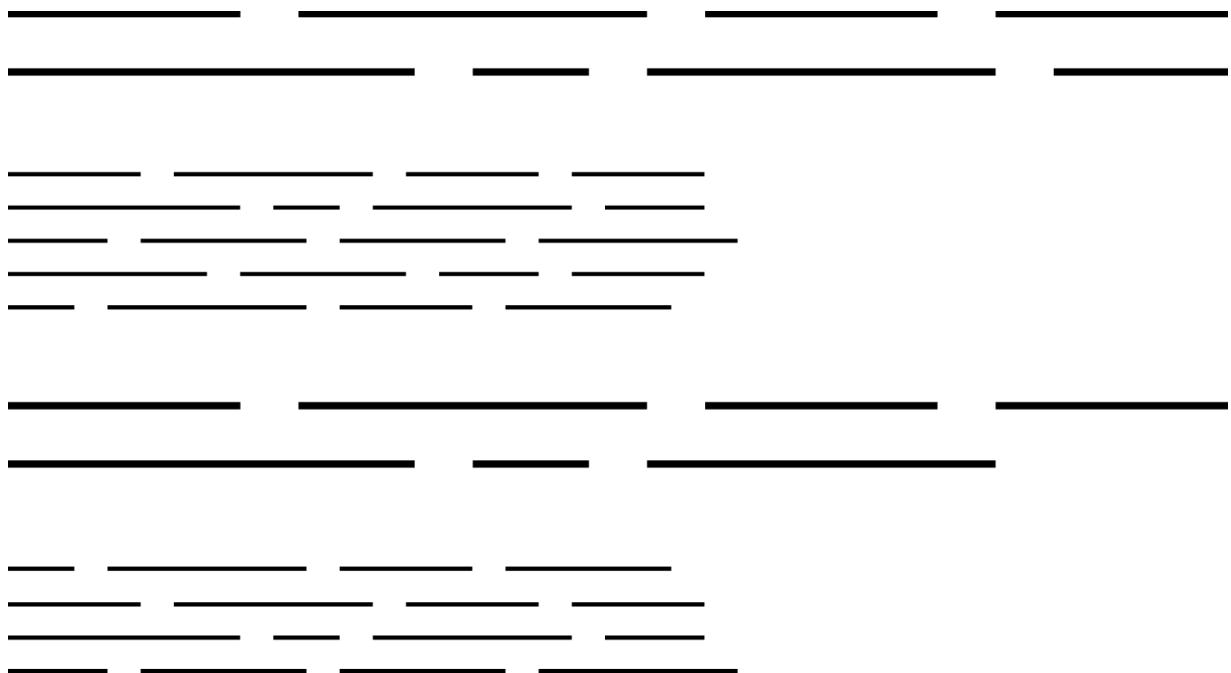
At the time of conceiving the axiom, you may not have pictured this specific visual effect. But that's not to say it isn't sound or desirable. In fact, it's your CSS doing exactly as you intended: maintaining a reasonable measure irrespective of the context.

Fundamentally, designing for the web is designing *without seeing*. You simply can't anticipate all of the visual combinations produced

by

1. The modular placement of your layout components
2. The circumstances and settings of each end user's setup

Instead of thinking of designing for the web as creating visual artefacts, think of it as writing *programs* for *generating* visual artefacts. Axioms are the rules that influence how those artefacts are created by the browser, and the better thought out they are the better the browser can accommodate the user.

# Global defaults

To realize the axiom, we need to ensure all applicable elements are subject to it. This is a question of selectors. We could create a class selector…

```
.measure-cap {
  max-inline-size: 60ch;
}
```

…but it's a mistake to think in terms of (utility) classes too early. It would mean applying the style manually, to individual elements in the HTML, wherever we felt it was applicable. Manual intervention is laborious, prone to error (missing elements out), and will lead to bloated markup.

Instead, we should ask ourselves which *types* of elements the rule might apply to. Certainly flow elements designed for text. Inline elements like `<em>` and `<small>` would not need to be included, since they would take up only a part of their parent flow elements' total measure.

```css
p,
h1,
h2,
h3,
h4,
h5,
h6,
li,
figcaption {
  max-inline-size: 60ch;
}
```

## Exception-based styling

It's difficult to know if we've remembered everything here. An exception based approach is smarter, since we only have to remember which elements should *not* be subject to the rule. Note that inline elements *would* be included in the following example but, since they would take up an equal or lesser horizontal space than their parents, no ill effects would emerge.

```
* {
  max-inline-size: 60ch;
}

html,
body,
div,
header,
nav,
main,
footer {
  max-inline-size: none;
}
```

The `<div>` element particularly tends to be used as a generic container/wrapper. It's likely some of these elements will contain multiple adjacent boxes, with one or more of each wishing to take up the full `60ch`. This makes their parents logical exceptions.

An exception-based approach to CSS lets us do *most* of our styling with the *least* of our code. If you are not taking an exception-based approach, it may be because making exceptions feels like *correcting mistakes*. But this is far from the case. CSS, with its cascade and other features ↗, is designed for this. In Harry Roberts' ITCSS (Inverted Triangle CSS) ↗ thesis, specificity (how specific selectors are) is inversely proportional to reach (how many elements they should affect).

# A universal value

Before we start using the measure value everywhere, we'd best define it as a custom property. That way, any change to the value will be propagated throughout the design.

Note that not all custom properties have to be global, but in this case we want our elements, props, and utility classes to agree. Therefore, we place the custom property on the `:root` element.

```
:root {
  --measure: 60ch;
}
```

This is passed into our universal block…

```
* {
  max-inline-size: var(--measure);
}

html,
body,
div,
header,
nav,
main,
footer {
  max-inline-size: none;
}
```

…and to any utility classes we may find we need.

```
.max-inline-size\:measure {
  max-inline-size: var(--measure);
}

.max-inline-size\:measure\/2 {
  max-inline-size: calc(var(--measure) / 2);
}
```

# Escaping

The backslashes are required in the previous example to escape the special forward slash and colon characters.

# Measure in composite layouts

Certain <u>layout primitives</u> inevitably accept measure-related props, and some set default values for those props using `var(--measure)`. The **Switcher** has a `threshold` prop that defines the container width at which the layout switches between a horizontal and vertical configuration:

```
get threshold() {
  return this.getAttribute('threshold') || 'var(--measure)';
}

set threshold(val) {
  return this.setAttribute('threshold', val);
}
```

This is a sensible default, but can easily be overridden with any string value:

```
<switcher-l threshold="20rem">...</switcher-l>
```

If we pass an illegitimate value to `threshold`, the declaration will be dropped, and the **Switcher's** fallback stylesheet will apply the default value anyway. Here's what that stylesheet looks like:

```css
switcher-l {
  display: flex;
  flex-wrap: wrap;
}

switcher-l > * {
  flex-basis: calc((var(--measure) - 100%) * 999);
  flex-grow: 1;
}
```

Our approach to measure is one where we assume control, but a tempered kind of control that's deferential towards the way browsers work and users operate them. Many of the 'axioms' that govern your design, like "*the body font will be Font X*" or "*headings will be dark blue*" will not have an impact on layout as such, making them much simpler to apply just with global styles. When layout comes into the equation, be wary of differing configurations and orientations. Choose properties, values, and units that enable the browser to calculate the most suitable layout on your behalf.
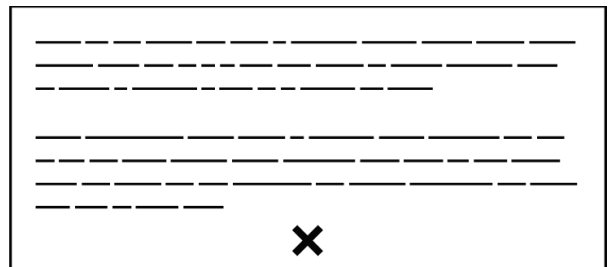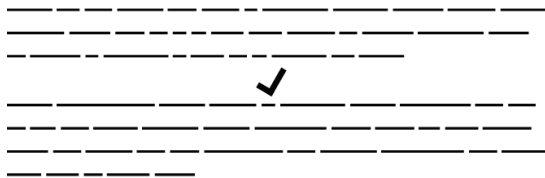
# The Stack

## The problem

Flow elements require space (sometimes referred to as *white space*) to physically and conceptually separate them from the elements that come before and after them. This is the purpose of the `margin` property.

However, design systems conceive elements and components in isolation. At the time of conception, it is not settled whether there will be surrounding content or what the nature of that content will be. One element or component is likely to appear in different contexts, and the requirement for spacing will differ.

We are in the habit of styling elements, or classes of elements, directly: we make style declarations *belong* to elements. Typically, this does not produce any issues, but `margin` is really a property of the *relationship* between two proximate elements. The following code is therefore problematic:

```
p {
  margin-bottom: 1.5rem;
}
```

Since the declaration is not context sensitive, any correct application of the margin is a matter of luck. If the paragraph is proceeded by another element, the effect is desirable. But a `:last-child` paragraph produces a redundant margin. Inside a padded parent element, this redundant margin combines with the parent's padding to produce double the intended space. This is just one problem this approach produces.

# The solution

The trick is to style the context, not the individual element(s). The **Stack** layout primitive injects margin between elements via their common parent:

```
.stack > * + * {
  margin-block-start: 1.5rem;
}
```

Using the adjacent sibling combinator (+), `margin-block-start` is only applied where the element is preceded by another element: no "left over" margin. The universal (or *wildcard*) selector (∗) ensures any and all elements are affected. The key ∗ + ∗ construct is known as the owl ↗.

## Line height and modular scale

In the previous example, we used a `margin-block-start` value of `1.5rem`. We're in the habit of using this value because it reflects our (usually preferred) body text `line-height` of `1.5`.

The vertical spacing of your design should be based on your standard `line-height` because text dominates most pages' layout, making one line of text a natural denominator.

If the body text `line-height` is `1.5` (i.e. `1.5` × the `font-size`), it makes sense to use `1.5` as the ratio for your modular scale. Read the introduction to modular scale, and how it can be expressed with CSS custom properties.

| |
|---|
| 1.5 |
| 1.5 |
| 1.5 |
| multiple or exponent of 1.5 |
| 1.5 |
| 1.5 |
| 1.5 |
| multiple or exponent of 1.5 |

# Recursion

In the previous example, the child combinator (>) ensures the margins only apply to children of the .stack element. However, it's possible to inject margins recursively by removing this combinator from the selector.

```
.stack * + * {
  margin-block-start: 1.5rem;
}
```

This can be useful where you want to affect elements at any nesting level, while retaining white space regularity.

—————————————————————
—————————————————————
—————————————



**First nesting level**

**Second nesting level**
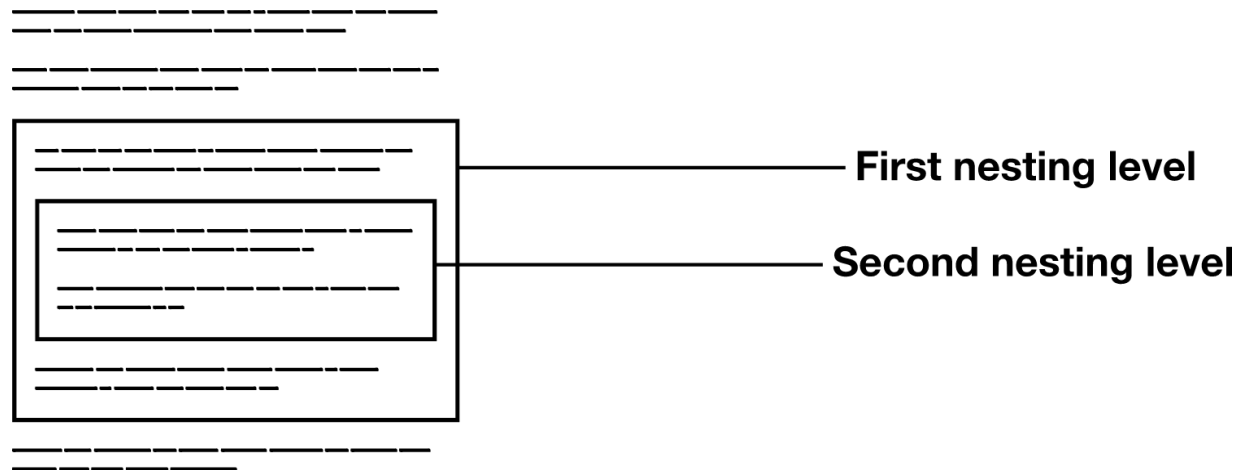
In the following demonstration (using the Stack component to follow) there are a set of box-shaped elements. Two of these are nested within another. Because recursion is applied, each box is evenly spaced using just one parent **Stack**.

This interactive demo is only available on the **Every Layout** site ↗.

You're likely to find the recursive mode affects unwanted elements. For example, generic list items that are typically not separated by margins will become unexpectedly *spread out*.

# Nested variants

Recursion applies the same margin no matter the nesting depth. A more deliberate approach would be to set up alternative non-recursive **Stacks** with different margin values, and nest them where suitable. Consider the following.

```css
[class^='stack'] > * {
  /* top and bottom margins in horizontal-tb writing mode */
  margin-block: 0;
}

.stack-large > * + * {
  margin-block-start: 3rem;
}

.stack-small > * + * {
  margin-block-end: 0.5rem;
}
```

This interactive demo is only available on the **Every Layout** site ↗.

The first declaration block's selector resets the vertical margin for all **Stack**-like elements (by matching class values that *begin* with `stack`). Importantly, only the vertical margins are reset, because the stack only *affects* vertical margin, and we don't want it to reach outside its remit. You may not need this reset if a universal reset for `margin` is already in place (see **Global and local styling**).

The following two blocks set up alternative **Stacks**, with different margin values. These can be nested to produce—for example—the illustrated form layout. Be aware that the `<label>` elements would need to have `display: block` applied to appear above the inputs, and

for their margins to actually produce spaces (the vertical margin of inline elements has no effect; see **The display property**).



In **Every Layout**, custom elements are used to implement layout components/primitives like the **Stack**. In the **Stack** component, the `space` prop (property; attribute) is used to define the spacing value. The modified classes example above is just for illustration. See the nested example.

# Exceptions

CSS works best as an exception-based language. You write far-reaching rules, then use the cascade to override these rules in special cases. As written in Managing Flow and Rhythm with CSS Custom Properties ↗, you can create per-element exceptions within a single **Stack** context (i.e. at the same nesting level).

```
.stack > * + * {
  margin-block-start: var(--space, 1.5em);
}

.stack-exception,
.stack-exception + * {
  --space: 3rem;
}
```

Note that we are applying the increased spacing above *and* below the `.exception` element, where applicable. If you only wanted to increase the space above, you would remove `.exception + *`.

This works because `*` has *zero* specificity, so `.stack > * + *` and `.stack-exception` are the same specificity and `.stack-exception` overrides `.stack > * + *` in the cascade (by appearing further down in the stylesheet).

# Splitting the stack

By making the **Stack** a Flexbox context, we can give it one final power: the ability to add an `auto` margin to a chosen element. This way, we can group elements to the top and bottom of the vertical space. Useful for card-like components.

In the following example, we've chosen to group elements *after* the second element towards the bottom of the space.

```
.stack {
  display: flex;
  flex-direction: column;
  justify-content: flex-start;
}

.stack > * + * {
  margin-block-start: var(--space, 1.5rem);
}

.stack > :nth-child(2) {
  margin-block-end: auto;
}
```

# Custom property placement

Importantly, despite now setting some properties on the parent
`.stack` element, we're still setting the `--space` value on the children,
not "hoisting" it up. If the parent is where the property is set, it will
get overridden if the parent becomes a child in nesting (see **Nested
variants**, above).

This can be seen working in context in the following demo depicting
a presentation/slides editor. The **Cover** element on the right has a
minimum height of `66.666vh`, forcing the left sidebar's height to be
taller than its content. This is what produces the gap between the
slide images and the *Add slide* button.

This interactive demo is only available on the **Every Layout** site ↗.

Where the **Stack** is the only child of its parent, nothing forces it to *stretch* as in the last example/demo. A height of `100%` ensures the **Stack's** height *matches* the parent's and the split can occur.

```
.stack:only-child {
  /* ↓ `height` in horizontal-tb writing mode */
  block-size: 100%;
}
```

# Use cases

The potential remit of the **Stack** layout can hardly be overestimated. Anywhere elements are stacked one atop another, it is likely a **Stack** should be in effect. Only adjacent elements (such as grid cells) should not be subject to a **Stack**. The grid cells *are* likely to be **Stacks**, however, and the grid itself a member of a **Stack**.
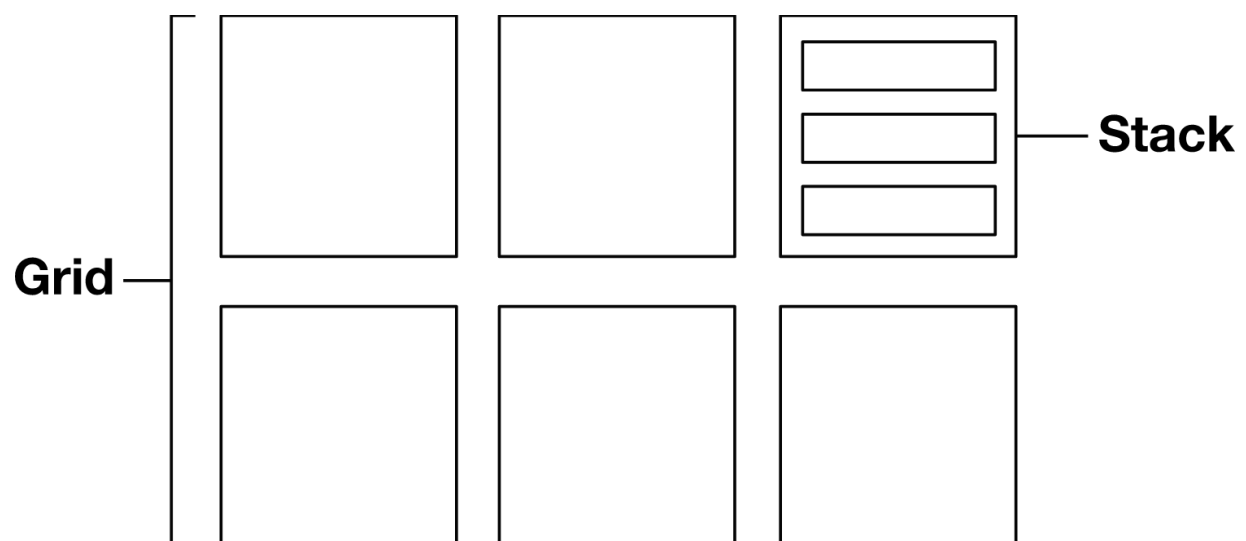
# The generator

The code generator tool is only available in <u>the accompanying documentation site</u> ↗. Here is the basic solution, with comments:

## CSS

```css
.stack {
  /* ↓ The flex context */
  display: flex;
  flex-direction: column;
  justify-content: flex-start;
}

.stack > * {
  /* ↓ Any extant vertical margins are removed */
  margin-block: 0;
}

.stack > * + * {
  /* ↓ Top margin is only applied to successive elements */
  margin-block-start: var(--space, 1.5rem;);
}
```

## HTML

```html
<div class="stack">
  <div><!-- child --></div>
  <div><!-- child --></div>
  <div><!-- etc --></div>
</div>
```

## The component

A custom element implementation of the Stack is available for
download ↗.

# Props API

The following props (attributes) will cause the **Stack** component to
re-render when altered. They can be altered by hand—in browser
developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
| --- | --- | --- | --- |
| **space** | string | "var(--s1)" | A CSS margin value |
| **recursive** | boolean | false | Whether the spaces apply recursively (i.e. regardless of nesting level) |
| **splitAfter** | number | | The element after which to *split* the stack with an auto margin |

# Examples

## Basic

```
<stack-l>
  <h2><!-- some text --></h2>
  <img src="path/to/some/image.svg" />
  <p><!-- more text --></p>
</stack-l>
```

## Nested

```
<stack-l space="3rem">
  <h2><!-- heading label --></h2>
  <stack-l space="1.5rem">
    <p><!-- body text --></p>
    <p><!-- body text --></p>
    <p><!-- body text --></p>
  </stack-l>
  <h2><!-- heading label --></h2>
  <stack-l space="1.5rem">
    <p><!-- body text --></p>
    <p><!-- body text --></p>
    <p><!-- body text --></p>
  </stack-l>
</stack-l>
```

## Recursive

```
<stack-l recursive>
  <div><!-- first level child --></div>
  <div><!-- first level sibling --></div>
  <div>
    <div><!-- second level child --></div>
    <div><!-- second level sibling --></div>
  </div>
</stack-l>
```

## List semantics

In some cases, browsers should interpret the **Stack** as a list for screen reader software. You can use the following ARIA attribution to achieve this.

```
<stack-l role="list">
  <div role="listitem"><!-- item 1 content --></div>
  <div role="listitem"><!-- item 2 content --></div>
  <div role="listitem"><!-- item 3 content --></div>
</stack-l>
```
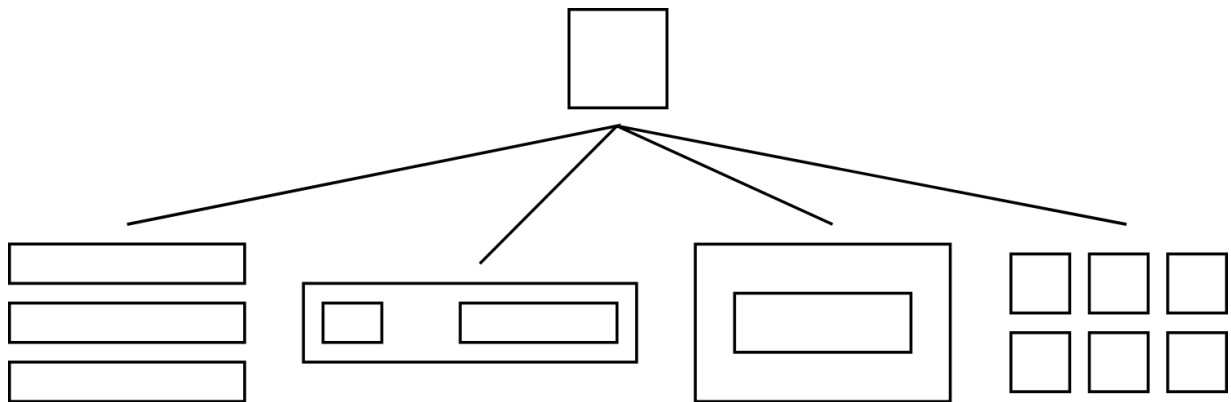
# The Box

## The problem

As I established in **Boxes**, every rendered element creates a box shape. So what is the use of a **Box** layout, encapsulated as a dedicated **Box** component?

All the ensuing layouts deal in arranging boxes *together*; distributing them in some way such that they form a composite visual structure. For example, the simple **Stack** layout takes a number of boxes and inserts vertical margin between them.

It is important the **Stack** is given no other purpose than to insert vertical margins. If it started to take on other responsibilities, its job description would become a nonsense, and the other layout primitives within the system wouldn't know how to behave around the **Stack**.

In other words, it's a question of separating concerns ↗. Just as in computer science, in visual design it benefits your system to give each working part a dedicated and unique responsibility. The design emerges through composition.

The **Box**'s role within this layout system is to take care of any styles that can be considered intrinsic to individual elements; styles which are not dictated, inherited, or inferred from the meta-layouts to which an individual element may be subjected. But which styles are these? It feels like they could be innumerable.

Not necessarily. While some approaches to CSS give you the power (or the *pain*, depending on your perspective) to apply any and every style to an individual element, there are plenty of styles that do not need to be written in this piecemeal way. Styles like `font-family`, `color`, and `line-height` can all be *inherited* or otherwise applied globally, as set out in **Global and local styling**. And they should, because setting these styles on a case-by-case basis is redundant.

```
:root {
  font-family: sans-serif;
}

.box {
  /* ↓ Not needed because the style is inherited */
  /* font-family: sans-serif; */
}
```

Of course, you are likely to employ more than one `font-family` in your design. But it is more efficient to apply default (or 'base') styles and later make *exceptions* than to style everything like it is a special case.

Conveniently, global styles tend to be *branding* related styles — styles that affect the aesthetics but not the *proportions* of the subject element(s). The purpose of this project is to explore the creation of a *layout system* specifically, and we are not interested in branding (or aesthetics) as such. We are building dynamic, responsive wireframes. Aesthetics can be applied on top.
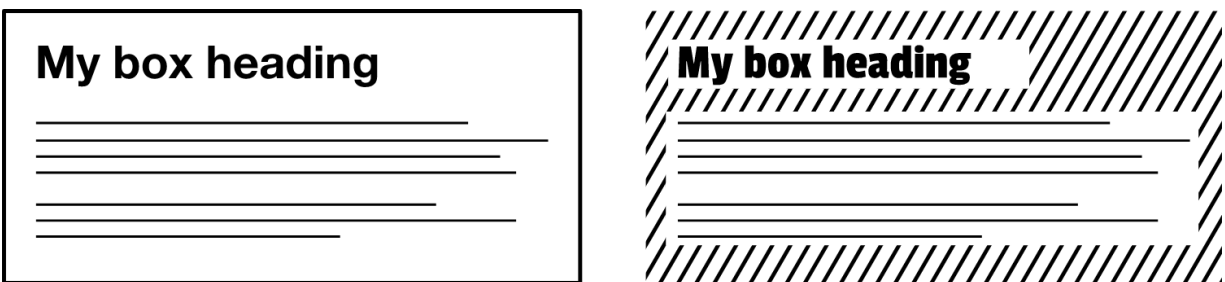


*Same layouts; different aesthetics*

This limits the number of properties we have to choose from. To reduce this set of potential properties further, we have to ask ourselves which layout-specific properties are better handled by parent or ancestor elements of the simple **Box**.

# The solution

Margin is applicable to the **Box**, but only as induced by context — as I've already established. Width and height should also be

inferred, either by an *extrinsic* value (such as the width calculated by `flex-basis`, `flex-grow`, and `flex-shrink` working together) or by the nature of the content held *inside* the **Box**.

Think of it like this: If you don't have anything to put in a box, you don't need a box. If you *do* have something to put in a box, the best box is one with just enough room and no more.

# Padding

Padding is different. Padding reaches *into* an element; it is introspective. Padding should be a **Box** styling option. The question is, how much control over `padding` for our **Box** is necessary? After all, CSS affords us `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`, as well as the `padding` shorthand.

Remember we are building a layout system, and not an API for creating a layout system. CSS itself is already the API. The **Box** element should have padding on *all* sides, or *no sides at all*. Why? Because an element with specific (and asymmetrical) padding is not a **Box**; it's something else trying to solve a more specific problem. More often than not, this problem relates to adding spacing between elements, which is what `margin` is for. Margin extends outside the element's border.
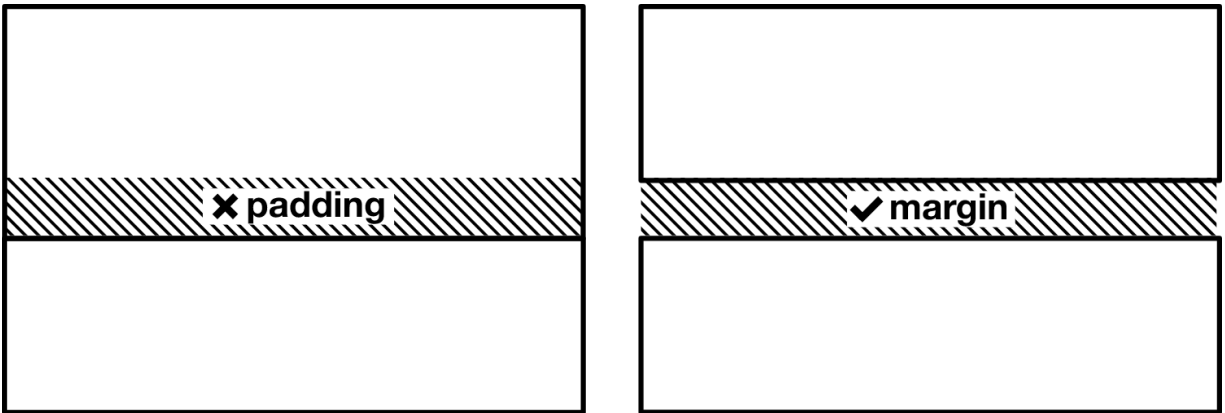
In the below example, I'm using a `padding` value corresponding to the first point on my <u>modular scale</u>. It is applied to all sides, and has the singular purpose of moving the **Box**'s content away from its edges.

```
.box {
  padding: var(--s1);
}
```

# The box model

As set out in **Boxes** you will avoid some sizing issues by applying `box-sizing: border-box`. However, this should already be applied to *all* elements, not just the named **Box**.

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

# The visible box

A **Box** is only really a **Box** if it has a box-like shape. Yes, <u>all elements are box-shaped</u>, but a **Box** should typically *show* you this. The most common methods use either `border` or a `background`.

Like `padding`, `border` should be applied on all sides or none at all. In cases where borders are used to *separate* elements, they should be applied contextually, via a parent, like `margin` is in the **Stack**. Otherwise, borders will come into contact and 'double up'.

<table>
<tr><td></td><td>* + *</td></tr>
<tr><td></td><td>* + *</td></tr>
<tr><td></td><td>* + *</td></tr>
<tr><td></td><td>* + *</td></tr>
</table>
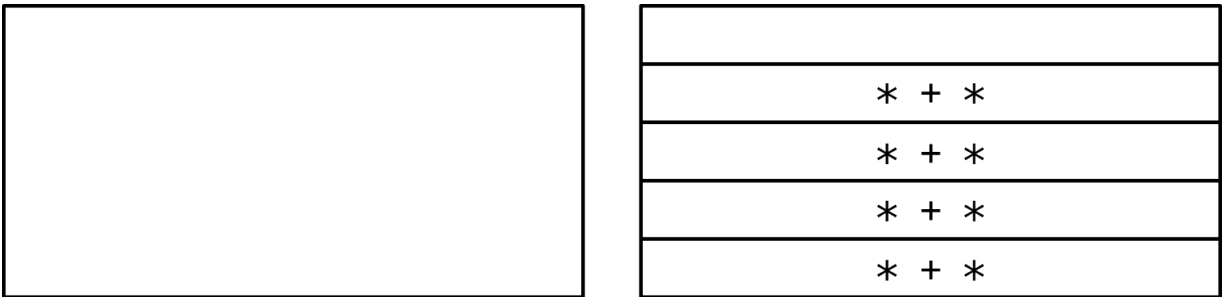
*By applying a `border-top` value via the `* + *` selector, only borders between child elements appear. None come into contact with the parent **Box's** bordered perimeter.*

If you've written CSS before, you've no doubt used `background-color` to create a visual box shape. Changing the `background-color` often requires you to change the `color` to ensure the content is still legible. This can be made easier by applying `color: inherit` to any elements inside that **Box**.

```
.box {
  padding: var(--s1);
}

.box * {
  color: inherit;
}
```

By forcing inheritance, you can change the `color`—along with the
`background-color`—in one place: on the **Box** itself. In the following
example, I am using an `.invert` class to swap the `color` and
`background-color` properties. Custom properties make it possible to
adjust the specific light and dark values in one place.

```
.box {
  --color-light: #eee;
  --color-dark: #222;
  color: var(--color-dark);
  background-color: var(--color-light);
  padding: var(--s1);
}

.box * {
  color: inherit;
}

.box.invert {
  /* ↓ Dark becomes light, and light becomes dark */
  color: var(--color-light);
  background-color: var(--color-dark);
}
```

# Filter inversion

In a greyscale design, it is possible to switch between dark-on-light
and light-on-dark with a simple `filter` declaration. Consider the

following code:

```css
.box {
  --color-light: hsl(0, 0%, 80%);
  --color-dark: hsl(0, 0%, 20%);
  color: var(--color-dark);
  background-color: var(--color-light);
}

.box.invert {
  filter: invert(100%);
}
```

Because `--color-light` is as light at `20%` as `--color-dark` is dark at `80%`, they are effectively opposites. When `filter: invert(100%)` is applied, they take each other's places. You can create a light/dark theme switcher ↗ with a similar technique.

