CSS HANDBOOK

FLAVIO COPES

Preface

This book aims to be an introduction to the CSS fundamentals.

This book was first published in 2019 and has been updated for 2025.

Legal

Introduction

I wrote this book to help you quickly learn CSS and get familiar with the advanced CSS topics.

CSS, a shorthand for Cascading Style Sheets, is one of the main building blocks of the Web. Its history goes back to the 90's and along with HTML it has changed a lot since its humble beginnings.

CSS is an amazing tool, and in the last few years it has grown a lot.

This handbook is aimed at a vast audience.

First, the beginner. I explain CSS from zero in a succinct but comprehensive way, so you can use this book to learn CSS from the basics.

Then, the professional. CSS is often considered like a secondary thing to learn, especially by JavaScript developers. They know CSS is not a real programming language, they are

programmers and therefore they should not bother learning CSS the right way. I wrote this book for you, too.

Next, the person that has used CSS for a few years but hasn't had the opportunity to learn the fundamentals of it.

Even if you don't write CSS for a living, knowing how CSS works can help save you some headaches when you need to understand it from time to time, for example while tweaking a web page.

How does CSS look like

CSS (an abbreviation of **Cascading Style Sheets**) is the language that we use to style an HTML file, and tell the browser how should it render the elements on the page.

A CSS file contains several CSS rules.

Each rule is composed by 2 parts:

- the **selector**
- the **declaration block**

The selector is a string that identifies one or more elements on the page, following a special syntax that we'll soon talk about extensively.

The declaration block contains one or more **declarations**, in turn composed by a **property** and **value** pair.

Those are all the things we have in CSS.

Carefully organizing properties, associating them values, and attaching those to specific elements of the page using a selector is the whole argument of this ebook.

A CSS **rule set** has one part called **selector**, and the other part called **declaration**. The declaration contains various **rules**, each composed by a **property**, and a **value**.

In this example, `p` is the selector, and applies one rule which sets the value `20px` to the `font-size` property:

```
p {
  font-size: 20px;
}
```

A selector can target HTML tags or HTML elements that contain a certain class attribute with `.my-class`, or HTML elements that have a specific `id` attribute with `#my-id`.

More advanced selectors allow you to choose items whose attribute matches a specific value, or also items which respond to pseudo-classes (more on that later).

Every CSS rule terminates with a semicolon. Semicolons are **not** optional, except after the last rule, but I suggest to always use them for consistency and to avoid errors if you add another property and forget to add the semicolon on the previous line.

There is no fixed rule for formatting. This CSS is valid:

```
p {
            font-size: 20px;}
a {color: blue;}
```

but a pain to see. Stick to some conventions, like the ones you see in the examples above: stick selectors and the closing brackets to the left, indent 2 spaces for each rule, have the opening bracket on the same line of the selector, separated by one space.

Correct and consistent use of spacing and indentation is a visual aid in understanding your code.

A brief history of CSS

Before moving on, I want to give you a brief recap of the history of CSS.

CSS was grown out of the necessity of styling web pages. Before CSS was introduced, people wanted a way to style their web pages, which looked all very similar and “academic” back in the day. You couldn’t do much in terms of personalisation.

HTML 3.2 introduced the option of defining colors inline as HTML element attributes, and presentational tags like `center` and `font`, but that escalated quickly into a far from ideal situation.

CSS let us move everything presentation-related from the HTML to the CSS, so that HTML could get back being the format that defines the structure of the document, rather than how things should look in the browser.

CSS is continuously evolving, and CSS you used 5 years ago might just be outdated, as new idiomatic CSS techniques emerged and browsers changed.

It’s hard to imagine the times when CSS was born and how different the web was.

At the time, we had several competing browsers, the main ones being Internet Explorer and Netscape Navigator.

Pages were styled by using HTML, with special presentational tags like `bold` and special attributes, most of which are now deprecated.

This meant you had a limited amount of customization opportunities.

The bulk of the styling decisions were left to the browser.

Also, you built a site specifically for one of them, because each one introduced different non-standard tags to give more power and opportunities.

Soon people realized the need for a way to style pages, in a way that would work across all browsers.

After the initial idea proposed in 1994, CSS got its first release in 1996, when the CSS Level 1 (“CSS 1”) recommendation was published.

CSS Level 2 (“CSS 2”) got published in 1998.

Since then, work began on CSS Level 3. The CSS Working Group decided to split every feature and work on it separately, in modules.

Browsers weren't especially fast at implementing CSS in those early days. We had to wait until 2002 to have the first browser implement the full CSS specification.

Historically, Internet Explorer implemented the box model incorrectly, which led to years of compatibility issues across browsers. Modern browsers now have excellent CSS support and standards compliance.

Today things are much, much better. We can just use the CSS standards without thinking about quirks, most of the time, and CSS has never been more powerful.

We don’t have official release numbers for CSS any more now, but the CSS Working Group releases a “snapshot” of the modules that are currently considered stable and ready to be included in browsers. This is the latest snapshot, from 2023: https://www.w3.org/TR/css-2023/

CSS Level 2 is still the base for the CSS we write today, and we have many more features built on top of it.

Adding CSS to an HTML page

CSS is attached to an HTML page in different ways.

1: Using the `link` tag

The `link` tag is the way to include a CSS file. This is the preferred way to use CSS as it’s intended to be used: one CSS file is included by all the pages of your site, and changing one line on that file affects the presentation of all the pages in the site.

To use this method, you add a `link` tag with the `href` attribute pointing to the CSS file you want to include. You add it inside the `head` tag of the site (not inside the `body` tag):

```
<link rel="stylesheet" type="text/css" href="myfile.css" />
```

The `rel` and `type` attributes are required too, as they tell the browser which kind of file we are linking to.

2: using the `style` tag

Instead of using the `link` tag to point to separate stylesheet containing our CSS, we can add the CSS directly inside a `style` tag. This is the syntax:

```
<style>
...our CSS...;
</style>
```

Using this method we can avoid creating a separate CSS file. I find this is a good way to experiment before “formalising” CSS to a separate file, or to add a special line of CSS just to a file.

3: inline styles

Inline styles are the third way to add CSS to a page. We can add a `style` attribute to any HTML tag, and add CSS into it.

```
<div style="">...</div>
```

Example:

```
<div style="background-color: yellow">...</div>
```

Selectors

A selector allows us to associate one or more declarations to one or more elements on the page.

Basic selectors

Suppose we have a `p` element on the page, and we want to display the words into it using the yellow color.

We can **target** that element using this selector `p`, which targets all the element using the `p` tag in the page. A simple CSS rule to achieve what we want is:

```
p {
  color: yellow;
```

```
}
```

Every HTML tag has a corresponding selector, for example: `div`, `span`, `img`.

If a selector matches multiple elements, all the elements in the page will be affected by the change.

HTML elements have 2 attributes which are very commonly used within CSS to associate styling to a specific element on the page: `class` and `id`.

There is one big difference between those two: inside an HTML document you can repeat the same `class` value across multiple elements, but you can only use an `id` once. As a corollary, using classes you can select an element with 2 or more specific class names, something not possible using ids.

Classes are identified using the `.` symbol, while ids using the `#` symbol.

Example using a class:

```
<p class="dog-name">Roger</p>
```

```
.dog-name {
  color: yellow;
}
```

Example using an id:

```
<p id="dog-name">Roger</p>
```

```
#dog-name {
  color: yellow;
}
```

Combining selectors

So far we've seen how to target an element, a class or an id. Let's introduce more powerful selectors.

Targeting an element with a class or id

You can target a specific element that has a class, or id, attached.

Example using a class:

```
<p class="dog-name">Roger</p>
```

```
p.dog-name {
  color: yellow;
}
```

Example using an id:

```
<p id="dog-name">Roger</p>
```

```
p#dog-name {
  color: yellow;
}
```

Why would you want to do that, if the class or id already provides a way to target that element? You might have to do that to have more specificity. We'll see what that means later.

Targeting multiple classes

You can target an element with a specific class using `.class-name`, as you saw previously. You can target an element with 2 (or more) classes by combining the class names separated with a dot, without spaces.

Example:

```
<p class="dog-name roger">Roger</p>
```

```
.dog-name.roger {
  color: yellow;
}
```

Combining classes and ids

In the same way, you can combine a class and an id.

Example:

```
<p class="dog-name" id="roger">Roger</p>
```

```
.dog-name#roger {
  color: yellow;
}
```

Grouping selectors

You can combine selectors to apply the same declarations to multiple selectors. To do so, you separate them with a comma.

Example:

```
<p>My dog name is:</p> <span class="dog-name">Roger</span>
```

```
p,.dog-name {
  color: yellow;
}
```

If you want you can add spaces, or put those declarations on different lines, to make them more clear:

```
p,
.dog-name {
  color: yellow;
}
```

Follow the document tree with selectors

We've seen how to target an element in the page by using a tag name, a class or an id.

You can create a more specific selector by combining multiple items to follow the document tree structure. For example, if you have a `span` tag nested inside a `p` tag, you can target that one without applying the style to a `span` tag not included in a `p` tag:

```
<span>
  Hello!
</span>
<p>
  My dog name is:
  <span class="dog-name">
    Roger
  </span>
</p>
```

```
p span {
  color: yellow;
}
```

See how we used a space between the two tokens `p` and `span`.

This works even if the element on the right is multiple levels deep.

To make the dependency strict on the first level, you can use the `>` symbol between the two tokens:

```
p > span {
  color: yellow;
}
```

In this case, if a `span` is not a first children of the `p` element, it's not going to have the new color applied.

Direct children will have the style applied:

```
<p>
  <span>
    This is yellow
  </span>
  <strong>
    <span>
      This is not yellow
    </span>
  </strong>
</p>
```

Adjacent sibling selectors let us style an element only if preceded by a specific element. We do so using the `+` operator:

Example:

```
p + span {
  color: yellow;
}
```

This will assign the color yellow to all span elements preceded by a `p` element:

```
<p>This is a paragraph</p>
<span>This is a yellow span</span>
```

We have a lot more selectors we can use:

- attribute selectors
- pseudo class selectors
- pseudo element selectors

We'll find all about them in the next sections.

Cascade

Cascade is a fundamental concept of CSS. After all, it's in the name itself, the first C of CSS - Cascading Style Sheets - it must be an important thing.

What does it mean?

Cascade is the process, or algorithm, that determines the properties applied to each element on the page. Trying to converge from a list of CSS rules that are defined in various places.

It does so taking in consideration:

- specificity
- importance
- inheritance
- order in the file

It also takes care of resolving conflicts.

Two or more competing CSS rules for the same property applied to the same element need to be elaborated according to the CSS spec, to determine which one needs to be applied.

Even if you just have one CSS file loaded by your page, there is other CSS that is going to be part of the process. We have the browser (user agent) CSS. Browsers come with a default set of rules, all different between browsers.

Then your CSS come into play.

Then the browser applies any user stylesheet, which might also be applied by browser extensions.

All those rules come into play while rendering the page.

We'll now see the concepts of specificity and inheritance.

Specificity

What happens when an element is targeted by multiple rules, with different selectors, that affect the same property?

For example, let's talk about this element:

```
<p class="dog-name">Roger</p>
```

We can have

```
.dog-name {
  color: yellow;
}
```

and another rule that targets `p`, which sets the color to another value:

```
p {
  color: red;
}
```

And another rule that targets `p.dog-name`. Which rule is going to take precedence over the others, and why?

Enter specificity. **The more specific rule will win**.
If two or more rules have the **same specificity, the one that appears last wins**.

Sometimes what is more specific in practice is a bit confusing to beginners. I would say it's also confusing to experts that do not look at those rules that frequently, or simply overlook them.

How to calculate specificity

Specificity is calculated using a convention.

We have 4 slots, and each one of them starts at 0: `0 0 0 0 0`. The slot at the left is the most important, and the rightmost one is the least important.

Like it works for numbers in the decimal system: `1 0 0 0` is higher than `0 1 0 0`.

Slot 1

The first slot, the rightmost one, is the least important.

We increase this value when we have an **element selector**. An element is a tag name. If you have more than one element selector in the rule, you increment accordingly the value stored in this slot.

Examples:

```
p {}
/* 0 0 0 1 */

span {}
/* 0 0 0 1 */

p span {}
```

```
/* 0 0 0 2 */

p > span {}
/* 0 0 0 2 */

div p > span {}
/* 0 0 0 3 */
```

Slot 2

The second slot is incremented by 3 things:

- class selectors
- pseudo-class selectors
- attribute selectors

Every time a rule meets one of those, we increment the value of the second column from the right.

Examples:

```
.name {}
/* 0 0 1 0 */

.users .name {}
/* 0 0 2 0 */

[href$='.pdf'] {}
/* 0 0 1 0 */

:hover {}
/* 0 0 1 0 */
```

Of course slot 2 selectors can be combined with slot 1 selectors:

```
div .name {}
/* 0 0 1 1 */

a[href$='.pdf'] {}
/* 0 0 1 1 */

.pictures img:hover {}
/* 0 0 2 1 */
```

One nice trick with classes is that you can repeat the same class and increase the specificity. For example:

```
.name {}
/* 0 0 1 0 */

.name.name {}
/* 0 0 2 0 */

.name.name.name {}
/* 0 0 3 0 */
```

Slot 3

Slot 3 holds the most important thing that can affect your CSS specificity in a CSS file: the `id`.

Every element can have an `id` attribute assigned, and we can use that in our stylesheet to target the element.

Examples:

```
#name {}
/* 0 1 0 0 */

.user #name {}
/* 0 1 1 0 */

#name span {}
/* 0 1 0 1 */
```

Slot 4

Slot 4 is affected by inline styles. Any inline style will have precedence over any rule defined in an external CSS file, or inside the `style` tag in the page header.

Example:

```
<p style="color: red">Test</p>
/* 1 0 0 0 */
```

Even if any other rule in the CSS defines the color, this inline style rule is going to be applied. Except for one case - if `!important` is used, which fills the slot 5.

Importance

Specificity does not matter if a rule ends with `!important`:

```
p {
  font-size: 20px !important;
}
```

That rule will take precedence over any rule with more specificity

Adding `!important` in a CSS rule is going to make that rule be more important than any other rule, according to the specificity rules. The only way another rule can take precedence is to have `!important` as well, and have higher specificity in the other less important slots.

Considerations on specificity

In general you should use the amount of specificity you need, but not more. In this way, you can craft other selectors to overwrite the rules set by preceding rules without going mad.

`!important` is a highly debated tool that CSS offers us. Many CSS experts advocate against using it. I find myself using it especially when trying out some style and a CSS rule has so much specificity that I need to use `!important` to make the browser apply my new CSS.

But generally, `!important` should have no place in your CSS files.

Using the `id` attribute to style CSS is also debated a lot, since it has a very high specificity. A good alternative is to use classes instead, which have less specificity, and so they are easier to work with, and they are more powerful (you can have multiple classes for an element, and a class can be reused multiple times).

You can use the site https://specificity.keegan.st/ to perform the specificity calculation for you automatically.

It's useful especially if you are trying to figure things out, as it can be a nice feedback tool.

Inheritance

When you set some properties on a selector in CSS, they are inherited by all the children of that selector.

I said *some*, because not all properties show this behaviour.

This happens because some properties make sense to be inherited. This helps us write CSS much more concisely, since we don't have to explicitly set that property again on every single children.

Some other properties make more sense to *not* be inherited.

Think about fonts: you don't need to apply the `font-family` to every single tag of your page. You set the `body` tag font, and every children inherits it, along with other properties.

The `background-color` property, on the other hand, makes little sense to be inherited.

Properties that inherit

Here is a list of the properties that do inherit. The list is non-comprehensive, but those rules are just the most popular ones you'll likely use:

- `border-collapse`
- `border-spacing`
- `caption-side`
- `color`
- `cursor`
- `direction`
- `empty-cells`
- `font-family`
- `font-size`
- `font-style`
- `font-variant`
- `font-weight`
- `font-size-adjust`
- `font-stretch`
- `font`
- `letter-spacing`
- `line-height`
- `list-style-image`
- `list-style-position`
- `list-style-type`
- `list-style`
- `orphans`
- `quotes`
- `tab-size`
- `text-align`
- `text-align-last`
- `text-decoration-color`
- `text-indent`
- `text-justify`
- `text-shadow`
- `text-transform`
- `visibility`
- `white-space`
- `widows`

- `word-break`
- `word-spacing`

Forcing properties to inherit

What if you have a property that's not inherited by default, and you want it to, in a children?

In the children, you set the property value to the special keyword `inherit`.

Example:

```
body {
  background-color: yellow;
}

p {
  background-color: inherit;
}
```

Forcing properties to NOT inherit

On the contrary, you might have a property inherited and you want to avoid so.

You can use the `revert` keyword to revert it. In this case, the value is reverted to the original value the browser gave it in its default stylesheet.

In practice this is rarely used, and most of the times you'll just set another value for the property to overwrite that inherited value.

Other special values

In addition to the `inherit` and `revert` special keywords we just saw, you can also set any property to:

- `initial`: use the default browser stylesheet if available. If not, and if the property inherits by default, inherit the value. Otherwise do nothing.
- `unset`: if the property inherits by default, inherit. Otherwise do nothing.

Import

From any CSS file you can import another CSS file using the `@import` directive.

Here is how you use it:

```
@import url(myfile.css);
```

url() can manage absolute or relative URLs.

One important thing you need to know is that `@import` directives must be put before any other CSS in the file, or they will be ignored.

You can use media descriptors to only load a CSS file on the specific media:

```
@import url(myfile.css) all;
@import url(myfile-screen.css) screen;
@import url(myfile-print.css) print;
```

CSS Nesting

CSS Nesting is now natively supported in all modern browsers, allowing you to write more maintainable and organized styles without a preprocessor like Sass or Less.

Nesting lets you write child selectors inside their parent's rule block, making the relationship between styles clearer and reducing repetition. This feature significantly improves code organization and readability.

Basic Nesting Syntax

You can nest selectors directly inside their parent rules:

```
/* Traditional CSS */
.card {
  padding: 1rem;
  background: white;
}
.card h2 {
  color: navy;
  font-size: 1.5rem;
}
.card p {
  color: gray;
  line-height: 1.6;
}

/* With CSS Nesting */
.card {
  padding: 1rem;
  background: white;

  h2 {
    color: navy;
    font-size: 1.5rem;
  }
```

```
  p {
    color: gray;
    line-height: 1.6;
  }
}
```

The nested version groups related styles together, making it immediately clear that `h2` and `p` styles apply only within `.card`.

Using the & Selector

The `&` symbol represents the parent selector and is required for certain nesting patterns:

```
.button {
  background: blue;
  color: white;

  /* Pseudo-classes need & */
  &:hover {
    background: darkblue;
  }

  &:active {
    transform: scale(0.98);
  }

  /* Modifier classes */
  &.large {
    padding: 1rem 2rem;
    font-size: 1.2rem;
  }

  &.disabled {
    opacity: 0.5;
    cursor: not-allowed;
  }
}
```

This compiles to `.button:hover`, `.button:active`, `.button.large`, and `.button.disabled`.

Complex Nesting Examples

Nesting can go multiple levels deep and combine various selectors:

```
nav {
  background: #333;
  padding: 1rem;

  ul {
    list-style: none;
    display: flex;
    gap: 2rem;

    li {
      position: relative;

      a {
        color: white;
        text-decoration: none;

        &:hover {
          color: #66b3ff;
        }
      }

      /* Dropdown menu */
      &:has(.dropdown) {
        > a::after {
          content: ' ▼';
          font-size: 0.8em;
        }
      }

      .dropdown {
        display: none;
        position: absolute;
        top: 100%;
        background: #444;

        /* Show on hover */
        @nest li:hover & {
          display: block;
        }
      }
    }
  }
}
```

Media Queries and Container Queries in Nesting

You can nest media queries and container queries directly inside rules:

```
.card {
  padding: 1rem;
  background: white;

  @media (min-width: 768px) {
    padding: 2rem;
    display: grid;
    grid-template-columns: 1fr 2fr;
  }

  @container (min-width: 400px) {
    h2 {
      font-size: 2rem;
    }
  }
}
```

CSS Cascade Layers

CSS Cascade Layers give you more control over the cascade, allowing you to organize your CSS into layers with explicit ordering. Think of layers like transparent sheets stacked on top of each other - you can control which sheet is on top and therefore which styles win when there are conflicts.

This feature solves a common problem in CSS: managing the specificity and order of styles from different sources (reset styles, third-party libraries, your own utilities, and component styles). Instead of fighting with specificity or using `!important`, you can organize your CSS into logical layers.

Creating and Using Layers

You can create layers using the `@layer` at-rule:

```
/* Define layer order first */
@layer reset, base, components, utilities;

/* Add styles to layers */
@layer reset {
  * {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
  }
}

@layer base {
  body {
```

```
    font-family: system-ui, sans-serif;
    line-height: 1.5;
    color: #333;
  }
}

@layer components {
  .button {
    background: blue;
    color: white;
    padding: 0.5rem 1rem;
    border-radius: 4px;
  }
}

@layer utilities {
  .text-center {
    text-align: center;
  }

  .mt-4 {
    margin-top: 1rem;
  }
}
```

The power of layers is that styles in later layers automatically override styles in earlier layers, regardless of specificity. In the example above, a utility class will always override a component style, even if the component has higher specificity. This makes your CSS more predictable and easier to maintain.

Anonymous and Nested Layers

You can create anonymous layers (without names) and nest layers inside other layers:

```
/* Anonymous layer */
@layer {
  .special-element {
    background: yellow;
  }
}

/* Nested layers */
@layer framework {
  @layer defaults {
    input {
      border: 1px solid #ccc;
    }
  }
```

```
  @layer theme {
    input {
      border-color: blue;
    }
  }
}
```

Anonymous layers are useful for one-off overrides, while nested layers help organize related styles into sub-categories. Nested layers are referenced using dot notation, like `framework.defaults` and `framework.theme`.

Importing into Layers

You can import external stylesheets directly into layers:

```
/* Import external styles into specific layers */
@import url('reset.css') layer(reset);
@import url('theme.css') layer(theme);
@import url('utilities.css') layer(utilities);

/* Or define the order and import */
@layer reset, theme, components, utilities;
@import url('components.css') layer(components);
```

This is incredibly useful when working with third-party CSS. You can put their styles in a lower layer, ensuring your custom styles always win without specificity battles.

CSS Logical Properties

CSS Logical Properties replace physical directions (left, right, top, bottom) with logical ones that adapt to different writing modes and text directions. This is essential for international websites that need to support languages like Arabic or Hebrew (right-to-left) or Japanese (vertical writing).

Instead of thinking in terms of "left" and "right", logical properties use "start" and "end" based on the text flow direction. Similarly, instead of "top" and "bottom", they use "block-start" and "block-end".

Common Logical Properties

Here are the most commonly used logical properties and their physical equivalents:

```
/* Traditional physical properties */
.old-way {
  margin-left: 20px;
  margin-right: 20px;
```

```
  padding-top: 10px;
  padding-bottom: 10px;
  border-left: 2px solid blue;
  width: 200px;
  height: 100px;
}

/* Modern logical properties */
.new-way {
  margin-inline-start: 20px;  /* replaces margin-left in LTR */
  margin-inline-end: 20px;    /* replaces margin-right in LTR */
  padding-block-start: 10px;  /* replaces padding-top */
  padding-block-end: 10px;    /* replaces padding-bottom */
  border-inline-start: 2px solid blue; /* replaces border-left in LTR */
  inline-size: 200px;         /* replaces width */
  block-size: 100px;          /* replaces height */
}
```

When the page direction changes (for example, when switching to Arabic), the logical properties automatically adapt. The `margin-inline-start` that was on the left in English will be on the right in Arabic, without changing your CSS.

Shorthand Logical Properties

Just like physical properties, logical properties have convenient shorthands:

```
.element {
  /* Inline axis (horizontal in LTR/RTL) */
  margin-inline: 20px;        /* start and end */
  margin-inline: 20px 40px;   /* start, then end */
  padding-inline: 10px 15px;

  /* Block axis (vertical in LTR/RTL) */
  margin-block: 30px;         /* start and end */
  margin-block: 30px 60px;    /* start, then end */
  padding-block: 20px 25px;

  /* Size properties */
  inline-size: 300px;         /* width in horizontal writing */
  block-size: 200px;          /* height in horizontal writing */
  min-inline-size: 100px;     /* min-width */
  max-block-size: 500px;      /* max-height */
}
```

These shorthands make your code cleaner while maintaining international compatibility. They work exactly like their physical counterparts but adapt to different writing modes automatically.

Logical Values for Positioning

Logical properties also extend to positioned elements:

```
.popup {
  position: absolute;

  /* Old way */
  top: 0;
  right: 0;

  /* Logical way */
  inset-block-start: 0;    /* top in horizontal writing */
  inset-inline-end: 0;     /* right in LTR, left in RTL */

  /* Or use the shorthand */
  inset: 0 0 auto auto;    /* block-start inline-end block-end inline-start
*/
}
```

The `inset` property is particularly powerful as it combines all four positioning values in one declaration, following the logical flow of block-start, inline-end, block-end, and inline-start.

Modern Color Spaces and Functions

CSS has evolved far beyond simple hex colors and RGB values. Modern CSS offers sophisticated color spaces and functions that give you more control over colors, better accessibility, and more intuitive color manipulation.

The new color capabilities include perceptually uniform color spaces (where equal numeric changes create equal visual changes), wide-gamut colors for modern displays, and functions that make color manipulation easier and more predictable.

The oklch() Color Space

The oklch() function represents colors using Lightness, Chroma (saturation), and Hue. This color space is perceptually uniform, meaning a 10% change in lightness looks like a 10% change regardless of the starting color:

```
.content {
  /* oklch(lightness chroma hue) */
  background: oklch(90% 0.1 250);     /* Light blue */
  color: oklch(30% 0.2 250);          /* Dark blue */

  /* Easy to create color variations */
  --primary: oklch(60% 0.3 250);
  --primary-light: oklch(80% 0.2 250);  /* Increase lightness, decrease
```

```
saturation */
  --primary-dark: oklch(40% 0.3 250);   /* Decrease lightness only */

  /* Create a whole palette from one hue */
  --blue: oklch(60% 0.3 250);
  --green: oklch(60% 0.3 140);
  --red: oklch(60% 0.3 30);
}
```

The beauty of oklch() is that you can create harmonious color palettes by keeping the lightness and chroma consistent while only changing the hue. This creates colors that feel like they belong together.

The color-mix() Function

The color-mix() function lets you blend two colors together, which is perfect for creating tints, shades, and color variations:

```
.variations {
  --brand-color: #0066cc;

  /* Mix with white for tints */
  background: color-mix(in srgb, var(--brand-color) 80%, white);

  /* Mix with black for shades */
  border-color: color-mix(in srgb, var(--brand-color) 70%, black);

  /* Mix two colors together */
  accent-color: color-mix(in srgb, blue 60%, purple 40%);

  /* Use different color spaces for different results */
  --smooth-mix: color-mix(in oklch, red, blue);
  --vibrant-mix: color-mix(in srgb, red, blue);
}
```

The color-mix() function is incredibly useful for creating hover states, disabled states, or generating entire color schemes from a base color. The choice of color space affects how the colors are interpolated, with oklch often producing more pleasing results.

Wide-Gamut Colors with display-p3

Modern displays can show colors beyond the traditional sRGB range. The display-p3 color space lets you use these vibrant colors:

```
.vibrant {
  /* Fallback for older browsers */
  background: rgb(255, 0, 0);
```

```
  /* Wide-gamut color for modern displays */
  background: color(display-p3 1 0 0);

  /* Use @supports to provide enhanced colors */
  @supports (color: color(display-p3 1 0 0)) {
    background: color(display-p3 1 0 0);  /* More vibrant red */
    color: color(display-p3 0 1 0);       /* More vibrant green */
  }
}
```

Wide-gamut colors are especially noticeable with vibrant reds, greens, and blues. They make images and designs pop on modern displays while gracefully falling back on older screens.

CSS Custom Properties Throughout

CSS Custom Properties (also called CSS Variables) should be used throughout modern CSS for maintainability and dynamic styling. Unlike preprocessor variables, CSS custom properties can be changed at runtime and participate in the cascade.

Custom properties transform how we write CSS by enabling theming, responsive design patterns, and dynamic calculations that were previously impossible without JavaScript.

Basic Usage and Patterns

Custom properties are defined with a double dash prefix and accessed using the var() function:

```
:root {
  /* Define global design tokens */
  --primary-color: oklch(60% 0.3 250);
  --secondary-color: oklch(60% 0.3 140);
  --text-color: oklch(20% 0.02 250);
  --background: oklch(98% 0.01 250);

  /* Spacing scale */
  --space-xs: 0.25rem;
  --space-sm: 0.5rem;
  --space-md: 1rem;
  --space-lg: 2rem;
  --space-xl: 4rem;

  /* Typography scale */
  --font-size-sm: clamp(0.875rem, 0.8rem + 0.25vw, 0.95rem);
  --font-size-base: clamp(1rem, 0.9rem + 0.5vw, 1.125rem);
  --font-size-lg: clamp(1.25rem, 1.1rem + 0.75vw, 1.5rem);
  --font-size-xl: clamp(1.5rem, 1.3rem + 1vw, 2rem);
```

```
}

.component {
  /* Use the properties */
  color: var(--text-color);
  padding: var(--space-md);
  font-size: var(--font-size-base);


  /* Provide fallbacks */
  background: blue;
  background: var(--primary-color, blue);
}
```

The var() function accepts a second parameter as a fallback value, which is used if the custom property isn't defined. This ensures your styles don't break if a variable is missing.

Dynamic Theming with Custom Properties

Custom properties enable instant theme switching without reloading:

```
/* Light theme (default) */
:root {
  --bg-primary: oklch(98% 0.01 250);
  --bg-secondary: oklch(95% 0.02 250);
  --text-primary: oklch(20% 0.02 250);
  --text-secondary: oklch(40% 0.02 250);
  --accent: oklch(60% 0.3 250);
}

/* Dark theme */
[data-theme="dark"] {
  --bg-primary: oklch(15% 0.02 250);
  --bg-secondary: oklch(20% 0.02 250);
  --text-primary: oklch(95% 0.01 250);
  --text-secondary: oklch(75% 0.01 250);
  --accent: oklch(70% 0.3 250);
}

/* Components use the variables */
.card {
  background: var(--bg-secondary);
  color: var(--text-primary);
  border: 1px solid var(--accent);
}

/* JavaScript switches themes */
/* document.documentElement.setAttribute('data-theme', 'dark'); */
```

This pattern allows users to switch themes instantly, with all components automatically updating their colors. The theme preference can be saved to localStorage and restored on page load.

Component-Scoped Properties

Custom properties can be scoped to components for modular, reusable styles:

```
.button {
  /* Component variables with defaults */
  --button-bg: var(--primary-color, blue);
  --button-color: white;
  --button-padding: 0.5rem 1rem;
  --button-radius: 4px;
  --button-hover-bg: color-mix(in oklch, var(--button-bg) 80%, black);

  /* Apply the variables */
  background: var(--button-bg);
  color: var(--button-color);
  padding: var(--button-padding);
  border-radius: var(--button-radius);
  transition: background-color 0.3s;

  &:hover {
    background: var(--button-hover-bg);
  }
}

/* Variations just override the variables */
.button--large {
  --button-padding: 0.75rem 1.5rem;
  font-size: 1.125rem;
}

.button--danger {
  --button-bg: oklch(55% 0.3 30);
  --button-hover-bg: oklch(45% 0.3 30);
}

.button--ghost {
  --button-bg: transparent;
  --button-color: var(--primary-color);
  border: 2px solid var(--primary-color);
}
```

Best Practices for CSS Nesting

1. **Don't nest too deeply** - More than 3 levels becomes hard to read

2. **Use & for clarity** - Even when optional, & makes parent references explicit
3. **Group related styles** - Keep modifiers and states near their base styles
4. **Consider specificity** - Nesting increases specificity, which can make overrides harder

```
/* Good - Clear and not too deep */
.article {
  padding: 2rem;

  h1 {
    font-size: 2rem;
    margin-bottom: 1rem;
  }

  .meta {
    color: #666;
    font-size: 0.9rem;

    time {
      font-weight: bold;
    }
  }
}

/* Avoid - Too deeply nested */
.article {
  .content {
    .section {
      .paragraph {
        .text {
          /* This is too deep! */
        }
      }
    }
  }
}
```

CSS Nesting makes stylesheets more maintainable by keeping related rules together and reducing selector repetition. It's particularly useful for component-based styling where all styles for a component can be grouped in one place.

Attribute selectors

We already introduced several of the basic CSS selectors: using element selectors, class, id, how to combine them, how to target multiple classes, how to style several selectors in the same rule, how to follow the page hierarchy with child and direct child selectors, and adjacent siblings.

In this section we'll analyze attribute selectors, and we'll talk about pseudo class and pseudo element selectors in the next 2 sections.

Attribute presence selectors

The first selector type is the attribute presence selector.

We can check if an element **has** an attribute using the `[]` syntax. `p[id]` will select all `p` tags in the page that have an `id` attribute, regardless of its value:

```
p[id] {
  /* ... */
}
```

Exact attribute value selectors

Inside the brackets you can check the attribute value using `=`, and the CSS will be applied only if the attribute matches the exact value specified:

```
p[id='my-id'] {
  /* ... */
}
```

Match an attribute value portion

While `=` let us check for exact value, we have other operators:

- `=` checks if the attribute contains the partial
- `^=` checks if the attribute starts with the partial
- `$=` checks if the attribute ends with the partial
- `|=` checks if the attribute starts with the partial and it's followed by a dash (common in classes, for example), or just contains the partial
- `~=` checks if the partial is contained in the attribute, but separated by spaces from the rest

All the checks we mentioned are **case sensitive**.

If you add an `i` just before the closing bracket, the check will be case insensitive. This is now widely supported across all modern browsers.

Pseudo-classes

Pseudo classes are predefined keywords that are used to select an element based on its **state**, or to target a specific child.

They start with a **single colon** `:`.

They can be used as part of a selector, and they are very useful to style active or visited links for example, change the style on hover, focus, or target the first child, or odd rows. Very handy in many cases.

These are the most popular pseudo classes you will likely use:

Pseudo class	Targets
`:active`	an element being activated by the user (e.g. clicked). Mostly used on links or buttons
`:checked`	a checkbox, option or radio input types that are enabled
`:default`	the default in a set of choices (like, option in a select or radio buttons)
`:disabled`	an element disabled
`:empty`	an element with no children
`:enabled`	an element that's enabled (opposite to `:disabled`)
`:first-child`	the first child of a group of siblings
`:focus`	the element with focus
`:hover`	an element hovered with the mouse
`:last-child`	the last child of a group of siblings
`:link`	a link that's not been visited
`:not()`	any element not matching the selector passed. E.g. `:not(span)`
`:nth-child()`	an element matching the specified position
`:nth-last-child()`	an element matching the specific position, starting from the end
`:only-child`	an element without any siblings
`:required`	a form element with the `required` attribute set
`:root`	represents the `html` element. It's like targeting `html`, but it's more specific. Useful in CSS Variables.
`:target`	the element matching the current URL fragment (for inner navigation in the page)
`:valid`	form elements that validated client-side successfully
`:visited`	a link that's been visited

Let's do an example. A common one, actually. You want to style a link, so you create a CSS rule to target the `a` element:

```
a {
  color: yellow;
```

```
}
```

Things seem to work fine, until you click one link. The link goes back to the predefined color (blue) when you click it. Then when you open the link and go back to the page, now the link is blue.

Why does that happen?

Because the link when clicked changes state, and goes in the `:active` state. And when it's been visited, it is in the `:visited` state. Forever, until the user clears the browsing history.

So, to correctly make the link yellow across all states, you need to write

```
a,
a:visited,
a:active {
  color: yellow;
}
```

`:nth-child()` deserves a special mention. It can be used to target odd or even children with `:nth-child(odd)` and `:nth-child(even)`.

It is commonly used in lists to color odd lines differently from even lines:

```
ul:nth-child(odd) {
  color: white;
  background-color: black;
}
```

You can also use it to target the first 3 children of an element with `:nth-child(-n+3)`. Or you can style 1 in every 5 elements with `:nth-child(5n)`.

Some pseudo classes are just used for printing, like `:first`, `:left`, `:right`, so you can target the first page, all the left pages, and all the right pages, which are usually styled slightly differently.

Modern Selectors

CSS has evolved with powerful new selectors that make complex selections much easier. These modern selectors are now widely supported across all major browsers.

The :has() Selector

The `:has()` selector, often called the "parent selector," allows you to select elements based on their descendants. This was a long-awaited feature in CSS.

Think of `:has()` as asking "does this element contain something specific?" For example, you might want to style an article differently if it contains images, or highlight a form that has validation errors. Before `:has()`, this was impossible with CSS alone - you needed JavaScript to check if an element contained certain children and then add classes. Now CSS can do this natively.

```
/* Select articles that contain images */
article:has(img) {
  border: 2px solid blue;
}
```

This rule finds all `<article>` elements that have at least one `<img>` element inside them, and gives them a blue border. This is incredibly useful for responsive layouts where you might want image-containing articles to have different spacing or backgrounds.

```
/* Select forms with invalid inputs */
form:has(input:invalid) {
  background-color: #fee;
}
```

Here we're selecting any form that contains an invalid input field. When a user enters invalid data (like an email without an @ symbol in an email field), the entire form gets a light red background, providing immediate visual feedback that something needs attention.

```
/* Select list items that have checked checkboxes */
li:has(input[type="checkbox"]:checked) {
  text-decoration: line-through;
  opacity: 0.6;
}
```

This creates a todo-list effect: when a checkbox inside a list item is checked, the entire list item gets a strikethrough and becomes semi-transparent. The magic is that we're styling the parent `<li>` based on the state of its child checkbox.

```
/* Select containers that have a specific child */
.card:has(.premium-badge) {
  border-color: gold;
}
```

This styles any card that contains a premium badge with a gold border. It's perfect for e-commerce sites where premium products need special visual treatment.

You can also use `:has()` to select based on following siblings:

```
/* Select h1 that is immediately followed by a paragraph */
h1:has(+ p) {
  margin-bottom: 0.5rem;
}
```

This reduces the margin under any `<h1>` heading that's immediately followed by a paragraph, creating tighter, more visually connected text blocks.

The :is() Selector

The `:is()` selector allows you to group multiple selectors, reducing repetition and making your CSS more maintainable. It takes the specificity of its most specific argument.

Imagine you're writing CSS and find yourself repeating the same selectors over and over. The `:is()` selector solves this by letting you group them together. It's like saying "any of these" in a compact way. This not only makes your CSS shorter but also easier to read and maintain.

```
/* Instead of writing this: */
article h1,
article h2,
article h3,
section h1,
section h2,
section h3 {
  color: blue;
}

/* You can write this: */
:is(article, section) :is(h1, h2, h3) {
  color: blue;
}
```

This example shows the power of `:is()`. Instead of writing six separate selectors for all combinations of article/section with h1/h2/h3, we can express it in one line. The rule applies to any h1, h2, or h3 that's inside either an article or section element.

```
/* Styling multiple states */
button:is(:hover, :focus, :active) {
  background-color: darkblue;
}
```

Here we're saying "when a button is in any of these states (hovered, focused, or active), give it a dark blue background." This is much cleaner than writing three separate rules or a long comma-separated selector list.

```
/* Complex selections simplified */
:is(header, main, footer) p:is(:first-child, :last-child) {
  margin: 0;
}
```

This advanced example removes margins from paragraphs that are either the first or last child, but only when they're inside a header, main, or footer element. Without `:is()`, you'd need to write six different selector combinations!

The :where() Selector

The `:where()` selector works exactly like `:is()`, but with zero specificity. This makes it perfect for creating default styles that are easy to override.

```
/* Base styles with zero specificity */
:where(h1, h2, h3, h4, h5, h6) {
  color: #333;
  line-height: 1.2;
}

/* These can be easily overridden */
.special h1 {
  color: blue; /* This will win due to higher specificity */
}

/* Resetting list styles */
:where(ul, ol) {
  list-style: none;
  padding: 0;
  margin: 0;
}
```

The key difference between `:is()` and `:where()`:

- `:is()` takes the specificity of its most specific selector
- `:where()` always has zero specificity

The :not() Selector (Enhanced)

While `:not()` has been around for a while, modern CSS allows complex selectors inside `:not()`:

```
/* Select all inputs except checkboxes and radios */
input:not([type="checkbox"], [type="radio"]) {
  width: 100%;
}
```

```
/* Select all list items except the last one */
li:not(:last-child) {
  border-bottom: 1px solid #ccc;
}

/* Complex combinations */
.card:not(:hover, :focus-within) {
  opacity: 0.8;
}
```

These modern selectors significantly reduce the amount of CSS you need to write and make your stylesheets more maintainable.

Pseudo-elements

Pseudo-elements are used to style a specific part of an element.

They start with a double colon `::`.

> Sometimes you will spot them in the wild with a single colon, but this is only a syntax supported for backwards compatibility reasons. You should use 2 colons to distinguish them from pseudo-classes.

`::before` and `::after` are probably the most used pseudo-elements. They are used to add content before or after an element, like icons for example.

Here's the list of the pseudo-elements:

Pseudo-element	Targets
`::after`	creates a pseudo-element after the element
`::before`	creates a pseudo-element before the element
`::first-letter`	can be used to style the first letter of a block of text
`::first-line`	can be used to style the first line of a block of text
`::selection`	targets the text selected by the user

Let's do an example. Say you want to make the first line of a paragraph slightly bigger in font size, a common thing in typography:

```
p::first-line {
  font-size: 2rem;
}
```

Or maybe you want the first letter to be bolder:

```
p::first-letter {
  font-weight: bolder;
}
```

`::after` and `::before` are a bit less intuitive. I remember using them when I had to add icons using CSS.

You specify the `content` property to insert any kind of content after or before an element:

```
p::before {
  content: url(/myimage.png);
}

.myElement::before {
  content: 'Hey Hey!';
}
```

Colors

By default an HTML page is rendered by web browsers quite sadly in terms of the colors used.

We have a white background, black color, and blue links. That's it.

Luckily CSS gives us the ability to add colors to our designs.

We have these properties:

- `color`
- `background-color`
- `border-color`

All of them accept a **color value**, which can be in different forms.

Named colors

First, we have CSS keywords that define colors. CSS started with 16, but today there is a huge number of colors names:

- `aliceblue`
- `antiquewhite`
- `aqua`
- `aquamarine`
- `azure`
- `beige`

- `bisque`
- `black`
- `blanchedalmond`
- `blue`
- `blueviolet`
- `brown`
- `burlywood`
- `cadetblue`
- `chartreuse`
- `chocolate`
- `coral`
- `cornflowerblue`
- `cornsilk`
- `crimson`
- `cyan`
- `darkblue`
- `darkcyan`
- `darkgoldenrod`
- `darkgray`
- `darkgreen`
- `darkgrey`
- `darkkhaki`
- `darkmagenta`
- `darkolivegreen`
- `darkorange`
- `darkorchid`
- `darkred`
- `darksalmon`
- `darkseagreen`
- `darkslateblue`
- `darkslategray`
- `darkslategrey`
- `darkturquoise`
- `darkviolet`
- `deeppink`
- `deepskyblue`
- `dimgray`
- `dimgrey`
- `dodgerblue`

- `firebrick`
- `floralwhite`
- `forestgreen`
- `fuchsia`
- `gainsboro`
- `ghostwhite`
- `gold`
- `goldenrod`
- `gray`
- `green`
- `greenyellow`
- `grey`
- `honeydew`
- `hotpink`
- `indianred`
- `indigo`
- `ivory`
- `khaki`
- `lavender`
- `lavenderblush`
- `lawngreen`
- `lemonchiffon`
- `lightblue`
- `lightcoral`
- `lightcyan`
- `lightgoldenrodyellow`
- `lightgray`
- `lightgreen`
- `lightgrey`
- `lightpink`
- `lightsalmon`
- `lightseagreen`
- `lightskyblue`
- `lightslategray`
- `lightslategrey`
- `lightsteelblue`
- `lightyellow`
- `lime`
- `limegreen`

- `linen`
- `magenta`
- `maroon`
- `mediumaquamarine`
- `mediumblue`
- `mediumorchid`
- `mediumpurple`
- `mediumseagreen`
- `mediumslateblue`
- `mediumspringgreen`
- `mediumturquoise`
- `mediumvioletred`
- `midnightblue`
- `mintcream`
- `mistyrose`
- `moccasin`
- `navajowhite`
- `navy`
- `oldlace`
- `olive`
- `olivedrab`
- `orange`
- `orangered`
- `orchid`
- `palegoldenrod`
- `palegreen`
- `paleturquoise`
- `palevioletred`
- `papayawhip`
- `peachpuff`
- `peru`
- `pink`
- `plum`
- `powderblue`
- `purple`
- `rebeccapurple`
- `red`
- `rosybrown`
- `royalblue`

- `saddlebrown`
- `salmon`
- `sandybrown`
- `seagreen`
- `seashell`
- `sienna`
- `silver`
- `skyblue`
- `slateblue`
- `slategray`
- `slategrey`
- `snow`
- `springgreen`
- `steelblue`
- `tan`
- `teal`
- `thistle`
- `tomato`
- `turquoise`
- `violet`
- `wheat`
- `white`
- `whitesmoke`
- `yellow`
- `yellowgreen`

plus `tranparent`, and `currentColor` which points to the `color` property, for example useful to make the `border-color` inherit it.

They are defined in the CSS Color Module, Level 4. They are case insensitive.

Wikipedia has a nice table which lets you pick the perfect color by its name.

Named colors are not the only option.

RGB and RGBa

You can use the `rgb()` function to calculate a color from its RGB notation, which sets the color based on its red-green-blue parts. From 0 to 255:

```
p {
  color: rgb(255, 255, 255); /* white */
```

```
  background-color: rgb(0, 0, 0); /* black */
}
```

`rgba()` lets you add the alpha channel to enter a transparent part. That can be a number from 0 to 1:

```
p {
  background-color: rgba(0, 0, 0, 0.5);
}
```

Hexadecimal notation

Another option is to express the RGB parts of the colors in the hexadecimal notation, which is composed by 3 blocks.

Black, which is `rgb(0,0,0)` is expressed as `#000000` or `#000` (we can shortcut the 2 numbers to 1 if they are equal).

White, `rgb(255,255,255)` can be expressed as `#ffffff` or `#fff`.

The hexadecimal notation lets express a number from 0 to 255 in just 2 digits, since they can go from 0 to “15” (f).

We can add the alpha channel by adding 1 or 2 more digits at the end, for example `#00000033`. Not all browsers support the shortened notation, so use all 6 digits to express the RGB part.

HSL and HSLa

This is a more recent addition to CSS.

HSL = Hue Saturation Lightness.

In this notation, black is `hsl(0, 0%, 0%)` and white is `hsl(0, 0%, 100%)`.

If you are more familiar with HSL than RGB because of your past knowledge, you can definitely use that.

You also have `hsla()` which adds the alpha channel to the mix, again a number from 0 to 1: `hsl(0, 0%, 0%, 0.5)`

Units

One of the things you’ll use every day in CSS are units. They are used to set lengths, paddings, margins, align elements and so on.

Things like `px`, `em`, `rem`, or percentages.

They are everywhere. There are some obscure ones, too. We'll go through each of them in this section.

Pixels

The most widely used measurement unit. A pixel does not actually correlate to a physical pixel on your screen, as that varies, a lot, by device (think high-DPI devices vs non-retina devices).

There is a convention that make this unit work consistently across devices.

Percentages

Another very useful measure, percentages let you specify values in percentages of that parent element's corresponding property.

Example:

```
.parent {
  width: 400px;
}

.child {
  width: 50%; /* = 200px */
}
```

Real-world measurement units

We have those measurement units which are translated from the outside world. Mostly useless on screen, they can be useful for print stylesheets. They are:

- `cm` a centimeter (maps to 37.8 pixels)
- `mm` a millimeter (0.1cm)
- `q` a quarter of a millimeter
- `in` an inch (maps to 96 pixels)
- `pt` a point (1 inch = 72 points)
- `pc` a pica (1 pica = 12 points)

Relative units

- `em` is the value assigned to that element's `font-size`, therefore its exact value changes between elements. It does not change depending on the font used, just on the font size. In typography this measures the width of the `m` letter.
- `rem` is similar to `em`, but instead of varying on the current element font size, it uses the root element (`html`) font size. You set that font size once, and `rem` will be a consistent

measure across all the page.

- `ex` is like `em`, but inserted of measuring the width of `m`, it measures the height of the `x` letter. It can change depending on the font used, and on the font size.
- `ch` is like `ex` but instead of measuring the height of `x` it measures the width of `0` (zero).

Viewport units

- `vw` the **viewport width unit** represents a percentage of the viewport width. `50vw` means 50% of the viewport width.
- `vh` the **viewport height unit** represents a percentage of the viewport height. `50vh` means 50% of the viewport height.
- `vmin` the **viewport minimum unit** represents the minimum between the height or width in terms of percentage. `30vmin` is the 30% of the current width or height, depending which one is smaller
- `vmax` the **viewport maximum unit** represents the maximum between the height or width in terms of percentage. `30vmax` is the 30% of the current width or height, depending which one is bigger

Fraction units

`fr` are fraction units, and they are used in CSS Grid to divide space into fractions.

We'll talk about them in the context of CSS Grid later on.

`url()`

When we talk about background images, `@import`, and more, we use the `url()` function to load a resource:

```
div {
  background-image: url(test.png);
}
```

In this case I used a relative URL, which searches the file in the folder where the CSS file is defined.

I could go one level back

```
div {
  background-image: url(../test.png);
}
```

or go into a folder

```
div {
  background-image: url(subfolder/test.png);
}
```

Or I could load a file starting from the root of the domain where the CSS is hosted:

```
div {
  background-image: url(/test.png);
}
```

Or I could use an absolute URL to load an external resource:

```
div {
  background-image: url(https://mysite.com/test.png);
}
```

`calc()`

The `calc()` function lets you perform basic math operations on values, and it's especially useful when you need to add or subtract a length value from a percentage.

This is how it works:

```
div {
  max-width: calc(80% - 100px);
}
```

It returns a length value, so it can be used anywhere you expect a pixel value.

You can perform

- additions using `+`
- subtractions using `-`
- multiplication using `*`
- division using `/`

> One caveat: with addition and subtraction, the space around the operator is mandatory, otherwise it does not work as expected.

Examples:

```
div {
  max-width: calc(50% / 3);
}
```

```
div {
  max-width: calc(50% + 3px);
}
```

Modern CSS Functions for Responsive Design

CSS now includes powerful comparison functions that make responsive design much more flexible and maintainable. These functions allow you to create fluid, responsive values without media queries.

The clamp() Function

The `clamp()` function lets you set a value that adjusts between a minimum and maximum based on a preferred value. It's perfect for responsive typography and spacing.

Think of `clamp()` as a smart value that knows its limits. It tries to use your preferred value (the middle one), but if that would be too small, it uses the minimum, and if it would be too large, it uses the maximum. This creates truly fluid, responsive designs without needing any media queries.

The syntax is: `clamp(minimum, preferred, maximum)`. The preferred value is usually relative (like viewport units), while the min and max are often fixed values.

```
/* Responsive font size: minimum 1rem, maximum 3rem */
h1 {
  font-size: clamp(1rem, 4vw, 3rem);
}
```

This heading will scale with the viewport width (4vw means 4% of viewport width). On a phone (say 400px wide), 4vw would be 16px, but we've set a minimum of 1rem (usually 16px), so it won't go smaller. On a large screen (2000px wide), 4vw would be 80px, but our maximum of 3rem (48px) prevents it from getting too large. In between, it scales smoothly.

```
/* Responsive padding */
.container {
  padding: clamp(1rem, 5%, 3rem);
}
```

Here, padding adjusts based on the container's width (5%). On narrow screens, it won't go below 1rem, keeping content readable. On wide screens, it caps at 3rem, preventing excessive whitespace.

```
/* Responsive width with limits */
.card {
```

```
  width: clamp(250px, 50%, 600px);
}
```

This card tries to be 50% of its container's width, but never smaller than 250px (ensuring content doesn't get squished) and never larger than 600px (maintaining readability).

The `clamp()` function is particularly useful for creating fluid typography that scales smoothly with viewport size while respecting minimum and maximum boundaries.

The min() Function

The `min()` function returns the smallest value from a list of comma-separated values. It's useful for setting maximum constraints.

```
/* Choose the smaller value */

/* Ensure element never exceeds viewport width or 1200px */
.container {
  width: min(100%, 1200px);
}

/* Responsive gap that shrinks on small screens */
.grid {
  gap: min(2rem, 5vw);
}

/* Font size that adapts but has a ceiling */
h2 {
  font-size: min(2.5rem, 8vw);
}
```

The max() Function

The `max()` function returns the largest value from a list. It's perfect for setting minimum constraints.

```
/* Choose the larger value */

/* Ensure minimum width on small screens */
.button {
  width: max(200px, 50%);
}

/* Minimum font size */
p {
  font-size: max(1rem, 2vw);
}
```

```
/* Responsive margin with minimum */
.section {
  margin-bottom: max(1.5rem, 10vh);
}
```

Combining Functions

These functions become even more powerful when combined:

```
/* Complex responsive sizing */
.hero {
  height: min(max(400px, 60vh), 800px);
  /* At least 400px, up to 60vh, but never more than 800px */
}

/* Responsive with calc() */
.sidebar {
  width: clamp(200px, calc(100% - 2rem), 350px);
}

/* Multiple values in min/max */
.element {
  padding: min(10vw, 5rem, calc(100% - 80px));
}
```

Practical Examples

Here are some common responsive patterns using these functions:

```
/* Fluid typography scale */
:root {
  --text-xs: clamp(0.75rem, 1.5vw, 0.875rem);
  --text-sm: clamp(0.875rem, 2vw, 1rem);
  --text-base: clamp(1rem, 2.5vw, 1.125rem);
  --text-lg: clamp(1.125rem, 3vw, 1.5rem);
  --text-xl: clamp(1.5rem, 4vw, 2rem);
  --text-2xl: clamp(2rem, 5vw, 3rem);
}

/* Responsive container with padding */
.container {
  width: min(100% - 2rem, 1400px);
  margin-inline: auto;
  padding: clamp(1rem, 5vw, 4rem);
}
```

```
/* Responsive grid */
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(min(100%, 300px), 1fr));
  gap: clamp(1rem, 3vw, 2rem);
}
```

These modern functions eliminate many media queries and create smoother responsive experiences that adapt fluidly to any screen size.

Backgrounds

The background of an element can be changed using several CSS properties:

- `background-color`
- `background-image`
- `background-clip`
- `background-position`
- `background-origin`
- `background-repeat`
- `background-attachment`
- `background-size`

and the shorthand property `background`, which allows to shorten definitions and group them on a single line.

`background-color` accepts a color value, which can be one of the color keywords, or an `rgb` or `hsl` value:

```
p {
  background-color: yellow;
}

div {
  background-color: #333;
}
```

Instead of using a color, you can use an image as background to an element, by specifying the image location URL:

```
div {
  background-image: url(image.png);
}
```

`background-clip` lets you determine the area used by the background image, or color. The default value is `border-box`, which extends up to the border outer edge.

Other values are

- `padding-box` to extend the background up to the padding edge, without the border
- `content-box` to extend the background up to the content edge, without the padding
- `inherit` to apply the value of the parent

When using an image as background you will want to set the position of the image placement using the `background-position` property: `left`, `right`, `center` are all valid values for the X axis, and `top`, `bottom` for the Y axis:

```
div {
  background-position: top right;
}
```

If the image is smaller than the background, you need to set the behavior using `background-repeat`. Should it `repeat-x`, `repeat-y` or `repeat` on all the axis? This last one is the default value. Another value is `no-repeat`.

`background-origin` lets you choose where the background should be applied: to the entire element including padding (default) using `padding-box`, to the entire element including the border using `border-box`, to the element without the padding using `content-box`.

With `background-attachment` we can attach the background to the viewport, so that scrolling will not affect the background:

```
div {
  background-attachment: fixed;
}
```

By default the value is `scroll`. There is another value, `local`. The best way to visualize their behavior is this Codepen.

The last background property is `background-size`. We can use 3 keywords: `auto`, `cover` and `contain`. `auto` is the default.

`cover` expands the image until the entire element is covered by the background.

`contain` stops expanding the background image when one dimension (x or y) covers the whole smallest edge of the image, so it's fully contained into the element.

You can also specify a length value, and if so it sets the width of the background image (and the height is automatically defined):

```
div {
  background-size: 100%;
}
```

If you specify 2 values, one is the width and the second is the height:

```
div {
  background-size: 800px 600px;
}
```

The shorthand property `background` allows to shorten definitions and group them on a single line.

This is an example:

```
div {
  background: url(bg.png) top left no-repeat;
}
```

If you use an image, and the image could not be loaded, you can set a fallback color:

```
div {
  background: url(image.png) yellow;
}
```

You can also set a gradient as background:

```
div {
  background: linear-gradient(#fff, #333);
}
```

Comments

CSS gives you the ability to write comments in a CSS file, or in the `style` tag in the page header

The format is the `/* this is a comment */` C-style (or JavaScript-style, if you prefer) comments.

This is a multiline comment. Until you add the closing `*/` token, the all the lines found after the opening one are commented.

Example:

```
/*
#name {
  display: block;
} */

#name {
  display: block;
  /* color: red; */
}
```

CSS does not have inline comments, like `//` in C or JavaScript.

Pay attention though - if you add `//` before a rule, the rule will not be applied, looking like the comment worked. In reality, CSS detected a syntax error and due to how it works it ignored the line with the error, and went straight to the next line.

Knowing this approach lets you purposefully write inline comments, although you have to be careful because you can't add random text like you can in a block comment.

For example:

```
// Nice rule!#name {
  display: block;
}
```

In this case, due to how CSS works, the `#name` rule is actually commented out. You can find more details here if you find this interesting. To avoid shooting yourself in the foot, just avoid using inline comments and rely on block comments.

Fonts

At the dawn of the web you only had a handful of fonts you could choose from.

Thankfully today you can load any kind of font on your pages.

CSS has gained many nice capabilities over the years in regards to fonts.

The `font` property is the shorthand for a number of properties:

- `font-family`
- `font-weight`
- `font-stretch`
- `font-style`
- `font-size`

Let's see each one of them and then we'll cover `font`.