`filter: invert(100%)`

**My box heading**

**My box heading**

When hue becomes involved it is inverted as well, and the effect is likely to be less desirable.

In the absence of a border, a `background-color` is insufficient for describing a box shape. This is because high contrast themes ↗

tend to eliminate backgrounds. However, by employing a transparent `outline` the box shape can be restored.

```
.box {
  --color-light: #eee;
  --color-dark: #222;
  color: var(--color-dark);
  background-color: var(--color-light);
  padding: var(--s1);
  outline: 0.125rem solid transparent;
  outline-offset: -0.125rem;
}
```

How does this work? When a high contrast theme is not running, the outline is invisible. The `outline` property also has no impact on layout (it grows out from the element to cover other elements if present). When <u>Windows High Contrast Mode is switched on ↗</u>, it gives the outline a color and the box is drawn.

The negative `outline-offset` moves the outline *inside* the **Box**'s perimeter so it behaves like a border and no longer increases the box's overall size.

## Use cases

The basic, and highly prolific, use case for a **Box** is to group and demarcate some content. This content may appear as a message or 'note' among other, textual flow content, as one <u>card ↗</u> in a grid of many, or as the inner wrapper of a positioned dialog element.
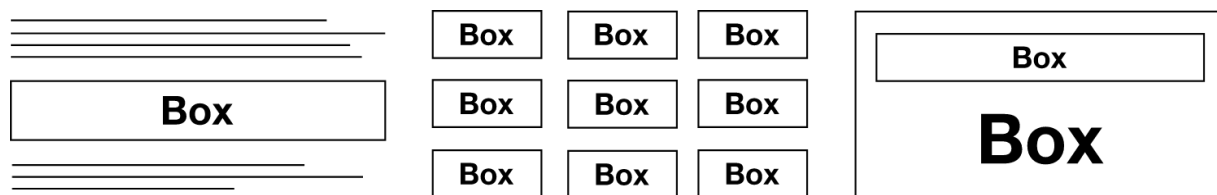
You can also combine just boxes to make some useful compositions. A **Box** with a 'header' element can be made from two sibling boxes, nested inside another, parent **Box**.
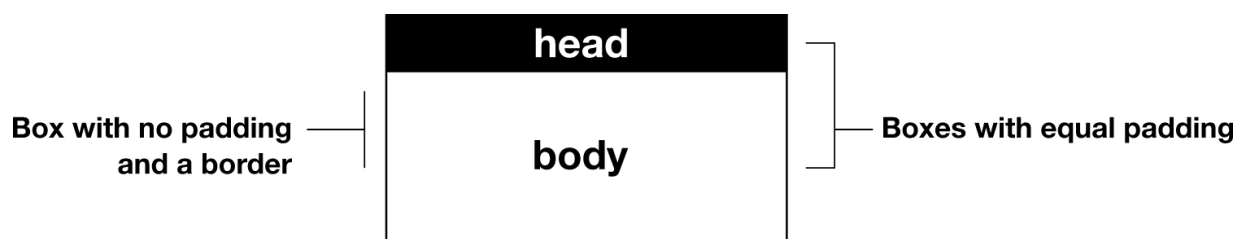


# The generator

Use this tool to generate basic **Box** CSS and HTML. It provides the ability to create basic, two-tone boxes, including light-on-dark and dark-on-light ('inverted') themes. See the **The visible box** section for more.

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments:

# CSS

```
.box {
  /* ↓ Padding set to the first point on the modular scale */
  padding: var(--s1);
  /* ↓ Assumes you have a --border-thin var */
  border: var(--border-thin) solid;
  /* ↓ Always apply the transparent outline, for high contrast mode */
  outline: var(--border-thin) transparent;
  outline-offset: calc(var(--border-thin) * -1);
  /* ↓ The light and dark color vars */
  --color-light: #fff;
  --color-dark: #000;
  color: var(--color-dark);
  background-color: var(--color-light);
}

.box * {
  /* ↓ Force colors to inherit from the parent
  and honor inversion (below) */
  color: inherit;
}

.box.invert {
  /* ↓ The color vars inverted */
  color: var(--color-light);
  background-color: var(--color-dark);
}
```

## HTML

```
<div class="box">
  <-- the box's contents -->
</div>
```

# The component

A custom element implementation of the Box is available for download ↗.

# Props API

The following props (attributes) will cause the **Box** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
| --- | --- | --- | --- |
| **padding** | `string` | `"var(--s1)"` | A CSS `padding` value |
| **borderWidth** | `string` | `"var(--border-thin)"` | A CSS `border-width` value |
| **invert** | `boolean` | `false` | Whether to apply an inverted theme. Only recommended for greyscale designs. |

# Examples

## Basic box

The **Box** comes with default padding and border. The `padding` value is set to the first point on the modular scale (`var(--s1)`) and the `border-width` uses the `var(--border-thin)` variable.

```
<box-l>
  <!-- contents of the box -->
</box-l>
```

## A Box within a Stack

In the context of a **Stack**, the **Box** will have `margin-top` applied if it is preceded by a sibling element.

```
<stack-l>
  <p>...</p>
  <blockquote>...</blockquote>
  <box-l>
    <!-- Box separated by vertical margins -->
  </box-l>
  <p>...</p>
  <div role="figure">...</div>
</stack-l>
```

## Box with a header

An implementation of the nested **Box** example from <u>Use cases</u>. The `invert` boolean attribute inverts the colors using `filter:` `invert(100%)`.

```
<box-l padding="0">
  <box-l borderWidth="0" invert>head</box-l>
  <box-l borderWidth="0">body</box-l>
</box-l>
```

# The Center

## The problem

In the early days of HTML, there were a number of presentational elements; elements devised purely to affect the appearance of their content. The <u>`<center>` ↗</u> was one such element, but has long since been considered obsolete. Curiously, it *is* still supported in some browsers, including Google's Chrome. Presumably this is because Google's search homepage still uses a `<center>` to center-justify its famous search input.

Tech' giants' whimsical usage of defunct elements aside, we mostly moved away from using presentational markup in the 2000s. By making styling the responsibility of a separate technology—CSS—we were able to manage style and structure separately. Consequently, a change in art direction would no longer mean reconstituting our content.

We later discovered that styling HTML purely in terms of semantics and context was rather ambitious, and led to some unwieldy selectors like

```
body > div > div > a {
  /*
  Link styles specifically for links
  nested two <div>s inside the body element
  */
}
```

For the sake of easier CSS maintenance and style modularity many of us adopted a compromise position using classes. Because classes can be placed on any element, we are free to style, say, a non-semantic `<div>` or a screen reader recognized `<nav>` in exactly the same way, using the same token, but without compromising on accessibility.

```
<div class="text-align:center"></div>

<nav class="text-align:center"></nav>
```

# Naming conventions

You'll notice my very *on the nose* naming convention in the preceding example. My choice of naming for **utility classes** is covered in the **Measure** section. In short, the `property-name:value` structure is designed to help with recollection.

All `<center>` did, and all `text-align: center` does, is center-justify text. And for most content—especially content that includes paragraph text—you'll want to avoid it. It's terrible for readability ↗.

But what *would* be useful is a component that can create a horizontally centered column. With such a component, we could create a centered 'stripe' of content within any container, capping its width to preserve a <u>reasonable measure</u>.

## The solution

One of the easiest ways to solve for a centered column is to use `auto` margins. The `auto` keyword, as its name suggests, instructs the browser to calculate the margin for you. It's perhaps one of the most rudimentary examples of an *algorithmic* CSS technique: one that defers to the browser's logic to determine the layout rather than 'hard coding' a specific value.
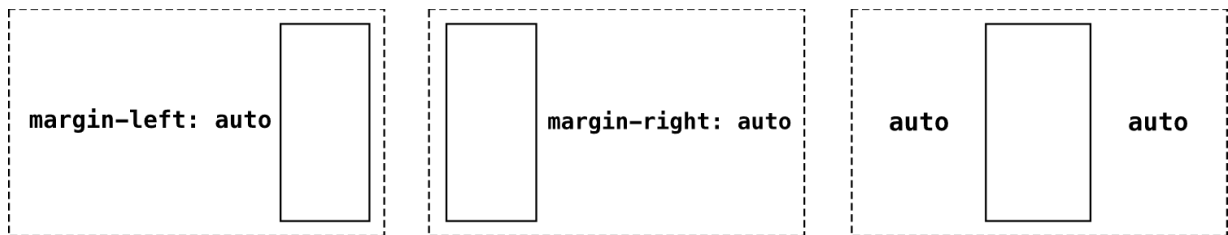


My first centered columns would use the `margin` shorthand, often on the `<body>` element.

```css
.center {
  max-width: 60ch;
  margin: 0 auto;
}
```

The trouble with the shorthand property—though it saves a few bytes—is that you have to declare certain values, even when they are not applicable. It's important to only set the CSS values needed to achieve the specific layout you are attempting. You never know what inferred or inherited values you might be undoing.

For example, I might want to place my <u><center-l></u> custom element within a **Stack** context. **Stack** sets `margin-top` on its children, and any `<center-l>` with `margin: 0 auto` would undo that.



```
* + * {
  margin-top: var(--s1);
}
```
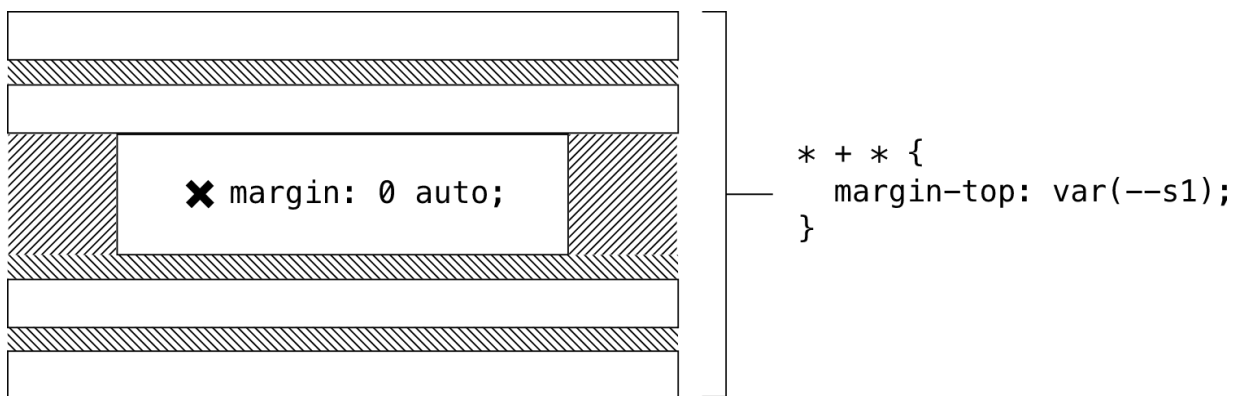
Instead, I could use the explicit `margin-left` and `margin-right` properties. Then, any vertical margins contextually applied would be preserved, and the `<center-l>` component would be primed for <u>composition</u>/nesting among other layout components.

```
.center {
  max-width: 60ch;
  margin-left: auto;
  margin-right: auto;
}
```

Even better, I could use a single `margin-inline` [logical property ↗](). As described in **Boxes**, logical properties pertain to direction and dimension mappings and are—as such—compatible with a wider range of languages. We are also using `max-inline-size` in place of `max-width`.

```
.center {
  max-inline-size: 60ch;
  margin-inline: auto;
}
```

# Measure

The `max-inline-size` should typically—as in the preceding code example—be set in `ch`, since achieving a reasonable measure is paramount. The **Axioms** section details how to set a reasonable measure.

# Minimum `margin`

In a context narrower than `60ch`, the contents will currently be flush with either side of the parent element or viewport. Rather than letting this happen, we should ensure a *minimum* space on either side.

I need to go about this in such a way that preserves centering, and the `60ch` maximum width. Since we can't enter `auto` into a calculation (like `calc(auto + 1rem)`), we should probably defer to padding.
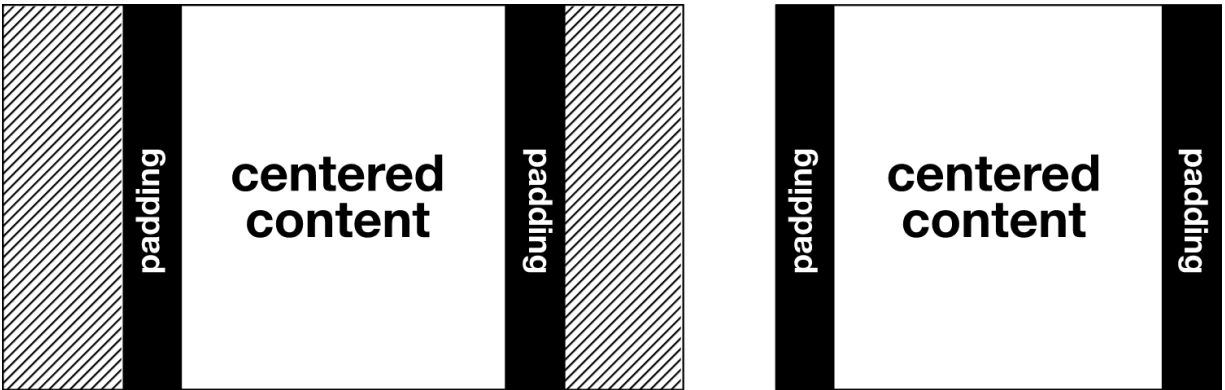
But I have to be wary of the box model. If, as suggested in **Boxes**, I have set all elements to adopt `box-sizing: border-box`, any padding added to my `<center-l>` will contribute to the `60ch` total. In other words, adding `padding` will make the *content* of my element narrower. However, as covered in **Axioms** CSS is designed for *exceptions*. I just need to override `border-box` with `content-box`, and allow the padding to 'grow out' from the `60ch` content threshold.
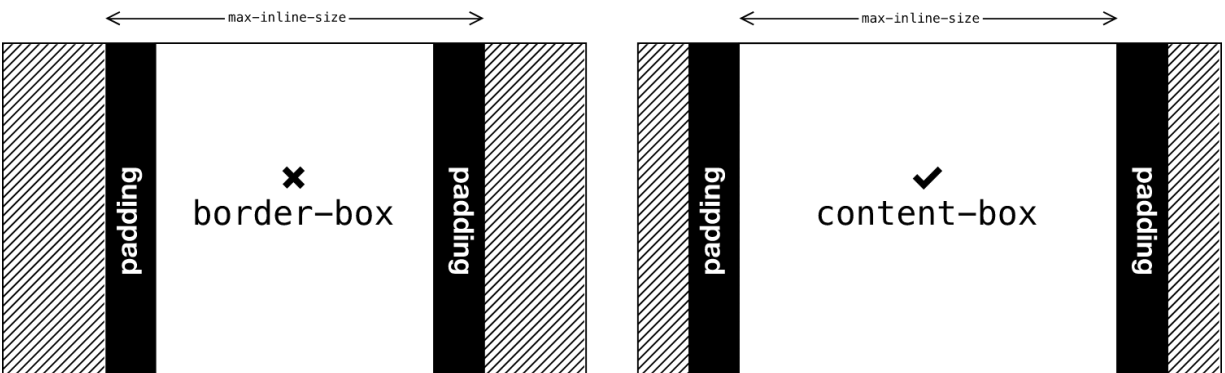


Here's a version that preserves the `60ch max-width`, but ensures there are, at least, `var(--s1)` "margins" on either side (`--s1` is the first point on the custom property-based modular scale).

```
.center {
  box-sizing: content-box;
  max-inline-size: 60ch;
  margin-inline: auto;
  padding-inline-start: var(--s1);
  padding-inline-end: var(--s1);
}
```

# Intrinsic centering

The `auto` margin solution is time-honoured and perfectly serviceable.
But there is an opportunity using the Flexbox layout module to
support *intrinsic* centering. That is, centering elements based on
their natural, content-based widths. Consider the following code.

```
.center {
  box-sizing: content-box;
  max-inline-size: 60ch;
  margin-inline: auto;
  display: flex;
  flex-direction: column;
  align-items: center;
}
```

Inside a `<center-l>` component, I would expect the contents to be
arranged vertically, as a column, hence `flex-direction: column`. This
allows me to set `align-items: center`, which will center any children
*regardless* of their width.

The upshot is any elements that are narrower than `60ch` will be
automatically centered within the `60ch`-wide area. These elements

can include naturally small elements like buttons, or elements with their own `max-width` set under `60ch`.



**Without intrinsic centering**          **With intrinsic centering**
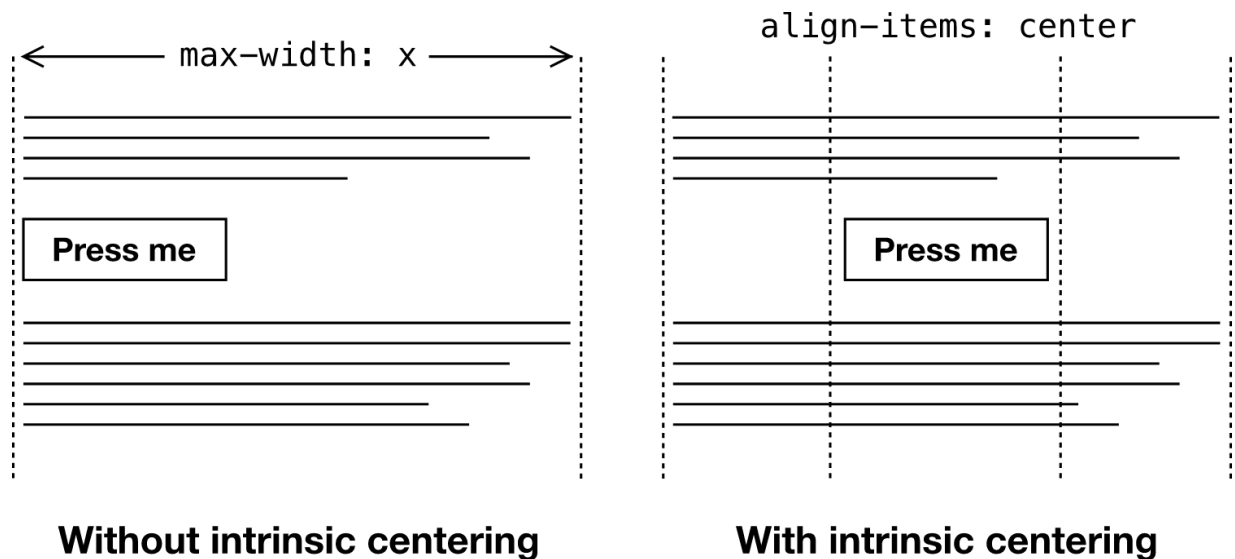
*The illustrated paragraphs are subject to `align-items: center`, but naturally take up all the available space (they are <u>block elements</u> with no set width).*

# ⚠ Accessibility

Be aware that, whenever you move content away from the left-hand edge (in a left-to-right writing direction), there's a potential accessibility issue. Where a user has zoomed the interface, it's possible the centered content will have moved out of the viewport. They may never realise it's there.

So long as your interface is flexible and responsive, and no fixed width is set on the container, the centered content should be visible in most circumstances.

# Use cases

Whenever you wish something to be horizontally centered, the **Center** is your friend. In the following example, I am emulating the basic layout for the **Every Layout** documentation site (which you may be looking at now, unless you're reading the EPUB). It comprises a **Sidebar**, with a **Center** to the right-hand side. Elements are vertically separated in both the sidebar and the **Center** using **Stacks**. A nested **Center** with the `intrinsic` boolean applied centers the 'Launch demo' button.

(You may need to launch it in its own (desktop) window to see the centering in action.)

This interactive demo is only available on the **Every Layout** site ↗.

# The Generator

Use this tool to generate basic **Center** CSS and HTML.

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments (omitting the intrinsic centering code):

# CSS

```css
.center {
  /* ↓ Remove padding from the width calculation */
  box-sizing: content-box;
  /* ↓ The maximum width is the maximum measure */
  max-inline-size: 60ch;
  /* ↓ Only affect horizontal margins */
  margin-inline: auto;
  /* ↓ Apply the minimum horizontal space */
  padding-inline-start: var(--s1);
  padding-inline-end: var(--s1);
}
```

## HTML

```html
<div class="center">
  <!-- centered content -->
</div>
```

# The Component

A custom element implementation of the Center is available for download ↗.

# Props API

The following props (attributes) will cause the **Center** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
| --- | --- | --- | --- |
| **max** | `string` | `"var(--measure)"` | A CSS `max-width` value |
| **andText** | `boolean` | `false` | Center align the text too (`text-align: center`) |
| **gutters** | `boolean` | `0` | The minimum space on either side of the content |
| **intrinsic** | `boolean` | `false` | Center child elements based on their content width |

# Examples

## Basic

You can create a single column web page just by nesting a **<u>Stack</u>** inside a **Center** inside a **<u>Box</u>**. The **Box** being padded by default means providing padding to either side of the **Center** using the `gutters` prop is not necessary.

```
<box-l>
  <center-l>
    <stack-l>
      <!-- Any flow content here (headings, paragraphs, etc) -->
    </stack-l>
  <center-l>
</box-l>
```

## Documentation layout

The markup from the example in Use cases. In the example, WAI-ARIA landmark roles have been added for screen reader support. Note that the **Center** has been wrapped in a generic `<div>` container. This `<div>` is subject to the **Sidebar's** layout logic, freeing the **Center** to apply its own logic inside it. The **Sidebar** wraps when this `<div>` starts to take up less that `66.666%` of the available horizontal space. Read **Sidebar** for a full explanation.

```
<sidebar-l contentMin="66.666%" sideWidth="10rem">
  <stack-l role="navigation">
    <!-- navigation items (API refs) -->
  </stack-l>
  <div>
    <center-l role="main">
      <!-- main content for the page -->
    </center-l>
  </div>
</sidebar-l>
```

## Vertical and horizontal centering

Using composition and the **Cover** component, it's trivial to horizontally *and* vertically center an element. The `intrinsic` boolean is used here to center the paragraph regardless of its content's width.

```
<cover-l centered="center-l">
  <center-l intrinsic>
    <p>I am in the absolute center.</p>
  </center-l>
</cover-l>
```

# The Cluster

## The problem

Sometimes <u>grids</u> are an appropriate framework for laying out content, because you want that content to align strictly to the horizontal and vertical lines that are those row and column boundaries.

But not everything benefits from this prescribed rigidity — at least not in all circumstances. Text itself cannot adhere to the strictures of a grid, because words come in different shapes and lengths. Instead, the browser's text wrapping algorithm distributes the text to fill the available space as best it can. Left-aligned text has a 'ragged' right edge, because each line will inevitably be of a different length.

Thanks to leading (`line-height`) and word spaces (the `U+0020` character, or a `SPACE` keypress to you), words can be reasonably evenly spaced, despite their diversity of form. Where we am dealing with groups of *elements* of an indeterminate size/shape, we should often like them to distribute in a similarly fluid way.

One approach is to set these elements' `display` value to `inline-block`. This gives you some control over `padding` and `margin` while retaining intrinsic sizing. That is, `inline-block` elements are still sized according to the dimensions of their content.

However, like words, `inline-block` elements are still separated by space characters (where present in the source). The width of this space will be added to any `margin` you apply. This space can be removed, but only by setting `font-size: 0` on the parent, and resetting the value on the children.

```css
.parent {
  font-size: 0;
}

.parent > * {
  font-size: 1rem;
}
```

This has the disadvantage that we can't use `em` on my child elements because it would be equal to `0`. Instead, we need to set the `font-size` relative to the `:root` element with the `rem` unit. Font size having to be reset in this fashion is somewhat restrictive.

Even with the space eliminated, there are still wrapping-related margin issues. If margin is applied to successive elements, the appearance is acceptable where *no* wrapping occurs. But where wrapping does occur, there are unexpected indents against the aligned side, and vertical spacing is missing entirely.
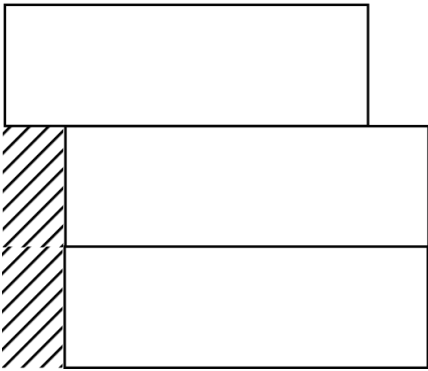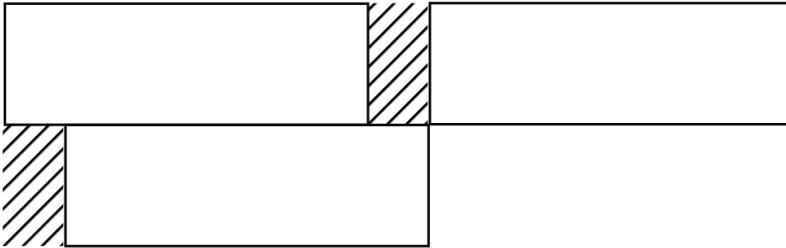
A partial fix is possible by placing right and bottom margins on each element.
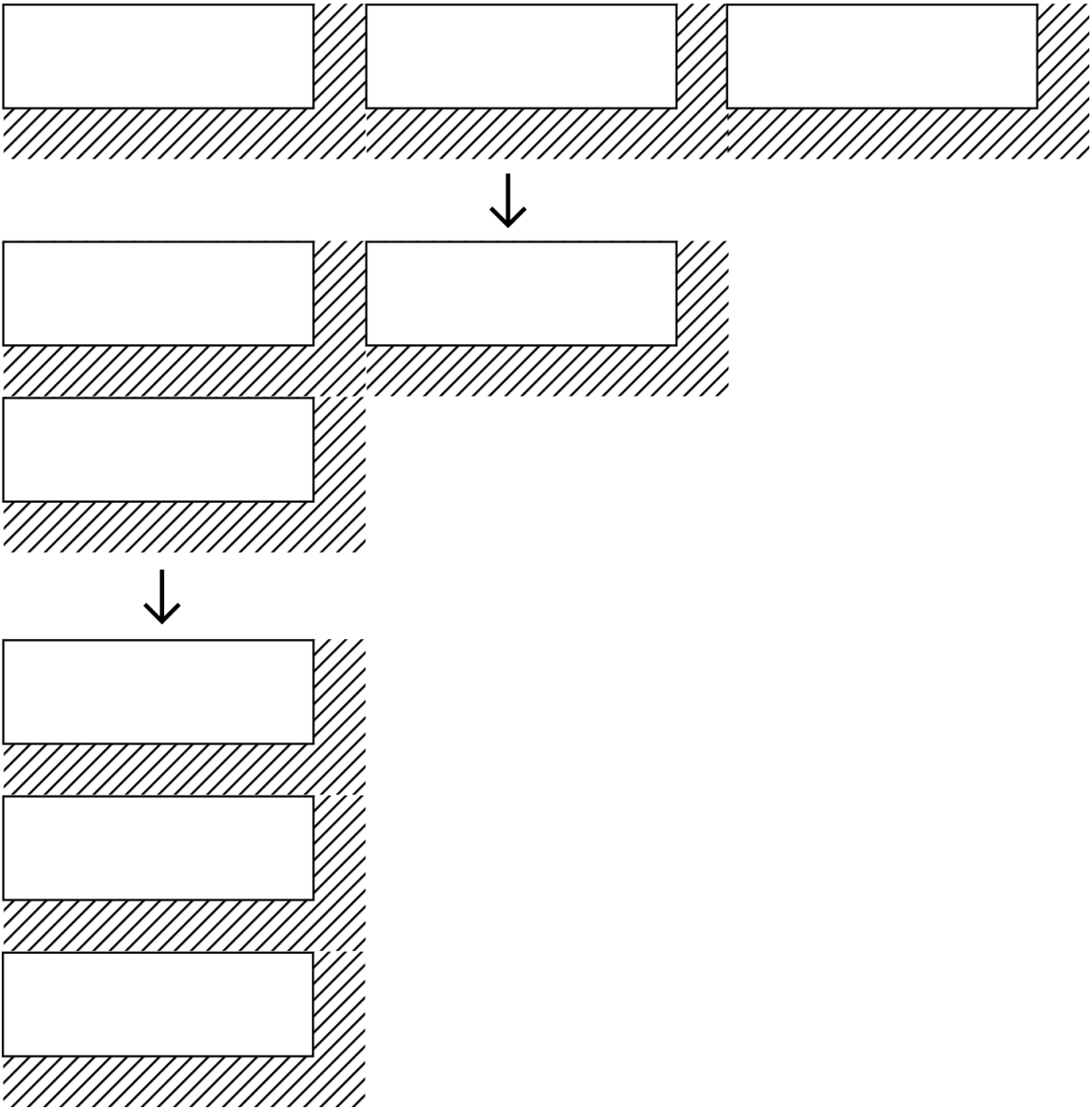
However, this only solves the left-aligned case — plus doubled-up space occurs where excess margin interacts with the padding of a parent element:
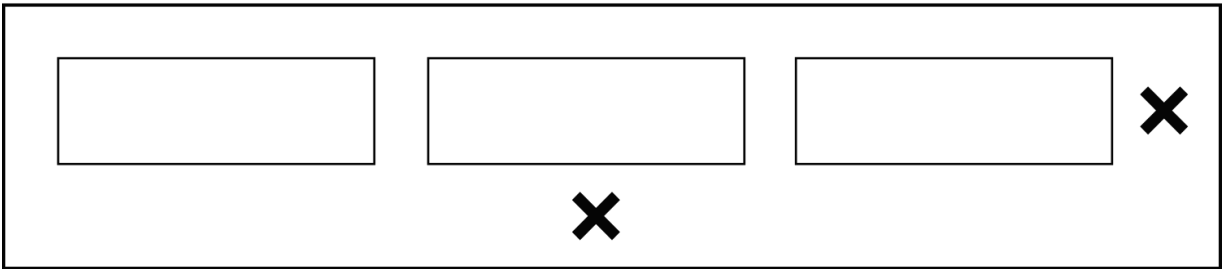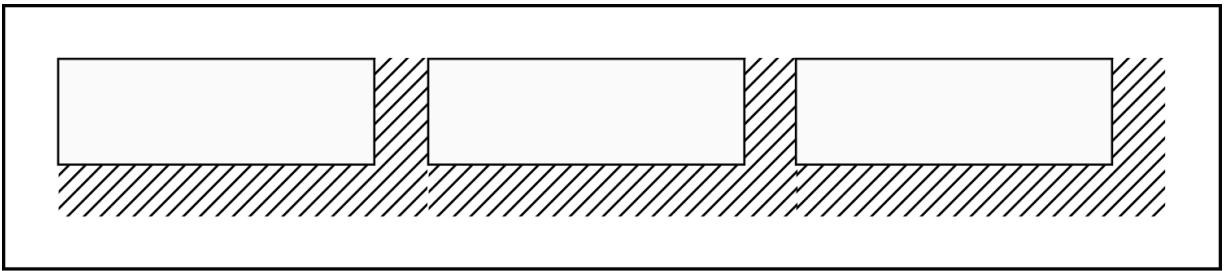
# The solution

To create an efficient and manageable design system, we need to devise robust, *general* solutions to our layout problems.