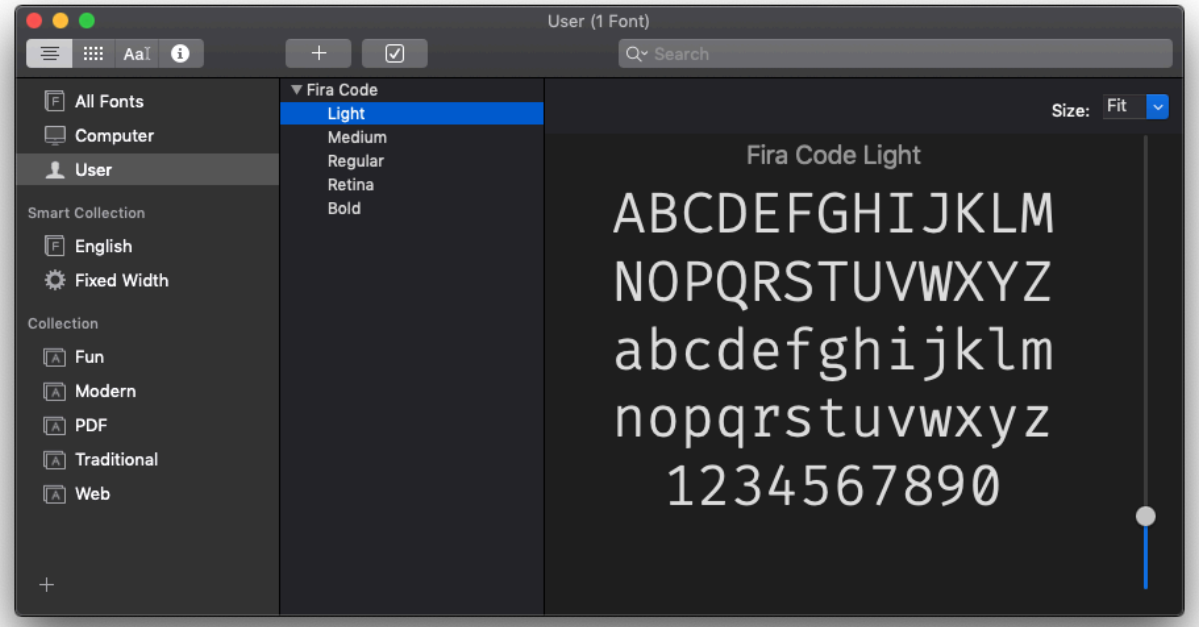
Then we'll talk about how to load custom fonts, using `@import` or `@font-face`, or by loading a font stylesheet.

`font-family`

Sets the font *family* that the element will use.

Why "family"? Because what we know as a font is actually composed of several sub-fonts. which provide all the style (bold, italic, light..) we need.

Here's an example from my Mac's Font Book app - the Fira Code font family hosts several dedicated fonts underneath:



This property lets you select a specific font, for example:

```
body {
  font-family: Helvetica;
}
```

You can set multiple values, so the second option will be used if the first cannot be used for some reason (if it's not found on the machine, or the network connection to download the font failed, for example):

```
body {
  font-family: Helvetica, Arial;
}
```

I used some specific fonts up to now, ones we call **Web Safe Fonts**, as they are pre-installed on different operating systems.

We divide them in Serif, Sans-Serif, and Monospace fonts. Here's a list of some of the most popular ones:

Serif

- Georgia
- Palatino
- Times New Roman
- Times

Sans-Serif

- Arial
- Helvetica
- Verdana
- Geneva
- Tahoma
- Lucida Grande
- Impact
- Trebuchet MS
- Arial Black

Monospace

- Courier New
- Courier
- Lucida Console
- Monaco

You can use all of those as `font-family` properties, but they are not guaranteed to be there for every system. Others exist, too, with a varying level of support.

You can also use generic names:

- `sans-serif` a font without ligatures
- `serif` a font with ligatures
- `monospace` a font especially good for code
- `cursive` used to simulate handwritten pieces
- `fantasy` the name says it all

Those are typically used at the end of a `font-family` definition, to provide a fallback value in case nothing else can be applied:

```
body {
  font-family: Helvetica, Arial, sans-serif;}
```

`font-weight`

This property sets the width of a font. You can use those predefined values:

- normal
- bold
- bolder (relative to the parent element)
- lighter (relative to the parent element)

Or using the numeric keywords

- 100
- 200
- 300
- 400, mapped to `normal`
- 500
- 600
- 700 mapped to `bold`
- 800
- 900

where 100 is the lightest font, and 900 is the boldest.

Some of those numeric values might not map to a font, because that must be provided in the font family. When one is missing, CSS makes that number be at least as bold as the preceding one, so you might have numbers that point to the same font.

`font-stretch`

Allows to choose a narrow or wide face of the font, if available.

This is important: the font must be equipped with different faces.

Values allowed are, from narrower to wider:

- `ultra-condensed`
- `extra-condensed`
- `condensed`
- `semi-condensed`
- `normal`
- `semi-expanded`

- `expanded`
- `extra-expanded`
- `ultra-expanded`

`font-style`

Allows you to apply an italic style to a font:

```
p {
  font-style: italic;
}
```

This property also allows the values `oblique` and `normal`. There is very little, if any, difference between using `italic` and `oblique`. The first is easier to me, as HTML already offers an `i` element which means italic.

`font-size`

This property is used to determine the size of fonts.

You can pass 2 kinds of values:

1. a length value, like `px`, `em`, `rem` etc, or a percentage
2. a predefined value keyword

In the second case, the values you can use are:

- xx-small
- x-small
- small
- medium
- large
- x-large
- xx-large
- smaller (relative to the parent element)
- larger (relative to the parent element)

Usage:

```
p {
  font-size: 20px;
}

li {
```

```
  font-size: medium;
}
```

`font-variant`

This property was originally used to change the text to small caps, and it had just 3 valid values:

- `normal`
- `inherit`
- `small-caps`

Small caps means the text is rendered in “smaller caps” beside its uppercase letters.

`font`

The `font` property lets you apply different font properties in a single one, reducing the clutter.

We must at least set 2 properties, `font-size` and `font-family`, the others are optional:

```
body {
  font: 20px Helvetica;
}
```

If we add other properties, they need to be put in the correct order.

This is the order:

```
font: <font-stretch> <font-style> <font-variant> <font-weight> <font-size>
<line-height> <font-family>;
```

Example:

```
body {
  font: italic bold 20px Helvetica;
}

section {
  font: small-caps bold 20px Helvetica;
}
```

Loading custom fonts using `@font-face`

`@font-face` lets you add a new font family name, and map it to a file that holds a font.

This font will be downloaded by the browser and used in the page, and it's been such a fundamental change to typography on the web - we can now use any font we want.

We can add `@font-face` declarations directly into our CSS, or link to a CSS dedicated to importing the font.

In our CSS file we can also use `@import` to load that CSS file.

A `@font-face` declaration contains several properties we use to define the font, including `src`, the URI (one or more URIs) to the font. This follows the same-origin policy, which means fonts can only be downloaded form the current origin (domain + port + protocol).

Fonts are usually in the formats

- `woff` (Web Open Font Format)
- `woff2` (Web Open Font Format 2.0)
- `eot` (Embedded Open Type)
- `otf` (OpenType Font)
- `ttf` (TrueType Font)

The following properties allow us to define the properties to the font we are going to load, as we saw above:

- `font-family`
- `font-weight`
- `font-style`
- `font-stretch`

A note on performance

Of course loading a font has performance implications which you must consider when creating the design of your page.

Typography

We already talked about fonts, but there's more to styling text.

In this section we'll talk about the following properties:

- `text-transform`
- `text-decoration`
- `text-align`
- `vertical-align`
- `line-height`
- `text-indent`

- `text-align-last`
- `word-spacing`
- `letter-spacing`
- `text-shadow`
- `white-space`
- `tab-size`
- `writing-mode`
- `hyphens`
- `text-orientation`
- `direction`
- `line-break`
- `word-break`
- `overflow-wrap`

`text-transform`

This property can transform the case of an element.

There are 4 valid values:

- `capitalize` to uppercase the first letter of each word
- `uppercase` to uppercase all the text
- `lowercase` to lowercase all the text
- `none` to disable transforming the text, used to avoid inheriting the property

Example:

```
p {
  text-transform: uppercase;
}
```

`text-decoration`

This property is sed to add decorations to the text, including

- `underline`
- `overline`
- `line-through`
- `blink`
- `none`

Example:

```
p {
  text-decoration: underline;
}
```

You can also set the style of the decoration, and the color.

Example:

```
p {
  text-decoration: underline dashed yellow;
}
```

Valid style values are `solid`, `double`, `dotted`, `dashed`, `wavy`.

You can do all in one line, or use the specific properties:

- `text-decoration-line`
- `text-decoration-color`
- `text-decoration-style`

Example:

```
p {
  text-decoration-line: underline;
  text-decoration-color: yellow;
  text-decoration-style: dashed;
}
```

`text-align`

By default text align has the `start` value, meaning the text starts at the “start”, origin 0, 0 of the box that contains it. This means top left in left-to-right languages, and top right in right-to-left languages.

Possible values are `start`, `end`, `left`, `right`, `center`, `justify` (nice to have a consistent spacing at the line ends):

```
p {
  text-align: right;
}
```

`vertical-align`

Determines how inline elements are vertically aligned.

We have several values for this property. First we can assign a length or percentage value. Those are used to align the text in a position higher or lower (using negative values) than the baseline of the parent element.

Then we have the keywords:

- `baseline` (the default), aligns the baseline to the baseline of the parent element
- `sub` makes an element subscripted, simulating the `sub` HTML element result
- `super` makes an element superscripted, simulating the `sup` HTML element result
- `top` align the top of the element to the top of the line
- `text-top` align the top of the element to the top of the parent element font
- `middle` align the middle of the element to the middle of the line of the parent
- `bottom` align the bottom of the element to the bottom of the line
- `text-bottom` align the bottom of the element to the bottom of the parent element font

`line-height`

This allows you to change the height of a line. Each line of text has a certain font height, but then there is additional spacing vertically between the lines. That's the line height:

```
p {
  line-height: 0.9rem;
}
```

`text-indent`

Indent the first line of a paragraph by a set length, or a percentage of the paragraph width:

```
p {
  text-indent: -10px;
}
```

`text-align-last`

By default the last line of a paragraph is aligned following the `text-align` value. Use this property to change that behavior:

```
p {
  text-align-last: right;
}
```

`word-spacing`

Modifies the spacing between each word.

You can use the `normal` keyword, to reset inherited values, or use a length value:

```
p {
  word-spacing: 2px;
}

span {
  word-spacing: -0.2em;
}
```

`letter-spacing`

Modifies the spacing between each letter.

You can use the `normal` keyword, to reset inherited values, or use a length value:

```
p {
  letter-spacing: 0.2px;
}

span {
  letter-spacing: -0.2em;
}
```

`text-shadow`

Apply a shadow to the text. By default the text has now shadow.

This property accepts an optional color, and a set of values that set

- the X offset of the shadow from the text
- the Y offset of the shadow from the text
- the blur radius

If the color is not specified, the shadow will use the text color.

Examples:

```
p {
  text-shadow: 0.2px 2px;
}

span {
  text-shadow: yellow 0.2px 2px 3px;
}
```

`white-space`

Sets how CSS handles the white space, new lines and tabs inside an element.

Valid values that collapse white space are:

- `normal` collapses white space. Adds new lines when necessary as the text reaches the container end
- `nowrap` collapses white space. Does not add a new line when the text reaches the end of the container, and suppresses any line break added to the text
- `pre-line` collapses white space. Adds new lines when necessary as the text reaches the container end

Valid values that preserve white space are:

- `pre` preserves white space. Does not add a new line when the text reaches the end of the container, but preserves line break added to the text
- `pre-wrap` preserves white space. Adds new lines when necessary as the text reaches the container end

`tab-size`

Sets the width of the tab character. By default it's 8, and you can set an integer value that sets the character spaces it takes, or a length value:

```
p {
  tab-size: 2;
}

span {
  tab-size: 4px;
}
```

`writing-mode`

Defines whether lines of text are laid out horizontally or vertically, and the direction in which blocks progress.

The values you can use are

- `horizontal-tb` (default)
- `vertical-rl` content is laid out vertically. New lines are put on the left of the previous
- `vertical-lr` content is laid out vertically. New lines are put on the right of the previous

`hyphens`

Determines if hyphens should be automatically added when going to a new line.

Valid values are

- `none` (default)
- `manual` only add an hyphen when there is already a visible hyphen or a hidden hyphen (a special character)
- `auto` add hyphens when determined the text can have a hyphen.

`text-orientation`

When `writing-mode` is in a vertical mode, determines the orientation of the text.

Valid values are

- `mixed` is the default, and if a language is vertical (like Japanese) it preserves that orientation, while rotating text written in western languages
- `upright` makes all text be vertically oriented
- `sideways` makes all text horizontally oriented

`direction`

Sets the direction of the text. Valid values are `ltr` and `rtl`:

```
p {
  direction: rtl;
}
```

`word-break`

This property specifies how to break lines within words.

- `normal` (default) means the text is only broken between words, not inside a word
- `break-all` the browser can break a word (but no hyphens are added)
- `keep-all` suppress soft wrapping. Mostly used for CJK (Chinese/Japanese/Korean) text.

Speaking of CJK text, the property `line-break` is used to determine how text lines break. I'm not an expert with those languages, so I will avoid covering it.

`overflow-wrap`

If a word is too long to fit a line, it can overflow outside of the container.

> This property is also known as word-wrap, although that is non-standard (but still works as an alias)

This is the default behavior (`overflow-wrap: normal;`).

We can use:

```
p {
  overflow-wrap: break-word;
}
```

to break it at the exact length of the line, or

```
p {
  overflow-wrap: anywhere;
}
```

if the browser sees there's a soft wrap opportunity somewhere earlier. No hyphens are added, in any case.

This property is very similar to `word-break` . We might want to choose this one on western languages, while `word-break` has special treatment for non-western languages.

Box Model

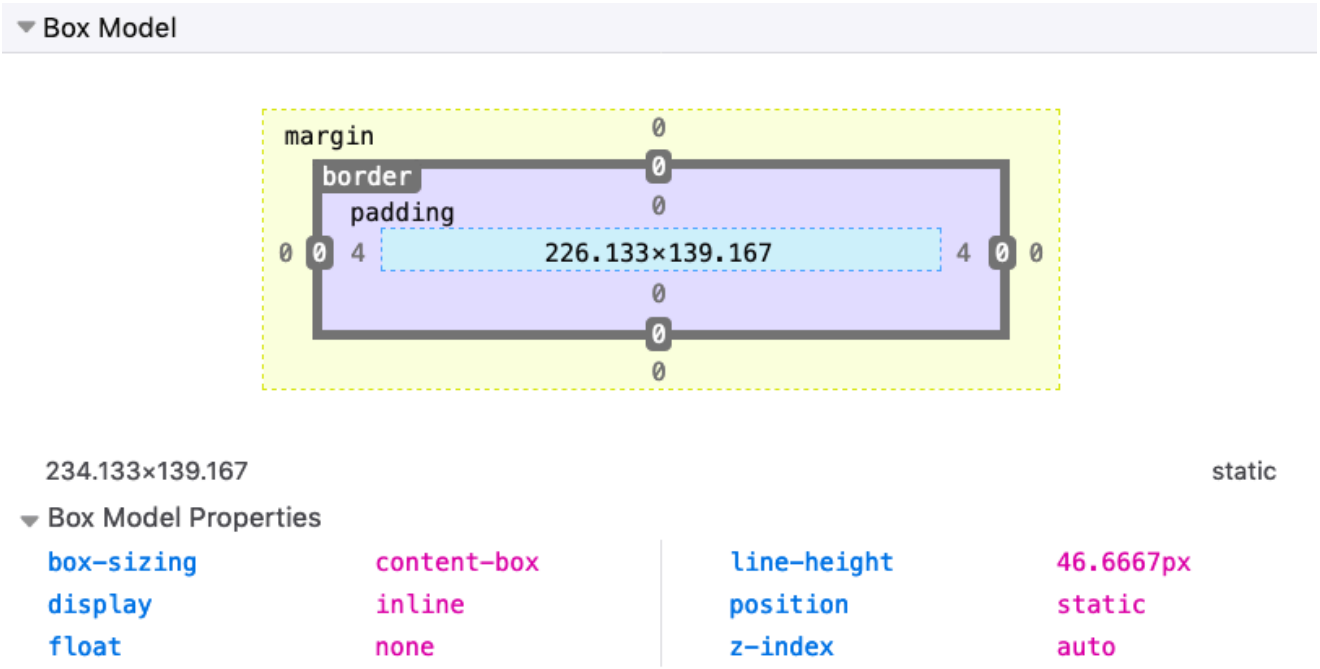
Every CSS element is essentially a box. Every element is a generic box.

The box model explains the sizing of the elements based on a few CSS properties.

From the inside to the outside, we have:

- the content area
- padding
- border
- margin

The best way to visualize the box model is to open the browser DevTools and check how it is displayed:



Here you can see how Firefox tells me the properties of a `span` element I highlighted. I right-clicked on it, pressed Inspect Element, and went to the Layout panel of the DevTools.

See, the light blue space is the content area. Surrounding it there is the padding, then the border and finally the margin.

By default, if you set a width (or height) on the element, that is going to be applied to the **content area**. All the padding, border, and margin calculations are done outside of the value, so you have to take this in mind when you do your calculation.

You can change this behavior using Box Sizing.

Border

The border is a thin layer between padding and margin. Editing the border you can make elements draw their perimeter on screen.

You can work on borders by using those properties:

- `border-style`
- `border-color`
- `border-width`

The property `border` can be used as a shorthand for all those properties.

`border-radius` is used to create rounded corners.

You also have the ability to use images as borders, an ability given to you by `border-image` and its specific separate properties:

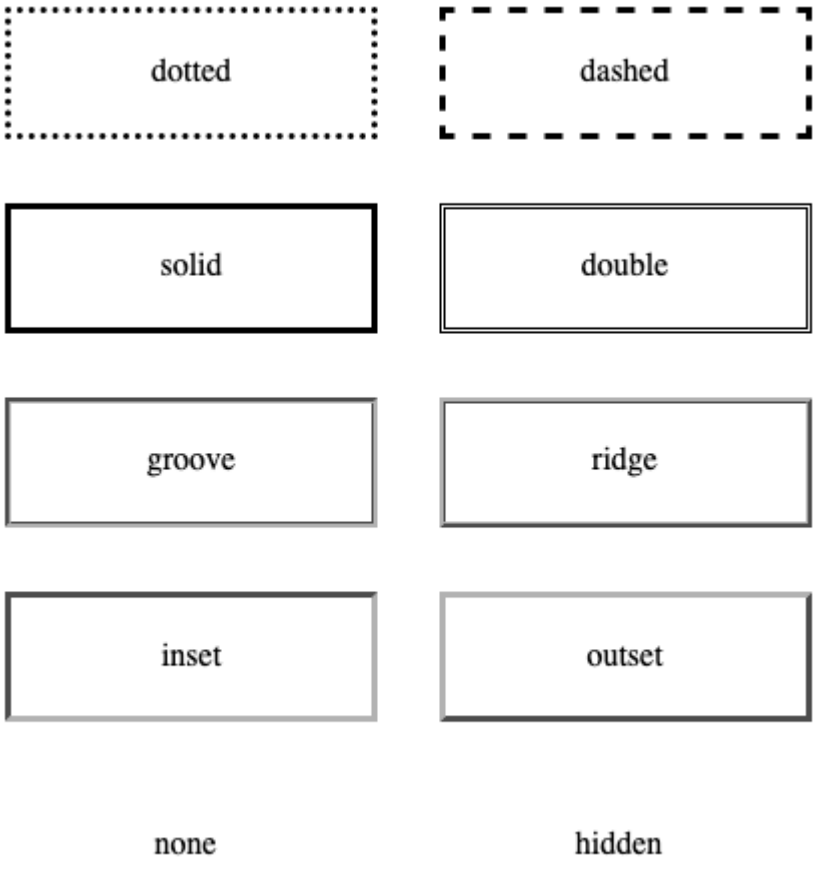
- `border-image-source`
- `border-image-slice`
- `border-image-width`
- `border-image-outset`
- `border-image-repeat`

Let's start with `border-style`.

The border style

The `border-style` property lets you choose the style of the border. The options you can use are:

- `dotted`
- `dashed`
- `solid`
- `double`
- `groove`
- `ridge`
- `inset`
- `outset`
- `none`
- `hidden`



Check this Codepen for a live example

The default for the style is `none`, so to make the border appear at all you need to change it to something else. `solid` is a good choice most of the times.

You can set a different style for each edge using the properties

- `border-top-style`
- `border-right-style`
- `border-bottom-style`
- `border-left-style`

or you can use `border-style` with multiple values to define them, using the usual Top-Right-Bottom-Left order:

```
p {
  border-style: solid dotted solid dotted;
}
```

The border width

`border-width` is used to set the width of the border.

You can use one of the pre-defined values:

- `thin`
- `medium` (the default value)
- `thick`

or express a value in pixels, em or rem or any other valid length value.

Example:

```
p {
  border-width: 2px;
}
```

You can set the width of each edge (Top-Right-Bottom-Left) separately by using 4 values:

```
p {
  border-width: 2px 1px 2px 1px;
}
```

or you can use the specific edge properties `border-top-width`, `border-right-width`, `border-bottom-width`, `border-left-width`.

The border color

`border-color` is used to set the color of the border.

If you don't set a color, the border by default is colored using the color of the text in the element.

You can pass any valid color value to `border-color`.

Example:

```
p {
  border-color: yellow;
}
```

You can set the color of each edge (Top-Right-Bottom-Left) separately by using 4 values:

```
p {
  border-color: black red yellow blue;
}
```

or you can use the specific edge properties `border-top-color`, `border-right-color`, `border-bottom-color`, `border-left-color`.

The border shorthand property

Those 3 properties mentioned, `border-width`, `border-style` and `border-color` can be set using the shorthand property `border`.

Example:

```
p {
  border: 2px black solid;
}
```

You can also use the edge-specific properties `border-top`, `border-right`, `border-bottom`, `border-left`.

Example:

```
p {
  border-left: 2px black solid;
  border-right: 3px red dashed;
}
```

The border radius

`border-radius` is used to set rounded corners to the border. You need to pass a value that will be used as the radius of the circle that will be used to round the border.

Usage:

```
p {
  border-radius: 3px;
}
```

You can also use the edge-specific properties `border-top-left-radius`, `border-top-right-radius`, `border-bottom-left-radius`, `border-bottom-right-radius`.

Using images as borders

One very cool thing with borders is the ability to use images to style them. This lets you go very creative with borders.

We have 5 properties:

- `border-image-source`
- `border-image-slice`
- `border-image-width`
- `border-image-outset`
- `border-image-repeat`

and the shorthand `border-image`. I won't go in much details here as images as borders would need a more in-depth coverage as the one I can do in this little chapter. I recommend reading the CSS Tricks almanac entry on border-image for more information.

Padding

The `padding` CSS property is commonly used in CSS to add space in the inner side of an element.

Remember:

- `margin` adds space outside an element border
- `padding` adds space inside an element border

Specific padding properties

`padding` has 4 related properties that alter the padding of a single edge at once:

- `padding-top`
- `padding-right`
- `padding-bottom`

- `padding-left`

The usage of those is very simple and cannot be confused, for example:

```
padding-left: 30px;
padding-right: 3em;
```

Using the `padding` shorthand

`padding` is a shorthand to specify multiple padding values at the same time, and depending on the number of values entered, it behaves differently.

Using a single value applies that to **all** the paddings: top, right, bottom, left.

```
padding: 20px;
```

Using 2 values applies the first to **bottom & top**, and the second to **left & right**.

```
padding: 20px 10px;
```

Using 3 values applies the first to **top**, the second to **left & right**, the third to **bottom**.

```
padding: 20px 10px 30px;
```

Using 4 values applies the first to **top**, the second to **right**, the third to **bottom**, the fourth to **left**.

```
padding: 20px 10px 5px 0px;
```

So, the order is *top-right-bottom-left*.

Values accepted

`padding` accepts values expressed in any kind of length unit, the most common ones are px, em, rem, but many others exist.

Margin

The `margin` CSS property is commonly used in CSS to add space around an element.

Remember:

- `margin` adds space outside an element border
- `padding` adds space inside an element border

Specific margin properties

`margin` has 4 related properties that alter the margin of a single edge at once:

- `margin-top`
- `margin-right`
- `margin-bottom`
- `margin-left`

The usage of those is very simple and cannot be confused, for example:

```
margin-left: 30px;
margin-right: 3em;
```

Using the `margin` shorthand

`margin` is a shorthand to specify multiple margins at the same time, and depending on the number of values entered, it behaves differently.

Using a single value applies that to **all** the margins: top, right, bottom, left.

```
margin: 20px;
```

Using 2 values applies the first to **bottom & top**, and the second to **left & right**.

```
margin: 20px 10px;
```

Using 3 values applies the first to **top**, the second to **left & right**, the third to **bottom**.

```
margin: 20px 10px 30px;
```

Using 4 values applies the first to **top**, the second to **right**, the third to **bottom**, the fourth to **left**.

```
margin: 20px 10px 5px 0px;
```

So, the order is *top-right-bottom-left*.

Values accepted

`margin` accepts values expressed in any kind of length unit, the most common ones are px, em, rem, but many others exist.

It also accepts percentage values, and the special value `auto`.

Using `auto` to center elements

`auto` can be used to tell the browser to select automatically a margin, and it's most commonly used to center an element in this way:

```
margin: 0 auto;
```

As said above, using 2 values applies the first to **bottom & top**, and the second to **left & right**.

The modern way to center elements is to use Flexbox, and its `justify-content: center;` directive.

All modern browsers have excellent Flexbox support. The `margin: 0 auto;` technique remains useful as an alternative approach for centering.

Using a negative margin

`margin` is the only property related to sizing that can have a negative value. It's extremely useful, too.
Setting a negative top margin makes an element move over elements before it, and given enough negative value it will move out of the page.

A negative bottom margin moves up the elements after it.

A negative right margin makes the content of the element expand beyond its allowed content size.

A negative left margin moves the element left over the elements that precede it, and given enough negative value it will move out of the page.

Box Sizing

The default behavior of browsers when calculating the width of an element is to apply the calculated width and height to the **content area**, without taking any of the padding, border and margin in consideration.

This approach has proven to be quite complicated to work with.

You can change this behavior by setting the `box-sizing` property.

The `box-sizing` property is a great help. It has 2 values:

- `border-box`
- `content-box`

`content-box` is the default, the one we had for ages before `box-sizing` became a thing.

`border-box` is the new and great thing we are looking for. If you set that on an element:

```
.my-div {
  box-sizing: border-box;
}
```

width and height calculation include the padding and the border. Only the margin is left out, which is reasonable since in our mind we also typically see that as a separate thing: margin is outside of the box.

This property is a small change but has a big impact. CSS Tricks even declared an international box-sizing awareness day, just saying, and it's recommended to apply it to every element on the page, out of the box, with this:

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

Aspect Ratio and Modern Sizing

Modern CSS provides powerful properties for maintaining aspect ratios and controlling element sizing more precisely.

Before the `aspect-ratio` property, maintaining consistent proportions (especially for responsive images and videos) required hacky techniques like the "padding-bottom trick." Now, we can simply tell an element what proportions to maintain, and CSS handles the rest. This is especially useful for responsive design where elements need to scale while keeping their shape.

The aspect-ratio Property

The `aspect-ratio` property lets you define an element's preferred aspect ratio, automatically calculating one dimension based on the other.

Think of aspect ratio like the shape of a TV screen or photo frame. A 16:9 ratio means for every 16 units of width, there are 9 units of height. The beauty of `aspect-ratio` is that you only need to set one dimension (width OR height), and CSS automatically calculates the other to maintain the proportion.

```
/* Square aspect ratio */
.square {
  aspect-ratio: 1; /* or 1/1 */
  width: 200px;
```

```
  /* Height automatically becomes 200px */
}
```

This creates a perfect square. Set the width to 200px, and the height automatically becomes 200px too. If the width changes (say, on a smaller screen), the height adjusts to maintain the square shape.

```
/* 16:9 video ratio */
.video-container {
  aspect-ratio: 16/9;
  width: 100%;
  /* Height automatically adjusts */
}
```

This is the standard widescreen video ratio. The container takes the full width of its parent, and its height is automatically calculated to be 9/16ths (or 56.25%) of the width. Perfect for responsive video embeds!

```
/* Portrait ratio */
.portrait {
  aspect-ratio: 3/4;
  height: 400px;
  /* Width automatically becomes 300px */
}
```

Here we're setting the height and letting CSS calculate the width. With a 3:4 ratio and 400px height, the width becomes 300px (3/4 of 400).

```
/* Common ratios */
.golden-ratio { aspect-ratio: 1.618; }
.widescreen { aspect-ratio: 16/9; }
.classic-tv { aspect-ratio: 4/3; }
.cinema { aspect-ratio: 2.35/1; }
.square { aspect-ratio: 1; }
.instagram-portrait { aspect-ratio: 4/5; }
```

Responsive Images and Videos

Using aspect-ratio for responsive media:

```
/* Responsive video wrapper */
.video-wrapper {
  aspect-ratio: 16/9;
  width: 100%;
  max-width: 800px;
```

```
  overflow: hidden;
}

.video-wrapper iframe {
  width: 100%;
  height: 100%;
  border: 0;
}

/* Responsive image with maintained ratio */
.image-container {
  aspect-ratio: 3/2;
  overflow: hidden;
}

.image-container img {
  width: 100%;
  height: 100%;
  object-fit: cover;
}
```

Object Fit and Position

Control how replaced elements (images, videos) fit within their containers:

```
/* object-fit values */
.cover {
  object-fit: cover; /* Fills container, may crop */
}

.contain {
  object-fit: contain; /* Fits entirely, may have space */
}

.fill {
  object-fit: fill; /* Stretches to fill (default) */
}

.none {
  object-fit: none; /* Natural size, may overflow */
}

.scale-down {
  object-fit: scale-down; /* Smaller of none or contain */
}

/* Control positioning within container */
.positioned {
```

```
  object-fit: cover;
  object-position: top left; /* Focus on top-left */
}

.centered {
  object-fit: cover;
  object-position: center; /* Default */
}

.custom-position {
  object-fit: cover;
  object-position: 25% 75%; /* Custom positioning */
}
```

Modern Sizing Units

Container Query Units

Container query units are relative to a container's dimensions:

```
/* Container query units */
.container {
  container-type: inline-size;
}

.child {
  /* cqw = 1% of container width */
  width: 50cqw;

  /* cqh = 1% of container height */
  height: 30cqh;

  /* cqi = 1% of container inline size */
  padding: 2cqi;

  /* cqb = 1% of container block size */
  margin: 1cqb;

  /* cqmin/cqmax = smaller/larger of cqi or cqb */
  font-size: 5cqmin;
}
```

Dynamic Viewport Units

Modern viewport units that account for mobile browser UI:

```css
/* Traditional viewport units */
.old-full-height {
  height: 100vh; /* Can be cut off on mobile */
}

/* Dynamic viewport units */
.dynamic-height {
  /* svh = Small viewport height (browser UI visible) */
  min-height: 100svh;

  /* lvh = Large viewport height (browser UI hidden) */
  max-height: 100lvh;

  /* dvh = Dynamic viewport height (updates as UI appears/disappears) */
  height: 100dvh;
}

/* Same for width */
.dynamic-width {
  width: 100dvw; /* Dynamic viewport width */
  width: 100svw; /* Small viewport width */
  width: 100lvw; /* Large viewport width */
}
```

Intrinsic Sizing

CSS intrinsic sizing keywords:

```css
/* min-content - smallest size that fits content */
.min {
  width: min-content;
}

/* max-content - preferred size without wrapping */
.max {
  width: max-content;
}

/* fit-content - smaller of max-content or available space */
.fit {
  width: fit-content;
}

/* With limits */
.limited {
  width: fit-content(300px);
}
```

```css
/* Practical use cases */
.button {
  width: fit-content;
  min-width: 100px;
  max-width: 200px;
  padding: 10px 20px;
}

.card {
  width: min(100%, max-content);
}
```

Practical Examples

```css
/* Responsive card with aspect ratio */
.card {
  aspect-ratio: 3/4;
  width: 100%;
  max-width: 300px;
  overflow: hidden;
  border-radius: 8px;
}

.card img {
  width: 100%;
  height: 60%;
  object-fit: cover;
}

/* Gallery with consistent ratios */
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  gap: 1rem;
}

.gallery-item {
  aspect-ratio: 1;
  overflow: hidden;
}

.gallery-item img {
  width: 100%;
  height: 100%;
  object-fit: cover;
  transition: transform 0.3s;
}
```

```
.gallery-item:hover img {
  transform: scale(1.1);
}

/* Responsive hero section */
.hero {
  width: 100%;
  aspect-ratio: 21/9;
  min-height: 400px;
  max-height: 100dvh;
  position: relative;
}

.hero video {
  width: 100%;
  height: 100%;
  object-fit: cover;
}
```

Display

The `display` property of an object determines how it is rendered by the browser.

It's a very important property, and probably the one with the highest number of values you can use.

Those values include:

- `block`
- `inline`
- `none`
- `contents`
- `flow`
- `flow-root`
- `table` (and all the `table-*` ones)
- `flex`
- `grid`
- `list-item`
- `inline-block`
- `inline-table`
- `inline-flex`
- `inline-grid`
- `inline-list-item`

plus others you will not likely use, like `ruby`.

Choosing any of those will considerably alter the behavior of the browser with the element and its children.

In this section we'll analyze the most important ones not covered elsewhere:

- `block`
- `inline`
- `inline-block`
- `none`

We'll see some of the others in later chapters, including coverage of `table`, `flex` and `grid`.

`inline`

Inline is the default display value for every element in CSS.

All the HTML tags are displayed inline out of the box except some elements like `div`, `p` and `section`, which are set as `block` by the user agent (the browser).

Inline elements don't have any margin or padding applied.

Same for height and width.

You *can* add them, but the appearance in the page won't change - they are calculated and applied automatically by the browser.

`inline-block`

Similar to `inline`, but with `inline-block` `width` and `height` are applied as you specified.

`block`

As mentioned, normally elements are displayed inline, with the exception of some elements, including

- `div`
- `p`
- `section`
- `ul`

which are set as `block` by the browser.

With `display: block`, elements are stacked one after each other, vertically, and every element takes up 100% of the page.

The values assigned to the `width` and `height` properties are respected, if you set them, along with `margin` and `padding`.

`none`

Using `display: none` makes an element disappear. It's still there in the HTML, but just not visible in the browser.

Positioning

Positioning is what makes us determine where elements appear on the screen, and how they appear.

You can move elements around, and position them exactly where you want.

In this section we'll also see how things change on a page based on how elements with different `position` interact with each other.

We have one main CSS property: `position`.

It can have those 5 values:

- `static`
- `relative`
- `absolute`
- `fixed`
- `sticky`

Static positioning

This is the default value for an element. Static positioned elements are displayed in the normal page flow.

Relative positioning

If you set `position: relative` on an element, you are now able to position it with an offset, using the properties

- top
- right
- bottom
- left

which are called **offset properties**. They accept a length value or a percentage.

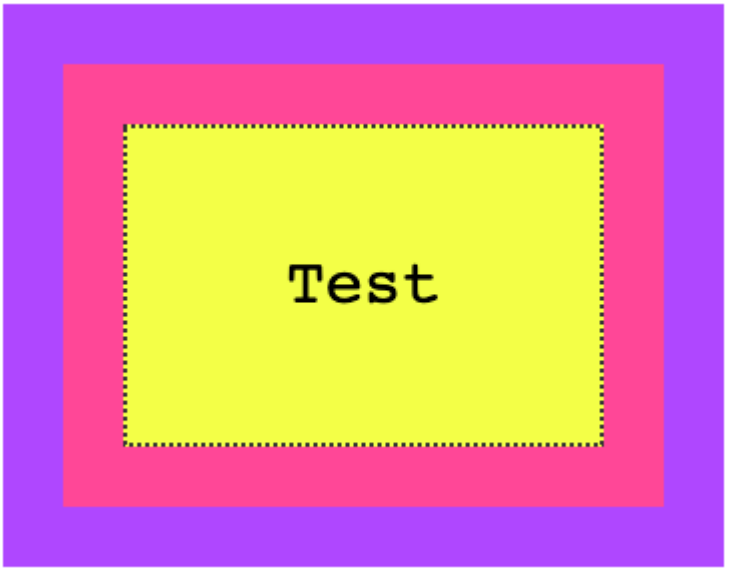
Take this example I made on Codepen. I create a parent container, a child container, and an inner box with some text:

```
<div class="parent">
  <div class="child">
    <div class="box">
      <p>Test</p>
    </div>
  </div>
</div>
```

with some CSS to give some colors and padding, but does not affect positioning:

```
.parent {
  background-color: #af47ff;
  padding: 30px;
  width: 300px
}
.child {
  background-color: #ff4797;
  padding: 30px
}
.box {
  background-color: #f3ff47;
  padding: 30px;
  border: 2px solid #333;
  border-style: dotted;
  font-family: courier;
  text-align: center;
  font-size: 2rem
}
```

here's the result:

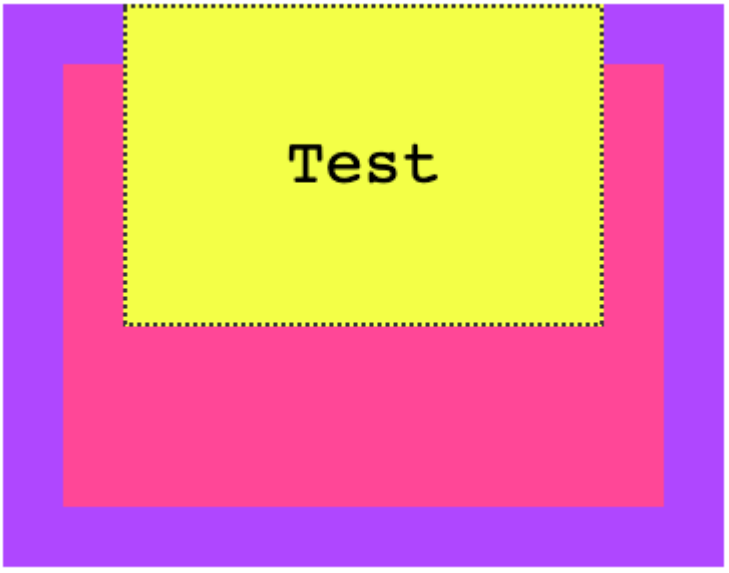


You can try and add any of the properties I mentioned before (`top` , `right` , `bottom` , `left`) to `.box` , and nothing will happen. The position is `static` .

Now if we set `position: relative` to the box, at first apparently nothing changes. But the element is now able to move using the `top` , `right` , `bottom` , `left` properties, and now you can alter the position of it relatively to the element containing it.

For example:

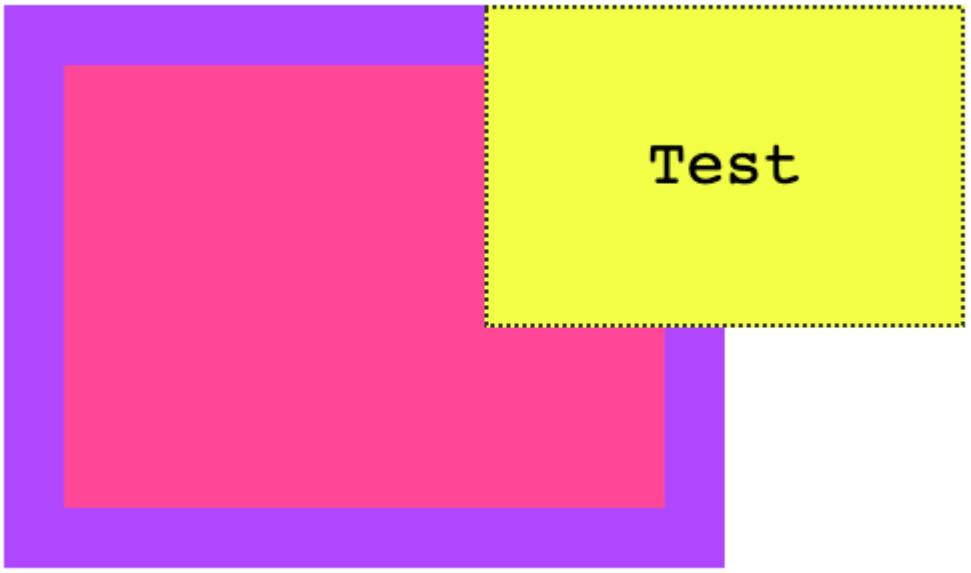
```
.box {
  /* ... */
  position: relative;
  top: -60px;
}
```



A negative value for `top` will make the box move up relatively to its container.

Or

```
.box {
  /* ... */
  position: relative;
  top: -60px;
  left: 180px;
}
```



Notice how the space that is occupied by the box remains preserved in the container, like it was still in its place.

Another property that will now work is `z-index` to alter the z-axis placement. We'll talk about it later on.

Absolute positioning

Setting `position: absolute` on an element will remove it from the document's flow, and it will not longer follow the original page positioning flow.

Remember in relative positioning that we noticed the space originally occupied by an element was preserved even if it was moved around?

With absolute positioning, as soon as we set `position: absolute` on `.box`, its original space is now collapsed, and only the origin (x, y coordinates) remain the same.

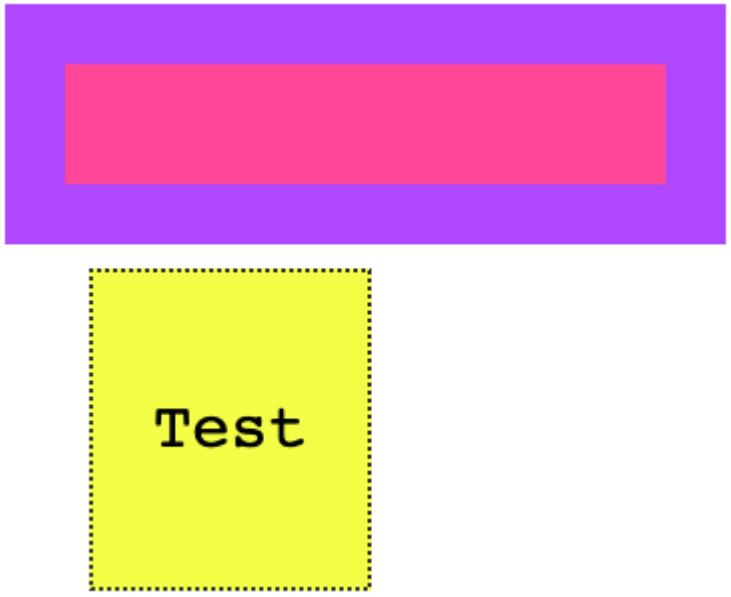
```
.box {
  /* ... */
  position: absolute;
}
```

We can now move the box around as we please, using the `top`, `right`, `bottom`, `left` properties:

```
.box {
  /* ... */
  position: absolute;
  top: 0px;
  left: 0px;
}
```

or

```
.box {
  /* ... */
  position: absolute;
  top: 140px;
  left: 50px;
}
```



The coordinates are relative to the closest container that is not `static`.

This means that if we add `position: relative` to the `.child` element, and we set `top` and `left` to 0, the box will not be positioned at the top left margin of the *window*, but rather it will be positioned at the 0, 0 coordinates of `.child`:

```
.child {
  /* ... */
  position: relative;
}

.box {
  /* ... */
  position: absolute;
  top: 0px;
  left: 0px;
}
```

Here's what happens if `.child` is static (the default):

```
.child {
  /* ... */
  position: static;
}

.box {
  /* ... */
  position: absolute;
  top: 0px;
  left: 0px;
}
```

Like for relative positioning, you can use `z-index` to alter the z-axis placement.

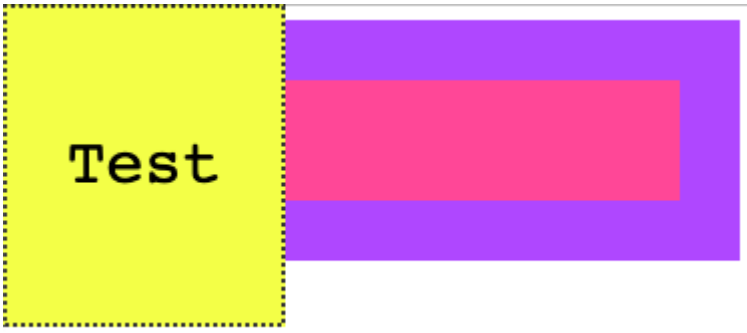
Fixed positioning

Like with absolute positioning, when an element is assigned `position: fixed` it's removed from the flow of the page.

The difference with absolute positioning is this: elements are now always positioned relative to the window, instead of the first non-static container.

```
.box {
  /* ... */
  position: fixed;
}
```


```
.box {
  /* ... */
  position: fixed;
  top: 0;
  left: 0;
}
```



Another big difference is that elements are not affected by scrolling. Once you put a sticky element somewhere, scrolling the page does not remove it from the visible part of the page.

Sticky positioning

While the above values have been around for a very long time, sticky positioning now has excellent support across all modern browsers.

The UITableView iOS component is the thing that comes to mind when I think about `position: sticky`. You know when you scroll in the contacts list and the first letter is sticked to the top, to let you know you are viewing that particular letter's contacts?

We used JavaScript to emulate that, but this is the approach taken by CSS to allow it natively.

CSS Transforms

CSS transforms allow you to modify the coordinate space of elements, enabling you to rotate, scale, skew, or translate elements. Transforms are hardware-accelerated in modern browsers, making them performant for animations.

Transforms let you manipulate elements in ways that were previously only possible with images or complex JavaScript. You can move, rotate, scale, and skew elements, all while maintaining their original space in the document flow. This means other elements won't reflow when you transform something, which is great for performance and prevents layout jumping.

2D Transforms

translate()

Move an element from its current position:

The `translate()` function moves an element from its current position without affecting the layout of other elements. Think of it like picking up a photo and sliding it to a new position on a table - the space where it was remains reserved.

```
/* Move 50px right and 100px down */
.element {
  transform: translate(50px, 100px);
}
```

This moves the element 50 pixels to the right and 100 pixels down from where it would normally be. Positive X values move right, negative left. Positive Y values move down, negative up.

```
/* Use percentages relative to element's size */
.centered {
  transform: translate(-50%, -50%);
}
```

This is a classic centering trick. When combined with `position: absolute; top: 50%; left: 50%;`, the translate pulls the element back by half its own width and height, perfectly centering it. The percentages refer to the element's own dimensions, not its parent's.

```
/* Single axis translation */
.slide-right {
  transform: translateX(100px);
}

.slide-up {
```

```
  transform: translateY(-50px);
}
```

Sometimes you only want to move in one direction. `translateX()` moves horizontally, `translateY()` moves vertically. These are shortcuts that make your intent clearer.

rotate()

Rotate an element around its center point:

```
/* Rotate 45 degrees clockwise */
.rotated {
  transform: rotate(45deg);
}

/* Negative values rotate counter-clockwise */
.counter-rotated {
  transform: rotate(-30deg);
}

/* Full rotation */
.spinner {
  transform: rotate(360deg);
}
```

scale()

Resize an element:

```
/* Scale to 1.5x size */
.larger {
  transform: scale(1.5);
}

/* Scale X and Y differently */
.stretched {
  transform: scale(2, 0.5);
}

/* Scale only horizontally or vertically */
.wide {
  transform: scaleX(1.5);
}

.tall {
  transform: scaleY(2);
}
```

skew()

Skew an element along the X and Y axes:

```
/* Skew 20 degrees on X axis */
.skewed {
  transform: skew(20deg);
}

/* Skew both axes */
.double-skewed {
  transform: skew(20deg, 10deg);
}

/* Individual axis skewing */
.skew-horizontal {
  transform: skewX(20deg);
}
```

Combining Transforms

You can apply multiple transforms by space-separating them:

```
.complex {
  transform: rotate(45deg) translate(100px) scale(1.5);
}

/* Order matters! */
.order-matters-1 {
  transform: translate(100px) rotate(45deg);
}

.order-matters-2 {
  transform: rotate(45deg) translate(100px);
  /* Different result! */
}
```

Transform Origin

Change the point around which transforms are applied:

```
/* Default is center */
.from-center {
  transform-origin: center;
  transform: rotate(45deg);
}
```

```css
/* Rotate from top-left corner */
.from-corner {
  transform-origin: top left;
  transform: rotate(45deg);
}

/* Use percentages or pixels */
.custom-origin {
  transform-origin: 25% 75%;
  transform: scale(1.5);
}
```

3D Transforms

CSS also supports 3D transforms for creating depth effects:

```css
/* Enable 3D space on parent */
.container {
  perspective: 1000px;
}

/* 3D rotation */
.flip-x {
  transform: rotateX(180deg);
}

.flip-y {
  transform: rotateY(180deg);
}

.rotate-3d {
  transform: rotateZ(45deg); /* Same as rotate() */
}

/* 3D translation */
.move-forward {
  transform: translateZ(100px);
}

/* Combined 3D transforms */
.card-flip {
  transform: rotateY(180deg) translateZ(100px);
  transform-style: preserve-3d;
}
```

Practical Examples

```css
/* Hover effect */
.button {
  transition: transform 0.3s;
}

.button:hover {
  transform: translateY(-5px) scale(1.05);
}

/* Card flip effect */
.card {
  transform-style: preserve-3d;
  transition: transform 0.6s;
}

.card.flipped {
  transform: rotateY(180deg);
}

/* Centering with transform */
.absolute-center {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}
```

CSS Transitions

Transitions allow property changes to occur smoothly over a specified duration, creating animation effects without JavaScript.

Basic Syntax

```css
/* Shorthand: property duration timing-function delay */
.element {
  transition: opacity 0.3s ease-in-out 0s;
}

/* Transition multiple properties */
.box {
  transition: width 0.3s, height 0.3s, background-color 0.5s;
}

/* Transition all properties */
.smooth {
```

```
  transition: all 0.3s ease;
}
```

Transition Properties

transition-property

Specify which properties to transition:

```
.element {
  transition-property: background-color, transform;
  transition-duration: 0.3s;
}

/* Common transitionable properties:
   - opacity
   - transform
   - background-color
   - width, height
   - padding, margin
   - border
   - color
   - box-shadow
*/
```

transition-duration

Set how long the transition takes:

```
.fast {
  transition-duration: 0.1s;
}

.medium {
  transition-duration: 0.3s;
}

.slow {
  transition-duration: 1s;
}
```

transition-timing-function

Control the transition's acceleration:

```
/* Predefined functions */
.ease {
  transition-timing-function: ease; /* Default - slow start, fast middle,
slow end */
}

.linear {
  transition-timing-function: linear; /* Constant speed */
}

.ease-in {
  transition-timing-function: ease-in; /* Slow start */
}

.ease-out {
  transition-timing-function: ease-out; /* Slow end */
}

.ease-in-out {
  transition-timing-function: ease-in-out; /* Slow start and end */
}

/* Custom cubic-bezier */
.custom {
  transition-timing-function: cubic-bezier(0.68, -0.55, 0.265, 1.55);
}

/* Steps for discrete animations */
.steps {
  transition-timing-function: steps(4, end);
}
```

transition-delay

Delay before the transition starts:

```
.delayed {
  transition: opacity 0.3s ease 0.5s; /* Wait 0.5s before starting */
}

/* Staggered animations */
.item:nth-child(1) { transition-delay: 0s; }
.item:nth-child(2) { transition-delay: 0.1s; }
.item:nth-child(3) { transition-delay: 0.2s; }
```

Practical Examples

```css
/* Button hover effect */
.button {
  background: #007bff;
  color: white;
  padding: 10px 20px;
  transition: all 0.3s ease;
}

.button:hover {
  background: #0056b3;
  transform: translateY(-2px);
  box-shadow: 0 4px 8px rgba(0,0,0,0.2);
}

/* Menu slide-in */
.menu {
  transform: translateX(-100%);
  transition: transform 0.3s ease-out;
}

.menu.open {
  transform: translateX(0);
}

/* Smooth color change */
.theme-switch {
  background: var(--bg-color);
  color: var(--text-color);
  transition: background-color 0.5s, color 0.5s;
}

/* Accordion expand */
.accordion-content {
  max-height: 0;
  overflow: hidden;
  transition: max-height 0.3s ease-out;
}

.accordion-content.open {
  max-height: 500px; /* Or use JavaScript for exact height */
}
```

CSS Animations

CSS animations allow you to create complex, multi-step animations with precise control over timing and behavior.

While transitions are great for simple changes between two states (like hover effects), animations let you create complex sequences with multiple steps. Think of animations as choreographed performances where you control every movement, while transitions are simple A-to-B movements. Animations can loop, reverse, pause, and include as many steps as you need.

Defining Keyframes

Animations are defined using `@keyframes`:

Keyframes are like a storyboard for your animation. You define what the element should look like at different points in time, and CSS smoothly animates between these points. The name you give to your `@keyframes` rule is how you'll reference it later.

```
/* Simple two-step animation */
@keyframes fadeIn {
  from {
    opacity: 0;
  }
  to {
    opacity: 1;
  }
}
```

This creates a simple fade-in effect. The element starts completely transparent (`from` or 0%) and ends completely opaque (`to` or 100%). CSS automatically calculates all the in-between states.

```
/* Or using percentages */
@keyframes fadeIn {
  0% {
    opacity: 0;
  }
  100% {
    opacity: 1;
  }
}
```

Using percentages gives you more control. This is identical to the previous example, but percentages let you add more steps...

```
/* Multi-step animation */
@keyframes slideBounce {
  0% {
    transform: translateX(0);
  }
```

```
  50% {
    transform: translateX(100px);
  }
  75% {
    transform: translateX(80px);
  }
  100% {
    transform: translateX(100px);
  }
}
```

This creates a sliding animation with a bounce effect. The element moves to 100px at the halfway point, then pulls back slightly to 80px at 75%, before settling at 100px. This creates a natural "overshoot and settle" effect that feels more organic than linear movement.

Applying Animations

Use the `animation` property to apply keyframes:

```
/* Shorthand: name duration timing-function delay iteration-count
direction fill-mode play-state */
.element {
  animation: fadeIn 1s ease-in-out 0s 1 normal forwards running;
}

/* Simple usage */
.fade {
  animation: fadeIn 0.5s;
}

/* Multiple animations */
.complex {
  animation:
    slideIn 0.5s ease-out,
    fadeIn 0.3s ease-in;
}
```

Animation Properties

animation-name

Reference the keyframes:

```
.animated {
  animation-name: slideIn;
}
```

animation-duration

How long one cycle takes:

```
.animated {
  animation-duration: 2s;
}
```

animation-timing-function

Same as transition timing functions:

```
.bounce {
  animation-timing-function: cubic-bezier(0.68, -0.55, 0.265, 1.55);
}
```

animation-delay

Wait before starting:

```
.delayed {
  animation-delay: 1s;
}
```

animation-iteration-count

How many times to repeat:

```
.once {
  animation-iteration-count: 1;
}

.three-times {
  animation-iteration-count: 3;
}

.infinite {
  animation-iteration-count: infinite;
}
```

animation-direction

Direction of the animation:

```
.normal {
  animation-direction: normal; /* Default */
}

.reverse {
  animation-direction: reverse;
}

.alternate {
  animation-direction: alternate; /* Back and forth */
}

.alternate-reverse {
  animation-direction: alternate-reverse;
}
```

animation-fill-mode

How to apply styles before/after animation:

```
.forwards {
  animation-fill-mode: forwards; /* Keep final state */
}

.backwards {
  animation-fill-mode: backwards; /* Apply initial state during delay */
}

.both {
  animation-fill-mode: both; /* Both forwards and backwards */
}
```

animation-play-state

Control playback:

```
.paused {
  animation-play-state: paused;
}

.running {
  animation-play-state: running;
}

/* Pause on hover */
.spinner:hover {
```

```
  animation-play-state: paused;
}
```

Practical Animation Examples

```
/* Loading spinner */
@keyframes spin {
  to {
    transform: rotate(360deg);
  }
}

.spinner {
  width: 50px;
  height: 50px;
  border: 5px solid #f3f3f3;
  border-top: 5px solid #3498db;
  border-radius: 50%;
  animation: spin 1s linear infinite;
}

/* Pulse effect */
@keyframes pulse {
  0%, 100% {
    transform: scale(1);
    opacity: 1;
  }
  50% {
    transform: scale(1.1);
    opacity: 0.7;
  }
}

.pulse {
  animation: pulse 2s ease-in-out infinite;
}

/* Typing effect */
@keyframes typing {
  from {
    width: 0;
  }
  to {
    width: 100%;
  }
}

@keyframes blink {
```

```
  50% {
    border-color: transparent;
  }
}

.typewriter {
  overflow: hidden;
  border-right: 3px solid black;
  white-space: nowrap;
  animation:
    typing 3s steps(30, end),
    blink 0.5s step-end infinite;
}

/* Bounce effect */
@keyframes bounce {
  0%, 100% {
    transform: translateY(0);
  }
  50% {
    transform: translateY(-20px);
  }
}

.bouncing {
  animation: bounce 1s ease-in-out infinite;
}

/* Fade in and slide up */
@keyframes fadeInUp {
  from {
    opacity: 0;
    transform: translateY(30px);
  }
  to {
    opacity: 1;
    transform: translateY(0);
  }
}

.fade-in-up {
  animation: fadeInUp 0.5s ease-out;
}
```

Performance Tips

1. **Use transform and opacity** - These properties are GPU-accelerated
2. **Avoid animating layout properties** - width, height, padding cause reflows

3. **Use will-change sparingly** - Hints to browser about upcoming changes
4. **Consider reduced motion** - Respect user preferences

```
/* Performance optimization */
.will-animate {
  will-change: transform, opacity;
}

/* Respect user preferences */
@media (prefers-reduced-motion: reduce) {
  * {
    animation-duration: 0.01ms !important;
    animation-iteration-count: 1 !important;
    transition-duration: 0.01ms !important;
  }
}
```

Float for Text Wrapping

The `float` property has one primary modern use case: wrapping text around images or other elements, like you see in magazines and newspapers.

Important: Float should NOT be used for page layouts. In the past, before Flexbox and Grid existed, developers used floats for creating columns and layouts, which led to many hacks and complications. Today, we use Grid and Flexbox for layouts (which we cover in other sections). Float is now used exclusively for its original purpose: making text flow around elements.

The `float` property supports 3 values:

- `left`
- `right`
- `none` (the default)

Say we have a box which contains a paragraph with some text, and the paragraph also contains an image.