First, we make the parent a Flexbox context. This allows us to configure the elements into clusters, without having to deal with undesirable word spaces. It also has several advantages over using floats: we do not need to provide a clear fix ↗ for one, and vertical alignment (using `align-items`) is possible.

```css
.cluster {
  display: flex;
  flex-wrap: wrap;
}
```

# Adding and obscuring margin

The only way we can currently add margins that respect wrapping behaviour, irrespective of the alignment chosen, is to add them *symmetrically*; to all sides. Unfortunately, this separates the elements from any edge with which they come into contact.
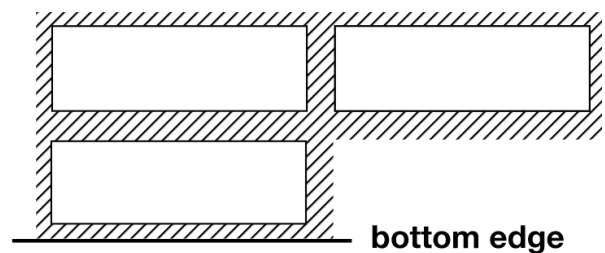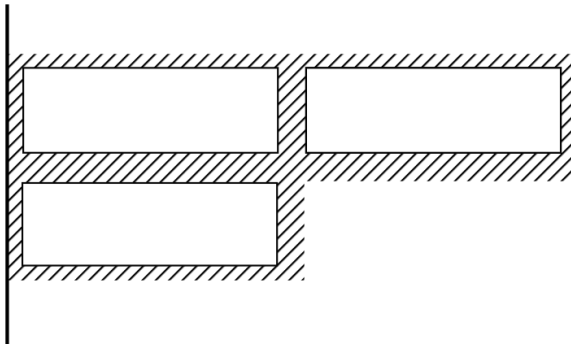


Note the value of the space between a child element and a parent element's edge is always *half* that of the space between two child elements (since their margins combine together). The solution is to use a negative margin on the parent to *pull* the children to its own edges:
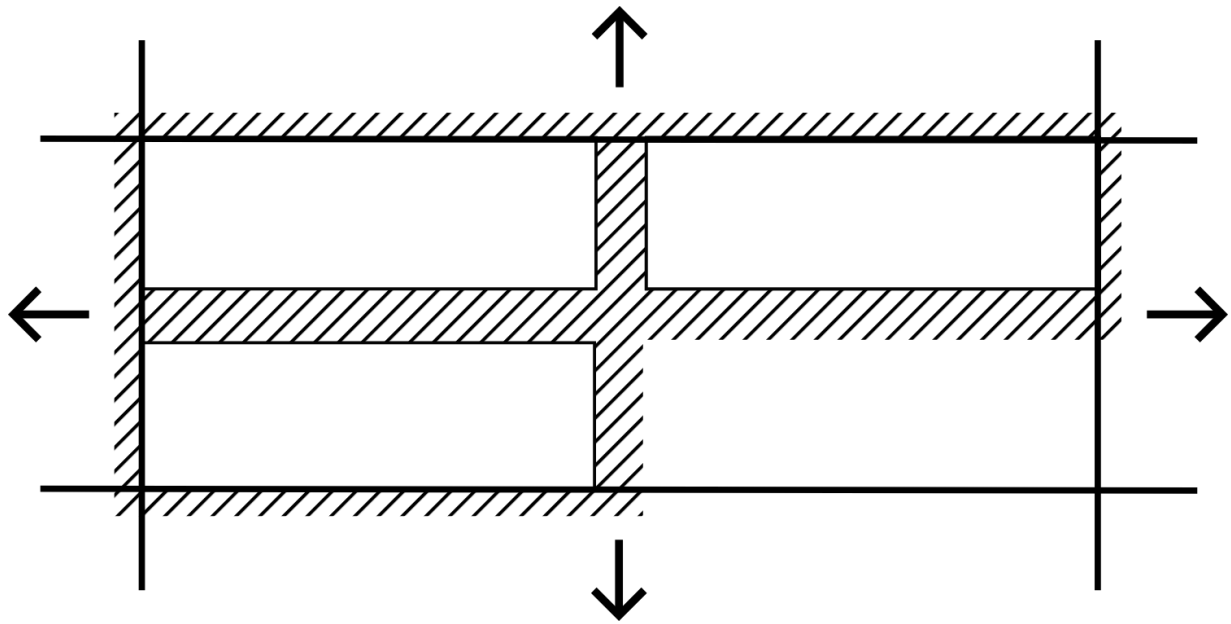
We can make authoring space in the **Cluster** component easier by using custom properties. The `--space` variable defines the desired spacing between elements, and `calc()` adapts this value accordingly. Note that a further wrapper element is included to *insulate* adjacent content from the negative margin. We still want the component to respect white space applied by a parent **Stack** component.

```
.cluster {
  --space: 1rem;
}

.cluster > * {
  display: flex;
  flex-wrap: wrap;
  /* ↓ multiply by -1 to negate the halved value */
  margin: calc(var(--space) / 2 * -1);
}

.cluster > * > * {
  /* ↓ half the value, because of the 'doubling up' */
  margin: calc(var(--space) / 2);
}
```

# The `gap` property

I think you'll agree the above technique is a bit unwieldy. It can also cause the horizontal scrollbar to appear, under some circumstances. Fortunately, as of mid-2021, <u>all major browsers now support the `gap` property with Flexbox ↗</u>. The `gap` property injects spacing *between* the child elements, doing away with the need for both negative margins and the additional wrapper element. Even the `calc()` can be retired, since the `gap` value is just that!

```
.cluster {
  display: flex;
  flex-wrap: wrap;
  gap: var(--space, 1rem);
}
```

# Fallback values

See how we're defining and declaring the `gap` value all in one line. The second argument to the `var()` function is the fallback value <u>for when the variable is otherwise undefined ↗</u>.

## Graceful degradation

Despite the reassuring support picture for `gap`, we should be mindful of the layout in browsers where it isn't supported. Problematically, `gap` may be supported for the Grid layout module (see **Grid**) but not for Flexbox, so using `gap` in a `@supports` block can give a false positive.

In browsers where `gap` is only supported for the Grid module, the following would lead to no margin *or* `gap` being applied.

```css
/* This won't work */
.cluster > * {
  display: flex;
  flex-wrap: wrap;
  margin: 1rem;
}

@supports (gap: 1rem) {
  .cluster > * {
    margin: 0;
  }

  .cluster {
    gap: var(--space, 1rem);
  }
}
```

As of today, we recommend using `gap` without feature detection, accepting that layouts will become *flush* in older browsers. We

include the negative margin technique above if that's your preference instead.

# Justification

Groups or *clusters* of elements can take any `justify-content` value, and the space/gap will be honored regardless of wrapping. Aligning the **Cluster** to the right would be a case for `justify-content: flex-end`.

In the demo to follow, a **Cluster** contains a list of linked keywords. This is placed inside a box with a `padding` value equal to that of the **Cluster's** space.

---

This interactive demo is only available on the **Every Layout** site ↗.

---

# Use cases

**Cluster** components suit any groups of elements that differ in length and are liable to wrap. Buttons that appear together at the end of forms are ideal candidates, as well as lists of tags, keywords, or other meta information. Use the **Cluster** to align any groups of horizontally laid out elements to the left or right, or in the center.

By applying `justify-content: space-between` and `align-items: center` you can even set out your page header's logo and navigation. This will wrap naturally, and without the need for an `@media` breakpoint:

*The navigation list will wrap below the logo at the point there is no room for its unwrapped content (its maximum width). This means we avoid the scenario where navigation links appear both beside and below the logo.*

Below is a demo of the aforementioned header layout, using a nested **Cluster** structure. The outer **Cluster** uses `justify-content: space-between` and `align-items: center`. The **Cluster** for the navigation links uses `justify-content: flex-start` to align its items to the left after wrapping.

> This interactive demo is only available on the **Every Layout** site ↗.

# The generator

Use this tool to generate basic **Cluster** CSS and HTML.

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments:

## CSS

```css
.cluster {
  /* ↓ Set the Flexbox context */
  display: flex;
  /* ↓ Enable wrapping */
  flex-wrap: wrap;
  /* ↓ Set the space/gap */
  gap: var(--space, 1rem);
  /* ↓ Choose your justification (flex-start is default) */
  justify-content: center;
  /* ↓ Choose your alignment (flex-start is default) */
  align-items: center;
}
```

## HTML

```html
<ul class="cluster">
  <li><!-- child --></li>
  <li><!-- child --></li>
  <li><!-- etc --></li>
</ul>
```

# The component

A custom element implementation of the Cluster is available for download ↗.

# Props API

The following props (attributes) will cause the **Cluster** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
| --- | --- | --- | --- |
| **justify** | string | "flex-start" | A CSS `justify-content` value |
| **align** | string | "flex-start" | A CSS `align-items` value |
| **space** | string | "var(--s1)" | A CSS `gap` value. The minimum space between the clustered child elements. |

# Examples

## Basic

Using the defaults.

```
<cluster-l>
  <!-- child element here -->
  <!-- another child element -->
  <!-- etc -->
  <!-- etc -->
  <!-- etc -->
  <!-- etc -->
</cluster-l>
```

## List

Since **Clusters** typically represent groups of similar elements, they benefit from being marked up as a list. List elements present information non-visually, to screen reader software. It's important screen reader users are aware there *is* a list present, and how many items it contains.

Since our custom element `<cluster-l>` is not a `<ul>` (and `<li>`
elements cannot exist without a `<ul>` parent) we can provide the list
semantics using ARIA instead: `role="list"` and `role="listitem"`:

```
<cluster-l role="list">
  <div role="listitem"><!-- content of first list item --></div>
  <div role="listitem"><!-- content of second list item --></div>
  <div role="listitem"><!-- etc --></div>
  <div role="listitem"><!-- etc --></div>
</cluster-l>
```

# The Sidebar

## The problem

When the dimensions and settings of the medium for your visual design are indeterminate, even something simple like *putting things next to other things* is a quandary. Will there be enough horizontal space? And, even if there is, will the layout make the most of the *vertical* space?
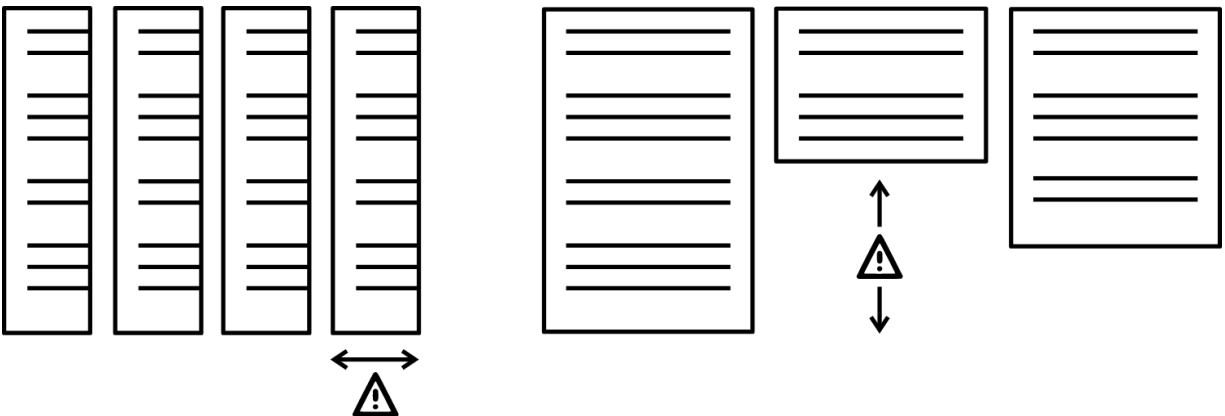
Where there's not enough space for two adjacent items, we tend to employ a breakpoint (a width-based `@media` query) to reconfigure the

layout, and place the two items one atop the other.

It's important we use *content* rather than *device* based `@media` queries. That is, we should intervene anywhere the content needs reconfiguration, rather than adhering to arbitrary widths like `720px` and `1024px`. The massive proliferation of devices means there's no real set of standard dimensions to design for.

But even this strategy has a fundamental shortcoming: `@media` queries for width pertain to the *viewport* width, and have no bearing on the actual available space. A component might appear within a `300px` wide container, or it might appear within a more generous `500px` wide container. But the width of the viewport is the same in either case, so there's nothing to "respond" to.
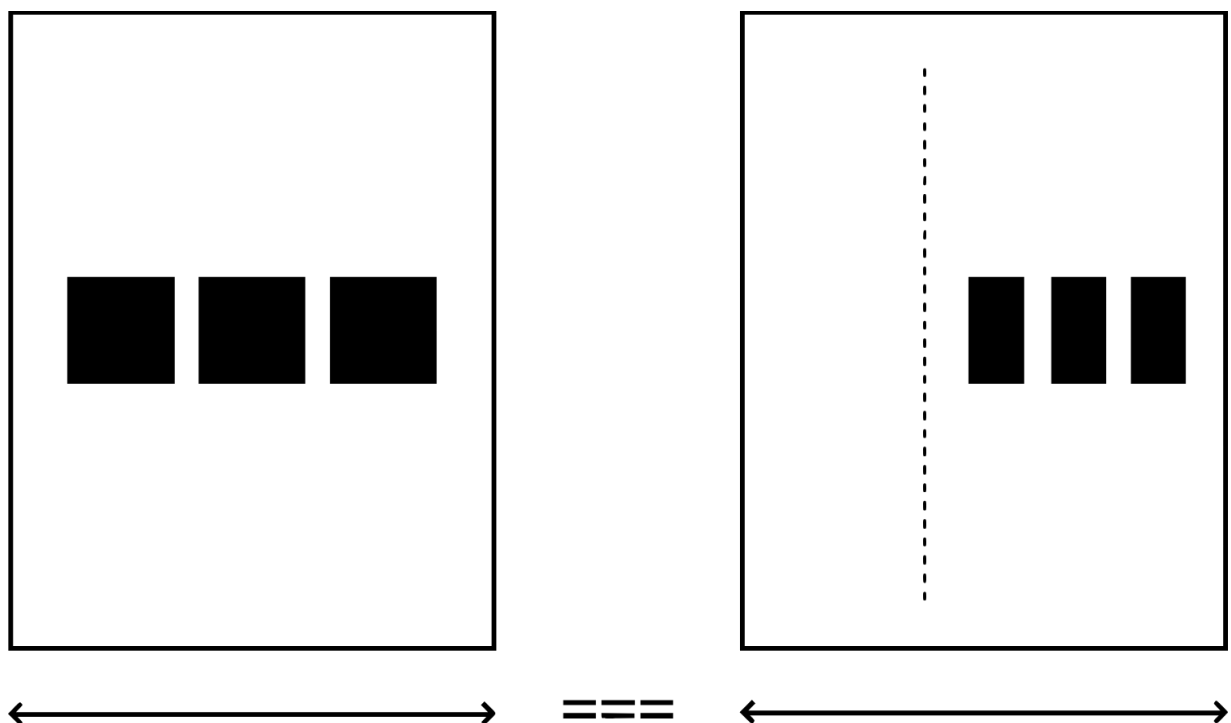
Design systems tend to catalogue components that can appear between different contexts and spaces, so this is a real problem. Only with a capability like the mooted <u>container queries ↗</u> might we teach our component layouts to be fully *context aware*.

In some respects, the CSS Flexbox module, with its provision of `flex-basis`, can already govern its own layout, per context, rather well. Consider the following code:

```
.parent {
  display: flex;
  flex-wrap: wrap;
}

.parent > * {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 30ch;
}
```

The `flex-basis` value essentially determines an *ideal* target width for the subject child elements. With growing, shrinking, and wrapping enabled, the available space is used up such that each element is as *close* to `30ch` wide as possible. In a `> 90ch` wide container, more than three children may appear per row. Between `60ch` and `90ch` only two items can appear, with one item taking up the whole of the final row (if the total number is odd).
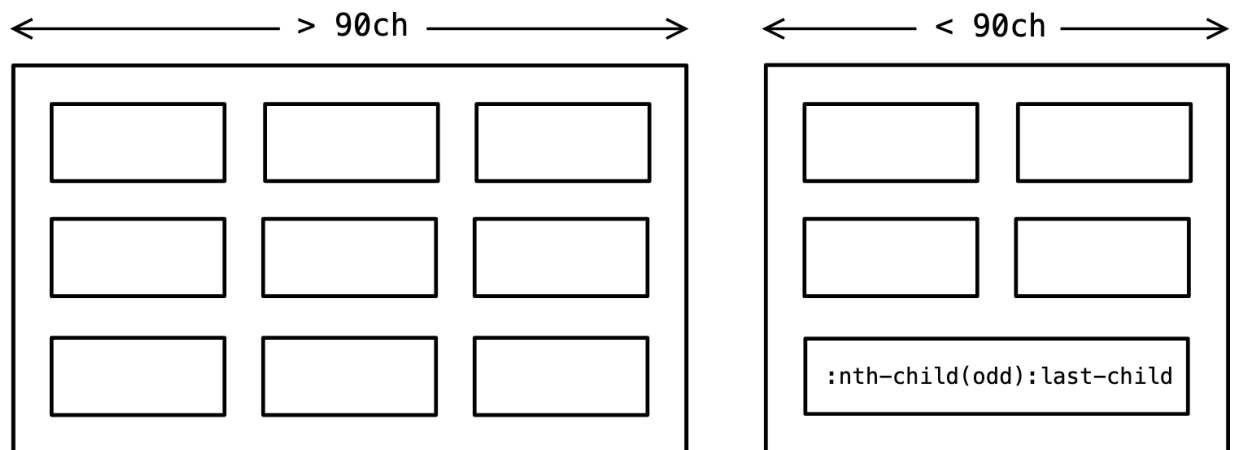
*An item with an odd index, which is also the last item, can be expressed by concatenating two pseudo selectors: `:nth-child(odd):last-child`*

By designing to *ideal* element dimensions, and tolerating reasonable variance, you can essentially do away with `@media` breakpoints. Your component handles its own layout, intrinsically, and without the need for manual intervention. Many of the layouts we're covering finesse this basic mechanism to give you more precise control over placement and wrapping.

For instance, we might want to create a classic sidebar layout, wherein one of two adjacent elements has a fixed width, and the other—the *principle* element, if you will—takes up the rest of the available space. This should be responsive, without `@media` breakpoints, and we should be able to set a *container* based breakpoint for wrapping the elements into a vertical configuration.

# The solution

The **Sidebar** layout is named for the element that forms the diminutive *sidebar*: the narrower of two adjacent elements. It is a *quantum* layout, existing simultaneously in one of the two configurations—horizontal and vertical—illustrated below. Which configuration is adopted is not known at the time of conception, and is dependent entirely on the space it is afforded when placed within a parent container.

**wide context**          **narrow context**

Where there is enough space, the two elements appear side-by-side. Critically, the sidebar's width is *fixed* while the two elements are adjacent, and the non-sidebar takes up the rest of the available space. But when the two elements wrap, *each* takes up `100%` of the shared container.

## Equal height

Note the two adjacent elements are the same height, regardless of the content they contain. This is thanks to a default `align-items`

value of `stretch`. In most cases, this is desirable (and was very difficult to achieve before the advent of Flexbox). However, you can "switch off" this behavior with `align-items: flex-start`.
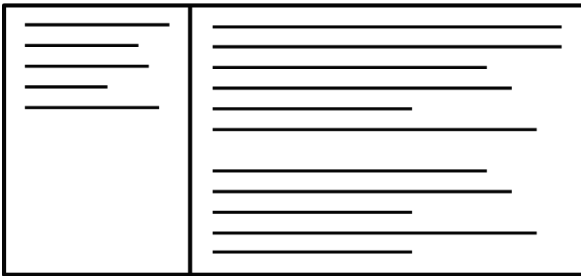


align-items: stretch              align-items: flex-start
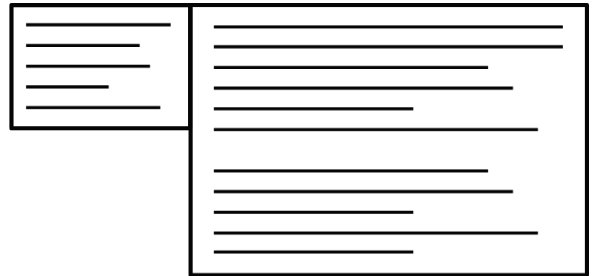
How to force wrapping at a certain point, we will come to shortly. First, we need to set up the horizontal layout.

```
.with-sidebar {
  display: flex;
  flex-wrap: wrap;
}

.sidebar {
  flex-basis: 20rem;
  flex-grow: 1;
}

.not-sidebar {
  flex-basis: 0;
  flex-grow: 999;
}
```

The key thing to understand here is the role of *available space*. Because the `.not-sidebar` element's `flex-grow` value is so high (`999`), it takes up all the available space. The `flex-basis` value of the `.sidebar` element is not counted as available space and is

subtracted from the total, hence the sidebar-like layout. The non-sidebar essentially squashes the sidebar down to its ideal width.

$$\longleftarrow n \longrightarrow$$

←
available space
(space - n)
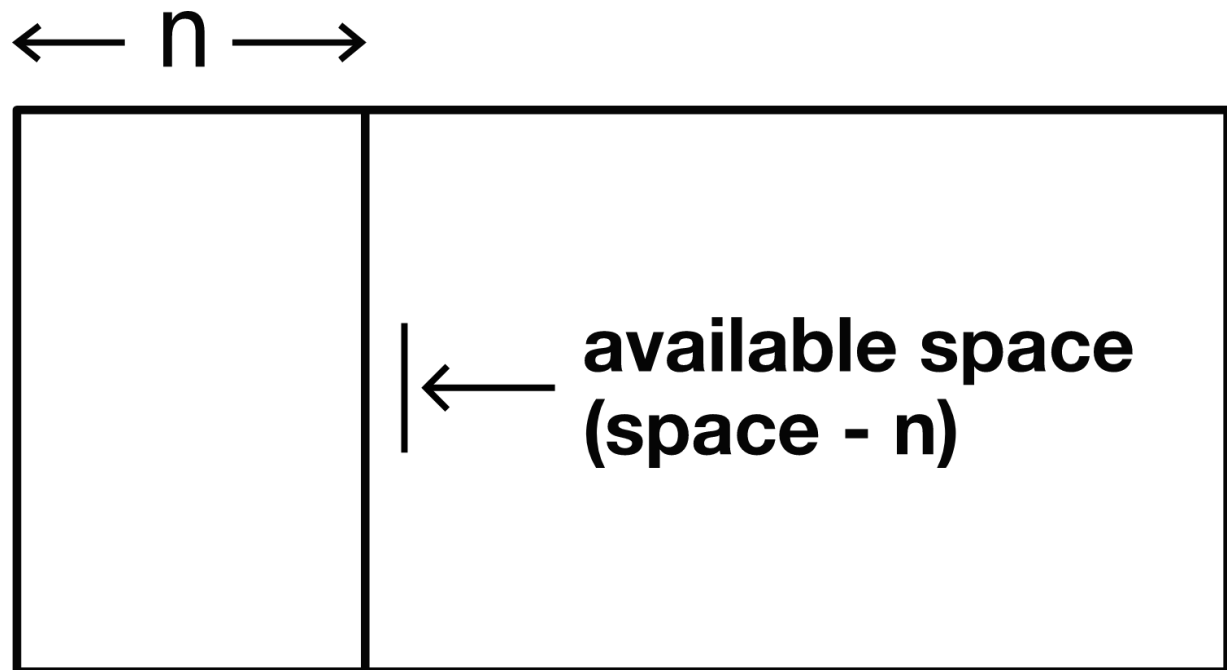
The `.sidebar` element is still technically allowed to grow, and is able to do so where `.not-sidebar` wraps beneath it. To control where that wrapping happens, we can use `min-inline-size`, which is equivalent to `min-width` in the default `horizontal-tb` writing mode.

```
.not-sidebar {
  flex-basis: 0;
  flex-grow: 999;
  min-inline-size: 50%;
}
```

Where `.not-sidebar` is destined to be less than or equal to `50%` of the container's width, it is forced onto a new line/row and grows to take

up all of its space. The value can be anything, but `50%` is apt since a sidebar ceases to be a sidebar when it is no longer the narrower of the two elements.



**(should have wrapped)**
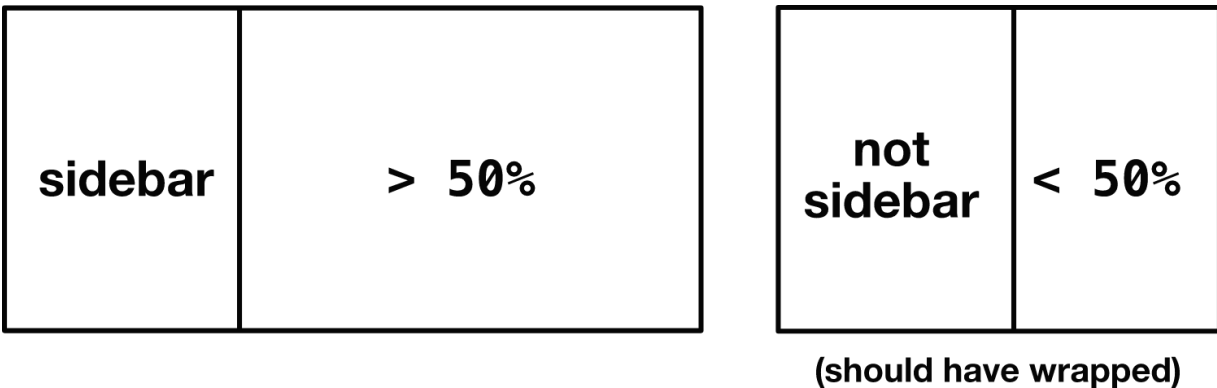
# The gutter

So far, we're treating the two elements as if they're touching. Instead, we might want to place a gutter/space between them. Since we want that space to appear between the elements regardless of the configuration and we *don't* want there to be extraneous margins on the outer edges, we'll use the `gap` property as we did for the **Cluster** layout.

For a gutter of `1rem`, the CSS now looks like the following.

```
.with-sidebar {
  display: flex;
  flex-wrap: wrap;
  gap: 1rem;
}

.sidebar {
  /* ↓ The width when the sidebar _is_ a sidebar */
  flex-basis: 20rem;
  flex-grow: 1;
}

.not-sidebar {
  /* ↓ Grow from nothing */
  flex-basis: 0;
  flex-grow: 999;
  /* ↓ Wrap when the elements are of equal width */
  min-inline-size: 50%;
}
```

This interactive demo is only available on the **Every Layout** site ↗.

# Intrinsic sidebar width

So far, we have been prescribing the width of our sidebar element (`flex-basis: 20rem`, in the last example). Instead, we might want to let the sidebar's *content* determine its width. Where we do not provide a `flex-basis` value at all, the sidebar's width is equal to the width of its contents. The wrapping behavior remains the same.
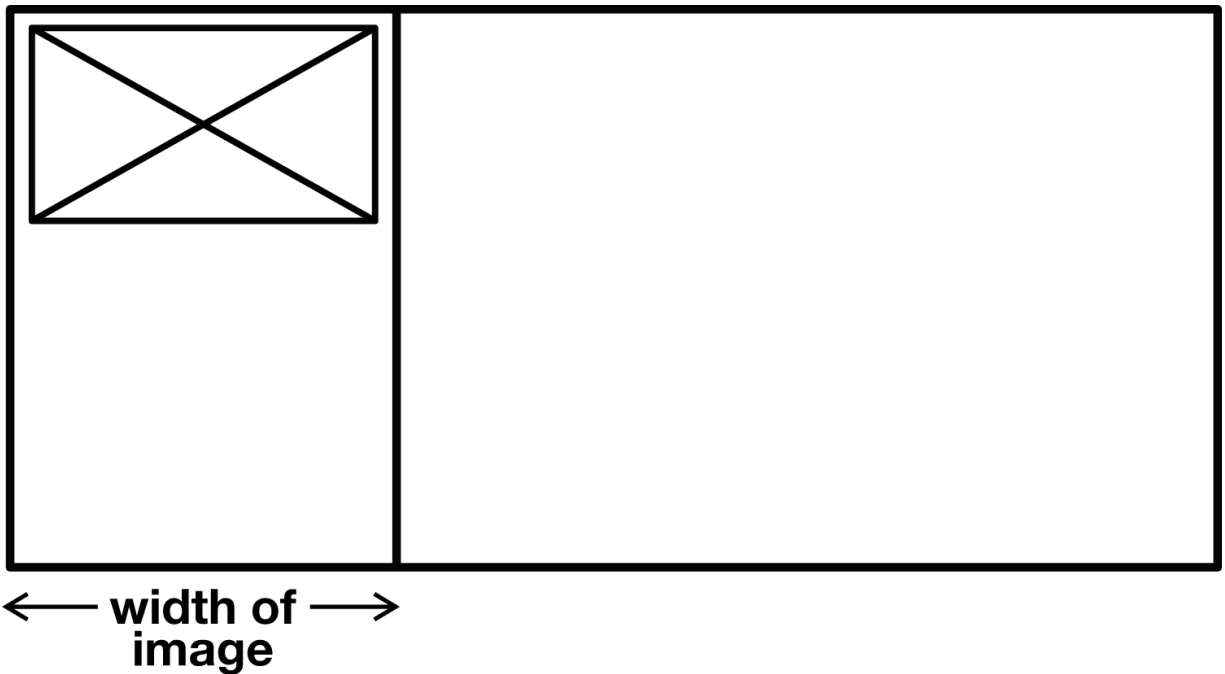
**←— width of —→**
**image**

If we set the width of an image inside of our sidebar to `15rem`, that will be the width of the sidebar in the horizontal configuration. It will grow to `100%` in the vertical configuration.

# Intrinsic web design

The term _Intrinsic Web Design_ ↗ was coined by Jen Simmons, and refers to a recent move towards tools and mechanisms in CSS that are more befitting of the medium. The kind of _algorithmic_, self-governing layouts set out in this series might be considered intrinsic design methods.

The term _intrinsic_ connotes introspective processes; calculations made by the layout pattern about itself. My use of 'intrinsic' in this section specifically refers to the inevitable width of an element as

determined by its contents. A button's width, unless explicitly set, is the width of what's inside it.

The CSS Box Sizing Module was formerly called the Intrinsic & Extrinsic Sizing Module, because it set out how elements can be sized both intrinsically and extrinsically. Generally, we should err on the side of intrinsic sizing. As covered in **Axioms**, we're better allowing the browser to size elements according to their content, and only provide *suggestions*, rather than *prescriptions*, for layout. We are *outsiders*.

# Use cases

The **Sidebar** is applicable to all sorts of content. The ubiquitous "media object" (the placing of an item of media next to a description) is a mainstay, but it can also be used to align buttons with form inputs (where the button forms the sidebar and has an *intrinsic*, content-based width).

The following example uses the component version, defined as a custom element.

```
<form>
  <sidebar-l side="right" space="0" contentMin="66.666%">
    <input type="text">
    <button>Search</button>
  </sidebar-l>
</form>
```

# The generator

Use this tool to generate basic **Sidebar** CSS and HTML.

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments. It is assumed the *non*-sidebar is the :last-child in this example.

# CSS

```
.with-sidebar {
  display: flex;
  flex-wrap: wrap;
  /* ↓ The default value is the first point on the modular scale */
  gap: var(--gutter, var(--s1));
}

.with-sidebar > :first-child {
  /* ↓ The width when the sidebar _is_ a sidebar */
  flex-basis: 20rem;
  flex-grow: 1;
}

.with-sidebar > :last-child {
  /* ↓ Grow from nothing */
  flex-basis: 0;
  flex-grow: 999;
  /* ↓ Wrap when the elements are of equal width */
  min-inline-size: 50%;
}
```

# HTML

(You don't *have* to use `<div>`s; use semantic elements where appropriate.)

```
<div class="with-sidebar">
  <div><!-- sidebar --></div>
  <div><!-- non-sidebar --></div>
</div>
```

# The component

A custom element implementation of the Sidebar is available for download ↗.

# Props API

The following props (attributes) will cause the **Sidebar** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|---|---|---|---|
| **side** | string | `"left"` | Which element to treat as the sidebar (all values but "left" are considered "right") |
| **sideWidth** | string |  | Represents the width of the sidebar *when* adjacent. If not set (`null`) it defaults to the sidebar's content width |
| **contentMin** | string | `"50%"` | A CSS **percentage** value. The minimum width of the content element in the horizontal configuration |
| **space** | string | `"var(--s1)"` | A CSS margin value representing the space between the two elements |
| **noStretch** | boolean | `false` | Make the adjacent elements adopt their natural height |

# Examples

## Media object

Uses the default `50%` "breakpoint" and an increased `space` value, taken from the custom property-based <u>modular scale</u>. The sidebar/image is `15rem` wide in the horizontal configuration.

Because the image is a flex child, `noStretch` must be supplied, to stop it distorting. If the image was placed inside a `<div>` (making the `<div>` the flex child) this would not be necessary.

```
<sidebar-l space="var(--s2)" sideWidth="15rem" noStretch>
  <img src="path/to/image" alt="Description of image" />
  <p><!-- the text accompanying the image --></p>
</sidebar-l>
```

## Switched media object

The same as the last example, except the text *accompanying* the
image is the sidebar (`side="right"`), allowing the image to grow when
the layout is in the horizontal configuration. The `<p>` sidebar has a
width (<u>measure ↗</u>) of `30ch` (approximately 30 characters) in the
horizontal configuration.

The image is contained in `<div>`, meaning `noStretch` is not necessary
in this case. The image should grow to use up the available space,
so the basic CSS for responsive images should be in your global
styles (`img { max-width: 100% }`).

```
<sidebar-l space="var(--s2)" side="right" sideWidth="30ch">
  <div>
    <image src="path/to/image" alt="Description of image">
  </div>
  <p><!-- the text accompanying the image --></p>
</sidebar-l>
```

# The Switcher

As we set out in **Boxes**, it's better to provide *suggestions* rather than diktats about the way the visual design is laid out. An overuse of `@media` breakpoints can easily come about when we try to *fix* designs to different contexts and devices. By only suggesting to the browser how it should arrange our layout boxes, we move from creating multiple layouts to single *quantum* layouts existing simultaneously in different states.

The `flex-basis` property is an especially useful tool when adopting such an approach. A declaration of `width: 20rem` means just that: make it `20rem` wide — regardless of circumstance. But `flex-basis: 20rem` is more nuanced. It tells the browser to consider `20rem` as an ideal or "target" width. It is then free to calculate just how closely the `20rem` target can be resembled given the content and available space. You empower the browser to make the right decision for the content, and the user, reading that content, given their circumstances.

Consider the following code.

```
.grid {
  display: flex;
  flex-wrap: wrap;
}

.grid > * {
  width: 33.333%;
}

@media (max-width: 60rem) {
  .grid > * {
    width: 50%;
  }
}

@media (max-width: 30rem) {
  .grid > * {
    width: 100%;
  }
}
```

The mistake here (aside from not using the logical property `inline-size` in place of `width`) is to adopt an *extrinsic* approach to the layout: we are thinking about the viewport first, then adapting our boxes to it. It's verbose, unreliable, and doesn't make the most of Flexbox's capabilities.

With `flex-basis`, it's easy to make a responsive Grid-like layout which is in no need of `@media` breakpoint intervention. Consider this alternative code:

```
.grid {
  display: flex;
  flex-wrap: wrap;
}

.grid > * {
  flex: 1 1 20rem;
}
```

Now I'm thinking *intrinsically* — in terms of the subject elements' own dimensions. That `flex` shorthand property ↗ translates to "*let each element grow and shrink to fill the space, but try to make it about `20rem` wide*". Instead of manually pairing the column count to the viewport width, I'm telling the browser to *generate* the columns based on my desired column width. I've automated my layout.

As Zoe Mickley Gillenwater ↗ has pointed out, `flex-basis`, in combination with `flex-grow` and `flex-shrink`, achieves something similar to an element/container query ↗ in that "breaks" occur, implicitly, according to the available space rather than the viewport width. My Flexbox "grid" will automatically adopt a different layout depending on the size of the container in which it is placed. Hence: *quantum layout*.

# Issues with two-dimensional symmetry

While this is a serviceable layout mechanism, it only produces two layouts wherein each element is the same width:

- The single-column layout (given the narrowest of containers)
- The regular multi-column layout (where each row has an equal number of columns)

In other cases, the number of elements and the available space conspire to make layouts like these:
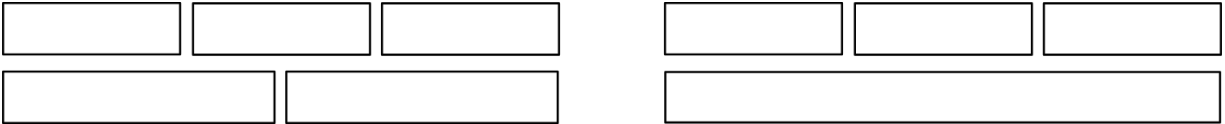
This is not *necessarily* a problem that needs to be solved, depending on the brief. So long as the content configures itself to remain in the space, unobscured, the most important battle has been won. However, for smaller numbers of subject elements, there may be cases where you wish to switch *directly* from a horizontal (one row) to a vertical (one column) layout and bypass the intermediary states.

Any element that has wrapped and grown to adopt a different width could be perceived by the user as being "picked out"; made to deliberately look different, or more important. We should want to avoid this confusion.
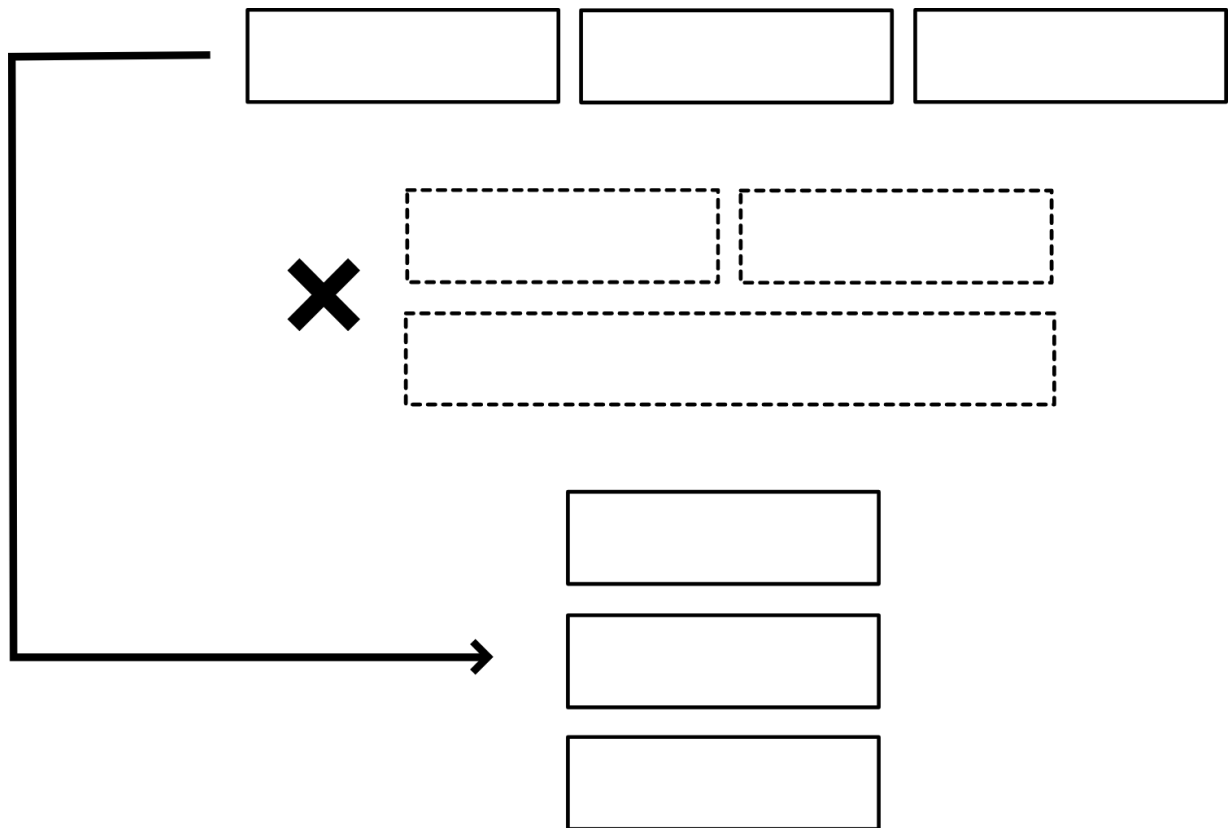
# The solution

The **Switcher** element (based on the bizarrely named <u>Flexbox Holy Albatross ↗</u>) switches a Flexbox context between a horizontal and a vertical layout at a given, *container*-based breakpoint. That is, if the breakpoint is `30rem`, the layout will switch to a vertical configuration when the parent element is less than `30rem` wide.

In order to achieve this switch, first a basic horizontal layout is instated, with wrapping and `flex-grow` enabled:

```
.switcher > * {
  display: flex;
  flex-wrap: wrap;
}

.switcher > * > * {
  flex-grow: 1;
}
```

The `flex-basis` value enters the (current) width of the container, expressed as `100%`, into a calculation with the designated `30rem` breakpoint.

```
30rem - 100%
```

Depending on the parsed value of `100%`, this will return either a *positive* or *negative* value: positive if the container is narrower than `30rem`, or negative if it is wider. This number is then multiplied by `999` to produce either a *very large* positive number or a *very large* negative number:

```
(30rem - 100%) * 999
```

Here is the `flex-basis` declaration in situ:

```
.switcher > * {
  display: flex;
  flex-wrap: wrap;
}

.switcher > * > * {
  flex-grow: 1;
  flex-basis: calc((30rem - 100%) * 999);
}
```

A negative `flex-basis` value is invalid, and dropped. Thanks to CSS's resilient error handling this means just the `flex-basis` line is ignored, and the rest of the CSS is still applied. The erroneous negative `flex-basis` value is corrected to `0` and—because `flex-grow` is present—each element grows to take up an equal proportion of horizontal space.

## ⚠ Content width

The previous statement, "*each element grows to take up an equal proportion of the horizontal space*" is true where the *content* of any one element does not exceed that alloted proportion. To keep things in order, nested elements can be given a `max-inline-size` of `100%`.

```
┌─────────────────────────────────────┐  ┌──────────────────┐  ┌──────────────────┐
│  ┌───────────────────────────────┐  │  │                  │  │                  │
│  │      inline-size: 30rem       │  │  │                  │  │                  │
│  └───────────────────────────────┘  │  │                  │  │                  │
│                                     │  │                  │  │                  │
└─────────────────────────────────────┘  └──────────────────┘  └──────────────────┘
```
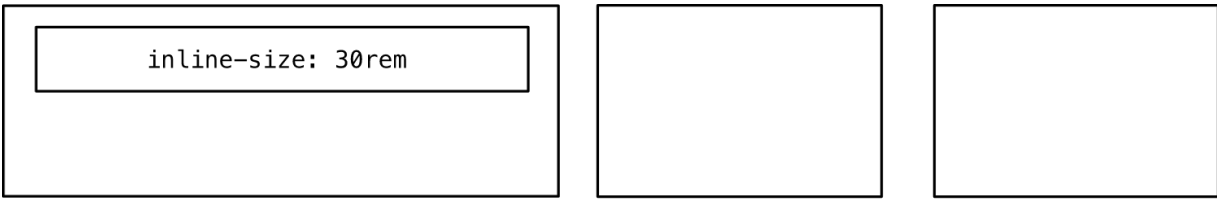
**Returned to equal width**

```
┌─────────────────────────────────────┐  ┌──────────────────┐  ┌──────────────────┐
│  ┌───────────────────────────────┐  │  │                  │  │                  │
│  │     max-inline-size: 100%     │  │  │                  │  │                  │
│  └───────────────────────────────┘  │  │                  │  │                  │
│                                     │  │                  │  │                  │
└─────────────────────────────────────┘  └──────────────────┘  └──────────────────┘
```
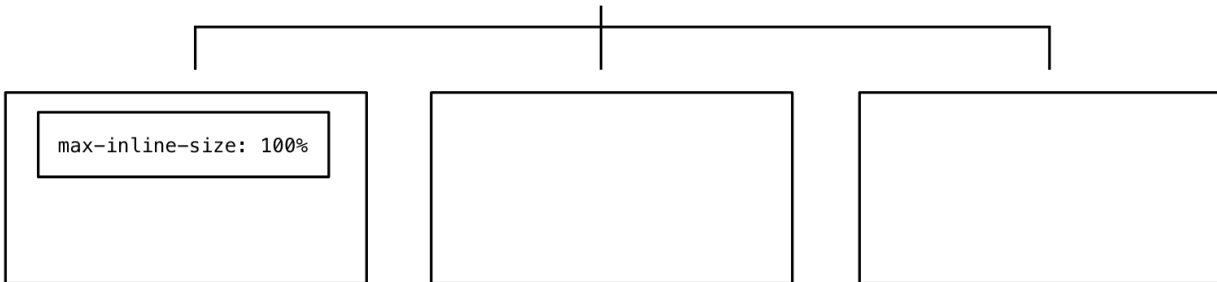
*As ever, setting fixed widths (or even `min-width`s) can be problematic. Instead, width should be suggested or inferred from context.*

If, on the other hand, the calculated `flex-basis` value is a large positive number, each element *maxes out* to take up a whole row. This results in the vertical configuration. Intermediary configurations are successfully bypassed.

## flex-basis

-n * 999                    n * 999
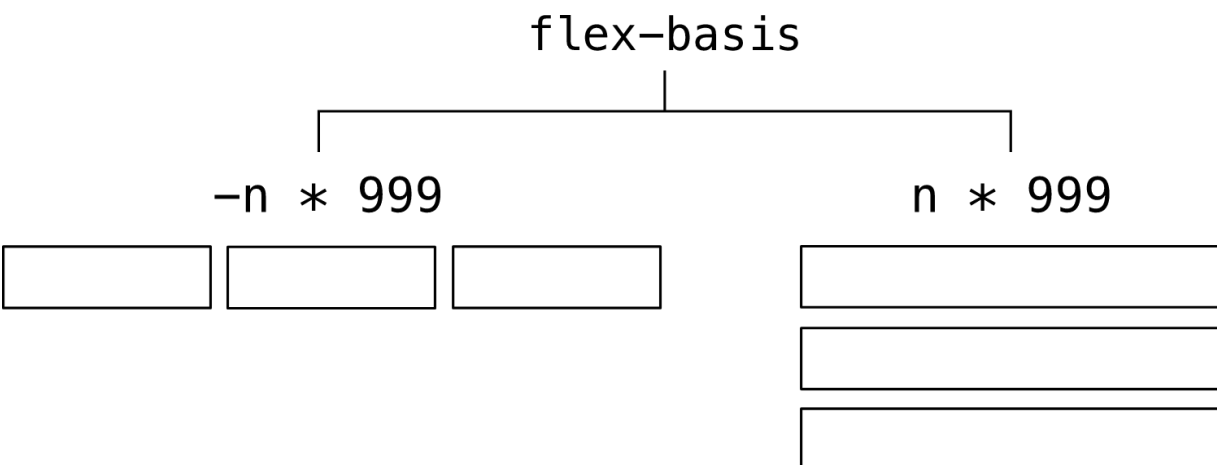
# Gutters

To support margins ('gutters'; 'gaps') between the subject elements, we could adapt the <u>negative margin technique covered in the</u> **<u>Cluster</u>** <u>documentation</u>. However, the `flex-basis` calculation would need to be adapted to compensate for the increased width produced by *stretching* the parent element. That is, by applying negative margins on all sides, the parent becomes wider than *its* container and their `100%` values no longer match.

```
.switcher {
  --threshold: 30rem;
  --space: 1rem;
}

.switcher > * {
  display: flex;
  flex-wrap: wrap;
  /* ↓ Multiply by -1 to make negative */
  margin: calc(var(--space) / 2 * -1);
}

.switcher > * > * {
  flex-grow: 1;
  flex-basis: calc((var(--threshold) - (100% - var(--space))) * 999);
  /* ↓ Half the value to each element, combining to make the whole */
  margin: calc(var(--space) / 2);
}
```

Instead, since `gap` is now supported in all major browsers, we don't have to worry about such calculations any more. The `gap` property represents the browser making such calculations for us. And it allows us to cut both the HTML and CSS code down quite a bit.
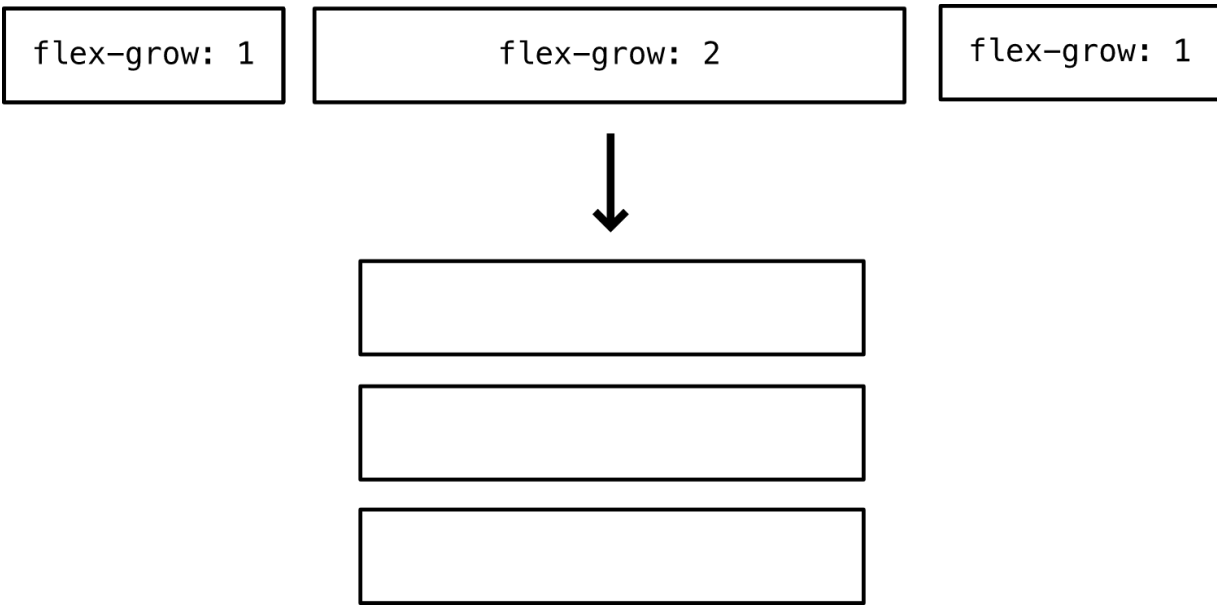
```
.switcher {
  display: flex;
  flex-wrap: wrap;
  gap: 1rem;
  --threshold: 30rem;
}

.switcher > * {
  flex-grow: 1;
  flex-basis: calc((var(--threshold) - 100%) * 999);
}
```

This interactive demo is only available on the **Every Layout** site ↗.

# Managing proportions

There is no reason why one or more of the elements, when in a horizontal configuration, cannot be alloted more or less of the available space. By giving the second element (`:nth-child(2)`) `flex-grow: 2` is will become twice as wide as its siblings (and the siblings will shrink to compensate).

```
.switcher > :nth-child(2) {
  flex-grow: 2;
}
```

```
┌─────────────────┐  ┌──────────────────────────────┐  ┌─────────────────┐
│  flex-grow: 1   │  │         flex-grow: 2         │  │  flex-grow: 1   │
└─────────────────┘  └──────────────────────────────┘  └─────────────────┘
```

↓

```
┌──────────────────────────────┐
│                              │
└──────────────────────────────┘

┌──────────────────────────────┐
│                              │
└──────────────────────────────┘

┌──────────────────────────────┐
│                              │
└──────────────────────────────┘
```

## Quantity threshold

In the horizontal configuration, the amount of space alloted each element is determined by two things:

- The total space available (the width of the container)
- The number of sibling elements

So far, my **Switcher** *switches* according to the available space. But we can add as many elements as we like, and they will lay out together horizontally above the breakpoint (or *threshold*). The more elements we add, the less space each gets alloted, and things can easily start to get squashed up.

This is something that could be addressed in documentation, or by providing warning or error messages in the developer's console. But that isn't very efficient or robust. Better to *teach* the layout to handle

this problem itself. The aim for each of the layouts in this project is to make them as self-governing as possible.

It is quite possible to style each of a group of sibling elements based on how many siblings there are in total. The technique is something called a <u>quantity query</u> ↗. Consider the following code.

```
.switcher > :nth-last-child(n+5),
.switcher > :nth-last-child(n+5) ~ * {
  flex-basis: 100%;
}
```

Here, we are applying a `flex-basis` of `100%` to each element, only where there are **five or more elements in total**. The `nth-last-child(n+5)` selector targets any elements that are more than 4 from the *end* of the set. Then, the general sibling combinator (~) applies the same rule to the rest of the elements (it matches anything after `:nth-last-child(n+5)`). If there are fewer that 5 items, no `:nth-last-child(n+5)` elements and the style is not applied.
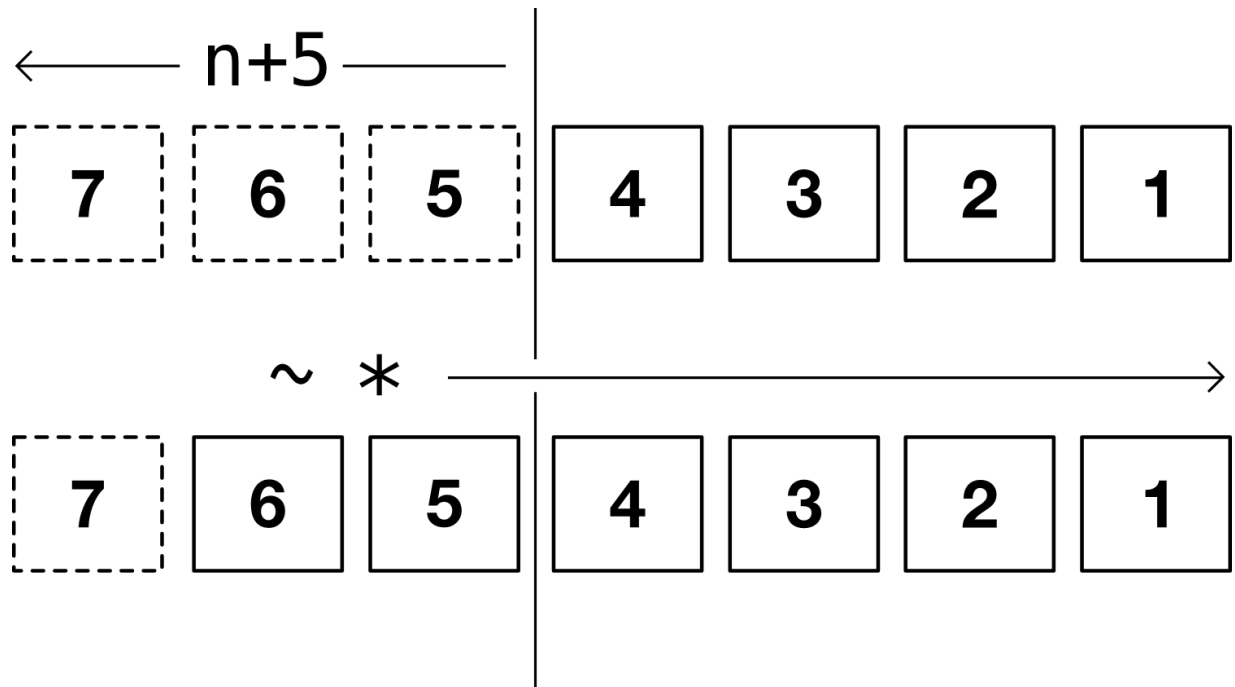
Now the layout has two kinds of threshold that it can handle, and is twice as robust as a result.

## Use cases

There are any number of situations in which you might want to switch directly between a horizontal and vertical layout. But it is especially useful where each element should be considered equal, or part of a continuum. Card components advertising products should share the same width no matter the orientation, otherwise one or more cards could be perceived as highlighted or featured in some way.

A set of numbered steps is also easier on cognition if those steps are laid out along one horizontal or vertical line.
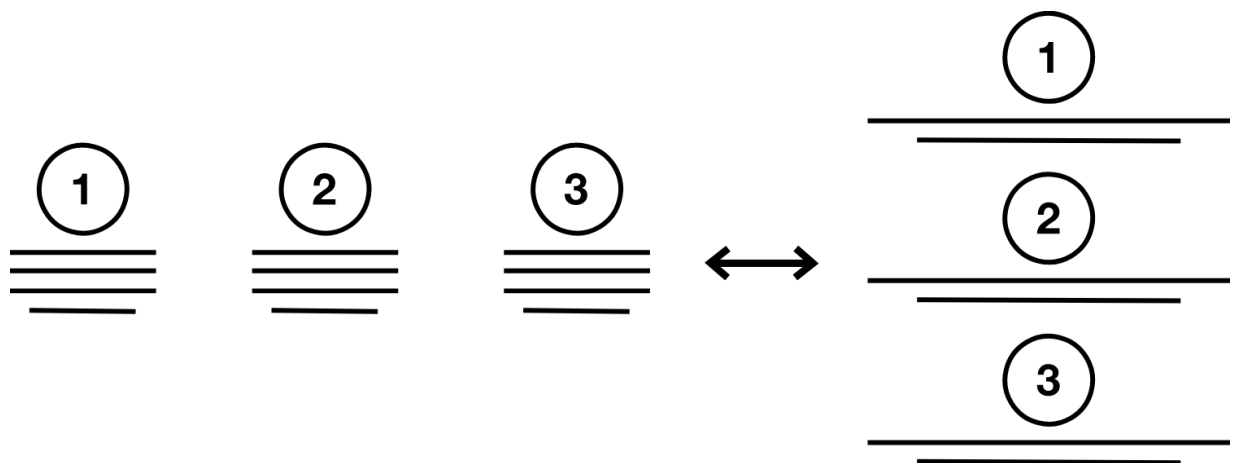
# The Generator

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments:

# CSS

```css
.switcher {
  display: flex;
  flex-wrap: wrap;
  /* ↓ The default value is the first point on the modular scale */
  gap: var(--gutter, var(--s1));
  /* ↓ The width at which the layout "breaks" */
  --threshold: 30rem;
}

.switcher > * {
  /* ↓ Allow children to grow */
  flex-grow: 1;
  /* ↓ Switch the layout at the --threshold */
  flex-basis: calc((var(--threshold) - 100%) * 999);
}

.switcher > :nth-last-child(n+5),
.switcher > :nth-last-child(n+5) ~ * {
  /* ↓ Switch to a vertical configuration if
  there are more than 4 child elements */
  flex-basis: 100%;
}
```

## HTML

```html
<div class="switcher">
  <div><!-- child element --></div>
  <div><!-- another child element --></div>
  <div><!-- etc --></div>
</div>
```

# The Component

A custom element implementation of the Switcher is available for download ↗.

# Props API

The following props (attributes) will cause the **Switcher** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|---|---|---|---|
| **threshold** | `string` | `"var(--measure)"` | A CSS `width` value (representing the 'container breakpoint') |
| **space** | `string` | `"var(--s1)"` | A CSS `margin` value |
| **limit** | `integer` | `4` | A number representing the maximum number of items permitted for a horizontal layout |

**The Cover**

# The problem

For years, there was consternation about how hard it was to horizontally and vertically center something with CSS. It was used by detractors of CSS as a kind of exemplary "proof" of its shortcomings.

The truth is, there are numerous ways to center content with CSS. However, there are only so many ways you can do it without fear of overflows, overlaps, or other such breakages. For example, we could use `relative` positioning and a `transform` to vertically center an element within a parent:

```
.parent {
  /* ↓ Give the parent the height of the viewport */
  block-size: 100vh;
}

.parent > .child {
  position: relative;
  /* ↓ Push the element down 50% of the parent */
  inset-block-start: 50%;
  /* ↓ Then adjust it by 50% of its own height */
  transform: translateY(-50%);
}
```

What's neat about this is the `translateY(-50%)` part, which
compensates for the height of the element itself—no matter what
that height is. What's less than neat is the top and bottom overflow
produced when the child element's content makes it taller than the
parent. We have not, so far, designed the layout to tolerate dynamic
content.
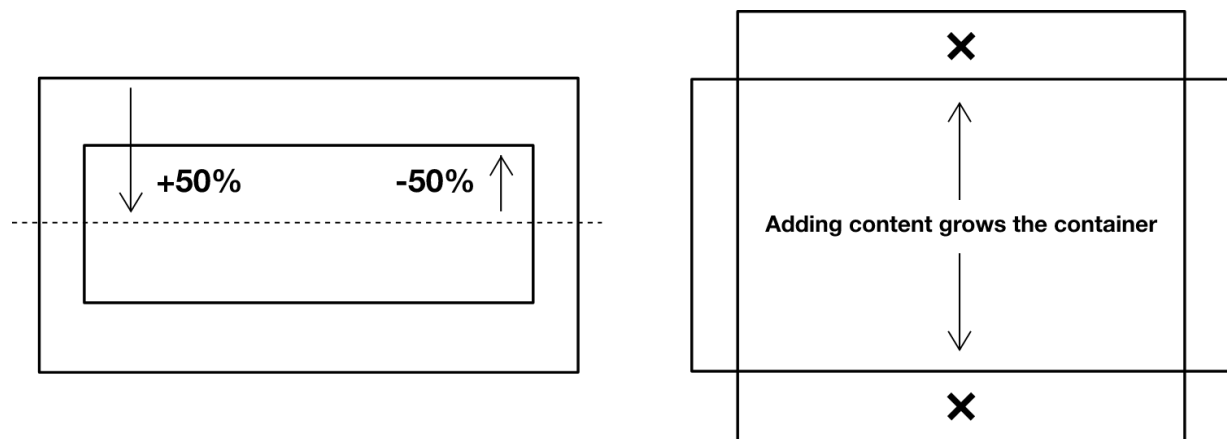


Perhaps the most robust method is to combine Flexbox's `justify-content: center` (horizontal) and `align-items: center` (vertical).

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

# Proper handling of height

Just applying the Flexbox CSS will not, on its own, have a visible effect on vertical centering because, by default, the `.centered` element's height is determined by the height of its content (implicitly, `block-size: auto`). This is something sometimes referred to as *intrinsic sizing*, and is covered in more detail in the **Sidebar** layout documentation.

Setting a fixed height—as in the unreliable `transform` example from before—would be foolhardy: we don't know ahead of time how much content there will be, or how much vertical space it will take up. In other words, there's nothing stopping overflow from happening.
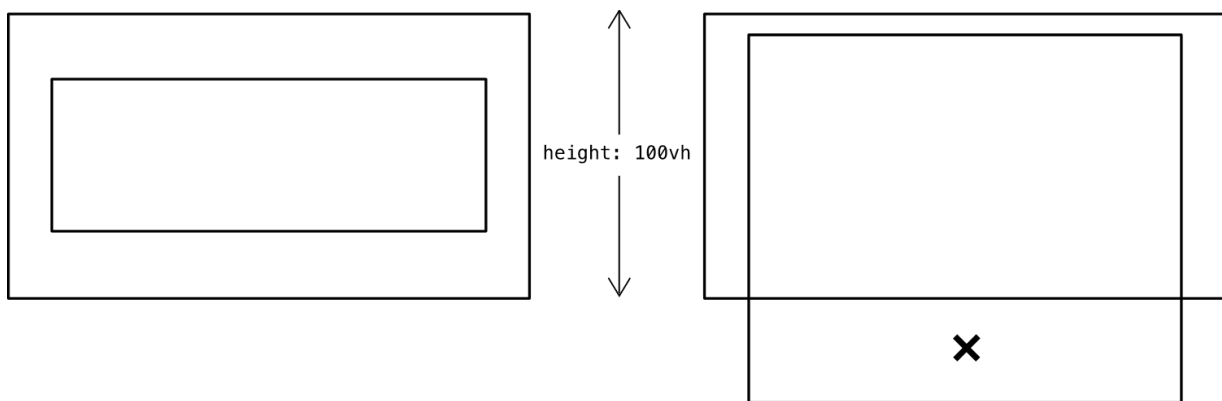
Instead, we can set a `min-block-size` (`min-height` in the `horizontal-tb` writing mode). This way, the element will expand vertically to accommodate the content, wherever the natural (`auto`) height happens to be more than the `min-block-size`. Where this happens, the provision of some vertical padding ensures the centered content does not meet the edges.
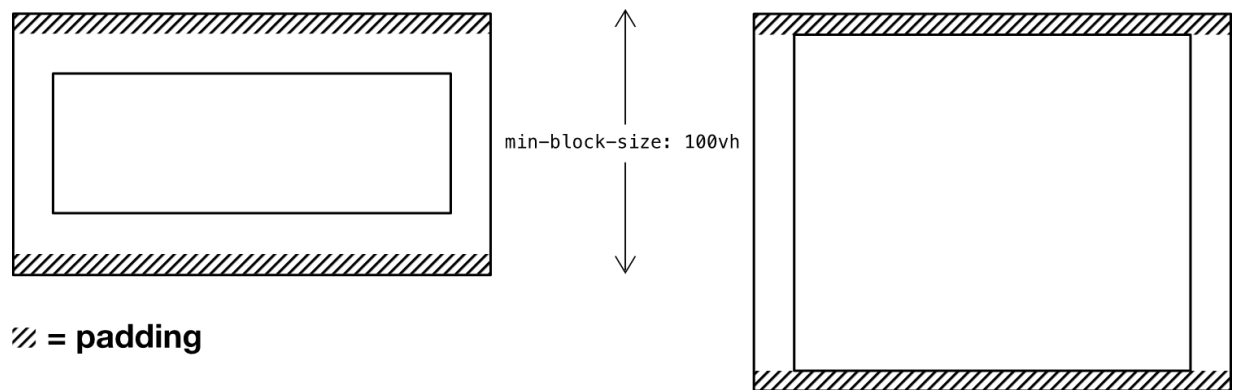


**⧄ = padding**

# Box sizing

To ensure the parent element retains a height of `100vh`, despite the additional padding, a `box-sizing: border-box` value must be applied. Where it is not, the padding is *added* to the total height.