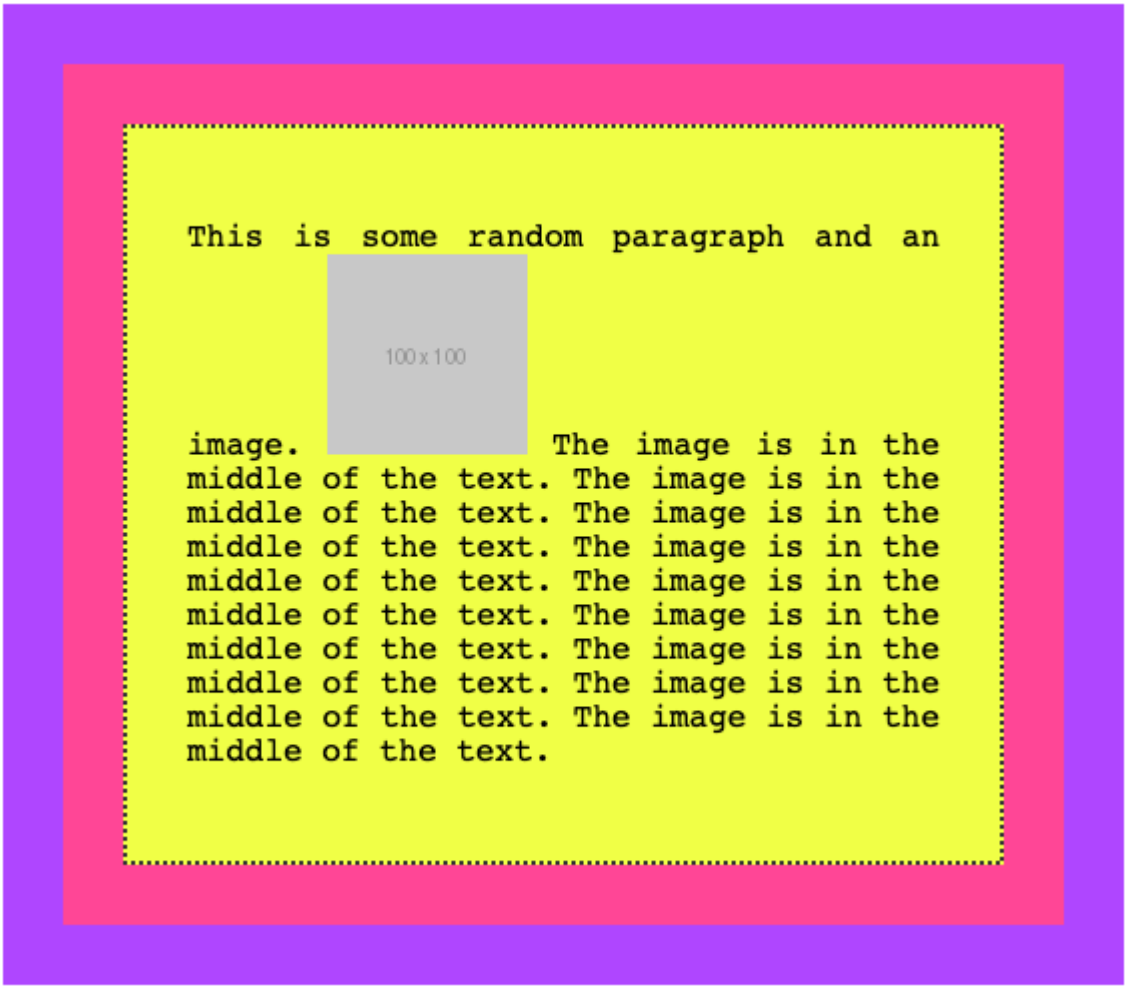
Here's some code:

```
<div class="parent">
  <div class="child">
    <div class="box">
      <p>
        This is some random paragraph and an image.
        <img src="https://via.placeholder.com/100x100"/>
        The image is in the middle of the text. The image is in the middle
```

```
of the text. The image is in the middle of the text. The image is in the
middle of the text. The image is in the middle of the text. The image is
in the middle of the text. The image is in the middle of the text. The
image is in the middle of the text. The image is in the middle of the
text.
      </p>
    </div>
  </div>
</div>
```

```
.parent {
  background-color: #af47ff;
  padding: 30px;
  width: 500px
}
.child {
  background-color: #ff4797;
  padding: 30px
}
.box {
  background-color: #f3ff47;
  padding: 30px;
  border: 2px solid #333;
  border-style: dotted;
  font-family: courier;
  text-align: justify;
  font-size: 1rem
}
```

and the visual appearance:

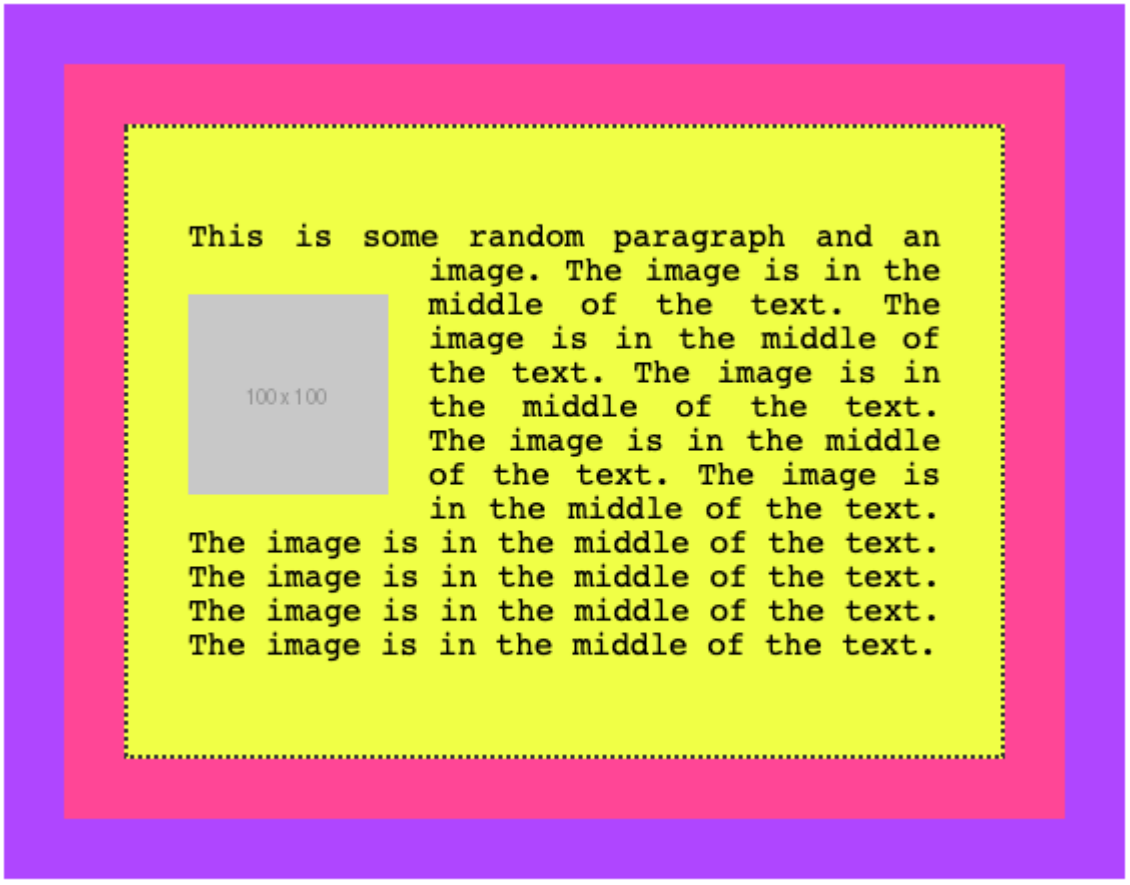


As you can see, the normal flow by default considers the image inline, and makes space for it in the line itself.

If we add `float: left` to the image, and some padding:

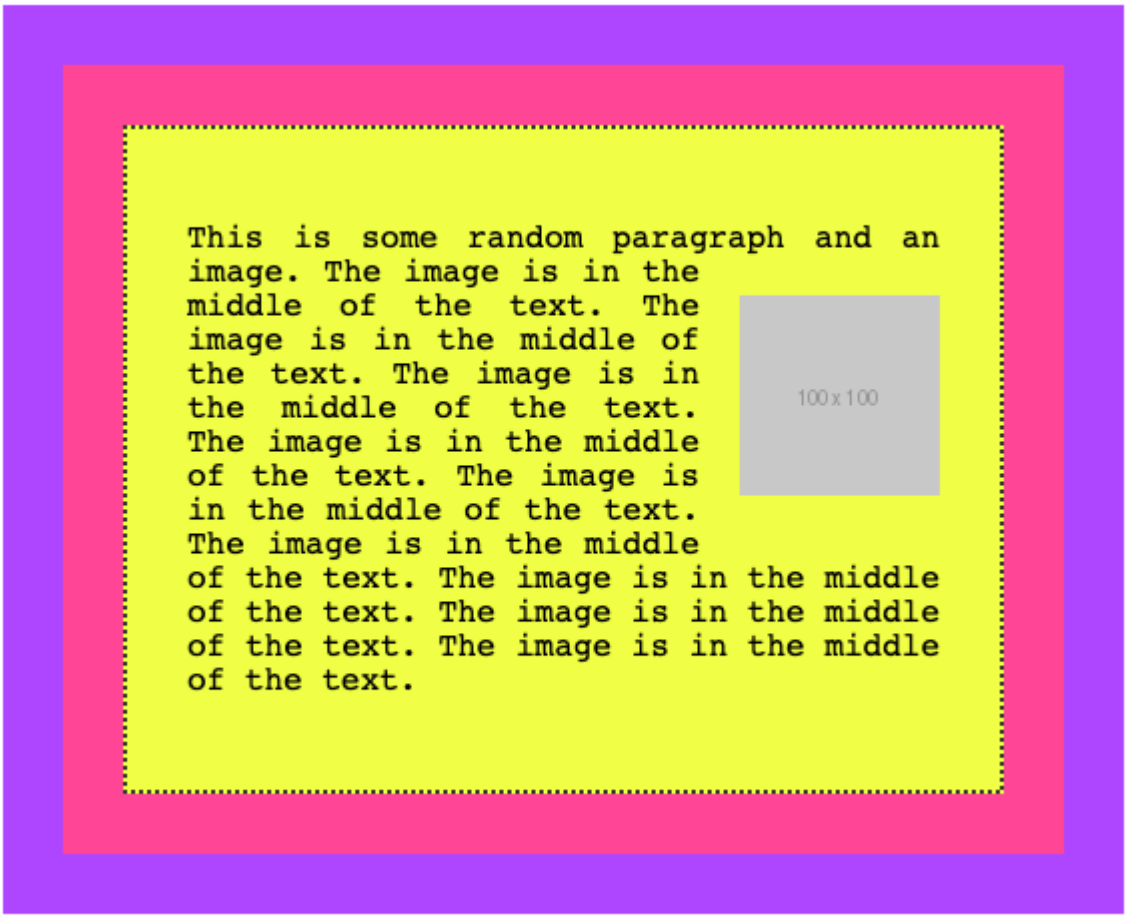
```
img {
  float: left;
  padding: 20px 20px 0px 0px;
}
```

this is the result:



and this is what we get by applying a float: right, adjusting the padding accordingly:

```
img {
  float: right;
  padding: 20px 0px 20px 20px;
}
```



A floated element is removed from the normal flow of the page, and the other content flows around it.

See the example on Codepen

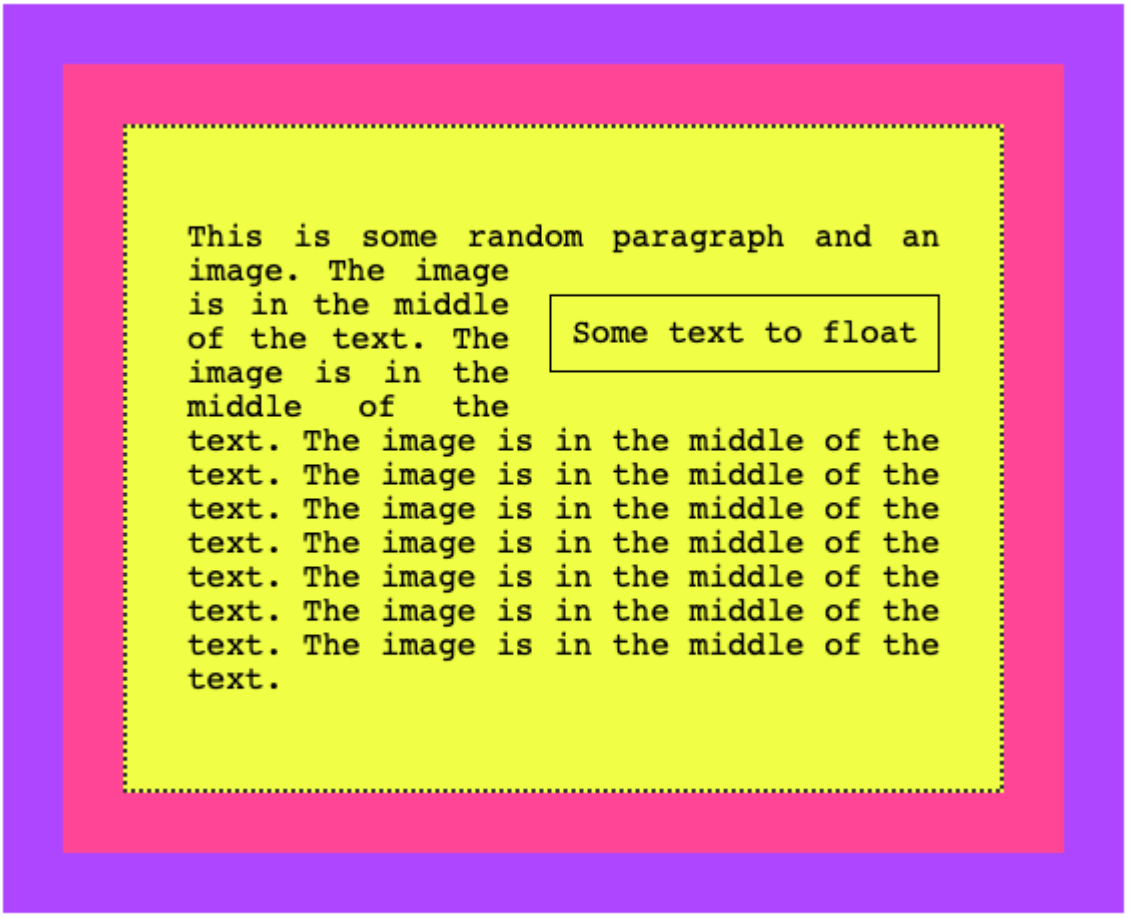
You are not limited to floating images, too. Here we switch the image with a `span` element:

```
<div class="parent">
  <div class="child">
    <div class="box">
      <p>
        This is some random paragraph and an image.
        <span>Some text to float</span> The image is in the middle of the
text.
        The image is in the middle of the text. The image is in the middle
of
        the text. The image is in the middle of the text. The image is in
the
        middle of the text. The image is in the middle of the text. The
image is
        in the middle of the text. The image is in the middle of the text.
The
        image is in the middle of the text.
      </p>
    </div>
```

```
  </div>
</div>
```

```
span {
  float: right;
  margin: 20px 0px 20px 20px;
  padding: 10px;
  border: 1px solid black;
}
```

and this is the result:

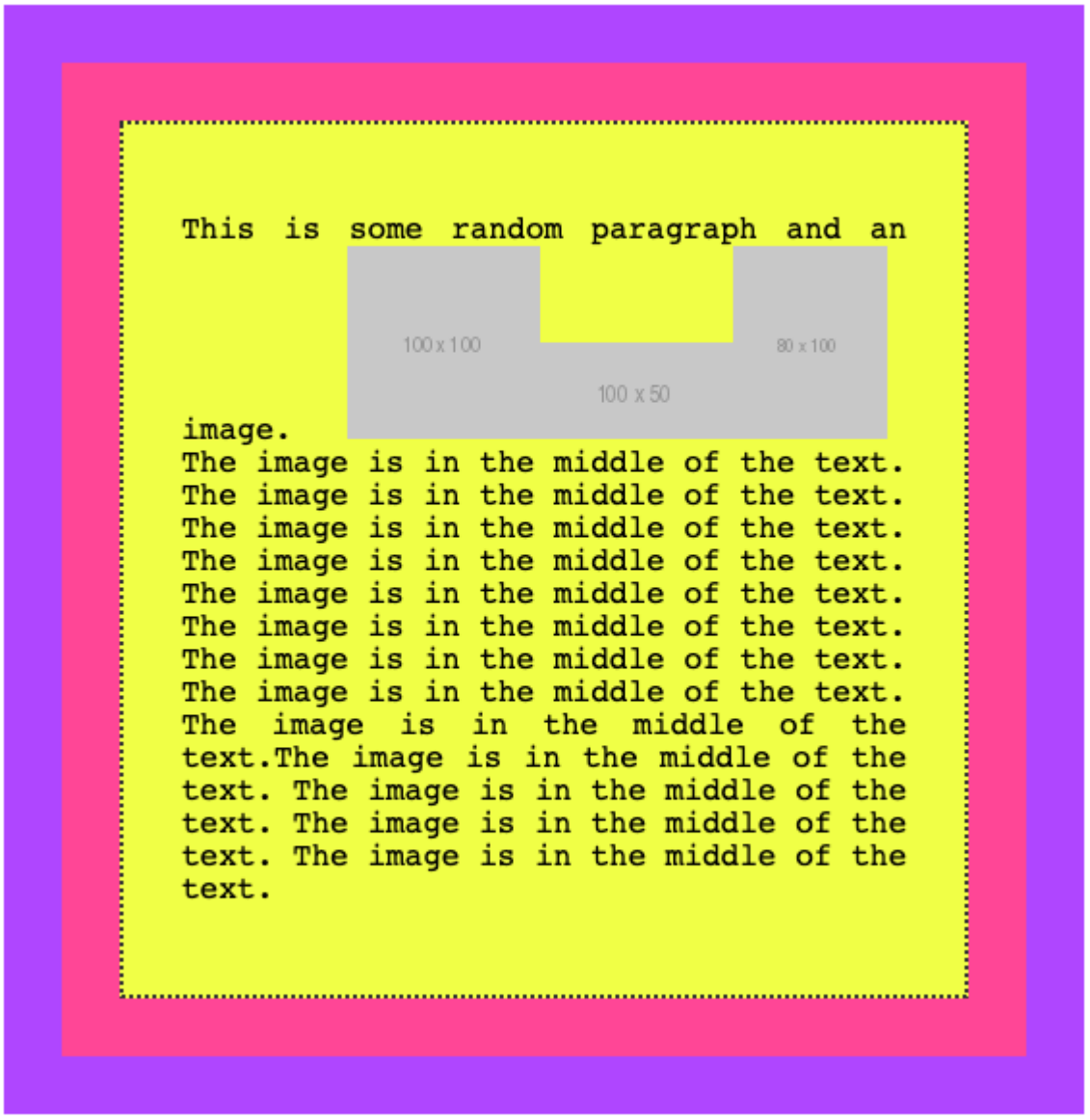


Clearing

What happens when you float more than one element?

If when floated they find another floated image, by default they are stacked up one next to the other, horizontally. Until there is no room, and they will start being stacked on a new line.

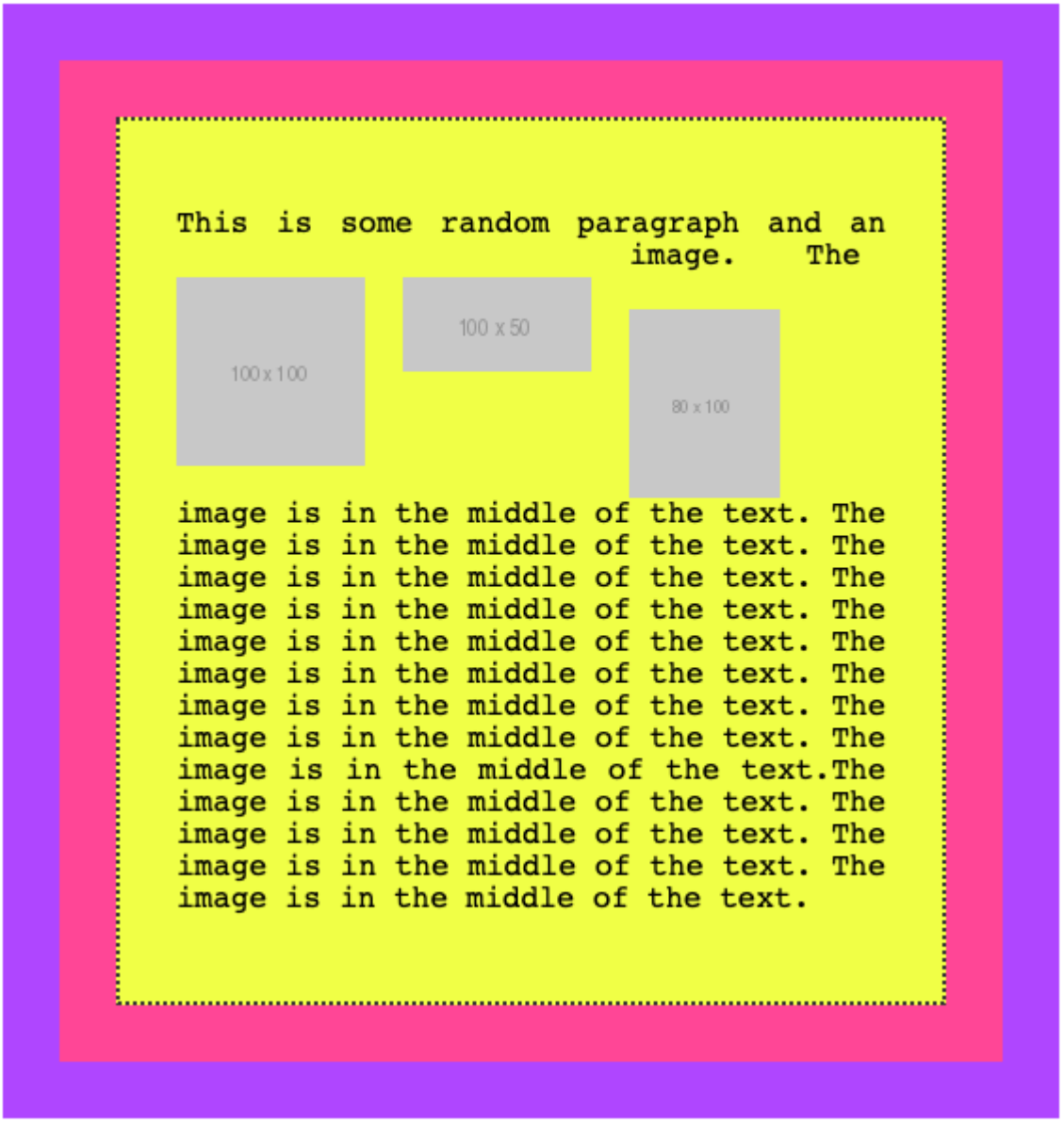
Say we had 3 inline images inside a `p` tag:



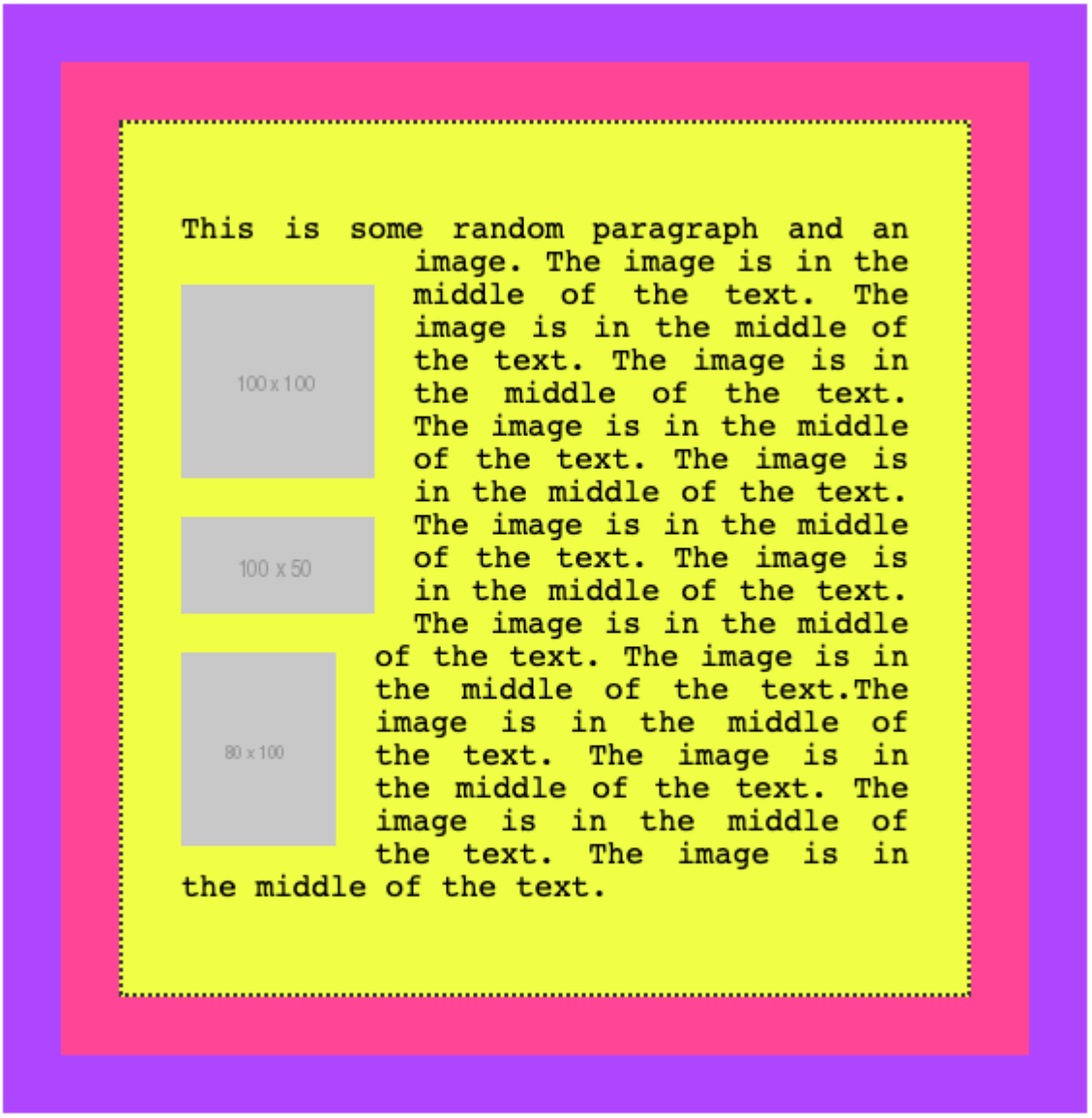
If we add `float: left` to those images:

```
img {
  float: left;
  padding: 20px 20px 0px 0px;
}
```

this is what we'll have:



if you add `clear: left` to images, those are going to be stacked vertically rather than horizontally:



The `clear` property helps manage how elements interact with floated elements that come before them. While float-based layouts have been replaced by Flexbox and Grid for most use cases, understanding `clear` is still useful when working with floated images or legacy code.

z-index

When we talked about positioning, I mentioned that you can use the `z-index` property to control the Z axis positioning of elements.

It's very useful when you have multiple elements that overlap each other, and you need to decide which one is visible, as nearer to the user, and which one(s) should be hidden behind it.

This property takes a number (without decimals) and uses that number to calculate which elements appear nearer to the user, in the Z axis.

The higher the z-index value, the more an element is positioned nearer to the user.

When deciding which element should be visible and which one should be positioned behind it, the browser does a calculation on the z-index value.

The default value is `auto`, a special keyword. Using `auto`, the Z axis order is determined by the position of the HTML element in the page - the last sibling appears first, as it's defined last.

By default elements have the `static` value for the `position` property. In this case, the `z-index` property does not make any difference - it must be set to `absolute`, `relative` or `fixed` to work.

Example:

```
.my-first-div {
  position: absolute;
  top: 0;
  left: 0;
  width: 600px;
  height: 600px;
  z-index: 10;
}

.my-second-div {
  position: absolute;
  top: 0;
  left: 0;
  width: 500px;
  height: 500px;
  z-index: 20;
}
```

The element with class `.my-second-div` will be displayed, and behind it `.my-first-div`.

Here we used 10 and 20, but you can use any number. Negative numbers too. It's common to pick non-consecutive numbers, so you can position elements in the middle. If you use consecutive numbers instead, you would need to re-calculate the z-index of each element involved in the positioning.

CSS Filters and Effects

CSS filters allow you to apply graphical effects like blur, brightness, and color shifting to elements. These effects are hardware-accelerated and can create stunning visual designs.

Filters work like Instagram effects for your web elements. You can blur images, adjust brightness and contrast, convert to black and white, and much more - all with CSS. These effects apply to the entire element including its children, and they're processed by the GPU

for smooth performance. Before CSS filters, these effects required image editing software or canvas manipulation with JavaScript.