The `box-sizing: border-box` is so desirable, it is usually applied to all elements in a global declaration block. The use of the ∗ (universal) selector means all elements are affected.

```
* {
  box-sizing: border-box;
  /* other global styles */
}
```

This is perfectly serviceable where only one centered element is in contention. But we have a habit of wanting to include other elements, above and below the centered one. Perhaps it's a close button in the top right, or a "read more" indicator in the bottom center. In any case, I need to handle these cases in a modular fashion, and without producing breakages.

## The solution

What I need is a layout component that can handle vertically centered content (under a `min-block-size` threshold) and can accommodate top/header and bottom/footer elements. To make the component truly *composable* I should be able to add and remove these elements in the HTML without having to *also* adapt the CSS. It should be modular, and therefore not a coding imposition on content editors.

The **Cover** component is a Flexbox context with `flex-direction: column`. This declaration means child elements are laid out vertically rather than horizontally. In other words, the 'flow direction' of the Flexbox formatting context is returned to that of a standard block element.

```
.cover {
  display: flex;
  flex-direction: column;
}
```

The **Cover** has one *principal* element that should always gravitate towards the center. In addition, it can have one top/header element and/or one bottom/footer element.
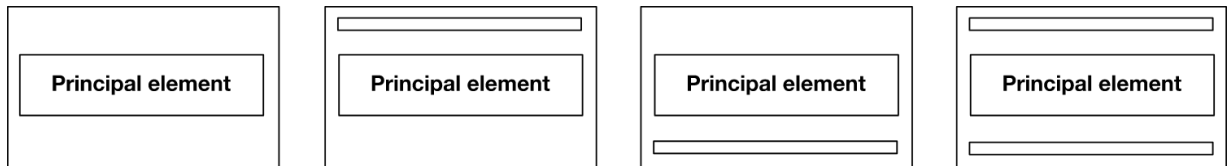


How do we manage all these cases without having to adapt the CSS? First, we give the centered element (`h1` in the example, but it can be any element) `auto` margins. This can be done in one declaration using `margin-block`:

```
.cover {
  display: flex;
  flex-direction: column;
}

.cover > h1 {
  margin-block: auto;
}
```

These *push* the element away from anything above and below it, moving it into the center of the available space. Critically, it will *push off* the inside edge of a parent *or* the top/bottom edge of a sibling element.
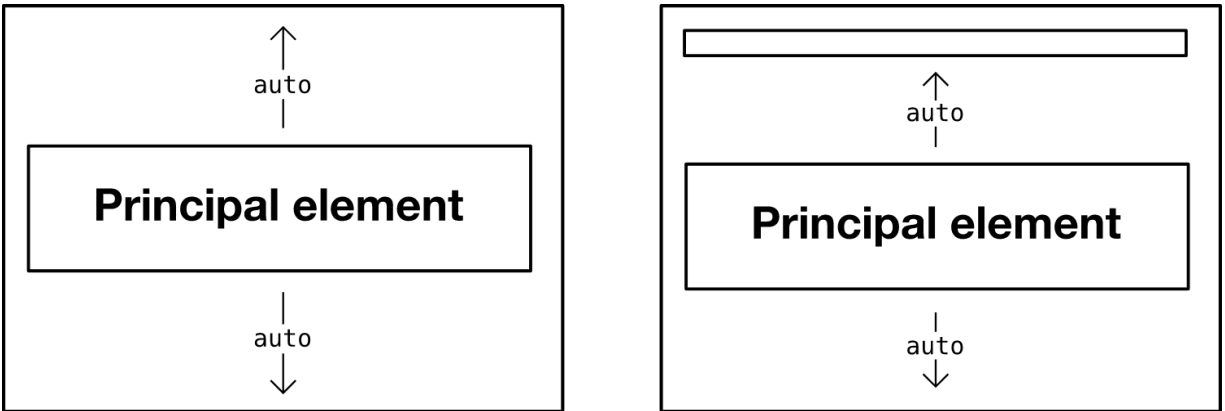
*Note that, in the right-hand configuration, the centered element is in the vertical center of the available space.*

All that remains is to ensure there is space between the (up to) three child elements where the `min-block-size` threshold has been breached.
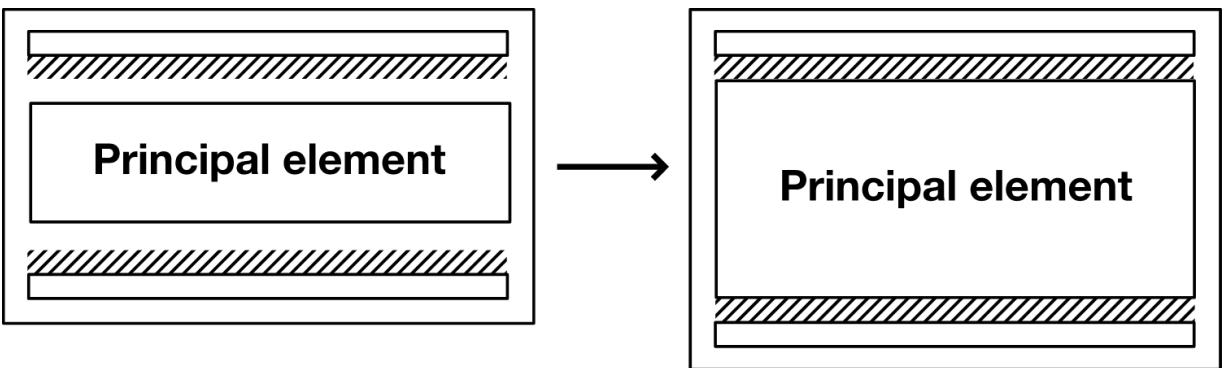


Currently, the `auto` margins simply collapse down to nothing. Since we can't enter `auto` into a `calc()` function to adapt the margin (`calc(auto + 1rem)` is invalid), the best we can do is to add `margin` to the header and footer elements contextually.

```css
.cover > * {
  margin-block: 1rem;
}

.cover > h1 {
  margin-block: auto;
}

.cover > :first-child:not(h1) {
  margin-block-start: 0;
}

.cover > :last-child:not(h1) {
  margin-block-end: 0;
}
```

Note, the use of <u>the cascade, specificity ↗</u> and negation to target the correct elements. First, we apply top and bottom margins to all the children, using a universal child selector. We then override this for the to-be-centered (`h1`) element to achieve the `auto` margins. Finally, we use the `:not()` function to remove extraneous margin from the top and bottom elements *only* if they are *not* the centered element. If there is a centered element and a footer element, but no header element, the centered element will be the `:first-child` and must retain `margin-block-start: auto`.

## ⚠ Shorthands

Notice how we use `margin-block: 1rem` and not `margin: 1rem 0`. The reason is that *this component* only cares about the vertical margins to achieve its layout. By making the inline (horizontal in the default writing mode) margins `0`, we might be unduly undoing styles applied or inherited by an ancestor component.

Only set what you need to set.

Now it is safe to add spacing around the inside of the **Cover** container using `padding`. Whether there are one, two or three elements present, spacing now remains *symmetrical*, and our component modular without styling intervention.

```
.cover {
  padding: 1rem;
  min-block-size: 100vh;
}
```

The `min-block-size` is set to `100vh`, so that the element *covers* 100% of the viewport's height (hence the name). However, there's no reason why the `min-block-size` cannot be set to another value. `100vh` is considered a *sensible default*, and is the default value for the `minHeight` prop in the custom element implementation to come.

# Horizontal centering

So far I've not tackled horizontal centering, and that's quite deliberate. Layout components should try to solve just one problem —and the modular centering problem is a peculiar one. The **Center layout** handles horizontal centering and can be used in composition

with the **Cover**. You might wrap the **Cover** in a **Center** or make a **Center** one or more of its children. It's all about *composition*.

# Use cases

A typical use for the **Cover** would be to create the "above the fold" introductory content for a web page. In the following demo, a nested **Cluster element** is used to lay out the logo and navigation menu. In this case, a utility class (`.text-align\:center`) is used to horizontally center the `<h1>` and footer elements.

This interactive demo is only available on the **Every Layout** site ↗.

It might be that you treat each section of the page as a **Cover**, and use the Intersection Observer API to animate aspects of the cover as it comes into view. A simple implementation is provided below (where the `data-visible` attribute is added as the element comes into view).

```
if ('IntersectionObserver' in window) {
  const targets = Array.from(document.querySelectorAll('cover-l'));
  targets.forEach(t => t.setAttribute('data-observe', ''));
  const callback = (entries, observer) => {
    entries.forEach(entry => {
      entry.target.setAttribute('data-visible', entry.isIntersecting);
    });
  };

  const observer = new IntersectionObserver(callback);
  targets.forEach(t => observer.observe(t));
}
```

# The generator

Use this tool to generate basic **Cover** CSS and HTML.

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments. It assumes the centered element is an `<h1>`, in this case, but it could be any element.

# CSS

```css
.cover {
  --space: var(--s1);
  /* ↓ Establish a columnal flex context */
  display: flex;
  flex-direction: column;
  /* ↓ Set a minimum height to match the viewport height
  (any minimum would be fine) */
  min-block-size: 100vh;
  /* Set a padding value */
  padding: var(--space);
}

.cover > * {
  /* ↓ Give each child a top and bottom margin */
  margin-block: var(--s1);
}

.cover > :first-child:not(h1) {
  /* ↓ Remove the top margin from the first-child
  if it _doesn't_ match the centered element */
  margin-block-start: 0;
}

.cover > :last-child:not(h1) {
  /* ↓ Remove the bottom margin from the last-child
  if it _doesn't_ match the centered element */
  margin-block-end: 0;
}

.cover > h1 {
  /* ↓ Center the centered element (h1 here)
  in the available vertical space */
  margin-block: auto;
}
```

# HTML

Assumes the centered element is an `<h1>`, and is in the `nth-child(2)` position.

```
<div class="cover">
  <div><!-- first child --></div>
  <h1><!-- centered child --></h1>
  <div><!-- third child --></div>
</div>
```

# The component

A custom element implementation of the Cover is available for download ↗.

# Props API

The following props (attributes) will cause the **Cover** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|------|------|---------|-------------|
| **centered** | `string` | `"h1"` | A simple selector such an element or class selector, representing the centered (main) element in the cover |
| **space** | `string` | `"var(--s1)"` | The minimum space between and around all of the child elements |
| **minHeight** | `string` | `"100vh"` | The minimum height (block-size) for the **Cover** |
| **noPad** | `boolean` | `false` | Whether the spacing is also applied as padding to the container element |

# Examples

## Basic

Just a centered element (an `<h1>`) with no header or footer companions. The context/parent adopts the default `min-height` of `100vh`.

```
<cover-l>
  <h1>Welcome!</h1>
</cover-l>
```

# ⚠ One `<h1>` per page

For reasons of accessible document structure, there should only be one `<h1>` element per page. This is the page's main heading to screen reader users. If you add several successive `<cover-l>`s, all but the first should have an `<h2>` to indicate it is a *subsection* in the document structure.

# The Grid

## The problem

Designers sometimes talk about designing *to a grid*. They put the grid—a matrix of horizontal and vertical lines—in place first, then they populate that space, making the words and pictures span the boxes those intersecting lines create.
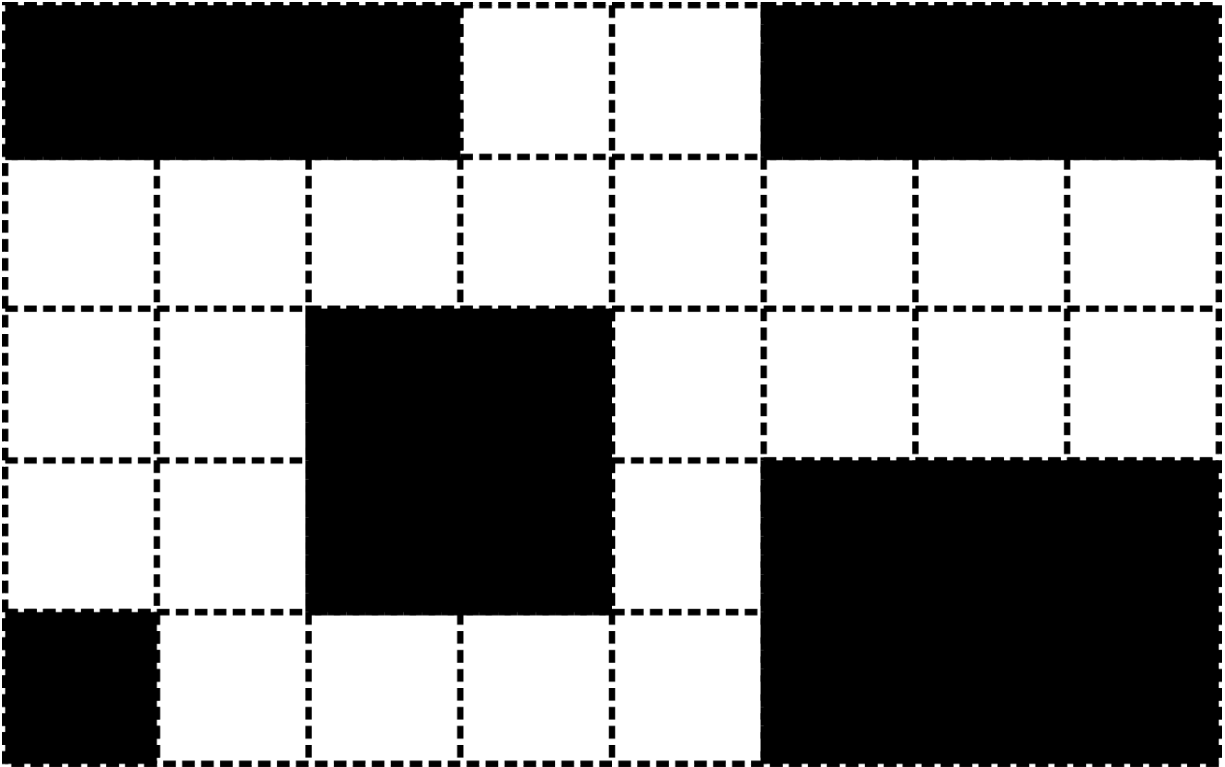
## ■ Content areas

A 'grid first' approach to layout is only really tenable where two things are known ahead of time:

1. The space
2. The content

For a paper-destined magazine layout, like the one described in **Axioms**, these things are attainable. For a screen and device-independent web layout containing dynamic (read: changeable) content, they fundamentally are not.

The CSS Grid module is radical because it lets you place content anywhere within a predefined grid, and as such brings *designing to a grid* to the web. But the more particular and deliberate the placement of grid content, the more manual adjustment, in the form of `@media` breakpoints, is needed to adapt the layout to different spaces. Either the grid definition itself, the position of content within it, or both will have to be changed by hand, and with additional code.

As I covered in **The Switcher**, `@media` breakpoints pertain to viewport dimensions only, and not the immediate available space offered by a parent container. That means layout components defined using `@media` breakpoints are fundamentally not context independent: a huge issue for a modular design system.

It is not, even theoretically, possible to design *to a grid* in a context-independent, automatically responsive fashion. However, it's possible to create basic grid-like formations: sets of elements divided into both columns and rows.
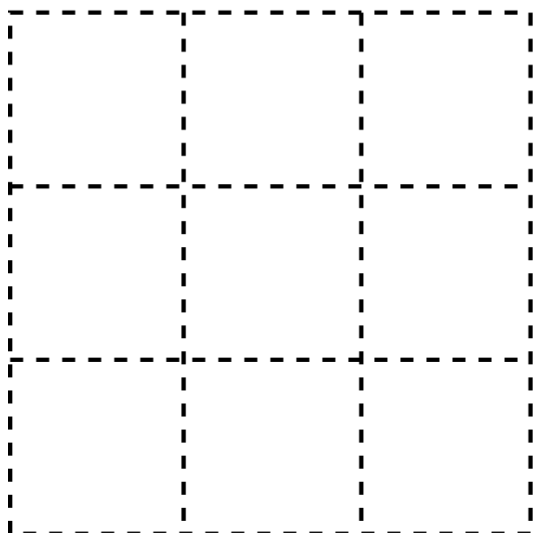
## A grid for content
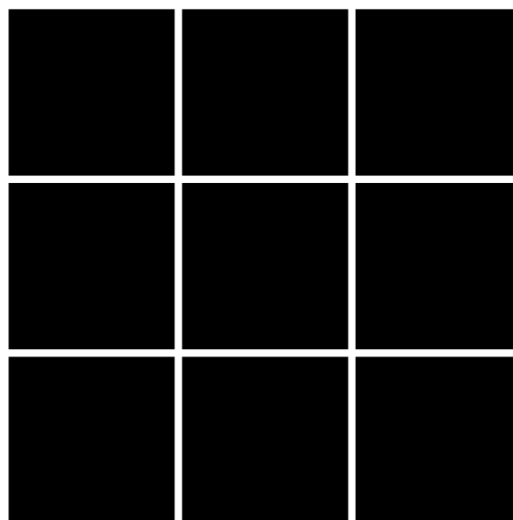
## A grid of content

*In **Every Layout**, we design with content. Without content, a grid needn't exist; with content, a grid formation may emerge from it*

Compromise is inevitable, so it's a question of finding the most archetypal yet efficient solution.

# Flexbox for grids

Using Flexbox, I can create a grid formation using `flex-basis` to determine an *ideal* width for each of the grid cells:

```
.flex-grid {
  display: flex;
  flex-wrap: wrap;
}

.flex-grid > * {
  flex: 1 1 30ch;
}
```

The `display: flex` declaration defines the Flexbox context, `flex-wrap: wrap` allows wrapping, and `flex: 1 1 30ch` says, "*the ideal width should be 30ch, but items should be allowed to grow and shrink according to the space available*". Importantly, the number of columns is not prescribed based on a fixed grid schematic; it's determined *algorithmically* based on the `flex-basis` and the available space. The content and the context define the grid, not a human arbiter.

In **The Switcher**, we identified an interaction between *wrapping* and *growth* that leads items to 'break' the grid shape under certain circumstances:
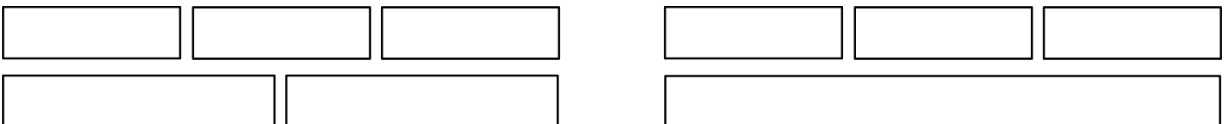
On the one hand, the layout takes up all its container's horizontal space, and there are no unsightly gaps. On the other, a generic grid formation should probably make each of its items align to both the horizontal and vertical rules.

# Mitigation

You'll recall the global measure rule explored in the **<u>Axioms</u>** section. This ensured all applicable elements could not become wider than a comfortably readable line-length.

Where a grid-like layout created with Flexbox results in a full-width `:last-child` element, the measure of its contained text elements would be in danger of becoming too long. Not with that global measure style in place. The benefit of global rules (*axioms*) is in not having to consider each design principle per-layout. Many are already taken care of.

**max-width for text elements**
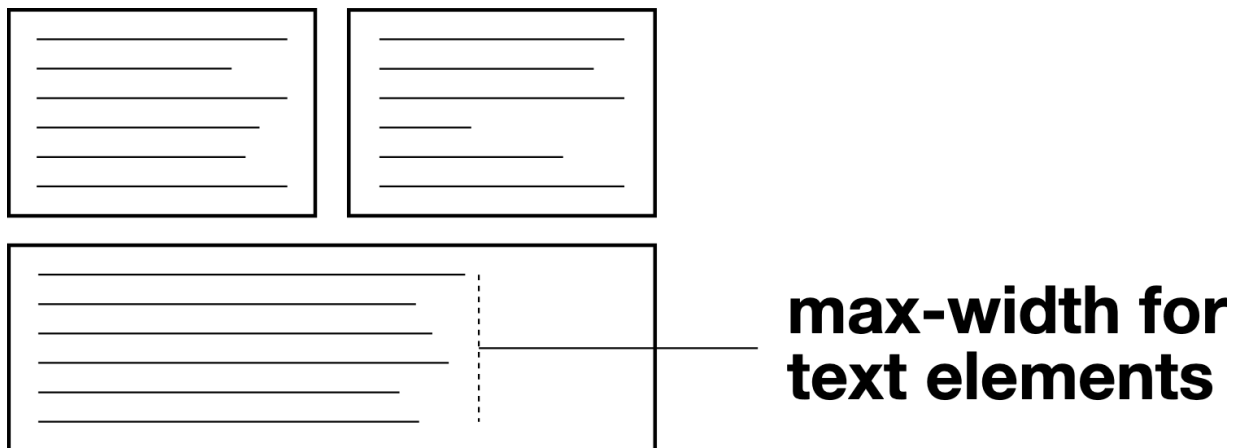
## *Grid* for grids

The aptly named CSS Grid module brings us closer to a 'true' responsive grid formation in one specific sense: It's possible to make items grow, shrink, and wrap together *without* breaching the column boundaries.
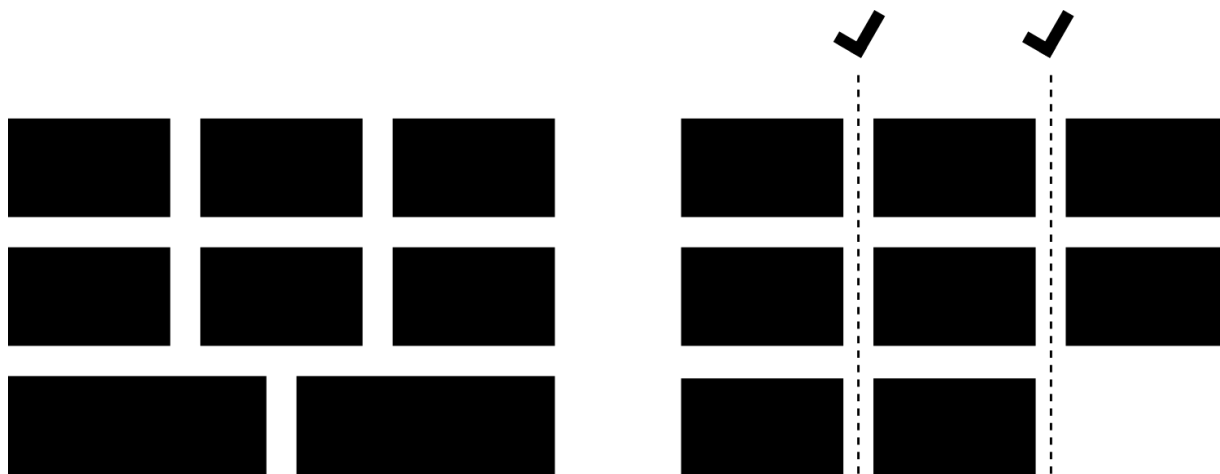
This behavior is closer to the archetypal responsive grid I have in mind, and will be the layout we pursue here. There's just one major implementation issue to quash. Consider the following code.

```
.grid {
  display: grid;
  grid-gap: 1rem;
  grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
}
```

This is the pattern, which I first discovered in Jen Simmon's Layout Land ↗ video series. To break it down:

1. `display: grid:` sets the grid context, which makes grid cells for its children.
2. `grid-gap:` places a 'gutter' *between* each grid item (saving us from having to employ the negative margin technique first described in **The Cluster**).

3. `grid-template-columns:` Would ordinarily define a rigid grid for *designing to*, but used with `repeat` and `auto-fit` allows the dynamic spawning and wrappping of columns to create a behavior similar to the preceding Flexbox solution.
4. `minmax:` This function ensures each column, and therefore each cell of content shares a width between a minimum and maximum value. Since `1fr` represents one part of the available space, columns grow together to fill the container.

The shortcoming of this layout is the minimum value in `minmax()`. Unlike `flex-basis`, which allows any amount of growing or shrinking from a single 'ideal' value, `minmax()` sets a scope with hard limits.

Without a fixed minimum (`250px`, in this case) there's nothing to *trigger* the wrapping. A value of `0` would just produce one row of ever-diminishing widths. But it being a fixed minimum has a clear consequence: in any context narrower than the minimum, overflow will occur.



**Hard minimum reached**
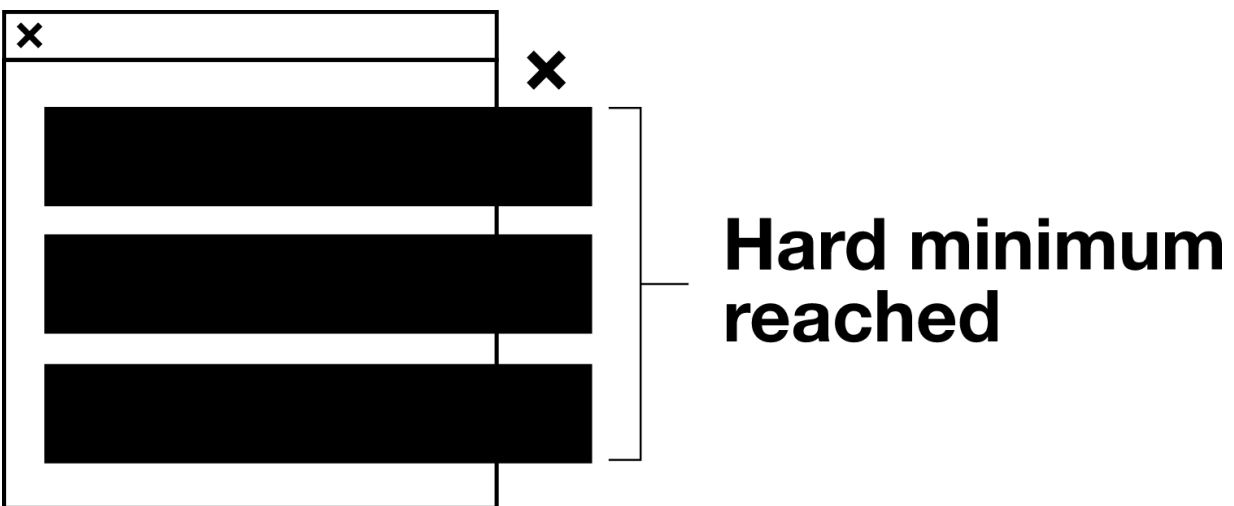
To put it simply: the pattern as it stands can only safely produce layouts where the columns converge on a width that is below the estimated minimum for the container. About `250px` is reasonably safe because most handheld device viewports are no wider. But what if I want my columns to grow considerably beyond this width, where the space is available? With Flexbox and `flex-basis` that is quite possible, but with CSS Grid it is not without assistance.

# The solution

Each of the layouts described so far in **Every Layout** have handled sizing and wrapping with just CSS, and without `@media` queries. Sometimes it's not possible to rely on CSS alone for automatic reconfiguration. In these circumstances, turning to `@media` breakpoints is out of the question, because it undermines the modularity of the layout system. Instead, I *might* defer to JavaScript. But I should do so *judiciously*, and using progressive enhancement.

<u>ResizeObserver</u> ↗ (soon to be <u>available in most modern browsers</u> ↗) is a highly optimized API for tracking and responding to changes in element dimensions. It is the most efficient method yet for creating <u>container queries</u> ↗ with JavaScript. I wouldn't recommend using it as a matter of course, but employed *only* for solving tricky layout issues is acceptable.

Consider the following code:

```
.grid {
  display: grid;
  grid-gap: 1rem;
}

.grid.aboveMin {
  grid-template-columns: repeat(auto-fit, minmax(500px, 1fr));
}
```

The `aboveMin` class presides over an overriding declaration that produces the responsive grid. `ResizeObserver` is then instructed to add and remove the `aboveMin` class depending on the container width. The minimum value of `500px` (in the above example) is *only* applied where the container itself is wider than that threshold. Here is a standalone function to activate the `ResizeObserver` on a grid element.

```
function observeGrid(gridNode) {
  // Feature detect ResizeObserver
  if ('ResizeObserver' in window) {
    // Get the min value from data-min="[min]"
    const min = gridNode.dataset.min;
    // Create a proxy element to measure and convert
    // the `min` value (which might be em, rem, etc) to `px`
    const test = document.createElement('div');
    test.style.width = min;
    gridNode.appendChild(test);
    const minToPixels = test.offsetWidth;
    gridNode.removeChild(test);

    const ro = new ResizeObserver(entries => {
      for (let entry of entries) {
        // Get the element's current dimensions
        const cr = entry.contentRect;
        // `true` if the container is wider than the minimum
        const isWide = cr.width > minToPixels;
        // toggle the class conditionally
        gridNode.classList.toggle('aboveMin', isWide);
      }
    });

    ro.observe(gridNode);
  }
}
```

If `ResizeObserver` is not supported, the fallback one-column layout is applied perpetually. This basic fallback is included here for brevity, but you could instead fallback to the serviceable-but-imperfect Flexbox solution covered in the previous section. In any case, no content is lost or obscured, and you have the ability to use larger `minmax()` minimum values for more expressive grid formations. And since we're no longer bound to absolute limits, we can begin employing relative units.
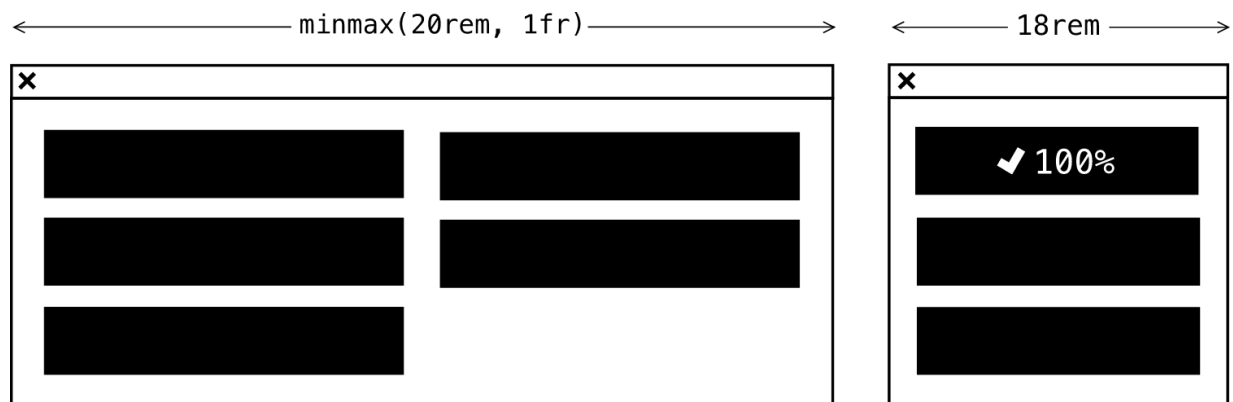
Here's an example initialization (code is elided for brevity):

```
<div class="grid" data-min="250px">
  <!-- Place children here -->
</div>

<script>
  const grid = document.querySelector('.grid');
  observeGrid(grid);
</script>
```

# The `min()` function

While it is worth covering `ResizeObserver` because it may serve you well in other circumstances, it is actually no longer needed to solve this particular problem. That's because we have the recently widely adopted ↗ CSS `min()` function. Sorry for the wild goose chase but we can, in fact, write this layout without JavaScript after all.

As a fallback, we configure the grid into a single column. Then we use `@supports` to test for `min()` and enhance from there:

```
.grid {
  display: grid;
  grid-gap: 1rem;
}

@supports (width: min(250px, 100%)) {
  .grid {
    grid-template-columns: repeat(auto-fit, minmax(min(250px, 100%), 1fr));
  }
}
```

The way `min()` works is it calculates the *shortest length* from a set of comma-separated values. That is: `min(250px, 100%)` would return `100%` where `250px` evaluates as *higher* than the evaluated `100%`. This useful little algorithm *decides for us* where the width must be capped at `100%`.

# <watched-box>

If you are looking for a general solution for container queries ↗, I have created `<watched-box>` ↗. It's straightforward and declarative, and it supports any CSS length units.

It is recommended `<watched-box>` is used as a "last resort" manual override. In all but unusual cases, one of the purely CSS-based layouts documented in **Every Layout** will provide context sensitive layout automatically.

# Use cases

Grids are great for browsing teasers for permalinks or products. I can quickly compose a card component to house each of my teasers using a **Box** and a **Stack**.

# Shared height

Each card shares the same height, regardless of its content, because the default value for `align-items` is `stretch`. This is fortuitous since few would expect different sized cards, or the unsightly gaps the unequal heights would create.

# The generator

Use this tool to generate basic **Grid** CSS and HTML.

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments:

# CSS

```
.grid {
  /* ↓ Establish a grid context */
  display: grid;
  /* ↓ Set a gap between grid items  */
  grid-gap: 1rem;
  /* ↓ Set the minimum column width */
  --minimum: 20ch;
}

@supports (width: min(var(--minimum), 100%)) {
  .grid {
    /* ↓ Enhance with the min() function
    into multiple columns */
    grid-template-columns: repeat(auto-fit, minmax(min(var(--minimum), 100%),
1fr));
  }
}
```

# Implicit single column layout

Note that `grid-template-columns` is not set except in the
enhancement (`@supports`) block. Implicitly, it is a single column grid
unless `min()` is supported.

# HTML

```
<div class="grid">
  <div><!-- child element --></div>
  <div><!-- another child element --></div>
  <div><!-- etc --></div>
</div>
```

# The component

A custom element implementation of the Grid is available for
[download ↗](#).

## Props API

The following props (attributes) will cause the **Grid** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|---|---|---|---|
| **min** | string | `"250px"` | A CSS length value representing x in `minmax(min(x, 100%), 1fr)` |
| **space** | string | `"var(--s1)"` | The space between grid cells |

# Examples

## Cards

The code for the cards example from **Use cases**. Note that the `min` value is a fraction of the standard *measure*. There's more on typographic measure in the **Axioms** rudiment.

```
<grid-l min="calc(var(--measure) / 3)">
  <box-l>
    <stack-l>
      <!-- card content -->
    </stack-l>
  </box-l>
  <box-l>
    <stack-l>
      <!-- card content -->
    </stack-l>
  </box-l>
  <box-l>
    <stack-l>
      <!-- card content -->
    </stack-l>
  </box-l>
  <box-l>
    <stack-l>
      <!-- card content -->
    </stack-l>
  </box-l>
  <!-- etc -->
</grid-l>
```

# The Frame

## The problem

Some things exist as relationships. A line exists as the relationship between two points; without both the points, the line cannot come into being.

When it comes to drawing lines, there are factors we don't necessarily know, and others we absolutely do. We don't necessarily know where, in the universe, each of the points will appear. That might be outside of our control. But we *do* know that, no matter where the points appear, we'll be able to draw a straight line between them.
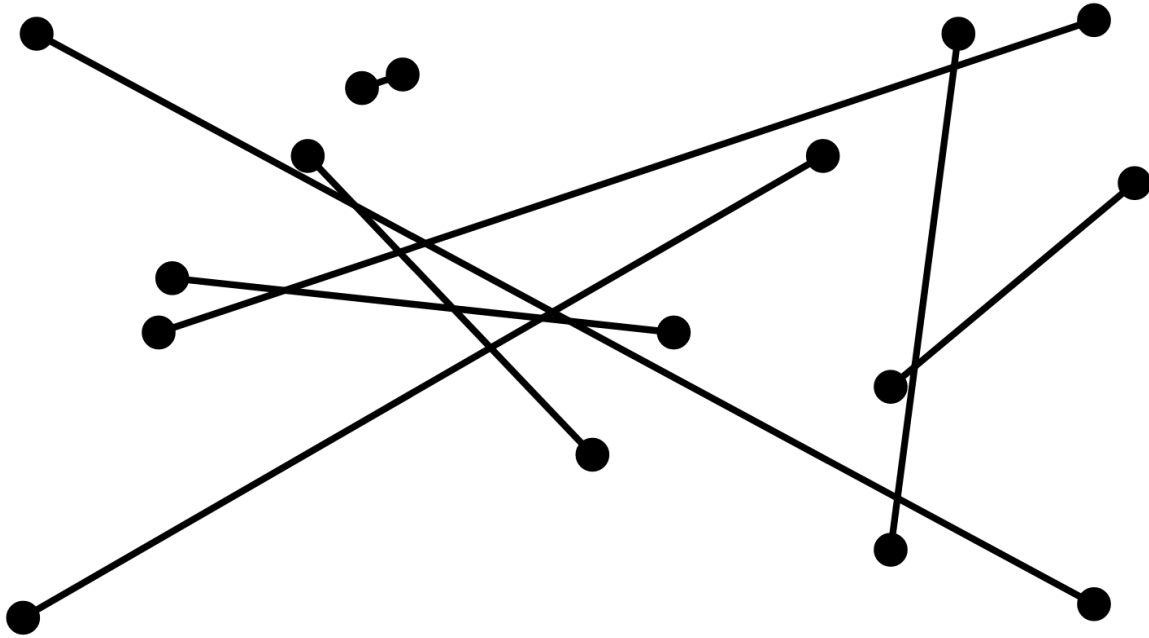
*Connecting randomly placed pairs of dots makes for some extremely pedestrian generative art.*

The position of the points is variable, but the nature of their relationship is constant. Capitalizing on the constants that exist in spite of the variables is how we shape dynamic systems.

# Aspect ratio

Aspect ratio is another constant that comes up a lot, especially when working with images. You find the aspect ratio by dividing the width of an image by its height.
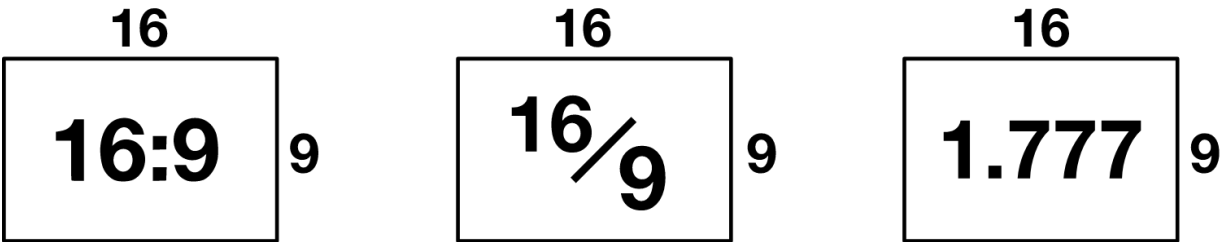
The `<img />` element is a replaced element ↗; it is an element *replaced* by the externally loaded source to which it points.

This source (an image file such as a PNG, JPEG, or SVG) has certain characteristics outside of your control as a writer of CSS. Aspect ratio is one such characteristic, and is determined when the image is originally created and cropped.

Making your images responsive is a matter of ensuring they don't overflow their container. A `max-inline-size` value of `100%` does just that.

```
img {
  max-inline-size: 100%;
}
```

# Global responsive images

Since this basic responsive behavior should be the default for all images, I apply the style with a non-specific element selector. Not all of your styles are component-specific; read **Global and local styling** for more info.

Now the image's width will match one of two values:

- Its own intrinsic/natural width, based on the file data
- The width of the horizontal space afforded by the container element

Importantly, the height—in either case—is determined by the aspect ratio. It's the same as writing `block-size: auto`, but that explicit declaration isn't needed by modern, compliant browsers.

```
height == width / aspect ratio
```

Sometimes we want to dictate the aspect ratio, rather than inheriting it from the image file. The only way to achieve this without squashing, or otherwise distorting, the image is to dynamically recrop it. Declaring `object-fit: cover` on an image will do just that: crop it to fit the space without augmenting its *own* aspect ratio. The container becomes a window onto the undistorted image.
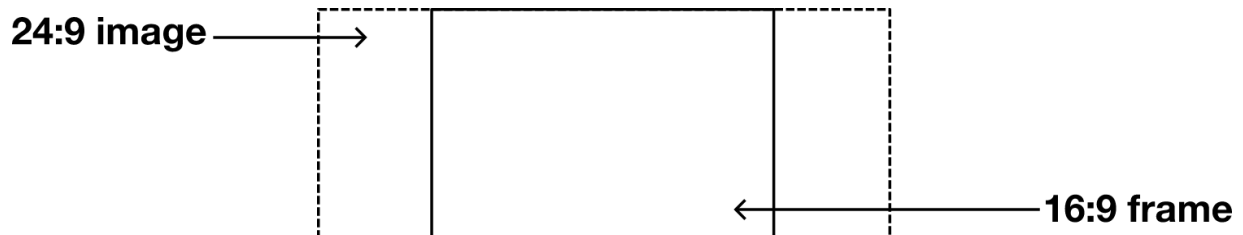


What might be useful is a general solution whereby we can draw a rectangle, based on a given aspect ratio, and make it a window onto

any content we place within it.

## The solution

The first thing we need to do is find a way to give an arbitrary element an aspect ratio *without* hard-coding its width and height. That is, we need to make a container behave like a (replaced) image.

For that, we have the aspect ratio property ↗ that would take an `x/n` value:

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

Before the advent of this property, we had to lean on an intrinsic ratio technique ↗ first written about as far back as 2009. The technique capitalizes on the fact that padding, even in the vertical dimension, is relative to the element's width. That is, `padding-bottom: 56.25%` will make an empty element (with no set height) *nine sixteenths as high as it is wide* — an aspect ratio of `16:9`. You find `56.25%` by dividing `9` (representing the height) by `16` (representing the width) — the opposite way around to finding the aspect ratio itself.

```
* + * {
    margin-top: var(--s1);
}
```
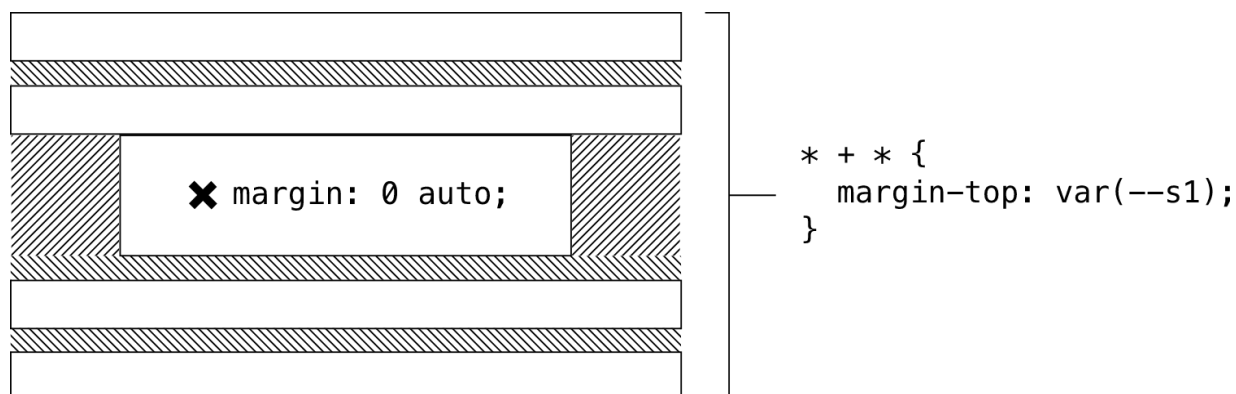
Using custom properties and `calc()`, we can create an interface that accepts any numbers for the left (numerator, or `n`) and right (denominator, or `d`) values of the ratio:

```
.frame {
    padding-bottom: calc(var(--n) / var(--d) * 100%);
}
```

Assuming `class="frame"` is a block level element (such as a `<div>`), its width will automatically match that of its parent. Whatever the calculated width value, the height is determined by multiplying it by `9 / 16`.

Since support is now good for the `aspect-ratio` property ↗, we can go ahead and use that instead of this elaborate hack.

# Cropping

So how does the cropping work? For replaced elements, like `<img />` and `<video />` elements, we just need to give them a `100%` width

and height, along with `object-fit: cover`:

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

.frame > img,
.frame > video {
  inline-size: 100%;
  block-size: 100%;
  object-fit: cover;
}
```

# Cropping position

Implicitly, the complementary `object-position` property's value is `50% 50%`, meaning the media is cropped around its center point. This is likely to be the most desirable cropping position (since most images have a focal point somewhere towards their middle). Be aware that `object-position` is at your disposal for adjustment.

The `object-fit` property is not designed for normal, non-replaced elements, so we'll have to include something more to handle them. Fortunately, Flexbox justification and alignment can have a similar effect. Since Flexbox has no affect on replaced elements, we can safely add these styles to the parent, with `overflow: hidden` preventing the content from escaping.

```
.frame {
  aspect-ratio: 16 / 9;
  overflow: hidden;
  display: flex;
  justify-content: center;
  align-items: center;
}

.frame > img,
.frame > video {
  inline-size: 100%;
  block-size: 100%;
  object-fit: cover;
}
```

Now any simple element will be placed in the center of the **Frame**, and cropped where it is taller or wider than the **Frame** itself. If the element's content makes it taller than the parent, it'll be cropped at the top *and* the bottom. Since inline content wraps, a specific set width might be needed to cause cropping on the left and right. To make sure the cropping happens in all contexts, and at all zoom levels, a %-based value will work.
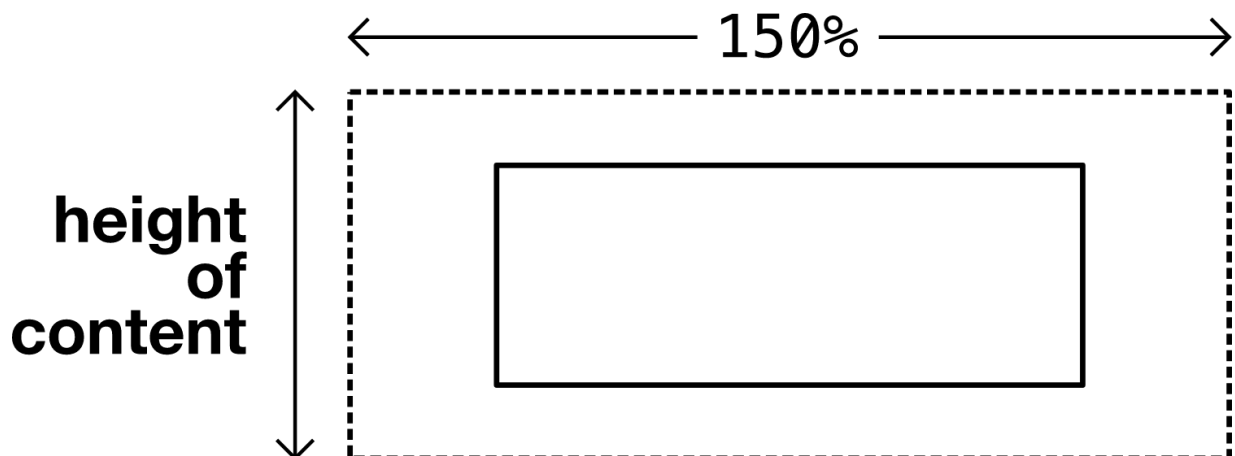


## ⚠ Background images

Another way to crop an image to cover its parent's shape is to supply it as a background image, and use `background-size: cover`. For this implementation, we are assuming the image should be treated as *content*, and therefore be supplied with alternative text ↗.

Background images cannot take alternative text directly, and are also removed by some high contrast modes/themes some of your users may be running. Using a "real" image, via an `<img />` tag, is usually preferable for accessibility.

# Use cases

The **Frame** is mostly useful for cropping media (videos and images) to a desired aspect ratio. Once you start controlling the aspect ratio, you can of course tailor it to the current circumstances. For example, you might want to give images a different aspect ratio depending on the viewport orientation.

It's possible to achieve this by changing the custom property values via an orientation `@media` query. In the following example, the **Frame** elements of the previous example are made square (rather than `16:9` landscape) where there is relatively more vertical space available.

```
@media (orientation: portrait) {
  .frame {
    aspect-ratio: 1 / 1;
  }
}
```

The Flexbox provision means you can crop any kind of HTML to the given aspect ratio, including `<canvas>` elements if those are your chosen means of creating imagery. A set of card-like components might each contain either an image or—where none is available—a textual fallback:

# The generator

Use this tool to generate basic **Frame** CSS and HTML.

The code generator tool is only available in the accompanying site ↗. Here is the basic solution, with comments.

# CSS

Replace the `--n` (numerator) and `--d` (denominator) values with whichever you wish, to create the aspect ratio.

```
.frame {
  --n: 16; /* numerator */
  --d: 9; /* denominator */
  aspect-ratio: var(--n) / var(--d);
  overflow: hidden;
  display: flex;
  justify-content: center;
  align-items: center;
}

.frame > img,
.frame > video {
  inline-size: 100%;
  block-size: 100%;
  object-fit: cover;
}
```

# HTML

The following example uses an image. There must be just one child element, whether it is a replaced element or otherwise.

```
<div class="frame">
  <img src="/path/to/image" alt="description of the image here" />
</div>
```

# The component

A custom element implementation of the Frame is available for download ↗.

# Props API

The following props (attributes) will cause the **Frame** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|------|------|---------|-------------|
| **ratio** | `string` | `"16:9"` | The element's aspect ratio |

# Examples

## Image frame

The custom element takes a `ratio` expression, like `4:3` (`16:9` is the default).

```
<frame-l ratio="4:3">
  <img src="/path/to/image" alt="description of the image here" />
</frame-l>
```

# The Reel

## The problem

When I'm sequencing music, I don't know how long the track I'm creating is going to be until I'm done. My sequencer software is aware of this and provisions time *on demand*, as I add bars of sound. Just as music sequencers dynamically provision time, web pages provision space. If all songs had to be four minutes and twenty six seconds long, or all web pages 768px high, well, that would be needlessly restrictive.
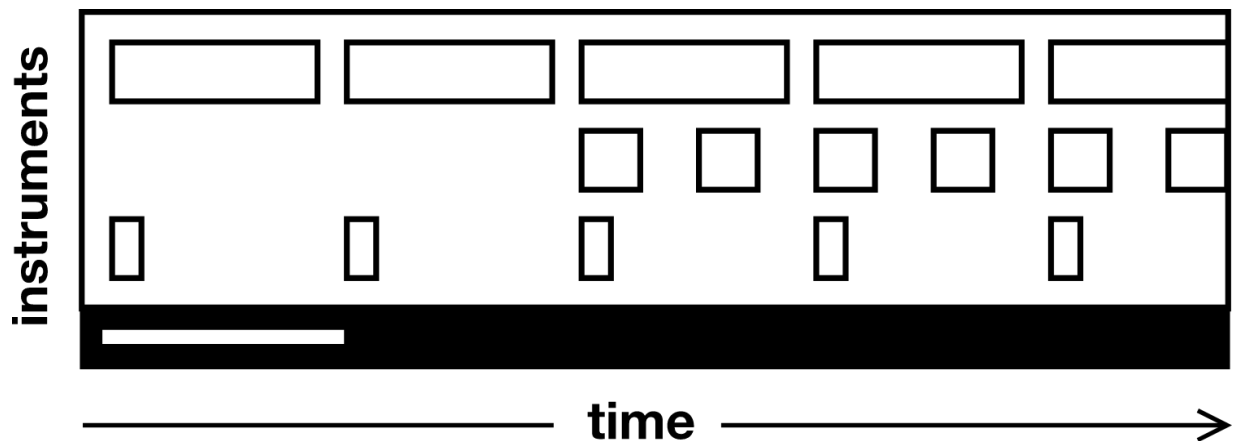
The mechanism whereby the provisioned space can be explored within a fixed "viewport" is called *scrolling*. Without it, everyone's devices would have to be exactly the same size, shape, and magnification level at all times. Writing content for such a space would become a formalist game, like writing haiku. Thanks to scrolling, you don't have to worry about space when writing web content. Writing for print does not have the same luxury.

The CSS `writing-mode` with which you are probably most familiar is `horizontal-tb`. In this mode, text and inline elements progress horizontally (either from left to right, as in English, or right to left) and block elements flow from top to bottom (that's the `tb` part). Since text and inline elements are instructed to *wrap*, the horizontal overflow which would trigger horizontal scrolling is generally avoided. Because content is not permitted to reach *outwards* is resolves to reach *downwards*. The vertical progression of block elements inevitably triggers vertical scrolling instead.
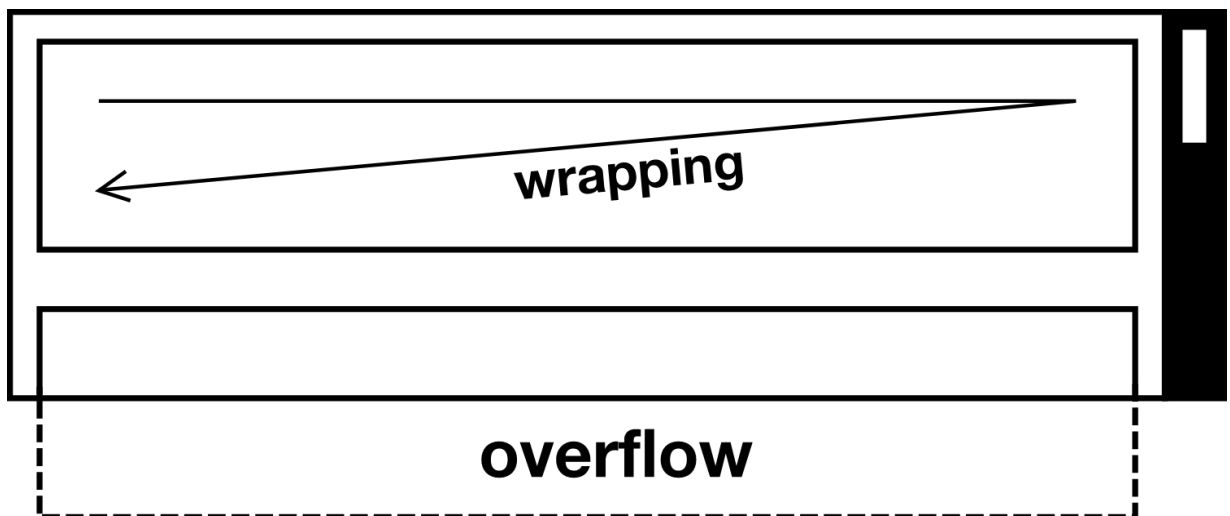
As a Western reader, accustomed to the `horizontal-tb` writing mode, vertical scrolling is conventional and expected. When you find the page needs to be scrolled vertically to see all the content, you don't think something has gone wrong. Where you encounter *horizontal* scrolling, it's not only unexpected but has clear usability implications: where overflow follows writing direction, each successive line of text has to be scrolled to be read.

All this is not to say that horizontal scrolling is strictly forbidden within a `horizontal-tb` writing mode. In fact, where implemented deliberately and clearly, horizontally scrolling sections within a vertically scrolling page can be an ergonomic way to browse content. Television streaming services tend to dissect their content by category vertically and programme horizontally, for example. The one thing you really want to avoid are single elements that scroll *bidirectionally*. This is considered a failure under WCAG's **1.4.10 Reflow** criterion.
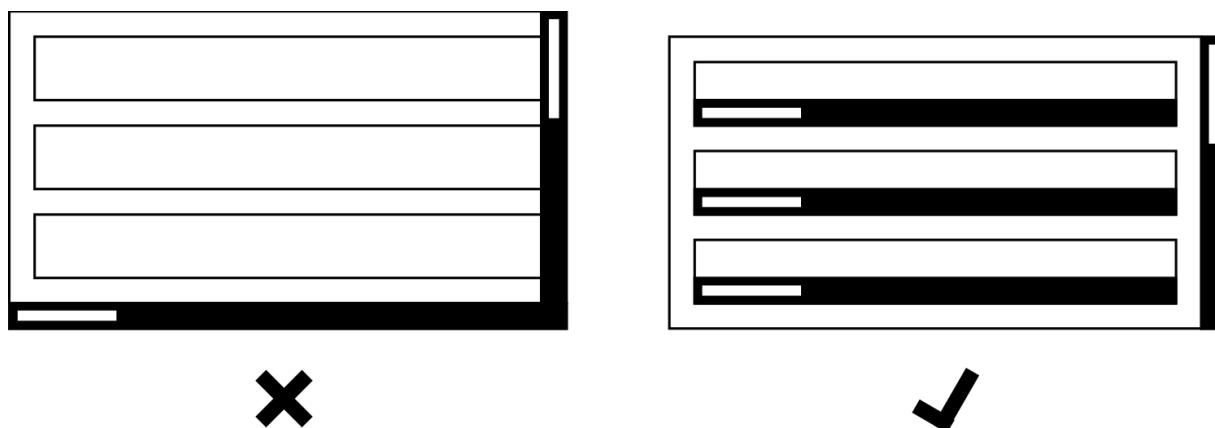


I formalized an accessible "carousel" component for the BBC ↗ which—instead of deferring entirely to JavaScript for the browsing functionality—simply invokes native scrolling with overflow. The

browsing buttons provided are merely a progressive enhancement, and increment the scroll position. Every Layout's **Reel** is similar, but foregoes the JavaScript to rely on standard browser scrolling behavior alone.

# The solution

As we set out in **The Cluster**, an efficient way to change the direction of block flow is to create a Flexbox context. By applying `display: flex` to an element, its children will switch from progressing downwards to progressing rightwards — at least where the default LTR (left-to-right) writing `direction` is in effect.

By omitting the often complementary `flex-wrap: wrap` declaration, elements are forced to maintain a single-file formation. Where this line of content is longer than the parent element is wide, overflow occurs. By default, this will cause the page itself to scroll horizontally. We don't want that, because it's only our Flexbox content that actually needs scrolling. It would be better that everything else stays still. So, instead, we apply `overflow: auto` to the Flex element, which automatically invokes scrolling *on that element* and only where overflow does indeed occur.

```
.reel {
  display: flex;
  /* ↓ We only want horizontal scrolling */
  overflow-x: auto;
}
```

**overflow**

**scrollbar instated ←**
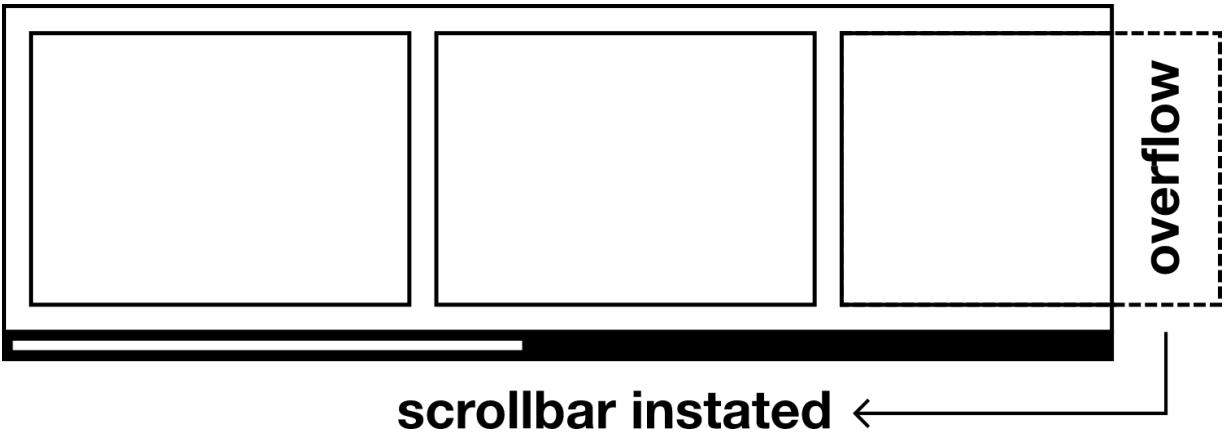
I'm yet to tackle affordance (making the element *look* scrollable), and there's the matter of spacing to address too, but this is the core of the layout. Because it capitalizes on standard browser behavior, it's extremely terse in code and robust — quite unlike your average carousel/slider jQuery plugin.

# The scrollbar

Scrolling is multimodal functionality: there are many ways to do it, and you can choose whichever suits you best. While touch, trackpad gestures, and arrow key presses may be some of the more ergonomic modes, clicking and dragging the scrollbar itself is probably the most familiar, especially to older users on desktop. Having a visible scrollbar has two advantages:

1. It allows for scrolling by dragging the scrollbar handle (or "*thumb*")
2. It indicates scrolling is available by this and other means (*increases affordance*)

Some operating systems and browsers hide the scrollbar by default, but there are CSS methods to reveal it. Webkit and Blink-based browsers proffer the following prefixed properties:

```
::-webkit-scrollbar {
}
::-webkit-scrollbar-button {
}
::-webkit-scrollbar-track {
}
::-webkit-scrollbar-track-piece {
}
::-webkit-scrollbar-thumb {
}
::-webkit-scrollbar-corner {
}
::-webkit-resizer {
}
```

As of version 64, there are also limited opportunities to style the scrollbar in Firefox, with the standardized `scrollbar-color` and `scrollbar-width` properties. Note that the `scrollbar-color` settings only take effect on MacOS where **Show scroll bars** is set to **Always** (in **Settings > General**).

Setting scrollbar colors is a question of aesthetics, which is not really what **Every Layout** is about. But it's important, for reasons of affordance, that scrollbars are *apparent*. The following black and white styles are chosen just to suit **Every Layout's** own aesthetic. You can adjust them as you wish.

```
.reel {
  display: flex;
  /* ↓ We only want horizontal scrolling */
  overflow-x: auto;
  /* ↓ First value: thumb; second value: track */
  scrollbar-color: var(--color-light) var(--color-dark);
}

.reel::-webkit-scrollbar {
  block-size: 1rem;
}

.reel::-webkit-scrollbar-track {
  background-color: var(--color-dark);
}

.reel::-webkit-scrollbar-thumb {
  background-color: var(--color-dark);
  background-image: linear-gradient(var(--color-dark) 0, var(--color-dark)
0.25rem, var(--color-light) 0.25rem, var(--color-light) 0.75rem, var(--color-
dark) 0.75rem);
}
```

Not all properties are supported for these proprietary pseudo-classes. Hence, visually *insetting* the thumb is a question of painting a centered stripe using a `linear-gradient` rather than attempting a margin or border.

**dark to light to dark `linear-gradient`**



# Height

What should the height of a **Reel** instance be? Probably shorter than the viewport, so that the whole **Reel** can be brought into view.

But should we be setting a height at all? Probably not. The best answer is *"as high as it needs to be"*, and is a question of the *content* height.

In the following demonstration, a **Reel** element houses a set of card-like components. The height of the **Reel** is determined by the height of the tallest card, which is the card with the most content. Note that the last element of each "card" (a simple attribution) is pushed to the bottom of the space, by using a **Stack** with `splitAfter="2"`.

For images, which may be very large or use differing aspect ratios, we may want to *set* the **Reel's** height. The common image `block-size` (`height` in a `horizontal-tb` writing mode) should accordingly be `100%`, and the width `auto`. This will ensure the images share a height but maintain their own aspect ratio.