Filter Functions

blur()

Apply a Gaussian blur:

The blur filter creates a soft, out-of-focus effect. The value you provide (in pixels) determines how much the element is blurred - higher values create more blur. This is perfect for creating depth, drawing attention to other elements, or creating frosted glass effects.

```
.blurred {
  filter: blur(5px);
}
```

This applies a medium blur, enough to make text unreadable but shapes still recognizable. Great for background images behind text overlays.

```
/* Subtle blur */
.soft {
  filter: blur(1px);
}
```

A 1-pixel blur creates a subtle softening effect, useful for reducing the harshness of images or creating a dreamy atmosphere without losing detail.

```
/* Heavy blur for backgrounds */
.background-blur {
  filter: blur(20px);
}
```

Heavy blur like this completely obscures details, perfect for creating abstract backgrounds from photos or for privacy screens where you want to hide sensitive information while maintaining color and general shape.

brightness()

Adjust the brightness (1 is normal):

```
.bright {
  filter: brightness(1.5); /* 150% brightness */
}
```

```
.dim {
  filter: brightness(0.5); /* 50% brightness */
}

.dark {
  filter: brightness(0);
}
```

contrast()

Adjust the contrast (1 is normal):

```
.high-contrast {
  filter: contrast(2);
}

.low-contrast {
  filter: contrast(0.5);
}
```

grayscale()

Convert to grayscale (0 to 1):

```
.grayscale {
  filter: grayscale(1); /* Fully grayscale */
}

.partial-grayscale {
  filter: grayscale(0.5); /* 50% grayscale */
}

/* Common hover effect */
img {
  filter: grayscale(1);
  transition: filter 0.3s;
}

img:hover {
  filter: grayscale(0);
}
```

sepia()

Apply a sepia tone:

```css
.vintage {
  filter: sepia(1);
}

.slight-sepia {
  filter: sepia(0.3);
}
```

saturate()

Adjust color saturation:

```css
.vibrant {
  filter: saturate(2); /* 200% saturation */
}

.desaturated {
  filter: saturate(0.5);
}

.no-saturation {
  filter: saturate(0); /* Same as grayscale(1) */
}
```

hue-rotate()

Rotate the hue of colors:

```css
.hue-shift {
  filter: hue-rotate(90deg);
}

.inverted-colors {
  filter: hue-rotate(180deg);
}

/* Animated color shift */
@keyframes rainbow {
  to {
    filter: hue-rotate(360deg);
  }
}

.rainbow {
  animation: rainbow 3s linear infinite;
}
```

invert()

Invert colors:

```
.inverted {
  filter: invert(1); /* Fully inverted */
}

.partial-invert {
  filter: invert(0.5);
}
```

opacity()

Adjust opacity (similar to opacity property):

```
.transparent {
  filter: opacity(0.5);
}
```

drop-shadow()

Apply a drop shadow (follows alpha channel):

```
.shadow {
  filter: drop-shadow(5px 5px 10px rgba(0,0,0,0.5));
}

/* Multiple shadows */
.multi-shadow {
  filter:
    drop-shadow(3px 3px 5px rgba(0,0,0,0.3))
    drop-shadow(-3px -3px 5px rgba(255,255,255,0.3));
}

/* Colored shadow */
.color-shadow {
  filter: drop-shadow(0 10px 20px rgba(0,100,200,0.5));
}
```

Combining Filters

Multiple filters can be combined:

```
.complex-filter {
  filter:
```

```
    contrast(1.2)
    brightness(1.1)
    blur(1px)
    drop-shadow(5px 5px 10px rgba(0,0,0,0.3));
}

/* Instagram-like filters */
.instagram-ludwig {
  filter:
    brightness(1.05)
    saturate(0.75)
    contrast(1.15);
}

.instagram-moon {
  filter:
    grayscale(1)
    contrast(1.1)
    brightness(1.1);
}
```

Backdrop Filter

Apply filters to the area behind an element:

```
/* Frosted glass effect */
.frosted-glass {
  background: rgba(255, 255, 255, 0.2);
  backdrop-filter: blur(10px);
  border: 1px solid rgba(255, 255, 255, 0.3);
}

/* Dark glassmorphism */
.glassmorphism {
  background: rgba(0, 0, 0, 0.5);
  backdrop-filter: blur(15px) saturate(1.5);
  border: 1px solid rgba(255, 255, 255, 0.1);
}

/* Modal overlay */
.modal-backdrop {
  backdrop-filter: blur(5px) brightness(0.7);
}
```

Mix Blend Mode

Control how elements blend with their background:

```css
.multiply {
  mix-blend-mode: multiply;
}

.screen {
  mix-blend-mode: screen;
}

.overlay {
  mix-blend-mode: overlay;
}

/* Common blend modes:
   - normal
   - multiply (darken)
   - screen (lighten)
   - overlay
   - darken
   - lighten
   - color-dodge
   - color-burn
   - hard-light
   - soft-light
   - difference
   - exclusion
   - hue
   - saturation
   - color
   - luminosity
*/

/* Text knockout effect */
.knockout-text {
  font-size: 100px;
  font-weight: bold;
  mix-blend-mode: multiply;
  background: white;
  color: black;
}

/* Creative overlays */
.color-overlay {
  background: linear-gradient(45deg, #ff0066, #00ff66);
  mix-blend-mode: overlay;
}
```

Clip Path

Create custom shapes by clipping elements:

```
/* Basic shapes */
.circle {
  clip-path: circle(50%);
}

.ellipse {
  clip-path: ellipse(50% 30%);
}

.triangle {
  clip-path: polygon(50% 0%, 0% 100%, 100% 100%);
}

.hexagon {
  clip-path: polygon(25% 0%, 75% 0%, 100% 50%, 75% 100%, 25% 100%, 0%
50%);
}

/* Animated clip path */
@keyframes morph {
  0% {
    clip-path: circle(40%);
  }
  50% {
    clip-path: polygon(50% 0%, 100% 50%, 50% 100%, 0% 50%);
  }
  100% {
    clip-path: circle(40%);
  }
}

.morphing {
  animation: morph 3s ease-in-out infinite;
}

/* Diagonal cut */
.diagonal {
  clip-path: polygon(0 0, 100% 0, 100% 80%, 0 100%);
}
```

Practical Examples

```
/* Card with glossy effect */
.glossy-card {
  background: linear-gradient(135deg, rgba(255,255,255,0.1),
rgba(255,255,255,0));
```

```
  backdrop-filter: blur(10px);
  border: 1px solid rgba(255,255,255,0.2);
  box-shadow: 0 8px 32px rgba(0,0,0,0.1);
}

/* Image hover effects */
.image-hover {
  filter: brightness(0.8) contrast(1.2);
  transition: filter 0.3s;
}

.image-hover:hover {
  filter: brightness(1) contrast(1) saturate(1.2);
}

/* Loading skeleton */
.skeleton {
  background: linear-gradient(90deg, #f0f0f0 25%, #e0e0e0 50%, #f0f0f0
75%);
  background-size: 200% 100%;
  animation: loading 1.5s infinite;
  filter: blur(0.5px);
}

@keyframes loading {
  to {
    background-position: -200% 0;
  }
}

/* Disabled state */
.disabled {
  filter: grayscale(1) opacity(0.6);
  pointer-events: none;
}
```

Lists

Lists are a very important part of many web pages.

CSS can style them using several properties.

`list-style-type` is used to set a predefined marker to be used by the list:

```
li {
  list-style-type: square;
}
```

We have lots of possible values, which you can see here https://developer.mozilla.org/en-US/docs/Web/CSS/list-style-type with examples of their appearance. Some of the most popular ones are `disc`, `circle`, `square` and `none`.

`list-style-image` is used to use a custom marker when a predefined marker is not appropriate:

```
li {
  list-style-image: url(list-image.png);
}
```

`list-style-position` lets you add the marker `outside` (the default) or `inside` of the list content, in the flow of the page rather than outside of it

```
li {
  list-style-position: inside;
}
```

The `list-style` shorthand property lets us specify all those properties in the same line:

```
li {
  list-style: url(list-image.png) inside;
}
```

Error handling

CSS is resilient. When it finds an error, it does not act like JavaScript which packs up all its things and goes away altogether, terminating all the script execution after the error is found.

CSS tries very hard to do what you want.

If a line has an error, it skips it and jumps to the next line without any error.

If you forget the semicolon on one line:

```
p {
  font-size: 20px
  color: black;
  border: 1px solid black;
}
```

the line with the error AND the next one will **not** be applied, but the third rule will be successfully applied on the page. Basically, it scans all until it finds a semicolon, but when it reaches it, the rule is now `font-size: 20px color: black;`, which is invalid, so it skips it.

Sometimes it’s tricky to realize there is an error somewhere, and where that error is, because the browser won’t tell us.

This is why tools like CSS Lint exist.

Flexbox

Flexbox, also called Flexible Box Module, is one of the two modern layouts systems, along with CSS Grid.

Compared to CSS Grid, which we’ll talk about later (which is bi-dimensional), flexbox is a **one-dimensional layout model**. It will control the layout based on a row or on a column, but not together at the same time.

The main goal of flexbox is to allow items to fill the whole space offered by their container, depending on some rules you set.

Let's dive into flexbox and become a master of it in a very short time.

A flexbox layout is applied to a container, by setting

```
display: flex;
```

or

```
display: inline-flex;
```

the content **inside the container** will be aligned using flexbox.

The difference between `display: inline-flex;` and `display: flex;` is that `inline-flex` makes the container behave like an inline element while still using flexbox layout for its children, whereas `flex` makes the container behave like a block element.

Container properties

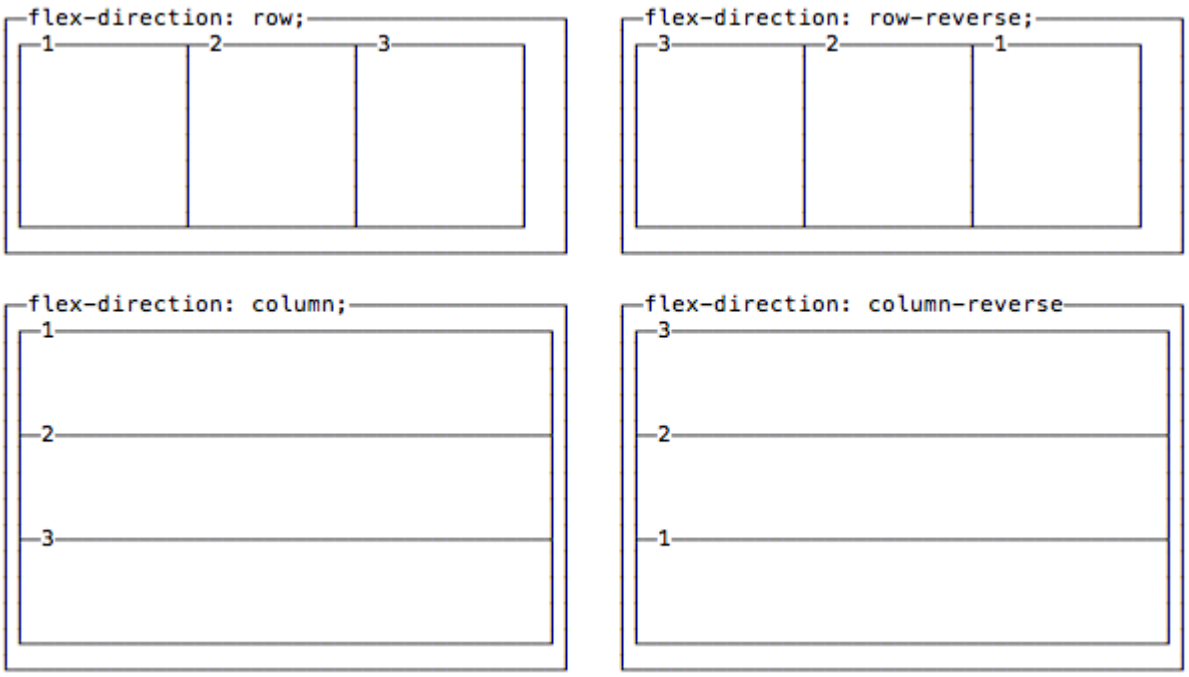
Some flexbox properties apply to the container, which sets the general rules for its items. They are

- `flex-direction`
- `justify-content`
- `align-items`
- `flex-wrap`
- `flex-flow`

Align rows or columns

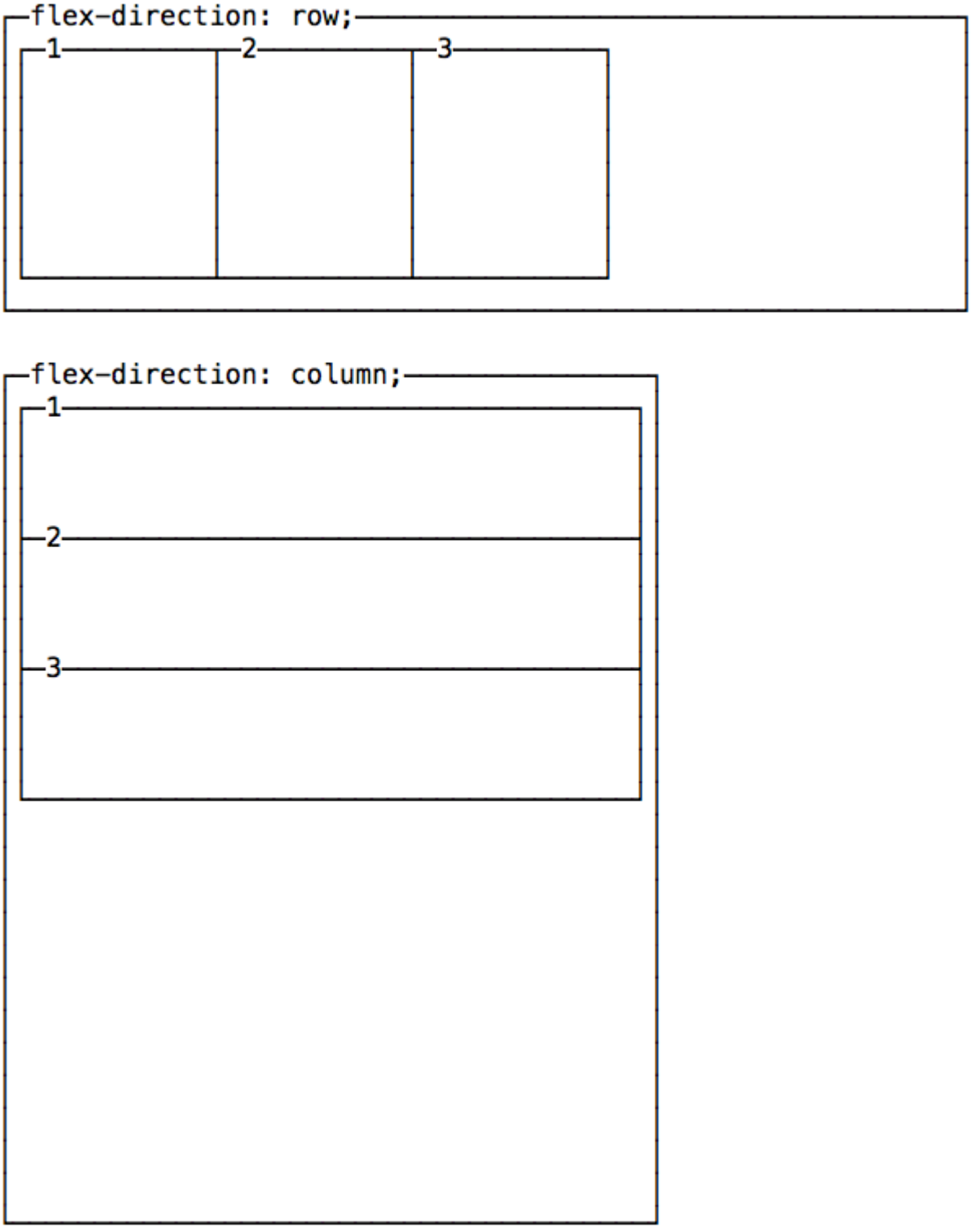
The first property we see, `flex-direction`, determines if the container should align its items as rows, or as columns:

- `flex-direction: row` places items as a **row**, in the text direction (left-to-right for western countries)
- `flex-direction: row-reverse` places items just like `row` but in the opposite direction
- `flex-direction: column` places items in a **column**, ordering top to bottom
- `flex-direction: column-reverse` places items in a column, just like `column` but in the opposite direction



Vertical and horizontal alignment

By default items start from the left if `flex-direction` is `row`, and from the top if `flex-direction` is `column`.



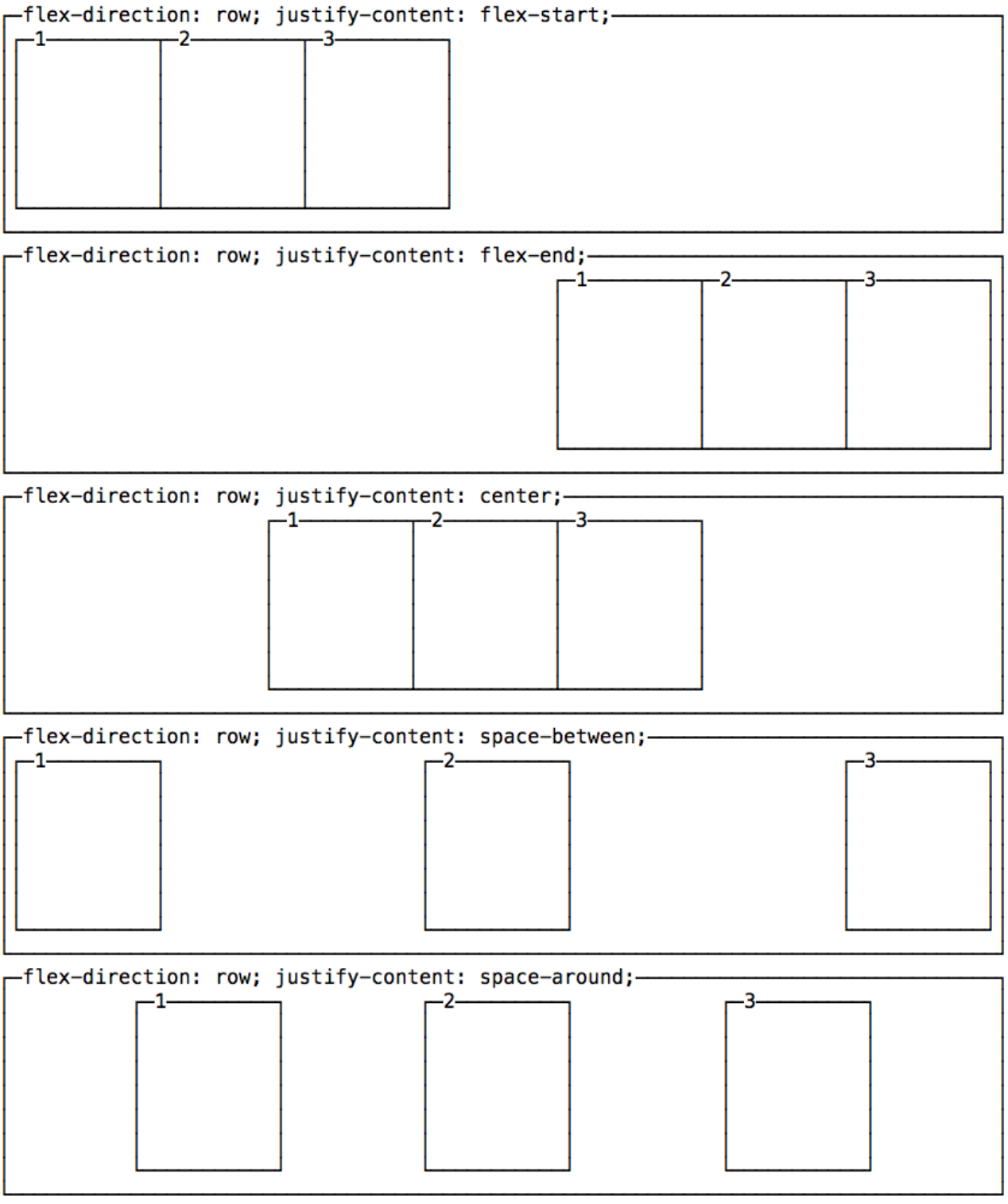
You can change this behavior using `justify-content` to change the horizontal alignment, and `align-items` to change the vertical alignment.

Change the horizontal alignment

`justify-content` has 5 possible values:

- `flex-start` : align to the left side of the container.
- `flex-end` : align to the right side of the container.
- `center` : align at the center of the container.

- `space-between` : display with equal spacing between them.
- `space-around` : display with equal spacing around them

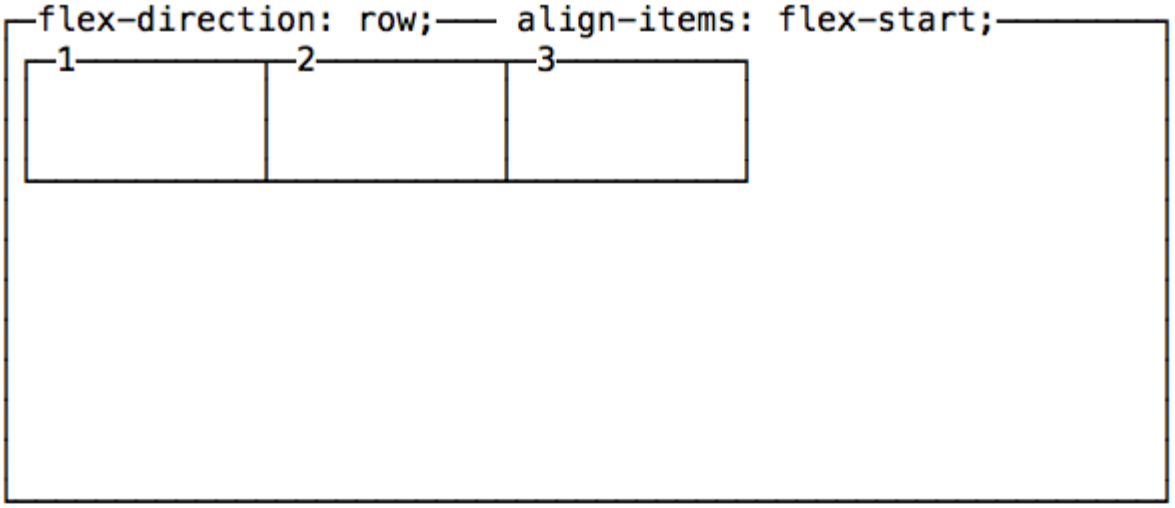


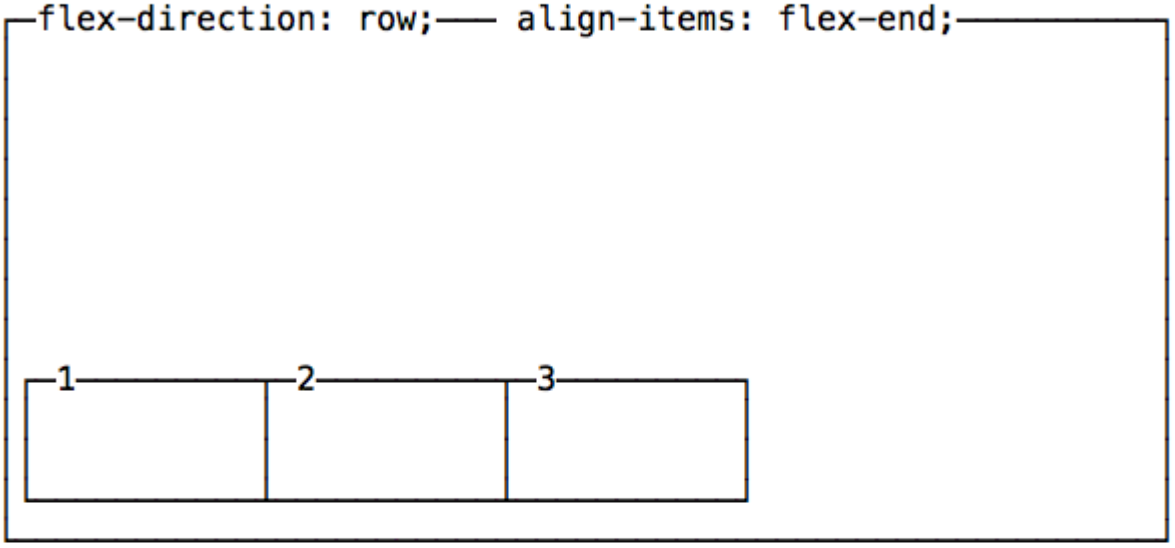
Change the vertical alignment

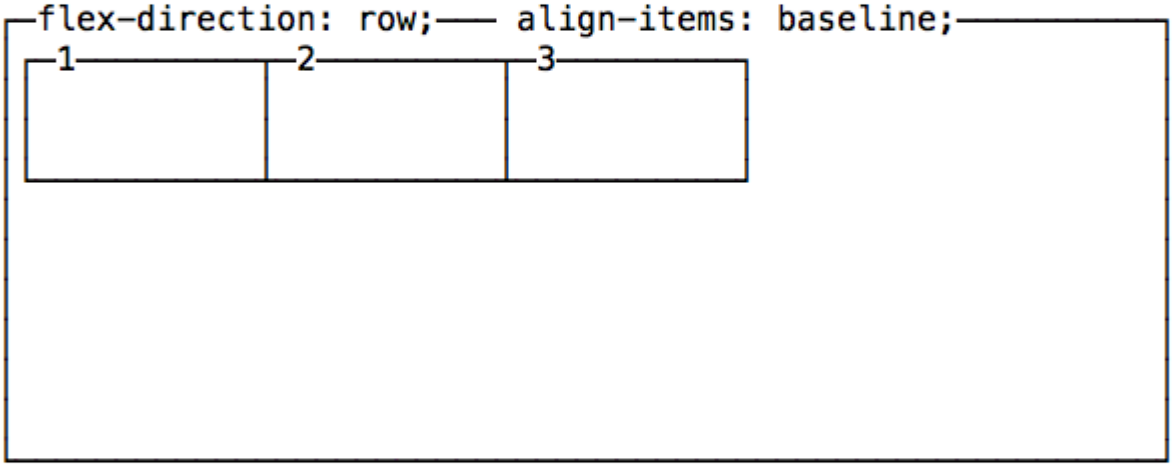
`align-items` has 5 possible values:

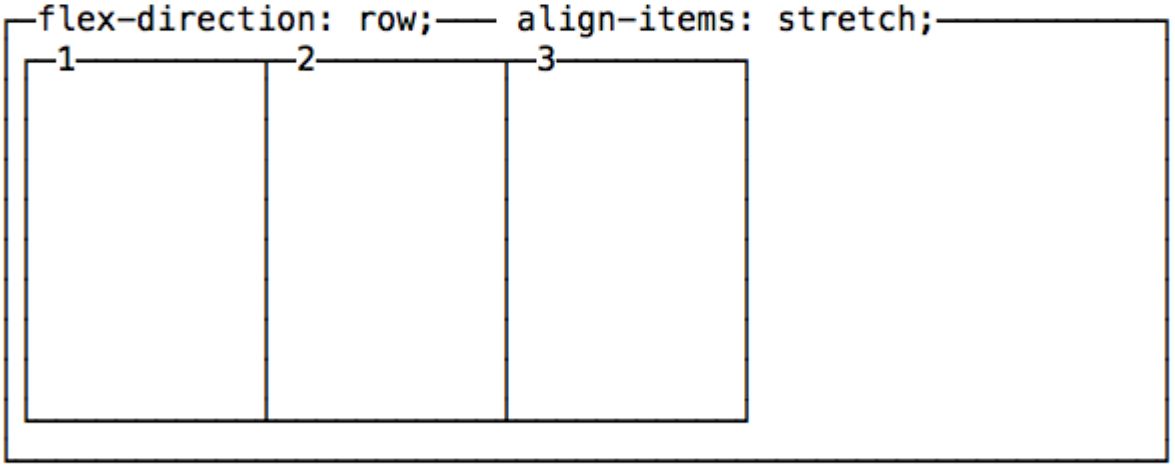
- `flex-start` : align to the top of the container.
- `flex-end` : align to the bottom of the container.
- `center` : align at the vertical center of the container.
- `baseline` : display at the baseline of the container.

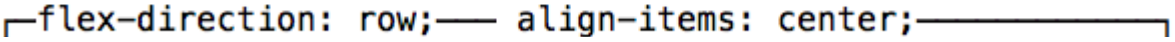
- `stretch`: items are stretched to fit the container.

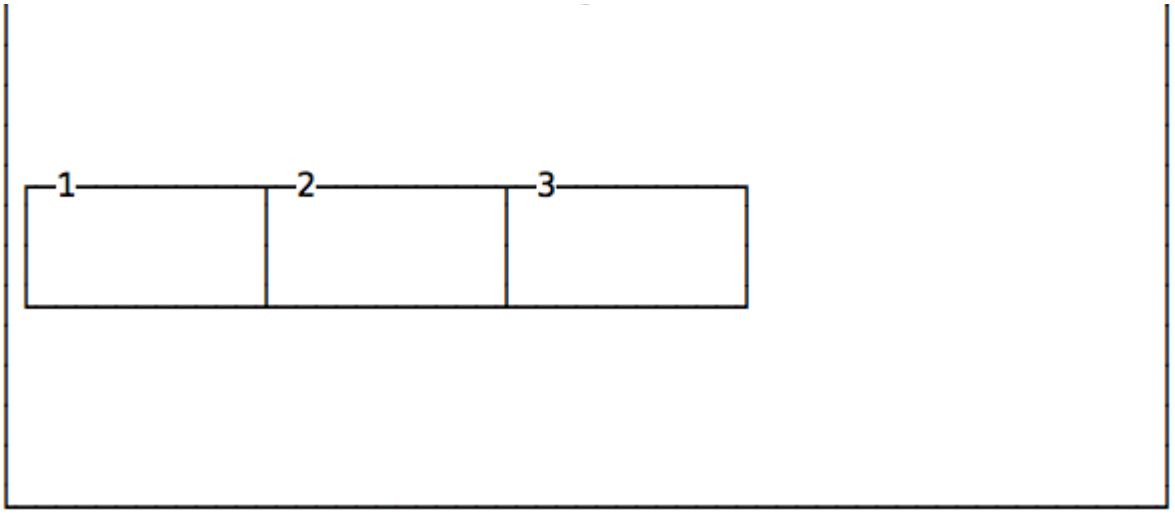

flex-direction: row; align-items: flex-start;
1
2
3


flex-direction: row; align-items: flex-end;
1
2
3


flex-direction: row; align-items: baseline;
1
2
3


flex-direction: row; align-items: stretch;
1
2
3


flex-direction: row; align-items: center;



A note on `baseline`

`baseline` looks similar to `flex-start` in this example, due to my boxes being too simple. Check out this Codepen to have a more useful example, which I forked from a Pen originally created by Martin Michálek. As you can see there, items dimensions are aligned.

Wrap

By default items in a flexbox container are kept on a single line, shrinking them to fit in the container.

To force the items to spread across multiple lines, use `flex-wrap: wrap`. This will distribute the items according to the order set in `flex-direction`. Use `flex-wrap: wrap-reverse` to reverse this order.

A shorthand property called `flex-flow` allows you to specify `flex-direction` and `flex-wrap` in a single line, by adding the `flex-direction` value first, followed by `flex-wrap` value, for example: `flex-flow: row wrap`.