```
.reel {
  block-size: 50vh;
}

.reel > img {
  block-size: 100%;
  width: auto;
}
```

# Child versus descendant selectors

Notice how we are using `.reel > img` and not `.reel img`. We only want to affect the layout of images *if* they are the direct descendants (or *children*) of the **Reel**. Hence, the `>` child combinator.

# Spacing

Spacing out the child elements used to be a surprisingly tricky business. A border is applied around the **Reel** in this case, to give it its shape.

Until recently, we would have had to use margin and the adjacent sibling combinator to add a space between the child elements. We use a logical property to ensure the **Reel** works in both writing directions.

```
.reel > * + * {
  margin-inline-start: var(--s1);
}
```

Now, since we're in a Flexbox context, we are also able to use the `gap` property, which is applied to the parent:

```
.reel {
  gap: var(--s1);
}
```

The main advantage of `gap` is ensuring the margins don't appear in the wrong places when elements wrap. Since the **Reel's** content is not designed to wrap, we shall use the `margin`-based solution instead. It's longer and better supported.

Adding spacing *around* the child elements (in between them and the parent `.reel` element) is a trickier business. Unfortunately, <u>padding interacts unexpectedly with scrolling</u> ↗. The effect on the right hand side is as if there were no padding at all:

### ▨ Padding



Scrolled to end ⎯⎤
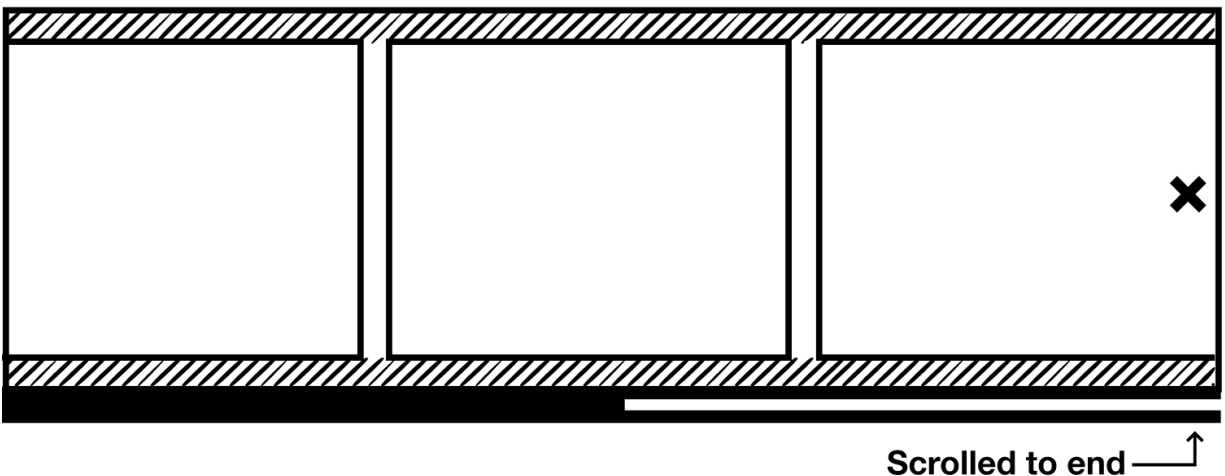
So, if we want spacing around the children, we take a different approach. We add margin to all but the right hand side of each child element, then insert space using pseudo-content on the last of those children.

```
.reel {
  border-width: var(--border-thin);
}

.reel > * {
  margin: var(--s0);
  margin-inline-end: 0;
}

.reel::after {
  content: '';
  flex-basis: var(--s0);
  /* ↓ Default is 1 so needs to be overridden */
  flex-shrink: 0;
}
```

# ⚠ Cascading border styles

Here, we are only applying the border width, and not the border color or style. For this to take effect, the `border-style` has to be applied somewhere already. In **Every Layout's** own stylesheet, the `border-style` is applied *universally*, making `border-width` the only ongoing concern for most border cases:

```
*,
*::before,
*::after {
  border-style: solid;
  /* ↓ 0 by default */
  border-width: 0;
}
```
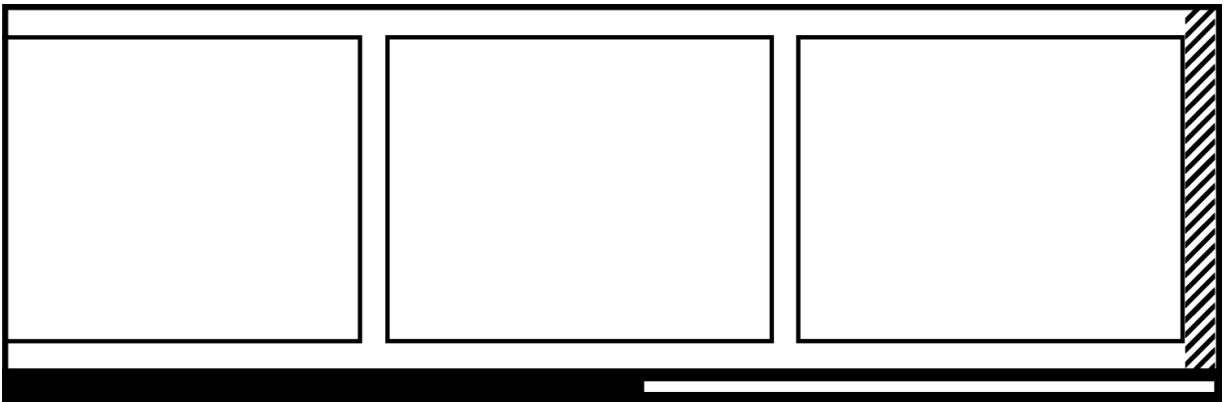
## ⧗ Pseudo-content

The implementation to follow assumes you do not need padding on the **Reel** element itself; the approach using `.reel > * + *` therefore suffices.

That just leaves the space between the children and the scrollbar (where present and visible) to handle. Not a problem, you may think: just add some padding to the bottom of the parent (`class="reel"` here). The trouble is, this creates a *redundant* space wherever the **Reel** is not overflowing and the scrollbar has not been invoked.

Ideally, there would be a pseudo-class for overflowing/scrolling elements. Then we could add the padding selectively. Currently, the `:overflowed-content` pseudo-class ↗ exists as little more than an idea. For now, we can apply the margin, and remove it using JavaScript and a simple `ResizeObserver`. Innately, this is a progressive enhancement technique: where JavaScript is unavailable, or `ResizeObserver` is not supported, the padding does not appear for an overflowing **Reel** — but with little detrimental effect. It just presses the scrollbar up against the content.

```
const reels = Array.from(document.querySelectorAll('.reel'));
const toggleOverflowClass = elem => {
  elem.classList.toggle('overflowing', elem.scrollWidth > elem.clientWidth);
};

for (let reel of reels) {
  if ('ResizeObserver' in window) {
    new ResizeObserver(entries => {
      toggleOverflowClass(entries[0].target);
    }).observe(reel);
  }
}
```

Inside the observer, the **Reel's** `scrollWidth` is compared to its `clientWidth`. If the `scrollWidth` is larger, the `overflowing` class is added.

```
.reel.overflowing {
  padding-block-end: var(--s0);
}
```

# Concatentating classes

See how the `reel` and `overflowing` classes are concatenated. This has the advantage that the `overflowing` styles defined here *only* apply to **Reel** components. That is, they can't unwittingly be applied to other elements and components that might also take an `overflowing` class.

Some developers use verbose namespacing to localize their styles, like prefixing each class associated with a component with the

component name (`reel--overflowing` etc.). Deliberate specification through class concatenation is less verbose and more elegant.

We're not quite done yet, because we haven't dealt with the case of child elements being dynamically removed from the **Reel**. That will affect `scrollWidth` too. We can abstract the class toggling function and add a `MutationObserver` that observes the `childList`. `MutationObserver` is almost ubiquitously supported ↗.

```
const reels = Array.from(document.querySelectorAll('.reel'));
const toggleOverflowClass = elem => {
  elem.classList.toggle('overflowing', elem.scrollWidth > elem.clientWidth);
};

for (let reel of reels) {
  if ('ResizeObserver' in window) {
    new ResizeObserver(entries => {
      toggleOverflowClass(entries[0].target);
    }).observe(reel);
  }

  if ('MutationObserver' in window) {
    new MutationObserver(entries => {
      toggleOverflowClass(entries[0].target);
    }).observe(reel, {childList: true});
  }
}
```

It's fair to say this is a bit *overkill* if only used to add or remove that bit of padding. But these observers can be used for other enhancements, even beyond styling. For example, it may be beneficial to keyboard users for an overflowing/scrollable **Reel** to take the `tabindex="0"` attribution. This would make the element focusable by keyboard and, therefore, scrollable using the arrow keys. If each *child element* is focusable, or includes focusable

content, this may not be necessary: focusing an element automatically brings it into view by changing the scroll position.

# Use cases

The **Reel** is a robust and efficient antidote to carousel/slider components. As already discussed and demonstrated, it is ideal for browsing categories of things: movies; products; news stories; photographs.

In addition, it can be used to supplant button-activated menu systems. What Bradley Taunt calls sausage links ↗ may well be more usable than "hamburger" menus for many. For such a use case, the visible scrollbar is probably rather heavy-handed. This is why the ensuing custom element implementation includes a Boolean `noBar` property.

This interactive demo is only available on the **Every Layout** site ↗.

There's no reason why the links have to be shaped like sausages, of course! That's just an etymological hangover. One thing to note, however, is the lack of affordance the omitted scrollbar represents. So long as the last visible child element on the right is partly obscured, it's relatively clear overflow is occurring and the ability to scroll is present. If this is not the case, it may appear that all of the available elements are already in view.
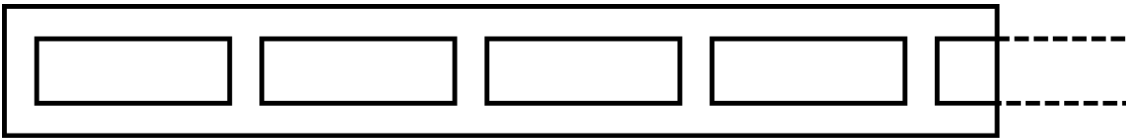
**"I need to scroll"**
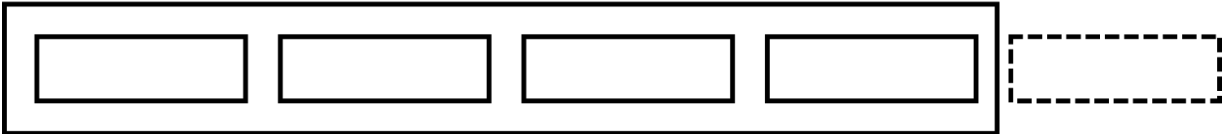


**"Everything seems to be here"**

From a layout perspective, you can reduce the likelihood of *"Everything seems to be here"* by avoiding certain types of width. Percentage widths like `25%` or `33.333%` are going to be problematic because—at least in the absence of spacing—this will fit the elements exactly within the space.

In addition, you can indicate the availability of scrolling by other means. You can capitalize on the observers' `overflowing` class to reveal a textual instruction (reading *"scroll for more"*, perhaps):

```
.reel.overflowing + .instruction {
  display: block;
}
```

However, this is not reactive to the *current* scroll position. You might use additional scripting to detect when the element is scrolled all the way to either side, and add `start` or `end` classes accordingly. The ever-innovative Lea Verou devised a way to achieve something similar using images and CSS alone ↗. Shadow background

images take `background-attachment: scroll` and remain at either end of the scrollable element. "Shadow cover" background images take `background-attachment: local`, moving *with* the content. Whenever the user reaches one end of the scrollable area, these "shadow cover" backgrounds obscure the shadows beneath them.

These considerations don't strictly relate to layout, more to communication, but are worth exploring further to improve usability.

# The generator

Use this tool to generate basic **Reel** CSS and HTML. You would want to include the `ResizeObserver` script along with the code generated. Here's a version implemented as an Immediately Invoked Function Expression (IIFE):

```
(function() {
  const className = 'reel';
  const reels = Array.from(document.querySelectorAll(`.${className}`));
  const toggleOverflowClass = elem => {
    elem.classList.toggle('overflowing', elem.scrollWidth > elem.clientWidth);
  };

  for (let reel of reels) {
    if ('ResizeObserver' in window) {
      new ResizeObserver(entries => {
        toggleOverflowClass(entries[0].target);
      }).observe(reel);
    }

    if ('MutationObserver' in window) {
      new MutationObserver(entries => {
        toggleOverflowClass(entries[0].target);
      }).observe(reel, {childList: true});
    }
  }
})();
```

The code generator tool is only available <u>in the accompanying site</u> ↗. Here is the basic solution, with comments. The code that hides the scrollbar has been removed for brevity, but is available via the `noBar` property in the <u>custom element implementation</u>.

# HTML

```
<div class="reel">
  <div><!-- child element --></div>
  <div><!-- another child element --></div>
  <div><!-- etc --></div>
  <div><!-- etc --></div>
</div>
```

# CSS

```css
.reel {
  /* ↓ Custom properties for ease of adjustment */
  --space: 1rem;
  --color-light: #fff;
  --color-dark: #000;
  --reel-height: auto;
  --item-width: 25ch;
  display: flex;
  block-size: var(--reel-height);
  /* ↓ Overflow */
  overflow-x: auto;
  overflow-y: hidden;
  /* ↓ For Firefox */
  scrollbar-color: var(--color-light) var(--color-dark);
}

reel-l::-webkit-scrollbar {
  /*
  ↓ Instead, you could make the scrollbar height
  a variable too. This is left as an exercise
  (be mindful of the linear-gradient!)
  */
  block-size: 1rem;
}

reel-l::-webkit-scrollbar-track {
  background-color: var(--color-dark);
}

reel-l::-webkit-scrollbar-thumb {
  background-color: var(--color-dark);
  /* ↓ Linear gradient 'insets' the white thumb within the black bar */
  background-image: linear-gradient(var(--color-dark) 0, var(--color-dark)
0.25rem, var(--color-light) 0.25rem, var(--color-light) 0.75rem, var(--color-
dark) 0.75rem);
}

.reel > * {
  /*
  ↓ Just a `width` wouldn't work because
  `flex-shrink: 1` (default) still applies
   */
  flex: 0 0 var(--item-width);
}

reel-l > img {
  /* ↓ Reset for images */
  block-size: 100%;
  flex-basis: auto;
  width: auto;
}
```

```css
.reel > * + * {
  margin-inline-start: var(--space);
}

.reel.overflowing:not(.no-bar) {
  /* ↓ Only apply if there is a scrollbar (see the JavaScript) */
  padding-block-end: var(--space);
}

/* ↓ Hide scrollbar with `no-bar` class */
.reel.no-bar {
  scrollbar-width: none;
}

.reel.no-bar::-webkit-scrollbar {
  display: none;
}
```

# JavaScript

Just an Immediately Invoked Function Expression (IIFE):

```
(function() {
  const className = 'reel';
  const reels = Array.from(document.querySelectorAll(`.${className}`));
  const toggleOverflowClass = elem => {
    elem.classList.toggle('overflowing', elem.scrollWidth > elem.clientWidth);
  };

  for (let reel of reels) {
    if ('ResizeObserver' in window) {
      new ResizeObserver(entries => {
        for (let entry of entries) {
          toggleOverflowClass(entry.target);
        }
      }).observe(reel);
    }

    if ('MutationObserver' in window) {
      new MutationObserver(entries => {
        for (let entry of entries) {
          toggleOverflowClass(entry.target);
        }
      }).observe(reel, {childList: true});
    }
  }
})();
```

# The component

A custom element implementation of the Reel is available for download ↗.

# Props API

The following props (attributes) will cause the **Reel** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|------|------|---------|-------------|
| **itemWidth** | `string` | `"auto"` | The width of each item (child element) in the Reel |
| **space** | `string` | `"var(--s0)"` | The space between Reel items (child elements) |
| **height** | `string` | `"auto"` | The height of the Reel itself |
| **noBar** | `boolean` | `false` | Whether to display the scrollbar |

# Examples

## Card slider

In this example, cards are given a `20rem` width. Note that a "fixed" width is acceptable in this circumstance, since the horizontal space is provisioned as needed, and wrapping takes care of text and inline elements: the cards are allowed to grow *downwards*.

```
<reel-l itemWidth="20rem">
  <box-l>
    <stack-l>
      <!-- card content -->
    <stack-l>
  </box-l>
  <box-l>
    <stack-l>
      <!-- card content -->
    <stack-l>
  </box-l>
  <box-l>
    <stack-l>
      <!-- card content -->
    <stack-l>
  </box-l>
  <!-- ad infinitum -->
</reel-l>
```

## Slidable links

Note the use of `role="list"` and `role="listitem"` to communicate the component as a list in screen reader output. This is customary for navigation regions.

```
<reel-l role="list" noBar>
  <div role="listitem">
    <a class="cta" href="/path/to/home">Home</a>
  </div>
  <div role="listitem">
    <a class="cta" href="/path/to/about">About</a>
  </div>
  <div role="listitem">
    <a class="cta" href="/path/to/pricing">Pricing</a>
  </div>
  <div role="listitem">
    <a class="cta" href="/path/to/docs">Documentation</a>
  </div>
  <div role="listitem">
    <a class="cta" href="/path/to/testimonials">Testimonials</a>
  </div>
</reel-l>
```

# The Imposter

## The problem

Positioning in CSS, using one or more instances of the `position` property's `relative`, `absolute`, and `fixed` values, is like manually overriding web layout. It is to switch off automatic layout and take matters into your own hands. As with piloting a commercial airliner, this is not a responsibility you would wish to undertake except in rare and extreme circumstances.

In the **Frame** documentation, you were warned of the perils of eschewing the browser's standard layout algorithms:

🖼️*When you give an element `position: absolute`, you remove it from the natural flow of the document. It lays itself out as if the elements around it don't exist. In most circumstances this is highly undesirable, and can easily lead to problems like overlapping and obscured*

*content.* 🖼️

But what if you *wanted* to obscure content, by placing other content over it? If you've been working in web development for more than

23 minutes, it's likely you have already done this, in the incorporation of a dialog element, "popup", or custom dropdown menu.

The purpose of the **Imposter** element is to add a general purpose *superimposition* element to your layouts suite. It will allow the author to centrally position an element over the viewport, the document, or a selected "positioning container" element.

# The solution

There are many ways to centrally position elements vertically, and many more to centrally position them horizontally (some alternatives were mentioned as part of the **Center** layout). However, there are only a few ways to position elements centrally *over* other elements/content.

The contemporaneous approach is to use CSS Grid ↗. Once your grid is established, you can arrange content by grid line number. The concept of flow ↗ is made irrelevant, and overlapping is eminently achievable wherever desired.

```
grid-area: 2 / 2 / 5 / 8
```
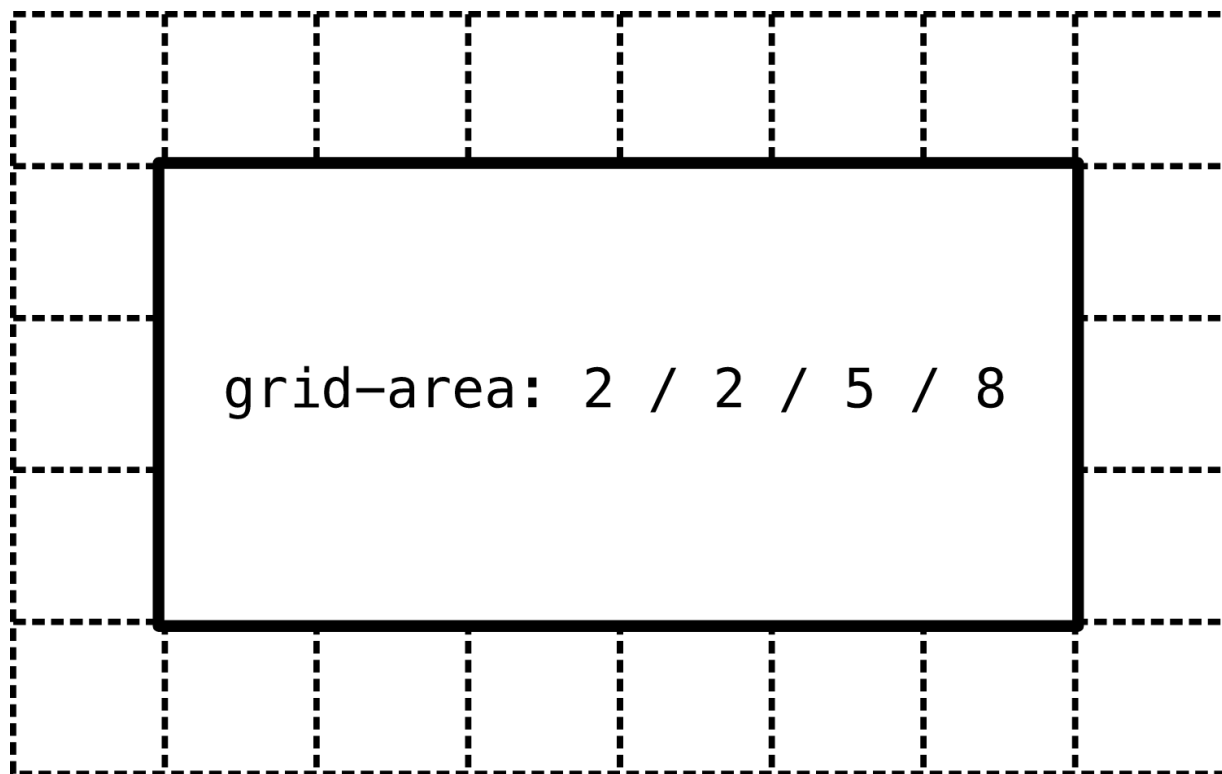
## Source order and layers

Whether you are positioning content according to Grid lines or doing so with the `position` property, which elements appear *over* which is, by default, a question of source order. That is: if two elements share the same space, the one that appears *above* the other will be the one that comes last in the source.
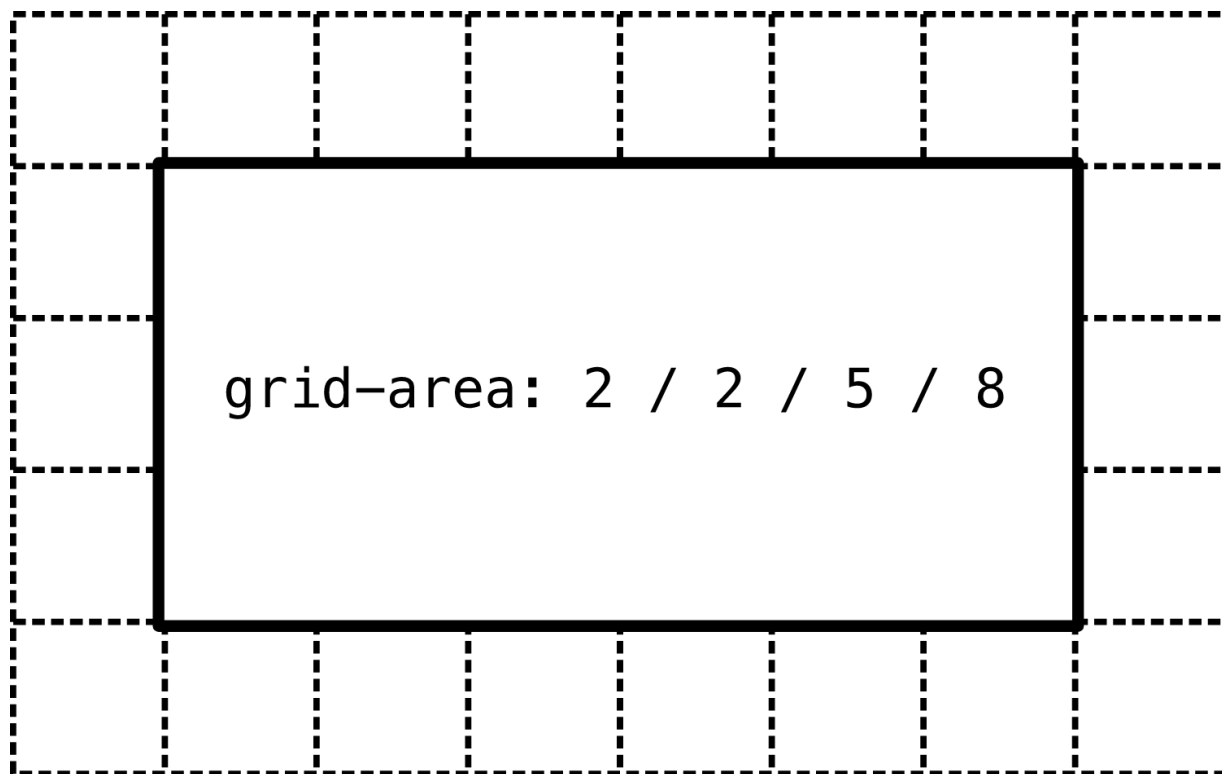
*Since you can place any elements along any grid lines you wish, an overlapping last-in-source element can appear first down the vertical axis*

This is often overlooked, and some believe that `z-index` needs to accompany `position: absolute` in all cases to determine the "layering". In fact, `z-index` is only necessary where you want to layer positioned elements irrespective of their source order. It's another kind of override, and should be avoided wherever possible.

An arms race of escalating `z-index` values is often cited as one of those irritating but necessary things you have to deal with using CSS. I rarely have `z-index` problems, because I rarely use positioning, and I'm mindful of source order when I do.

CSS Grid does not precipitate a general solution, because it would only work where your positioning element is set to `display: grid`

ahead of time, and the column/row count is suitable. We need something more flexible.
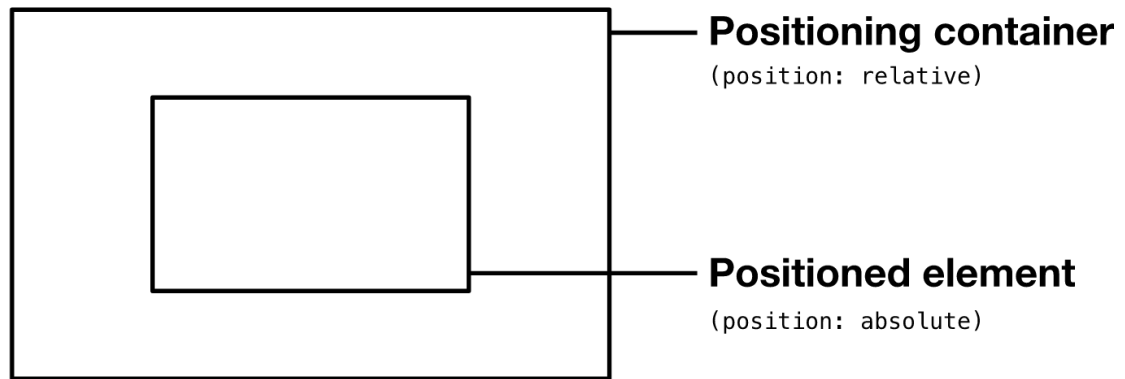
# Positioning

You can position an element to one of three things ("*positioning contexts*" from here on):

1. The viewport
2. The document
3. An ancestor element

To position an element relative to the viewport, you would use `position: fixed`. To position it relative to the document, you use `position: absolute`.

Positioning it relative to an ancestor element is possible where that element (the "*positioning container*" from here on) is also explicitly positioned. The easiest way to do this is to give the ancestor element `position: relative`. This sets the localized positioning context *without* moving the position of the ancestor element, or taking it out of the document flow.

*The `static` value for the `position` property is the default, so you will rarely see or use it except to reset the value.*

# Centering

How do we position the **Imposter** element over the *center* of the document, viewport, or positioning container? For positioned elements, techniques like `margin: auto` or `place-items: center` do not work. In *manual override*, we have to use a combination of the `top`, `left`, `bottom`, and/or `right` properties. Importantly, the values for each of these properties relate to the positioning context—not to the immediate parent element.
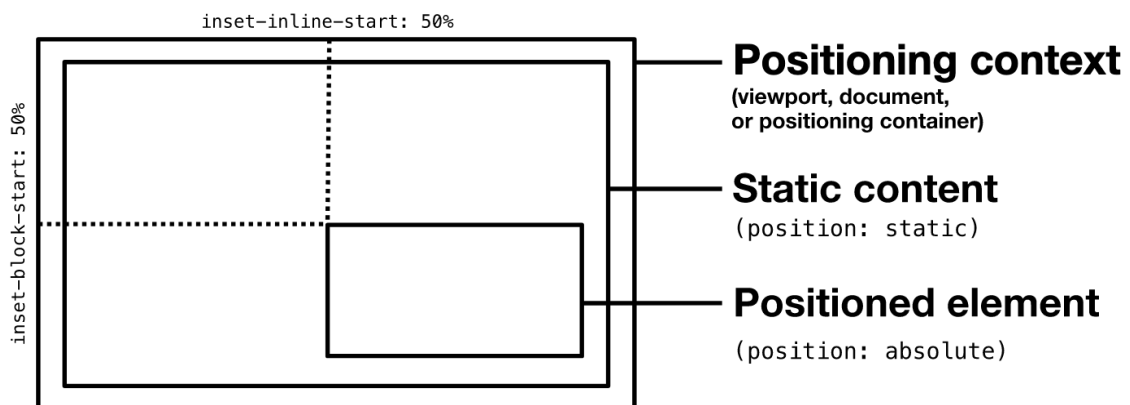


*The `static` value for the `position` property is the default, so you will rarely see or use it.*

So far, so bad: we want the element itself to be centered, not its top corner. Where we know the `width` of the element, we can compensate by using negative margins. For example, `margin-inline-start: -20rem` and `margin-block-start: -10rem` will recenter an element that is `40rem` wide and `20rem` tall (the negative value is always half the dimension).
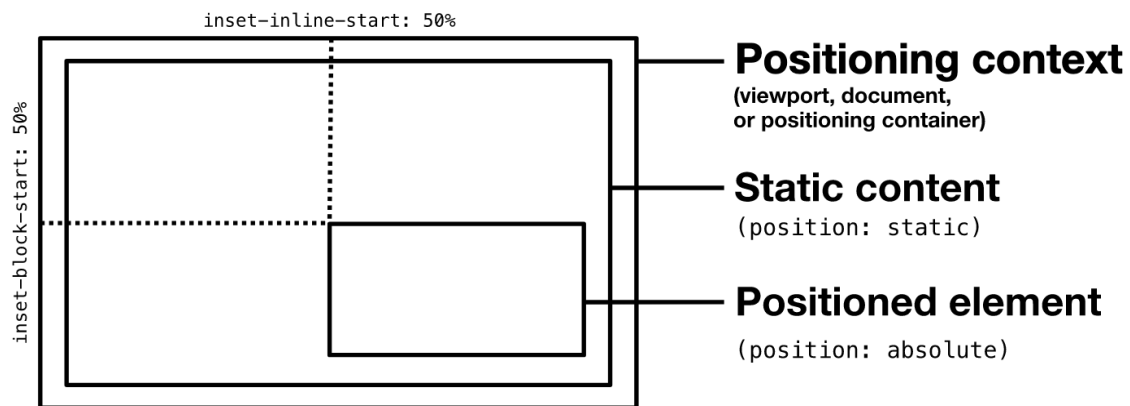


We avoid hard coding dimensions because, like positioning, it dispenses with the browser's algorithms for arranging elements according to available space. Wherever you code a fixed width on an element, the chances of that element or its contents becoming obscured on *somebody's* device *somewhere* are close to inevitable. Not only that, but we might not know the element's width or height ahead of time. So we wouldn't know which negative margin values with which to complement it.

Instead of designing layout, we design *for layout*, allowing the browser to have the final say. In this case, it's a question of using transforms. The `transform` property arranges elements according to their own dimensions, whatever they are at the given time. In short: `transform: translate(-50%, -50%)` will *translate* the element's position

by -50% of its *own* width and height respectively. We don't need to know the element's dimensions ahead of time, because the browser can calculate them and act on them for us.

Centering the element over its positioning container, no matter its dimensions, is therefore quite simple:

```
.imposter {
  /* ↓ Position the top left corner in the center */
  position: absolute;
  inset-block-start: 50%;
  inset-inline-start: 50%;
  /* ↓ Reposition so the center of the element
  is the center of the positioning container */
  transform: translate(-50%, -50%);
}
```

It should be noted at this point that a block-level **Imposter** element set to `position: absolute` no longer takes up the available space along the element's writing direction (usually horizontal; left-to-right). Instead, the element "shrink wraps" its content as if it were inline.