Gap Property for Flexbox

The `gap` property is now fully supported in Flexbox, making spacing between items much easier. Previously, you had to use margins on flex items, which required adjusting for first/last items. Now, `gap` handles this automatically:

```
/* Modern way - using gap */
.flex-container {
  display: flex;
  gap: 1rem; /* Adds 1rem space between all items */
}

/* Separate row and column gaps */
.flex-container {
  display: flex;
  flex-wrap: wrap;
```

```
  row-gap: 2rem;    /* Space between rows */
  column-gap: 1rem; /* Space between columns */
}

/* Shorthand: row-gap column-gap */
.flex-container {
  display: flex;
  flex-wrap: wrap;
  gap: 2rem 1rem;
}
```

The `gap` property only adds space between items, not around the edges of the container. This is much cleaner than the old margin approach:

```
/* Old way - using margins (avoid this) */
.flex-item {
  margin-right: 1rem;
}
.flex-item:last-child {
  margin-right: 0; /* Had to remove margin from last item */
}

/* New way - using gap (use this) */
.flex-container {
  display: flex;
  gap: 1rem; /* Much simpler! */
}
```

Gap works with all flex directions and wrapping modes, making it the preferred method for spacing flex items in modern CSS.

Properties that apply to each single item

Since now, we've seen the properties you can apply to the container.

Single items can have a certain amount of independence and flexibility, and you can alter their appearance using those properties:

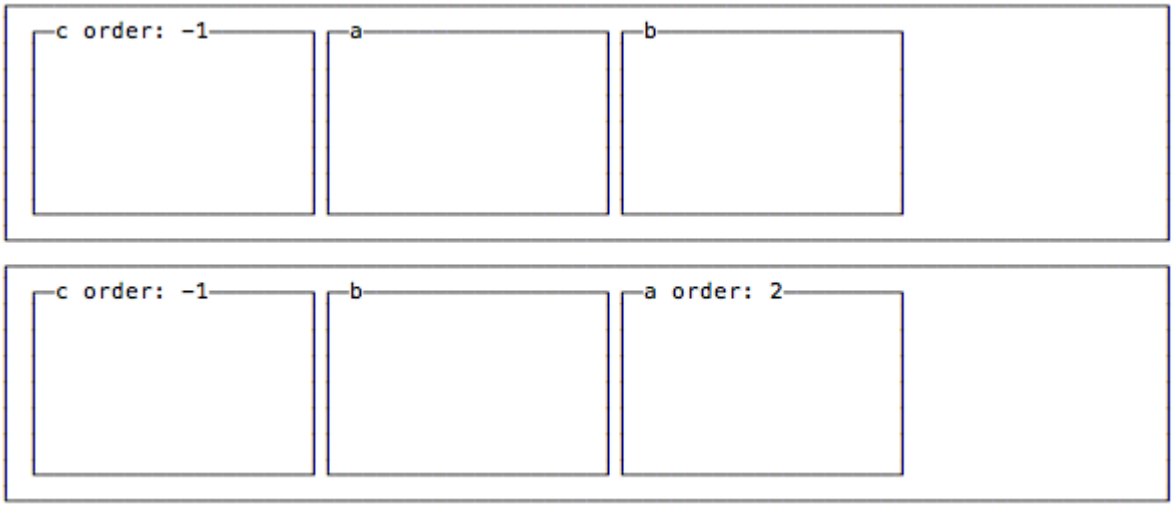
- `order`
- `align-self`
- `flex-grow`
- `flex-shrink`
- `flex-basis`
- `flex`

Let's see them in detail.

Moving items before / after another one using order

Items are ordered based on a order they are assigned. By default every item has order `0` and the appearance in the HTML determines the final order.

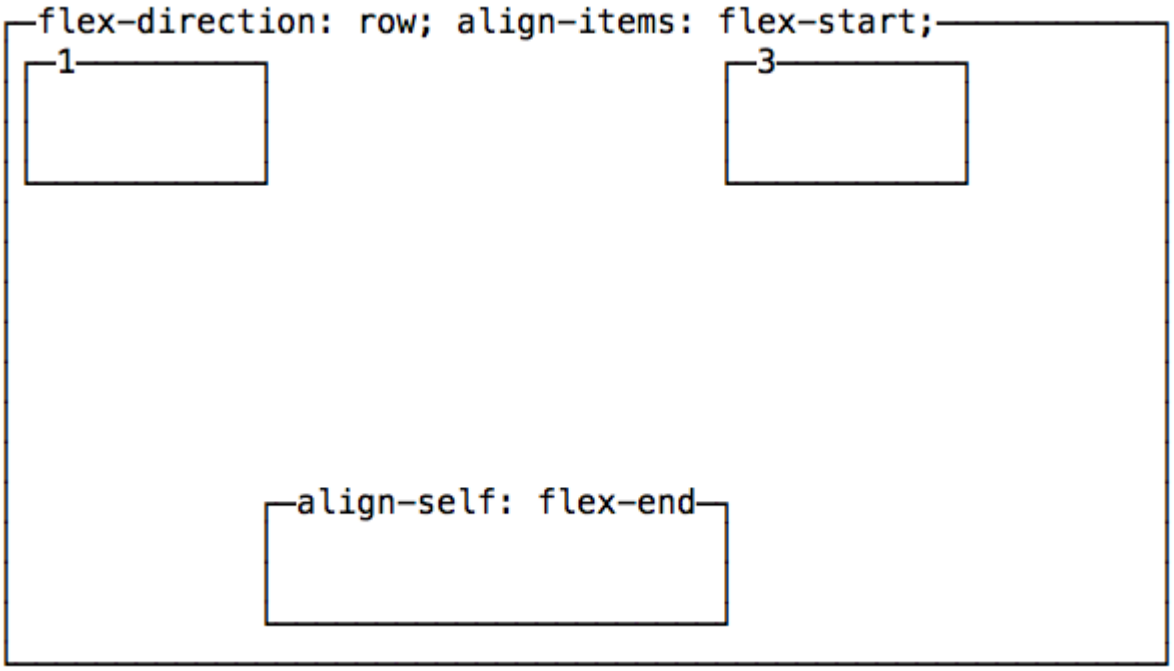
You can override this property using `order` on each separate item. This is a property you set on the item, not the container. You can make an item appear before all the others by setting a negative value.



Vertical alignment using align-self

An item can choose to **override** the container `align-items` setting, using **`align-self`**, which has the same 5 possible values of `align-items`:

- `flex-start`: align to the top of the container.
- `flex-end`: align to the bottom of the container.
- `center`: align at the vertical center of the container.
- `baseline`: display at the baseline of the container.
- `stretch`: items are stretched to fit the container.



Grow or shrink an item if necessary

- **`flex-grow`**
  The defaut for any item is 0.
  If all items are defined as 1 and one is defined as 2, the bigger element will take the space of two "1" items.
- **`flex-shrink`**
  The defaut for any item is 1.
  If all items are defined as 1 and one is defined as 3, the bigger element will shrink 3x the other ones. When less space is available, it will take 3x less space.
- **`flex-basis`**
  If set to `auto`, it sizes an item according to its width or height, and adds extra space based on the `flex-grow` property.
  If set to 0, it does not add any extra space for the item when calculating the layout.
  If you specify a pixel number value, it will use that as the length value (width or height depends if it's a row or a column item)
- **`flex`**
  This property combines the above 3 properties, `flex-grow`, `flex-shrink` and `flex-basis`, and provides a shorthand syntax: `flex: 0 1 auto`

CSS Grid

CSS Grid is a fundamentally new approach to building layouts using CSS.

CSS Grid is not a competitor to Flexbox. They interoperate and collaborate on complex layouts, because CSS Grid works on 2 dimensions (rows AND columns) while Flexbox works on a single dimension (rows OR columns).

The basics

The CSS Grid layout is activated on a container element (which can be a `div` or any other tag) by setting `display: grid`.

As with flexbox, you can define some properties on the container, and some properties on each individual item in the grid.

These properties combined will determine the final look of the grid.

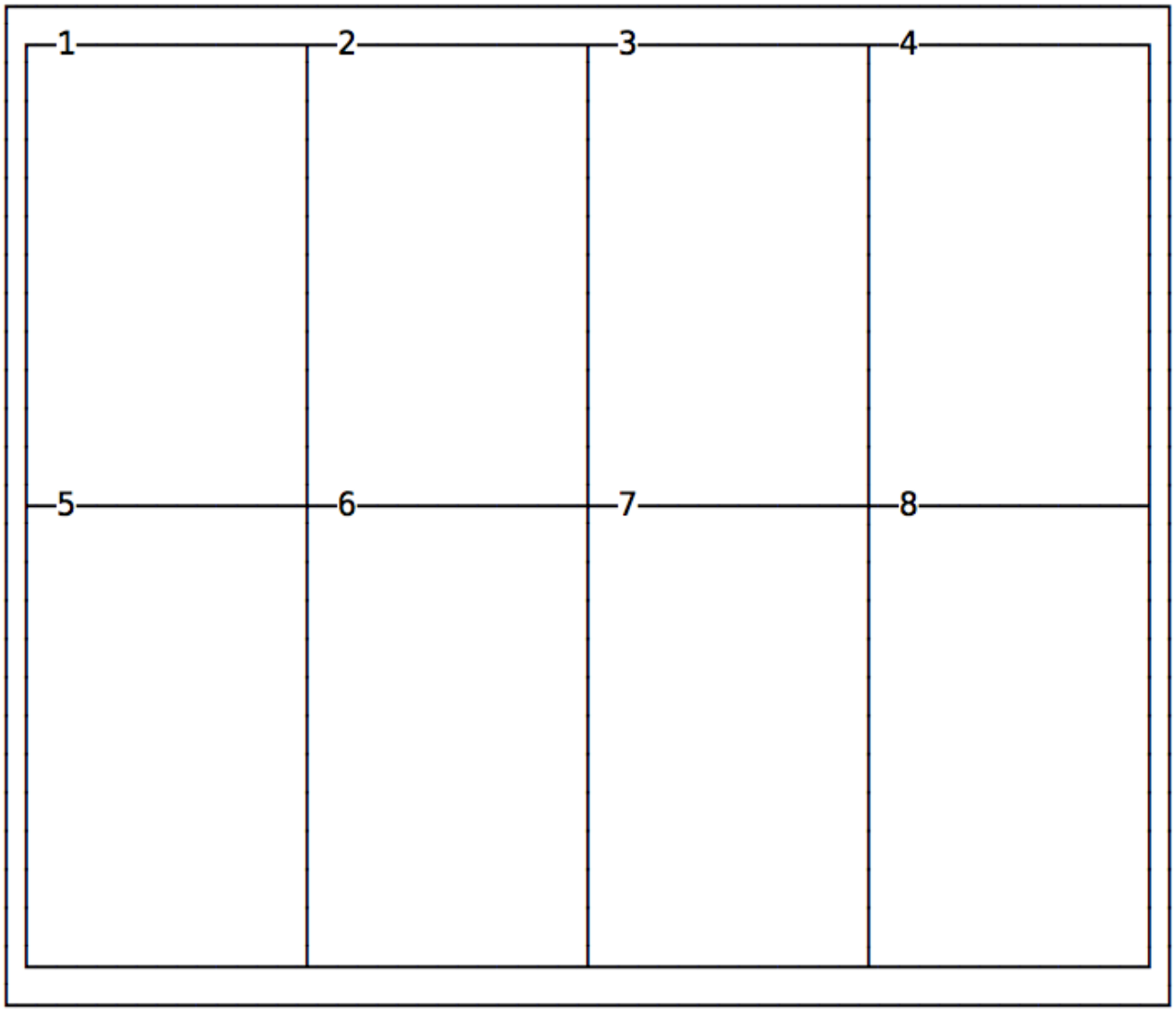
The most basic container properties are `grid-template-columns` and `grid-template-rows`.

grid-template-columns and grid-template-rows

Those properties define the number of columns and rows in the grid, and they also set the width of each column/row.

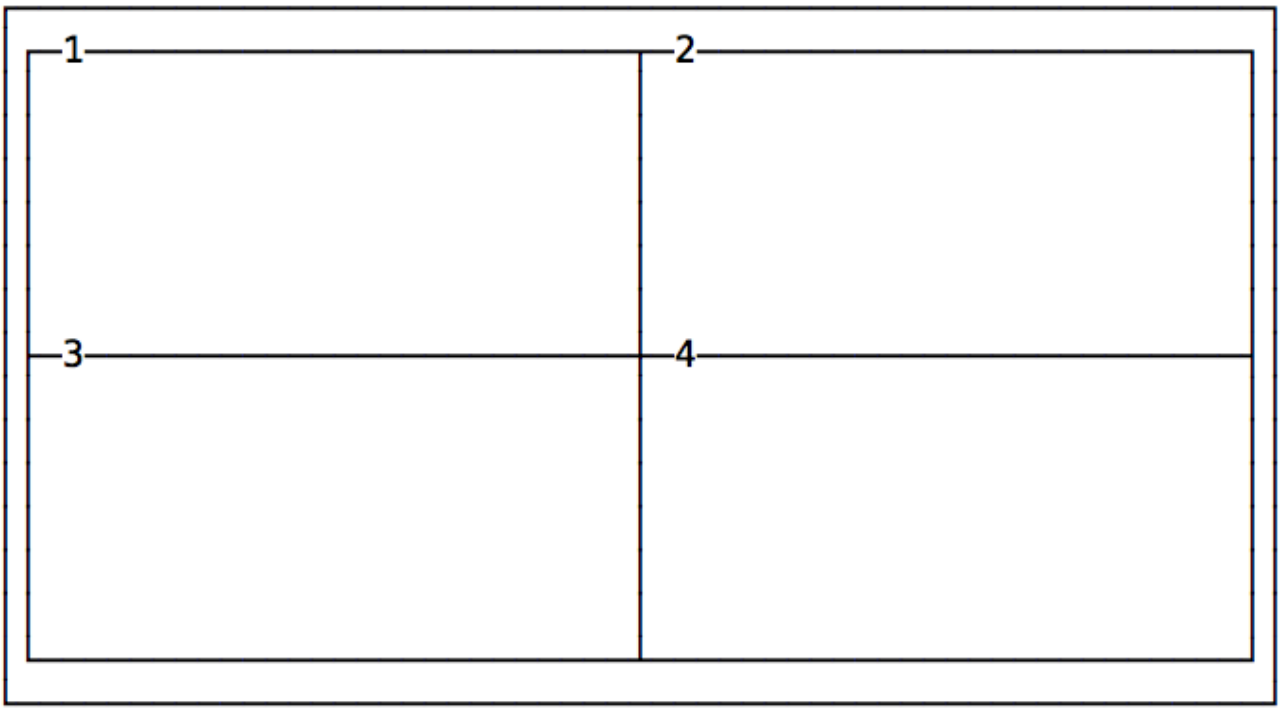
The following snippet defines a grid with 4 columns each 200px wide, and 2 rows with a 300px height each.

```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px 200px;
  grid-template-rows: 300px 300px;
}
```



Here's another example of a grid with 2 columns and 2 rows:

```
.container {
  display: grid;
  grid-template-columns: 200px 200px;
  grid-template-rows: 100px 100px;
}
```



Automatic dimensions

Many times you might have a fixed header size, a fixed footer size, and the main content that is flexible in height, depending on its length. In this case you can use the `auto` keyword:

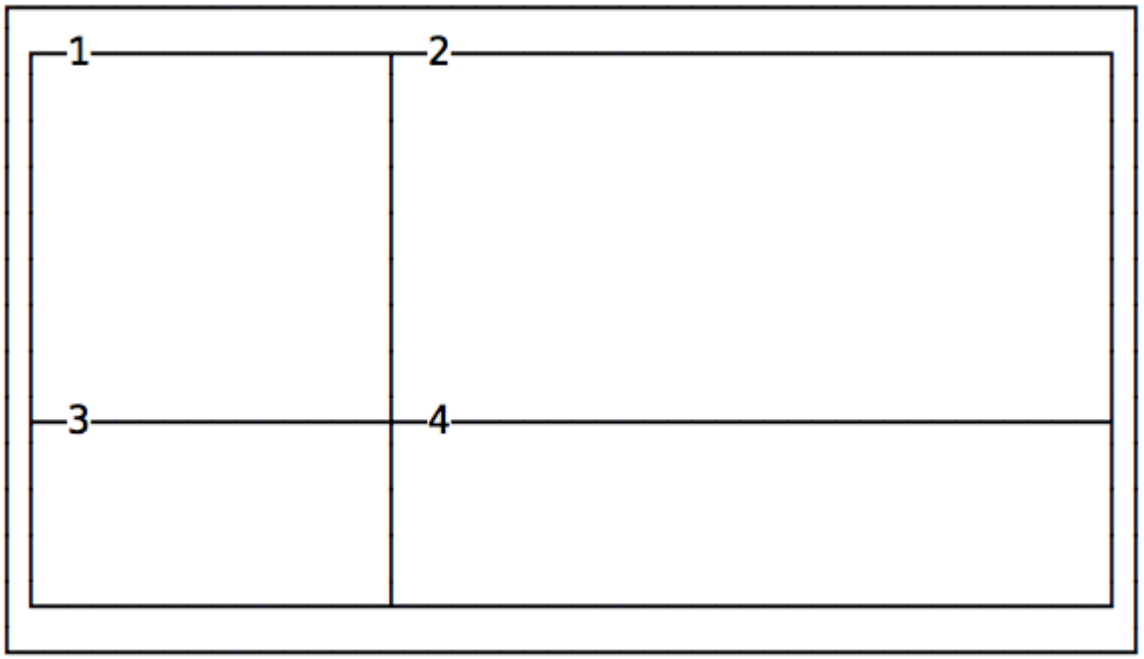
```
.container {
  display: grid;
  grid-template-rows: 100px auto 100px;
}
```

Different columns and rows dimensions

In the above examples we made pretty, regular grids by using the same values for rows and the same values for columns.

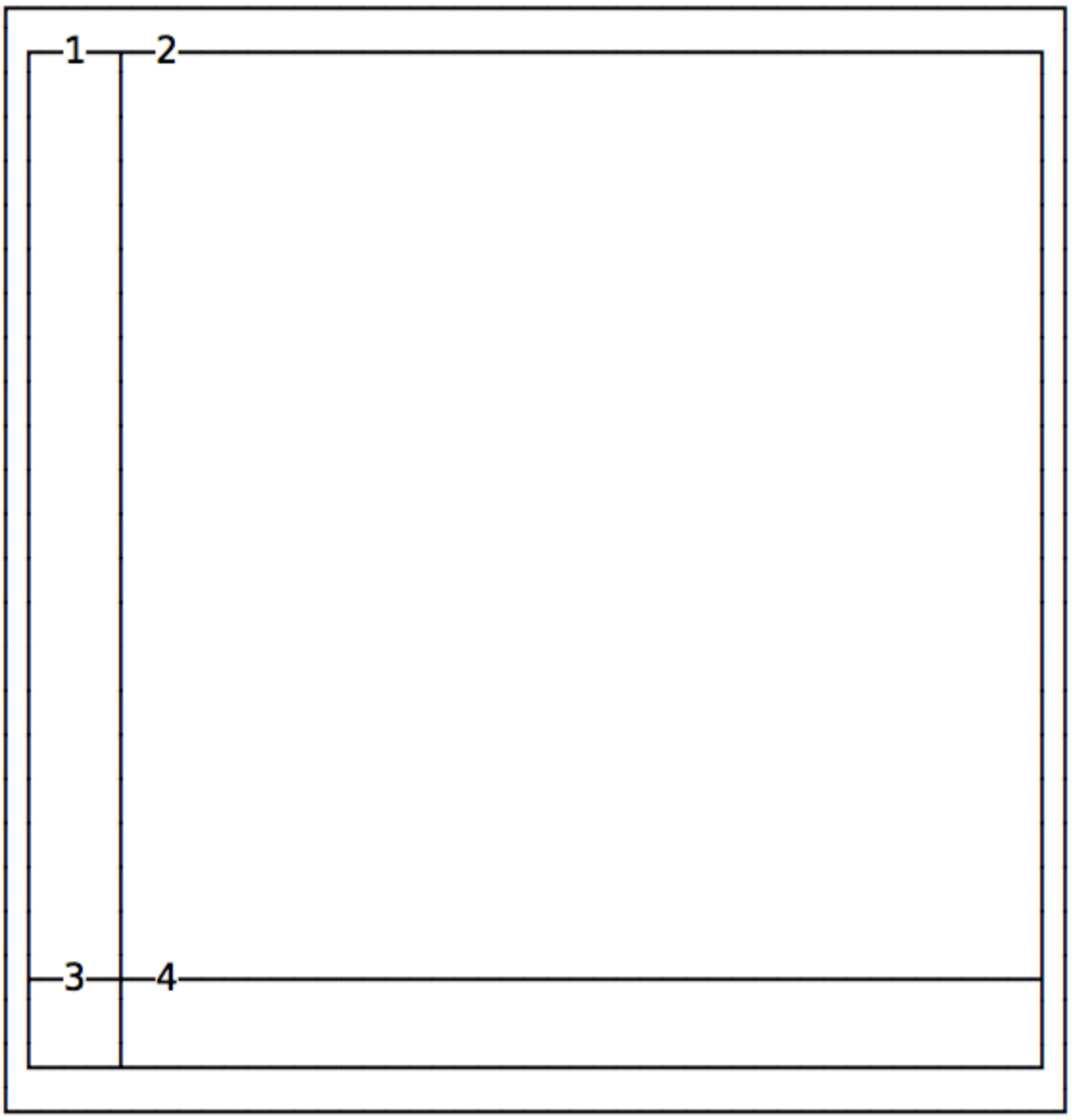
You can specify any value for each row/column, to create a lot of different designs:

```
.container {
  display: grid;
  grid-template-columns: 100px 200px;
  grid-template-rows: 100px 50px;
}
```



Another example:

```
.container {
  display: grid;
  grid-template-columns: 10px 100px;
  grid-template-rows: 100px 10px;
}
```



Adding space between the cells

Unless specified, there is no space between the cells.

You can add spacing by using those properties:

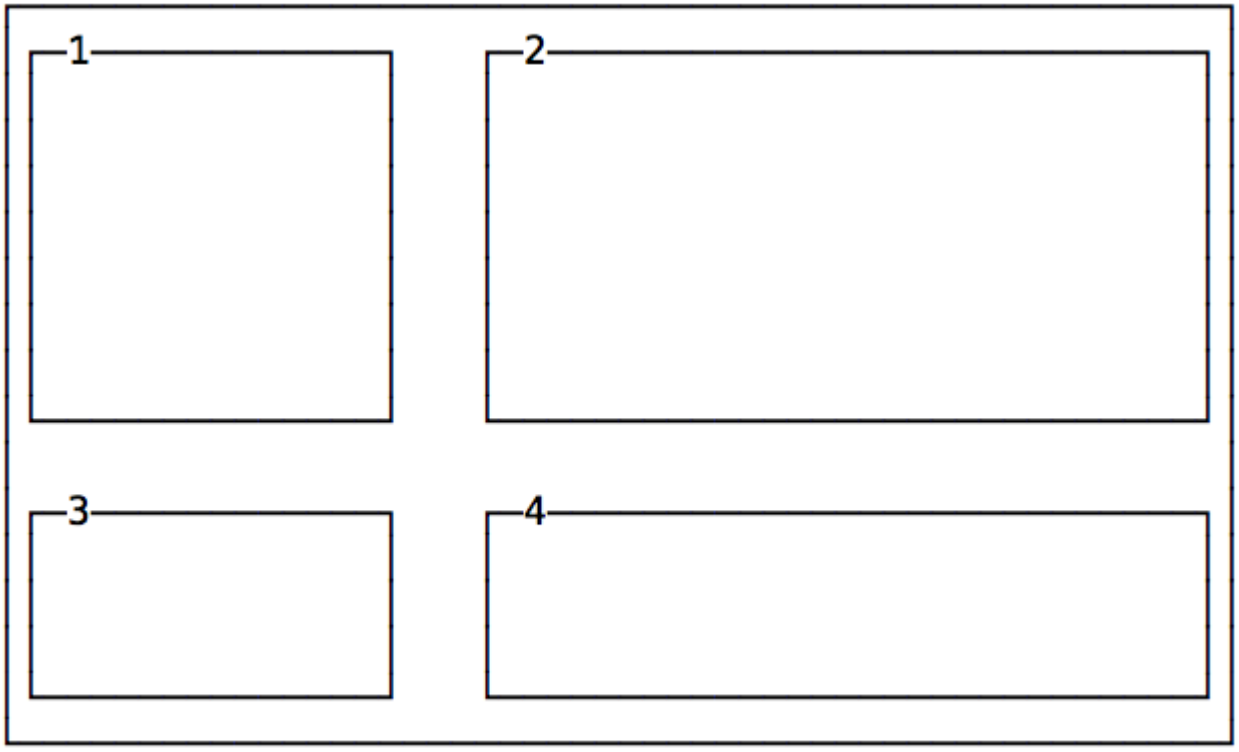
- `grid-column-gap`
- `grid-row-gap`

or the shorthand syntax `grid-gap`.

Example:

```
.container {
  display: grid;
  grid-template-columns: 100px 200px;
  grid-template-rows: 100px 50px;
  grid-column-gap: 25px;
```

```
  grid-row-gap: 25px;
}
```



The same layout using the shorthand:

```
.container {
  display: grid;
  grid-template-columns: 100px 200px;
  grid-template-rows: 100px 50px;
  grid-gap: 25px;
}
```

Span items on multiple columns and/or rows

Every cell item has the option to occupy more than just one box in the row, and expand horizontally or vertically to get more space, while respecting the grid proportions set in the container.

Those are the properties we'll use for that:

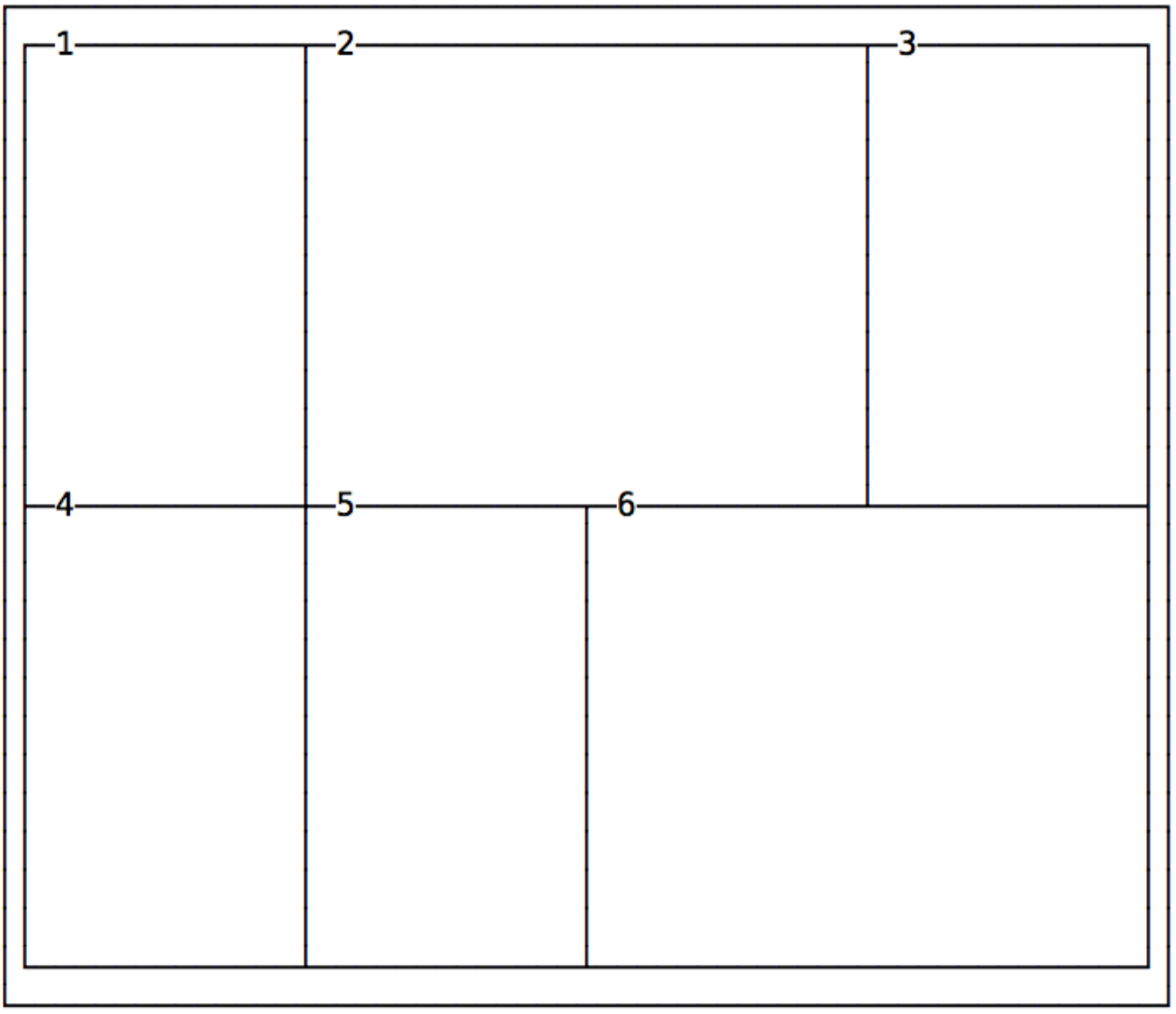
- `grid-column-start`
- `grid-column-end`
- `grid-row-start`
- `grid-row-end`

Example:

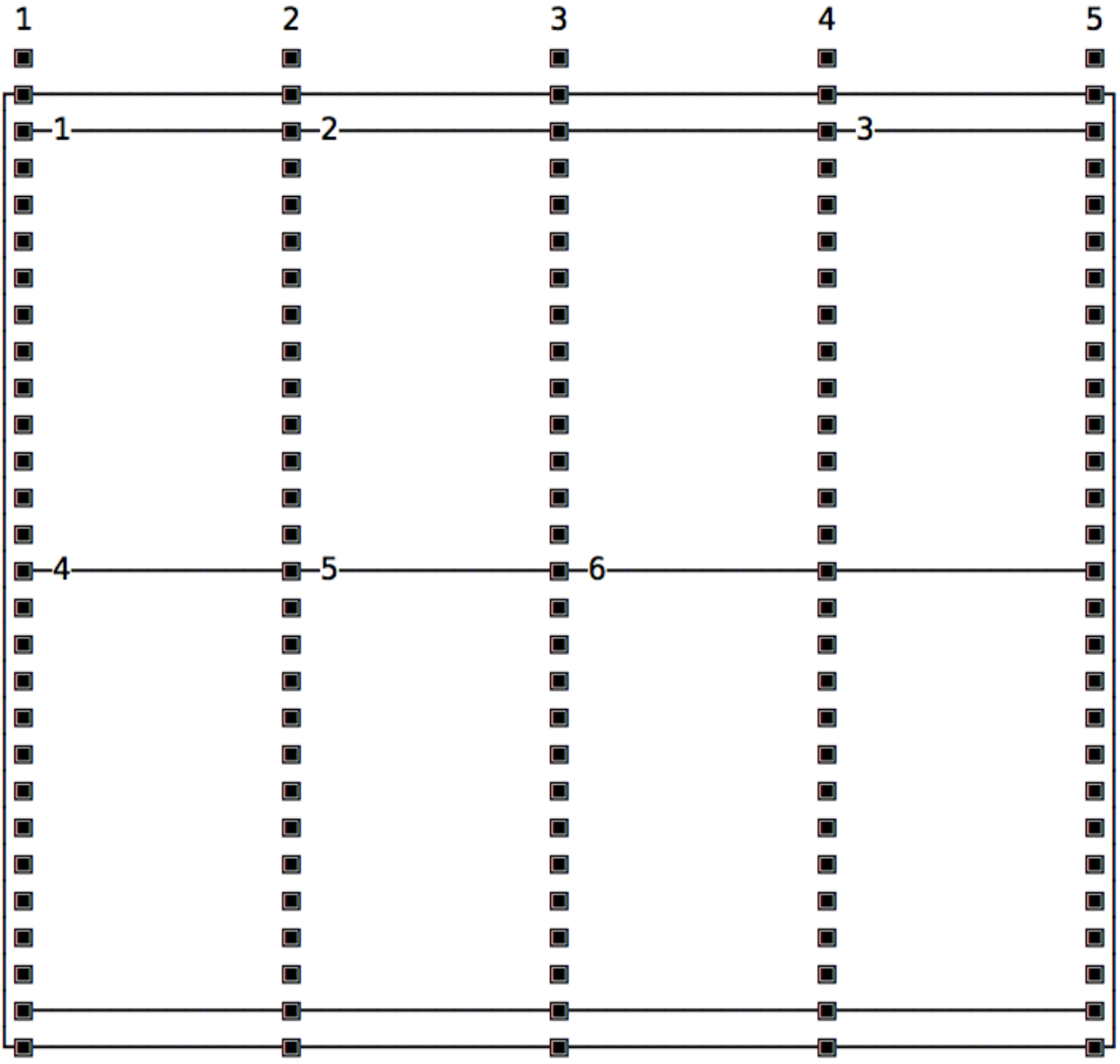
```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px 200px;
  grid-template-rows: 300px 300px;
}

.item1 {
  grid-column-start: 2;
  grid-column-end: 4;
}

.item6 {
  grid-column-start: 3;
  grid-column-end: 5;
}
```



The numbers correspond to the vertical line that separates each column, starting from 1:



The same principle applies to `grid-row-start` and `grid-row-end`, except this time instead of taking more columns, a cell takes more rows.