```
transform: translate(-50%, -50%)
```
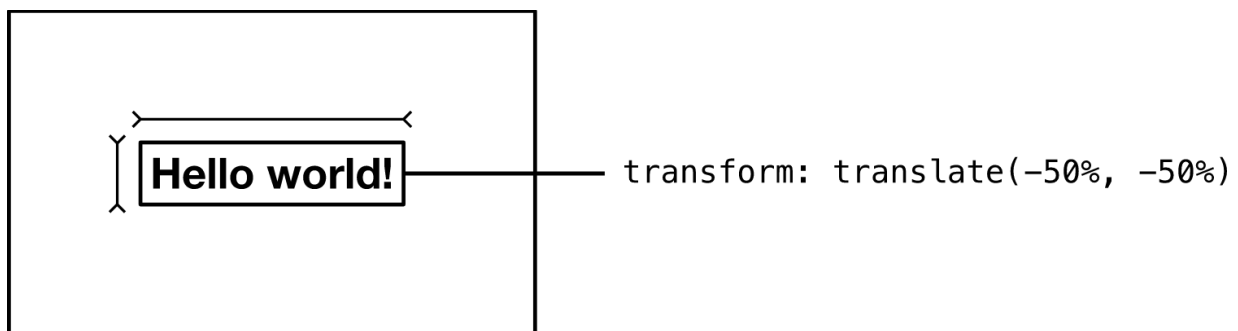
By default, the element's `width` will be 50%, or less if its content takes up less than 50% of the positioning container. If you add an

explicit `width` or `height`, it will be honoured *and* the element will continue to be centered within the positioning container — the internal translation algorithm sees to that.

# Overflow

What if the positioned **Imposter** element becomes wider or taller than its positioning container? With careful design and content curation, you should be able to create the generous tolerances that prevent this from happening under most circumstances. But it may still happen.

By default, the effect will see the **Imposter** *poking out* over the edges of the positioning container — and may be in danger of obscuring content around that container. In the following illustration, an **Imposter** is taller than its positioning container.
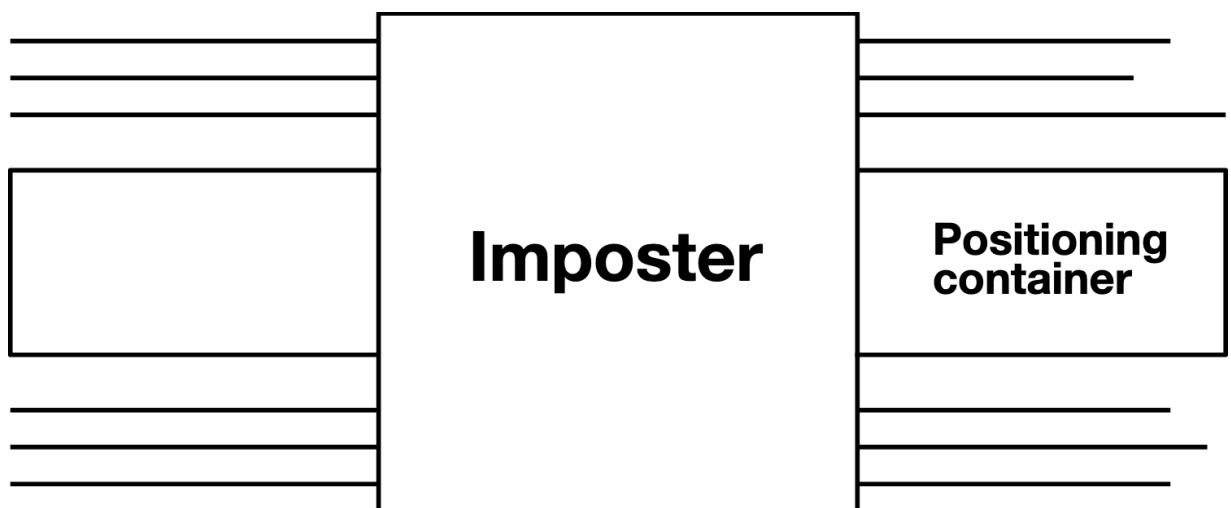


Since `max-width` and `max-height` override `width` and `height` respectively, we can allow authors to set dimensions—or minimum

dimensions—but still ensure the element is contained. All that's left is to add `overflow: auto` to ensure the constricted element's contents can be scrolled into view.

```
.imposter {
  position: absolute;
  /* ↓ equivalent to `top` in horizontal-tb writing mode */
  inset-block-start: 50%;
  /* ↓ equivalent to `left` in horizontal-tb writing mode */
  inset-inline-start: 50%;
  transform: translate(-50%, -50%);
  /* ↓ equivalent to `max-width` in horizontal-tb writing mode */
  max-inline-size: 100%;
  /* ↓ equivalent to `max-height` in horizontal-tb writing mode */
  max-block-size: 100%;
}
```

## Margin

In some cases, it will be desirable to have a minimum gap (space; margin; whatever you want to call it) between the **Imposter** element and the inside edges of its positioning container. For two reasons, we can't achieve this by adding padding to the positioning container:

1. It would inset any static content of the container, which is likely not to be a desirable visual effect
2. Absolute positioning does not respect padding: our **Imposter** element would ignore and overlap it

The answer, instead, is to adjust the `max-width` and `max-height` values. The `calc()` function is especially useful for making these kinds of

adjustments.

```
.imposter {
  position: absolute;
  inset-block-start: 50%;
  inset-inline-start: 50%;
  transform: translate(-50%, -50%);
  max-inline-size: calc(100% - 2rem);
  max-block-size: calc(100% - 2rem);
}
```

The above example would create a minimum gap of `1rem` on all sides: the `2rem` value is `1rem` removed for each end.
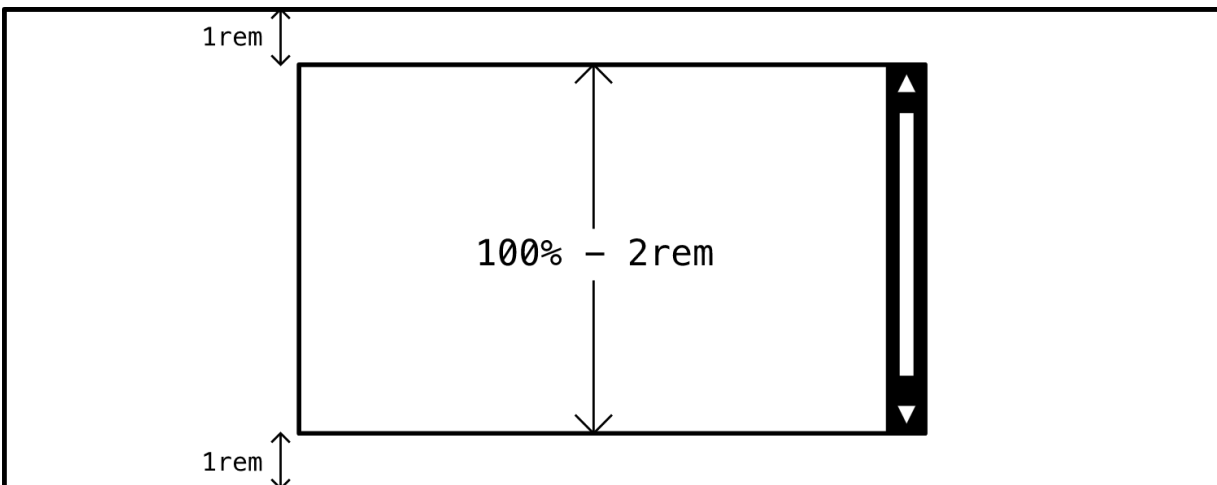


# Fixed positioning

Where you wish the **Imposter** to be fixed relative to the *viewport*, rather than the document or an element (read: positioning container) within the document, you should replace `position: absolute` with `position: fixed`. This is often desirable for dialogs, which should

follow the user as they scroll the document, and remain in view until tended to.

In the following example, the **Imposter** element has a `--positioning` custom property with a default value of `absolute`.

```
.imposter {
  position: var(--positioning, absolute);
  inset-block-start: 50%;
  inset-inline-start: 50%;
  transform: translate(-50%, -50%);
  max-inline-size: calc(100% - 2rem);
  max-block-size: calc(100% - 2rem);
}
```

As described in the **Every Layout** article Dynamic CSS Components Without JavaScript ↗, you can override this default value inline, inside a `style` attribute for special cases:

```
<div class="imposter" style="--positioning: fixed">
  <!-- imposter content -->
</div>
```

In the custom element implementation to follow (under **The Component**) an equivalent mechanism takes the form of a Boolean `fixed` prop'. Adding the `fixed` attribute overrides the `absolute` positioning that is default.

## ⚠ Fixed positioning and Shadow DOM

In most cases, using a `fixed` value for `position` will position the element relative to the viewport. That is, any candidate positioning containers (positioned ancestor elements) will be ignored.

However, a `shadowRoot` ↗ host element will be treated as the outer element of a nested document. Therefore, any element with `position: fixed` found inside Shadow DOM will instead be positioned relative to the `shadowRoot` host (the element that hosts the Shadow DOM). In effect, it becomes a positioning container as in previous examples.



**Normal element**                    **Shadow host**
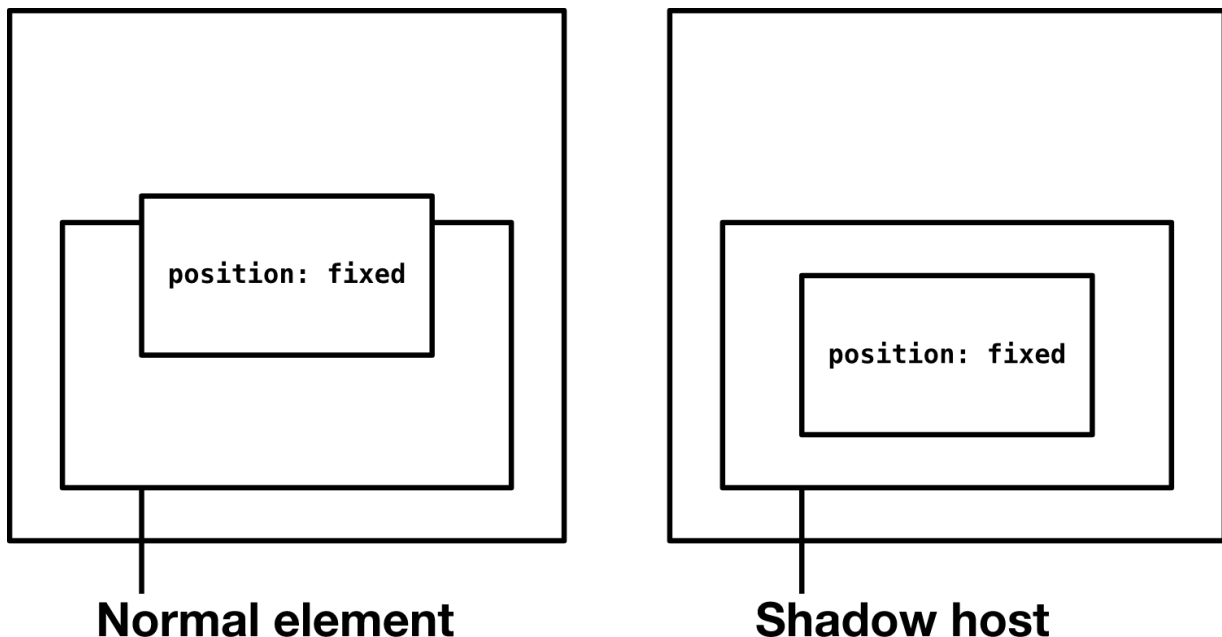
# Use cases

Wherever content needs to be deliberately obscured, the **Imposter** pattern is your friend. It may be that the content is yet to be made

available. In which case, the **Imposter** may consist of a call-to-action to unlock that content.

It may be that the artifacts obscured by the **Imposter** are more decorative, and do not need to be revealed in full.

When creating a dialog using an **Imposter**, be wary of the accessibility considerations that need to be included—especially those relating to keyboard focus management. Inclusive Components ↗ has a chapter on dialogs which describes these considerations in detail.

# The generator

Use this tool to generate basic **Imposter** CSS and HTML.

The code generator tool is only available in the accompanying documentation site ↗. Here is the basic solution, with comments. The `.contain` version contains the element within its positioning container, and handles overflow.

## CSS

```css
.imposter {
  /* ↓ Choose the positioning element */
  position: var(--positioning, absolute);
  /* ↓ Position the top left corner in the center */
  inset-block-start: 50%;
  inset-inline-start: 50%;
  /* ↓ Reposition so the center of the element
  is the center of the container */
  transform: translate(-50%, -50%);
}

.imposter.contain {
  /* ↓ Include a unit, or the calc function will be invalid */
  --margin: 0px;
  /* ↓ Provide scrollbars so content is not obscured */
  overflow: auto;
  /* ↓ Restrict the height and width, including optional
  spacing/margin between the element and positioning container */
  max-inline-size: calc(100% - (var(--margin) * 2));
  max-block-size: calc(100% - (var(--margin) * 2));
}
```

# HTML

An ancestor with a positioning value of `relative` or `absolute` must be provided. This becomes the "positioning container" over which the `.imposter` element is positioned. In the following example, a simple `<div>` with the `position: relative` inline style is used.

```html
<div style="position: relative">
  <p>Static content</p>
  <div class="imposter">
    <p>Superimposed content</p>
  </div>
</div>
```

# The component

A custom element implementation of the Imposter is available for download ↗.

# Props API

The following props (attributes) will cause the **Imposter** component to re-render when altered. They can be altered by hand—in browser developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|---|---|---|---|
| **breakout** | `boolean` | `false` | Whether the element is allowed to break out of the container over which it is positioned |
| **margin** | `string` | `0` | The minimum space between the element and the inside edges of the positioning container over which it is placed (where `breakout` is not applied) |
| **fixed** | `boolean` | `false` | Whether to position the element relative to the viewport |

# Examples

## Demo example

The code for the demo in the **Use cases** section. Note the use of `aria-hidden="true"` on the superimposed sibling content. It's likely the superimposed content should be unavailable to screen readers, since it is unavailable (or at least mostly obscured) visually.

```
<div style="position: relative">
  <text-l words="150" aria-hidden="true"></text-l>
  <imposter-l>
    <box-l style="background-color: var(--color-light)">
      <p class="h4"><strong>You can't see all the content, because of this box.
</strong></p>
    </box-l>
  </imposter-l>
</div>
```

# Dialog

The **Imposter** element could take the ARIA attribute `role="dialog"` to be communicated as a dialog in screen readers. Or, more simply, you could just place a `<dialog>` inside the **Imposter**. Note that the **Imposter** takes `fixed` here, to switch from an `absolute` to `fixed` position. This means the dialog would stay centered in the viewport as the document is scrolled.

```
<imposter-l fixed>
  <dialog aria-labelledby="message">
    <p id="message">It's decision time, sunshine!</p>
    <button type="button">Yes</button>
    <button type="button">No</button>
  </dialog>
</imposter-l>
```

# The Icon

## The problem

Most of the layouts in **Every Layout** take the form of *block components*, if you'll excuse the expression. That is, they set a block-level ↗ context wherein they affect the layout of child elements in their control. As you will discover in **Boxes**, elements with display values of either `block`, `flex`, or `grid` are themselves block-level (`flex` and `grid` differ by affecting their child elements in a special way).

Here, we shall be looking at something a lot smaller, and it doesn't get much smaller than an icon. This will be the first layout for which the custom element will retain its default `inline` display mode.

Getting things to line up and look right, *inline* can be a precarious business. When it comes to icons, we have to worry about things like:

- The distance between the icon and the text
- The height of the icon compared with the height of the text

- The vertical alignment of the icon to the text
- What happens when the text comes *after* the icon, rather than before
- What happens when you resize the text

# A simple icon

Before looking into any of these, I'm first going to give you a really quick crash course in SVG iconography, since SVG is the *de facto* iconography format on the web. Consider the following code:

```
<svg viewBox="0 0 10 10" width="0.75em" height="0.75em" stroke="currentColor"
stroke-width="2">
  <line x1="1" y1="1" x2="9" y2="9" />
  <line x1="9" y1="1" x2="1" y2="9" />
</svg>
```

This defines a simple icon: a cross. Let me explain each of the key features:

- `viewBox`: This defines the coordinate system for the SVG. The `0 0` part means *"count from the top left corner"* and the *"10 10"* part means give the SVG "canvas" 10 horizontal, and 10 vertical, coordinates. We are defining a square, since all our icons will occupy a square space.
- `width` **and** `height`: This sets the size of the icon. I shall explain why it uses the `em` unit, and is set to `0.75em` shortly. For now, be aware that we set the width and height *on* the SVG, rather than in CSS, because we want to keep the icon small even if CSS

fails to load. SVGs are displayed very large in most browsers by default.

- **`stroke` and `stroke-width`**: These presentational attributes give the `<line />` elements visible form. They can be written, or overridden, in CSS. But since we aren't using many, it's better to make sure these too are CSS-independent.
- **`<line />`**: The `<line />` element draws a simple line. Here we have one drawn from the top left to the bottom right, followed by one drawn from the top right to the bottom left (making our cross). I'm using `1` and `9`, not `0` and `10`, to compensate for the line's `stroke-width` of `2`. Otherwise the line would overflow the SVG "canvas".

1,1   9,1

1,9   9,9

stroke-width: 2

stroke: currentColor
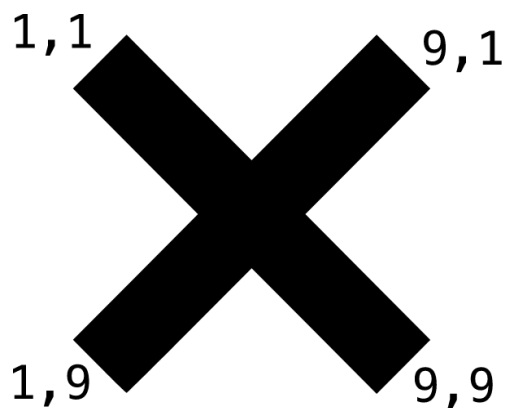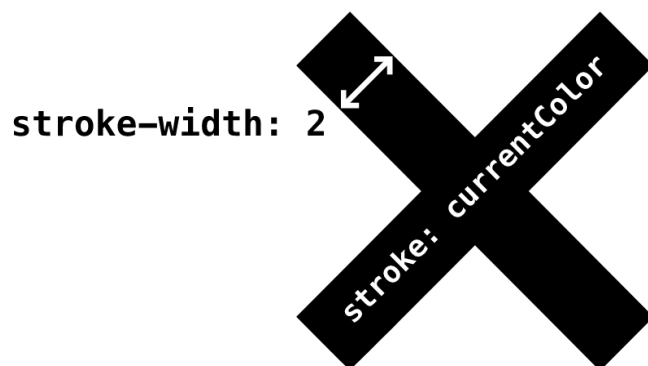
# Coordinates          # Line styles

There are many ways to draw the same cross shape. Perhaps the most efficient is to use a `<path />` element. A path lets you place all the coordinates in one `d` attribute. The `M` symbol marks the start of each line's separate coordinates:

```
<svg viewBox="0 0 10 10" width="0.75em" height="0.75em" stroke="currentColor"
stroke-width="2">
  <path d="M1,1 9,9 M9,1 1,9" />
</svg>
```

When your SVG data is this terse, there's no reason not to include it *inline* rather than using an `<img />` pointing to an SVG `src` file.

There are other advantages besides being able to dispense with an HTTP request, like the ability to use `currentColor` as shown. This keyword makes your inline SVG adopt the `color` of the surrounding text. For the demo icons to follow, the icons are included via the `<use>` element, which references one of many icon `<symbol>`s defined in a single **icons.svg** file (and therefore HTTP request). The `currentColor` technique still works when referencing SVG data in this way.

```
<svg class="icon">
  <use href="/images/icons/icons.svg#cross"></use>
</svg>
```

In any case, SVG is an efficient *scalable* format, much better suited to iconography than raster images like PNGs, and without the attendant accessibility issues ↗ of icon fonts.

Our task here is to create a dependable SVG canvas for square icons, and ensure it fits seamlessly alongside text, with the minimum of manual configuration.

# The solution

## Vertical alignment

As the previous note on `currentColor` suggests, we are going to treat our icons like text, and get them to accompany text as seamlessly as possible. Fortunately, the SVG will sit on the text's `baseline` by default, as if it were a letter.

For taller icons, you may expect to be able to use `vertical-align: middle`. However, contrary to popular belief this does not align around the vertical middle of the font, but *the vertical middle of the lowercase letters of the font*. Hence, the result will probably be undesirable.



`vertical-align: baseline`            `vertical-align: middle`

Instead, adjusting the vertical alignment for a taller icon will probably be a matter of supplying the `vertical-align` attribute with a length. This length represents the distance above the baseline, and can take a negative value.

# ✕Close

`vertical-align: -0.125em`

For our **Icon** layout, we shall stick to sitting icons on the baseline. This is the most robust approach since icons that hang below the baseline may collide with a successive line of text where wrapping occurs.

## Matching height

A suitable icon height, starting at the baseline, depends somewhat on the casing of the font and the presence or absence of descenders ↗. Where the letters are all lowercase, and include descenders, things can look particularly unbalanced.

This interactive demo is only available on the **Every Layout** site ↗.

This perceptual issue can be mitigated by ensuring the first letter of the icon's accompanying text is always uppercase, and that the icon itself is the height of an uppercase letter.

Actually matching the uppercase letter height *per font* is another matter. You might expect `1em` to be the value, but that is rarely the case. `1em` more closely matches the height of the font itself. By making selections of text from a few fonts, you'll see the font height is often taller than its capital letters. To put it another way: `1em` corresponds to font metrics, not letter metrics.

In my experimentation, I found that `0.75em` more closely matches uppercase letter height. Hence, the presentation attributes for my cross icon being `0.75em` each, to make a square following the precedent set by the `viewBox`.

```
<svg viewBox="0 0 10 10" width="0.75em" height="0.75em" stroke="currentColor"
stroke-width="2">
  <path d="M1,1 9,9 M9,1 1,9" />
</svg>
```

✕ Close  ✕ Close  ✕ Close  ✕ Close

*From left to right: Arial, Georgia, Trebuchet, and Verdana. For each icon, `0.75em` matches the uppercase letter height.*

However, the emerging `cap` unit ↗ promises to evaluate the individual font for a more accurate match. Since it is currently not supported very well, we can use `0.75em` as a fallback in our CSS:

```
.icon {
  height: 0.75em;
  height: 1cap;
  width: 0.75em;
  width: 1cap;
}
```

Better to have the `0.75em` values in the CSS as well, in case an author has omitted the presentational attributes.

## Height and width are not logical?

Generally, we should use logical properties to make our measurements compatible with a wider range of languages. In this case, since the `width` and `height` are the same, the writing mode doesn't matter. We use `width` and `height` over `inline-size` and `block-size` for the more longstanding browser support.

As Andy wrote in <u>Relative Sizing With EM units ↗</u>, the icon will now scale automatically with the text: `0.75em` is relative to the `font-size` for the context. For example:

```
.small {
  font-size: 0.75em;
}

.small .icon {
  /* Icon height will automatically be
     0.75 * 0.75em */
}

.big {
  font-size: 1.25em;
}

.big .icon {
  /* Icon height will automatically be
     1.25 * 0.75em */
}
```

This interactive demo is only available on the **Every Layout** site ↗.

## Matching lowercase letter height

If your icon text is to be lowercase, you may bet better results by matching the icon height to a lowercase letter. This is already possible using the ex unit which pertains to the height of a lowercase 'x'. You might want to enforce lowercase type as well.

```
.icon {
  width: 1ex;
  height: 1ex;
}

/* Assume this is the parent or ancestor element for the icon */
.with-icon {
  text-transform: lowercase;
}
```

# Spacing between icon and text

To establish how we manage the spacing of our icons, we have to weigh efficiency against flexibility. In design systems, sometimes inflexibility can be a virtue, since it enforces regularity and consistency.

Consider our cross icon in context, inside a button element and alongside the text "Close":

```
<button>
  <svg class="icon">...</svg> Close
</button>
```

Note the space (unicode point U+0020, if you want to be scientific) between the SVG and the text node. This adds a visible space between the icon and the text, as I'm sure you can imagine. Now, you don't have control over this space. Even adding an extra space of the same variety in the source will not help you, since it will be collapsed down to a single space by the browser. But it is a *suitable* space, because it matches the space between any words in the same context. Again, we are treating the icon like text.

A couple of other neat things about using simple text spacing with your icons:

1. If the icon appears on its own, the space does not appear (making the spacing inside the button look uneven) even if it remains in the source. It is collapsed under this condition too.

2. You can use the `dir` attribute with the `rtl` (right-to-left) value to swap the icon visually from left to right. The space will still appear *between* the icon and text becuse the text direction, including the spacing, has been reversed.

```
<button dir="rtl">
  <svg class="icon"></svg> Close
</button>
```



It's neat when we can use a fundamental feature of HTML to reconfigure our design, rather than having to write bespoke styles and attach them to arbitrary classes.

If you *do* want control of the length of the space, you have to accept an increase in complexity, and a diminishing of reusability: It's not really possible without setting a context for the icon in order to remove the extant space first. In the following code, the context is set by the `.with-icon` element and the word space eliminated by making it `inline-flex`.

```
.icon {
  height: 0.75em;
  height: 1cap;
  width: 0.75em;
  width: 1cap;
}

.with-icon {
  display: inline-flex;
  align-items: baseline;
}
```

The `inline-flex` display value behaves as its name suggests: it
creates a `flex` context, but the element creating that context itself
displays as inline. Employing `inline-flex` eliminates the word space,
freeing us to create a space/gap purely with margin.

**✕ Close**  **✕Close**  **✕ Close**
default space    display: inline-flex    with margin

Now we can add some margin. How do we add it in such a way that
it always appears in the correct place, like the space did? If I use
`margin-right: 0.5em`, it will work where the icon is on the left, before
the text. But if I add `dir="rtl"` that margin remains on the right,
creating a gap on the wrong side.

The answer is <u>CSS Logical Properties ↗</u>. While `margin-top`, `margin-`
`right`, `margin-bottom`, and `margin-left` all pertain to *physical*
orientation and placement, logical properties honor instead the
direction of the *content*. This differs depending on the flow and
writing direction, as explained in **Boxes**.

In this case, I would use `margin-inline-end` with the icon element. This applies margin *after* the element in the direction of the text (hence `-inline-`):

```css
.icon {
  height: 0.75em;
  height: 1cap;
  width: 0.75em;
  width: 1cap;
}

.with-icon {
  display: inline-flex;
  align-items: baseline;
}

.with-icon .icon {
  margin-inline-end: var(--space, 0.5em);
}
```



One disadvantage with this more flexible approach to spacing is that the margin remains even where text is not supplied. Unfortunately, although you can target lone elements with `:only-`

`child`, you cannot target lone elements *unaccompanied by text nodes*. So it is not possible to remove the margin with just CSS.

Instead, you could just remove the `with-icon` class, since it only creates the conditions for manual spacing by `margin`. This way, spaces will remain (and collapse automatically as described). In the custom element implementation to come, only if the `space` prop is supplied will the `<icon-l>` be made an `inline-flex` element, and the word space removed.

# Use cases

You've seen icons before, right? Most frequently you find them as part of button controls or links, supplementing a label with a visual cue. Too often our controls *only* use icons. This is okay for highly familiar icons/symbols like the cross icon from previous examples, but more esoteric icons should probably come with a textual description — at least in the early stages of the interface's usage.

Where no (visible) textual label is provided, it's important there is at least a screen reader perceptible label of some form. You can do one of the following:

1. Visually hide ↗ a textual label (probably supplied in a `<span>`)
2. Add a `<title>` to the `<svg>`
3. Add an `aria-label` directly to the parent `<button>` element

In the component, if a `label` prop is added to `<icon-l>`, the element itself is treated as an image, with `role="img"` and `aria-label="[the label value]"` applied. Encountered by screen reader *outside* of a button or link, the icon will be identified as an image or graphic, and the `aria-label` value read out. Where `<icon-l>` is placed *inside* a button or link, the image role is not announced. The pseudo-image element is simply purposed as the label.

# The generator

Use this tool to generate basic **Icon** CSS and HTML.

The code generator tool is only available in the accompanying site ↗. Here is the basic solution, with comments.

# HTML

We can employ the <use> element ↗ to embed the icon from a remote `icons.svg` file.

```
<span class="with-icon">
  <svg class="icon">
    <use href="/path/to/icons.svg#cross"></use>
  </svg>
  Close
</span>
```

# CSS

The `with-icon` class is only necessary if you wish to eliminate the natural word space and employ `margin` instead.

```css
.icon {
  height: 0.75em;
  /* ↓ Override the em value with `1cap`
  where `cap` is supported */
  height: 1cap;
  width: 0.75em;
  width: 1cap;
}

.with-icon {
  /* ↓ Set the `inline-flex` context,
  which eliminates the word space */
  display: inline-flex;
  align-items: baseline;
}

.with-icon .icon {
  /* ↓ Use the logical margin property
  and a --space variable with a fallback */
  margin-inline-end: var(--space, 0.5em);
}
```

As outlined in our blog post <u>Dynamic CSS Components Without JavaScript ↗</u>, you can adjust the space value declaratively, on the element itself, using the `style` attribute:

```
<span class="with-icon">
  <svg class="icon" style="--space: 0.333em">
    <use href="/images/icons/icons.svg#cross"></use>
  </svg>
  Close
</span>
```

# The component

A custom element implementation of the Icon is available for
download ↗.

# Props API

The following props (attributes) will cause the **Icon** component to
re-render when altered. They can be altered by hand—in browser
developer tools—or as the subjects of inherited application state.

| Name | Type | Default | Description |
|------|------|---------|-------------|
| **space** | string | null | The space between the text and the icon. If null, natural word spacing is preserved |
| **label** | string | null | Turns the element into an image in assistive technologies and adds an aria-label of the value |

# Examples

## Button with icon and accompanying text

In the following example, the `<icon-l>` provides an icon and accompanying text to a button. The assumes the button's accessible name, meaning the button will be announced as *"Close, button"* (or equivalent) in screen reader software. The SVG is ignored, since it provides no textual information.

In this case, an explicit space/margin of `0.5em` has been set.

```
<button>
  <icon-l space="0.5em">
    <svg>
      <use href="/images/icons/icons.svg#cross"></use>
    </svg>
    Close
  </icon-l>
</button>
```

## Button with just an icon

Where not accompanying text is provided, the button is in danger of not having an accessible name. By providing a `label` prop, the `<icon-l>` is communicated as a labeled image to screen reader software (using `role="img"` and `aria-label="[the prop value]"`). This is the authored code:

```
<button>
  <icon-l label="Close">
    <svg>
      <use href="/path/to/icons.svg#cross"></use>
    </svg>
  </icon-l>
</button>
```

And this is the code after instantiation:

```
<button>
  <icon-l space="0.5em" label="Close" role="img" aria-label="Close">
    <svg>
      <use href="/path/to/icons.svg#cross"></use>
    </svg>
  </icon-l>
</button>
```