Shorthand syntax

Those properties have a shorthand syntax provided by:

- `grid-column`
- `grid-row`

The usage is simple, here's how to replicate the above layout:

```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px 200px;
  grid-template-rows: 300px 300px;
}

.item1 {
```

```
  grid-column: 2 / 4;
}

.item6 {
  grid-column: 3 / 5;
}
```

Using `grid-area` as a shorthand

The `grid-area` property can be used as a shorthand for the `grid-column` and `grid-row` shorthands, when you need to apply both to a single element. Instead of having:

```
.item1 {
  grid-row: 1 / 4;
  grid-column: 3 / 5;
}
```

You can use

```
.item1 {
  grid-area: 1 / 3 / 4 / 5;
}
```

(grid-row-start / grid-column-start / grid-row-end / grid-column-end)

Using `span`

Another approach is to set the starting column/row, and set how many it should occupy using `span`:

```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px 200px;
  grid-template-rows: 300px 300px;
}

.item1 {
  grid-column: 2 / span 2;
}

.item6 {
  grid-column: 3 / span 2;
}
```

`span` works also with the non-shorthand syntax:

```
.item1 {
  grid-column-start: 2;
  grid-column-end: span 2;
}
```

and you can also use on the start property. In this case, the end position will be used as a reference, and `span` will count "back":

```
.item1 {
  grid-column-start: span 2;
  grid-column-end: 3;
}
```

More grid configuration

Using fractions

Specifying the exact width of each column or row is not ideal in every case.

A fraction is a unit of space.

The following example divides a grid into 3 columns with the same width, 1/3 of the available space each.

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

Using percentages and rem

You can also use percentages, and mix and match fractions, pixels, rem and percentages:

```
.container {
  grid-template-columns: 3rem 15% 1fr 2fr
}
```

Using `repeat()`

`repeat()` is a special function that takes a number that indicates the number of times a row/column will be repeated, and the length of each one.

If every column has the same width you can specify the layout using this syntax:

```
.container {
  grid-template-columns: repeat(4, 100px);
```

```
}
```

This creates 4 columns with the same width.

Or using fractions:

```
.container {
  grid-template-columns: repeat(4, 1fr);
}
```

Specify a minimum width for a row

Common use case: Have a sidebar that never collapses more than a certain amount of pixels when you resize the window.

Here's an example where the sidebar takes 1/4 of the screen and never takes less than 200px:

```
.container {
  grid-template-columns: minmax(200px, 3fr) 9fr;
}
```

You can also set just a maximum value using the `auto` keyword:

```
.container {
  grid-template-columns: minmax(auto, 50%) 9fr;
}
```

or just a minimum value:

```
.container {
  grid-template-columns: minmax(100px, auto) 9fr;
}
```

Positioning elements using `grid-template-areas`

By default elements are positioned in the grid using their order in the HTML structure.

Using `grid-template-areas` You can define template areas to move them around in the grid, and also to span an item on multiple rows / columns instead of using `grid-column`.

Here's an example:

```
<div class="container">
  <main>
```

```
    ...
  </main>
  <aside>
    ...
  </aside>
  <header>
    ...
  </header>
  <footer>
    ...
  </footer>
</div>
```

```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px 200px;
  grid-template-rows: 300px 300px;
  grid-template-areas:
    "header header header header"
    "sidebar main main main"
    "footer footer footer footer";
}

main {
  grid-area: main;
}

aside {
  grid-area: sidebar;
}

header {
  grid-area: header;
}

footer {
  grid-area: footer;
}
```

Despite their original order, items are placed where `grid-template-areas` define, depending on the `grid-area` property associated to them.

Adding empty cells in template areas

You can set an empty cell using the dot `.` instead of an area name in `grid-template-areas`:

```
.container {
  display: grid;
  grid-template-columns: 200px 200px 200px 200px;
  grid-template-rows: 300px 300px;
  grid-template-areas:
    ". header header ."
    "sidebar . main main"
    ". footer footer .";
}
```

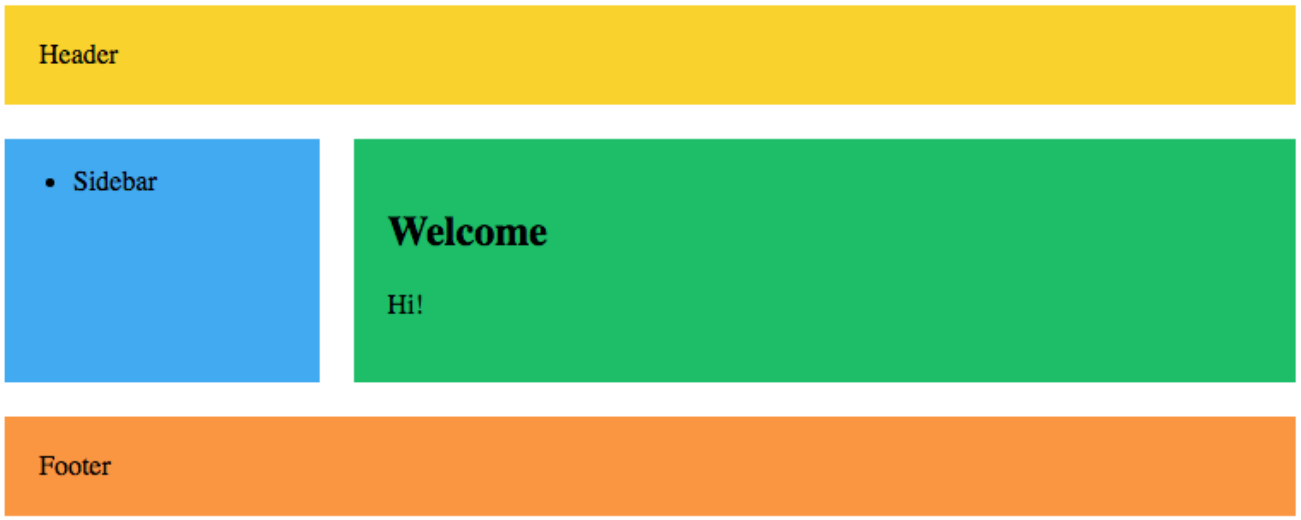
Fill a page with a grid

You can make a grid extend to fill the page using `fr`:

```
.container {
  display: grid;
  height: 100vh;
  grid-template-columns: 1fr 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
}
```

An example: header, sidebar, content and footer

Here is a simple example of using CSS Grid to create a site layout that provides a header op top, a main part with sidebar on the left and content on the right, and a footer afterwards.



Here's the markup:

```
<div class="wrapper">
  <header>Header</header>
  <article>
    <h1>Welcome</h1>
    <p>Hi!</p>
```

```
  </article>
  <aside><ul><li>Sidebar</li></ul></aside>
  <footer>Footer</footer>
</div>
```

and here's the CSS:

```
header {
  grid-area: header;
  background-color: #fed330;
  padding: 20px;
}

article {
  grid-area: content;
  background-color: #20bf6b;
  padding: 20px;
}

aside {
  grid-area: sidebar;
  background-color: #45aaf2;
}

footer {
  padding: 20px;
  grid-area: footer;
  background-color: #fd9644;
}

.wrapper {
  display: grid;
  grid-gap: 20px;
  grid-template-columns: 1fr 3fr;
  grid-template-areas:
    "header  header"
    "sidebar content"
    "footer  footer";
}
```

I added some colors to make it prettier, but basically it assigns to every different tag a `grid-area` name, which is used in the `grid-template-areas` property in `.wrapper`.

When the layout is smaller we can put the sidebar below the content using a media query:

```
@media (max-width: 500px) {
  .wrapper {
    grid-template-columns: 4fr;
```

```
    grid-template-areas:
      "header"
      "content"
      "sidebar"
      "footer";
  }
}
```

Custom Properties (or CSS Variables)

CSS is not a programming language like JavaScript/TypeScript, C, Python, PHP, Ruby or Go where variables are key to do something useful. CSS is very limited in what it can do, and it's mainly a declarative syntax to tell browsers how they should display an HTML page.

But a variable is a variable: a name that refers to a value, and variables in CSS helps reduce repetition and inconsistencies in your CSS, by centralizing the values definition.

And it introduces a way to access and change the value of a CSS Variable programmatically using JavaScript.

A CSS Variable is defined with a special syntax, prepending **two dashes** to a name (`--variable-name`), then a colon and a value. Like this:

```
:root {
  --primary-color: yellow;
}
```

(more on `:root` later)

You can access the variable value using `var()` :

```
p {
  color: var(--primary-color);
}
```

`var()` accepts a second parameter, which is the default fallback value when the variable value is not set:

```
p {
  color: var(--primary-color, 30px);
}
```

The variable value can be any valid CSS value, for example:

```
:root {
  --default-padding: 30px 30px 20px 20px;
  --default-color: red;
  --default-background: #fff;
}
```

CSS Variables can be defined inside any element. Some examples:

```
:root {
  --default-color: red;
}

body {
  --default-color: red;
}

main {
  --default-color: red;
}

p {
  --default-color: red;
}

span {
  --default-color: red;
}

a:hover {
  --default-color: red;
}
```

What changes in those different examples is the **scope**.

Adding variables to a selector makes them available to all the children of it.

In the example above you saw the use of `:root` when defining a CSS variable:

```
:root {
  --primary-color: yellow;
}
```

`:root` is a CSS pseudo-class that identifies the root element of a tree.

In the context of an HTML document, using the `:root` selector points to the `html` element, except that `:root` has higher specificity (takes priority).

In the context of an SVG image, `:root` points to the `svg` tag.

Adding a CSS custom property to `:root` makes it available to all the elements in the page.

If you add a variable inside a `.container` selector, it's only going to be available to children of `.container`:

```
.container {
  --secondary-color: yellow;
}
```

and using it outside of this element is not going to work.

Variables can be **reassigned**:

```
:root {
  --primary-color: yellow;
}

.container {
  --primary-color: blue;
}
```

Outside `.container`, `--primary-color` will be *yellow*, but inside it will be *blue*.

You can also assign or overwrite a variable inside the HTML using **inline styles**:

```
<main style="--primary-color: orange;">
  <!-- ... -->
</main>
```

> CSS Variables follow the normal CSS cascading rules, with precedence set according to specificity

The coolest thing with CSS Variables is the ability to access and edit them using JavaScript.

Here's how you set a variable value using plain JavaScript:

```
const element = document.getElementById('my-element')
element.style.setProperty('--variable-name', 'a-value')
```

This code below can be used to access a variable value instead, in case the variable is defined on `:root`:

```
const styles = getComputedStyle(document.documentElement)
const value = String(styles.getPropertyValue('--variable-name')).trim()
```

Or, to get the style applied to a specific element, in case of variables set with a different scope:

```
const element = document.getElementById('my-element')
const styles = getComputedStyle(element)
const value = String(styles.getPropertyValue('--variable-name')).trim()
```

If a variable is assigned to a property which does not accept the variable value, it's considered invalid.

For example you might pass a pixel value to a `position` property, or a rem value to a color property.

In this case the line is considered invalid and ignored.

Note that CSS Variables are case sensitive, this variable `--width: 100px;` is different than the variable `--Width: 100px;`.

To do math in CSS Variables, you need to use `calc()`, for example:

```
:root {
  --default-left-padding: calc(10px * 2);
}
```

Media types

Used in media queries and `@import` declarations, media types allow us to determine on which media a CSS file, or a piece of CSS, is loaded.

We have the following **media types**

- `all` means all the media
- `print` used when printing
- `screen` used when the page is presented on a screen
- `speech` used for screen readers

`screen` is the default.

In the past we had more of them, but most are deprecated as they proven to not be an effective way of determining device needs.

We can use them in @import statements like this:

```
@import url(myfile.css) screen;
@import url(myfile-print.css) print;
```

We can load a CSS file on multiple media types separating each with a comma:

```
@import url(myfile.css) screen, print;
```

The same works for the `link` tag in HTML:

```
<link rel="stylesheet" type="text/css" href="myfile.css" media="screen" />
<link rel="stylesheet" type="text/css" href="another.css" media="screen,
print" />
```

We're not limited to just using media types in the `media` attribute and in the `@import` declaration.

Using **media feature descriptors,** we can add more information to the `media` attribute of `link` or to the `@import` declaration, to express more conditionals over the loading of the CSS.

Here's the list of them:

- `width`
- `height`
- `device-width`
- `device-height`
- `aspect-ratio`
- `device-aspect-ratio`
- `color`
- `color-index`
- `monochrome`
- `resolution`
- `orientation`
- `scan`
- `grid`

Each of them have a corresponding min- *and* max-, for example:

- `min-width` , `max-width`

- `min-device-width`, `max-device-width`

and so on.

Some of those accept a length value which can be expressed in `px` or `rem` or any length value. It's the case of `width`, `height`, `device-width`, `device-height`.

For example:

```
@import url(myfile.css) screen and (max-width: 800px);
```

Notice that we wrap each block using media feature descriptors in parentheses.

Some accept a fixed value. `orientation`, used to detect the device orientation, accepts `portrait` or `landscape`.

Example:

```
<link rel="stylesheet" type="text/css" href="myfile.css" media="screen and
(orientation: portrait)" />
```

`scan`, used to determine the type of screen, accepts `progressive` (for modern displays) or `interlace` (for older CRT devices)

Some others want an integer.

Like `color` which inspects the number of bits per color component used by the device. Very low-level, but you just need to know it's there for your usage (like `grid`, `color-index`, `monochrome`).

`aspect-ratio` and `device-aspect-ratio` accept a ratio value representing the width to height viewport ratio, which is expressed as a fraction.

Example:

```
@import url(myfile.css) screen and (aspect-ratio: 4/3);
```

`resolution` represents the pixel density of the device, expressed in a resolution data type like `dpi`.

Example:

```
@import url(myfile.css) screen and (min-resolution: 100dpi);
```

We can combine rules using `and`:

```
<link rel="stylesheet" type="text/css" href="myfile.css" media="screen and
(max-width: 800px)" />
```

We can perform an "or" type of logic operation using commas, which combines multiple media queries:

```
@import url(myfile.css) screen, print;
```

We can use `not` to negate a media query:

```
@import url(myfile.css) not screen;
```

> Important: not can only be used to negate an entire media query, so it must be placed at the beginning of it (or after a comma)

Media queries

All those above rules we saw applied to @import or to the `link` HTML tag can be applied inside the CSS, too.

You need to wrap them in a `@media () {}` structure.

Example:

```
@media screen and (max-width: 800px) {
  /* enter some CSS */
}
```

and this is the foundation for **responsive design**.

Media queries can be quite complex. This example applies the CSS only if it's a screen device, the width is between 600 and 800 pixels, and the orientation is landscape:

```
@media screen and (max-width: 800px) and (min-width: 600px) and
(orientation: landscape) {
  /* enter some CSS */
}
```

A common strategy is to design for mobile devices first, then use media queries to enhance the layout for larger screens:

```
/* Base styles for mobile */
.container {
  width: 100%;
```

```
  padding: 10px;
}

/* Tablet styles */
@media screen and (min-width: 768px) {
  .container {
    width: 750px;
    padding: 20px;
  }
}

/* Desktop styles */
@media screen and (min-width: 1024px) {
  .container {
    width: 980px;
  }
}
```

While you should design breakpoints based on your content, these are commonly used widths:

- 320px — 480px: Mobile devices
- 481px — 768px: iPads, tablets
- 769px — 1024px: Small screens, laptops
- 1025px — 1200px: Desktops, large screens
- 1201px and more: Extra large screens, TV

Be careful with these media query issues:

- **Too many breakpoints:** This can make maintenance difficult. Focus on content-driven breakpoints.
- **Overlapping queries:** Ensure your media queries don't contradict each other with overlapping conditions.
- **Device-specific media queries:** Target ranges of screen sizes rather than specific devices.

Dark Mode Support with prefers-color-scheme

Modern operating systems offer dark mode, and CSS can detect and respond to the user's preference using the `prefers-color-scheme` media query. This allows your website to automatically adapt to the user's system settings.

Dark mode isn't just a trendy feature - it reduces eye strain in low-light conditions, saves battery on OLED screens, and many users simply prefer it. The `prefers-color-scheme` media query lets your website automatically match the user's system preference, creating a

seamless experience. When users switch their device to dark mode, your website switches too, without them having to find a toggle button.

Basic Dark Mode Implementation

The simplest way to implement dark mode is to detect the preference and adjust colors accordingly:

The key to maintainable dark mode is using CSS custom properties (variables). Define your colors once as variables, then change only the variable values for dark mode. This approach keeps your actual CSS rules unchanged - only the color definitions swap.

```
/* Light mode (default) */
:root {
  --background: #ffffff;
  --text-color: #000000;
  --card-bg: #f5f5f5;
  --link-color: #0066cc;
  --border-color: #dddddd;
}
```

These are your default colors, typically for light mode. The `:root` selector (which targets the `<html>` element) is where we define global CSS variables. Each variable starts with `--` and can be named anything you want.

```
/* Dark mode */
@media (prefers-color-scheme: dark) {
  :root {
    --background: #1a1a1a;
    --text-color: #e0e0e0;
    --card-bg: #2a2a2a;
    --link-color: #66b3ff;
    --border-color: #444444;
  }
}
```

When the user's system is in dark mode, this media query activates and overrides the color variables. Notice how we're not using pure black (#000000) for the background - that's too harsh. Dark grays are easier on the eyes.

```
/* Apply the CSS variables */
body {
  background-color: var(--background);
  color: var(--text-color);
}
```

```
.card {
  background-color: var(--card-bg);
  border: 1px solid var(--border-color);
}
```

Now your components reference the variables, not hard-coded colors. When dark mode activates, these components automatically use the dark color values without any changes to their CSS.

Detecting Light Mode Explicitly

You can also explicitly target light mode:

```
@media (prefers-color-scheme: light) {
  /* Light mode specific styles */
  .hero {
    background-image: url('light-hero.jpg');
  }
}

@media (prefers-color-scheme: dark) {
  /* Dark mode specific styles */
  .hero {
    background-image: url('dark-hero.jpg');
  }
}
```

Images and Dark Mode

Images might need adjustments in dark mode. You can use filters or provide alternative images:

```
/* Dim images in dark mode */
@media (prefers-color-scheme: dark) {
  img {
    opacity: 0.8;
    filter: brightness(0.9);
  }

  /* Except for images that should stay bright */
  img.logo {
    opacity: 1;
    filter: none;
  }
}

/* Or swap images entirely */
```

```
.logo {
  content: url('logo-light.svg');
}

@media (prefers-color-scheme: dark) {
  .logo {
    content: url('logo-dark.svg');
  }
}
```

Respecting User Preference with JavaScript

You can also detect and respond to color scheme preferences with JavaScript:

```
/* Add a class-based override system */
body.light-mode {
  --background: #ffffff;
  --text-color: #000000;
}

body.dark-mode {
  --background: #1a1a1a;
  --text-color: #e0e0e0;
}
```

Advanced Dark Mode Patterns

Here's a comprehensive example with smooth transitions:

```
/* Define color schemes with semantic names */
:root {
  /* Light mode colors */
  --color-bg-primary: #ffffff;
  --color-bg-secondary: #f8f9fa;
  --color-bg-tertiary: #e9ecef;
  --color-text-primary: #212529;
  --color-text-secondary: #6c757d;
  --color-accent: #0066cc;
  --color-success: #28a745;
  --color-warning: #ffc107;
  --color-danger: #dc3545;
  --shadow-sm: 0 1px 2px rgba(0,0,0,0.1);
  --shadow-md: 0 4px 6px rgba(0,0,0,0.1);
}

@media (prefers-color-scheme: dark) {
  :root {
```

```
    --color-bg-primary: #121212;
    --color-bg-secondary: #1e1e1e;
    --color-bg-tertiary: #2a2a2a;
    --color-text-primary: #e0e0e0;
    --color-text-secondary: #a0a0a0;
    --color-accent: #66b3ff;
    --color-success: #4caf50;
    --color-warning: #ff9800;
    --color-danger: #f44336;
    --shadow-sm: 0 1px 2px rgba(0,0,0,0.3);
    --shadow-md: 0 4px 6px rgba(0,0,0,0.3);
  }
}

/* Smooth transitions when changing themes */
body {
  background-color: var(--color-bg-primary);
  color: var(--color-text-primary);
  transition: background-color 0.3s ease, color 0.3s ease;
}

/* Components using the color system */
.card {
  background: var(--color-bg-secondary);
  border: 1px solid var(--color-bg-tertiary);
  box-shadow: var(--shadow-sm);
}

.button-primary {
  background: var(--color-accent);
  color: var(--color-bg-primary);
}
```

Best Practices for Dark Mode

1. **Use semantic color names** rather than literal ones (e.g., `--color-text-primary` instead of `--color-black`)
2. **Test contrast ratios** in both modes to ensure accessibility
3. **Consider reduced contrast in dark mode** - pure black on white becomes off-white on dark gray
4. **Adjust shadows and elevations** - shadows may need to be lighter or darker depending on the mode
5. **Test with real content** - ensure images, videos, and third-party embeds look good in both modes

The `prefers-color-scheme` media query is widely supported in modern browsers and provides a seamless way to respect user preferences for dark or light interfaces.

Feature queries

Feature queries allow you to apply CSS rules only when the browser supports a particular CSS feature. This is done using the @supports rule, which checks if a browser supports a specific property or value before applying styles.

The basic syntax is:

```
@supports (property: value) {
  /* CSS rules that will only be applied if the feature is supported */
}
```

For example, to apply styles only when CSS Grid is supported:

```
@supports (display: grid) {
  .container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    gap: 20px;
  }
}
```

You can also check for the lack of support using the not operator:

```
@supports not (display: grid) {
  /* Fallback styles for browsers that don't support grid */
  .container {
    display: flex;
    flex-wrap: wrap;
  }
}
```

Feature queries support logical operators for complex conditions:

- **and**: Requires all conditions to be true
- **or**: Requires at least one condition to be true
- **not**: Negates a condition

Here's an example combining multiple conditions:

```
@supports (display: grid) and (gap: 20px) {
  /* CSS that uses both grid and gap properties */
}
```

Using Feature Queries with @supports

You can combine media queries with feature queries to create even more specific conditions:

```
@media screen and (min-width: 800px) {
  @supports (display: grid) {
    .container {
      display: grid;
      grid-template-columns: 1fr 1fr;
    }
  }
}
```

Feature queries are particularly useful when:

- Implementing progressive enhancement strategies
- Providing fallbacks for newer CSS features
- Creating layouts that adapt to browser capabilities, not just screen sizes

Container Queries

Container queries represent the next evolution in responsive design, allowing you to style elements based on the size of their parent container rather than just the viewport size. This provides more granular control for component-based designs.

While media queries are useful for adapting layouts to different screen sizes, they fall short when you need to reuse components in different contexts. Container queries solve this problem by letting components adapt based on their immediate container's size, not just the viewport.

```
/* Define a container */
.card-container {
  container-type: inline-size;
  container-name: card;
}

/* Apply styles based on container width */
@container card (min-width: 400px) {
  .card {
    display: flex;
    flex-direction: row;
  }

  .card-image {
    width: 30%;
  }
}

@container card (max-width: 399px) {
```

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

  .card-image {
    width: 100%;
  }
}
```

Setting Up Container Queries

- **`container-type`**: Establishes an element as a query container. Values include `size`, `inline-size`, and `normal`.
- **`container-name`**: Assigns a name to the container for targeting in container queries.
- **`container`**: Shorthand property for setting both type and name.

You can nest container queries for complex layouts that respond to multiple container contexts:

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

.inner-container {
  container-type: inline-size;
}

@container (min-width: 700px) {
  /* Styles for elements in containers at least 700px wide */

  @container (min-width: 400px) {
    /* Styles for elements in nested containers at least 400px wide */
  }
}
```

Container queries are now supported in all major modern browsers. You can use feature queries to provide fallbacks:

```
@supports (container-type: inline-size) {
  /* Container query styles */
}

@supports not (container-type: inline-size) {
  /* Fallback styles */
}
```

Container queries complement media queries rather than replace them. Use media queries for page-level layouts and container queries for component-level responsiveness.

CSS Scroll Snap

Scroll snap allows you to create smooth, controlled scrolling experiences with precise stopping points. It's perfect for carousels, galleries, and full-page sections.

Have you ever tried to scroll through a carousel and wished it would automatically align each item perfectly? Or scrolled through a presentation where slides didn't quite line up? Scroll snap solves this by creating magnetic stopping points. When users scroll, the browser automatically adjusts the final position to align with your defined snap points, creating a polished, app-like experience without any JavaScript.

Scroll Snap Container

Define the scroll container and snap behavior:

The container is where you define the snap rules. You tell it which axis to snap on (horizontal, vertical, or both) and how aggressively to snap. The children elements then define where the snap points are.

```
/* Basic horizontal scroll snap */
.carousel {
  scroll-snap-type: x mandatory;
  overflow-x: scroll;
  display: flex;
}
```

This creates a horizontal carousel. The `x` means we're snapping horizontally, and `mandatory` means the browser will always snap to a point - even small scrolls will jump to the nearest snap position. The container needs `overflow-x: scroll` to be scrollable.

```
/* Vertical scroll snap */
.vertical-sections {
  scroll-snap-type: y mandatory;
  overflow-y: scroll;
  height: 100vh;
}
```

Perfect for full-page sections that should fill the viewport. Users scroll vertically, and each section snaps into full view. The `100vh` height ensures we see one section at a time.

```
/* Proximity vs mandatory */
.gentle-snap {
  scroll-snap-type: x proximity; /* Snaps only when close */
```

```
}

.strict-snap {
  scroll-snap-type: x mandatory; /* Always snaps */
}
```

`proximity` is more subtle - it only snaps when the user stops scrolling near a snap point. This feels more natural for content where exact alignment isn't critical. `mandatory` always forces a snap, which is better for slideshows or image galleries where you always want perfect alignment.

```
/* Both axes */
.matrix {
  scroll-snap-type: both mandatory;
  overflow: scroll;
}
```

Scroll Snap Alignment

Control where items snap to:

```
/* Snap to start of container */
.item {
  scroll-snap-align: start;
}

/* Snap to center */
.centered-item {
  scroll-snap-align: center;
}

/* Snap to end */
.end-item {
  scroll-snap-align: end;
}

/* Different alignment for each axis */
.custom-align {
  scroll-snap-align: start center;
  /* x-axis: start, y-axis: center */
}
```

Scroll Snap Stop

Control whether scrolling can skip snap points:

```
/* Prevent skipping this snap point */
.important-slide {
  scroll-snap-stop: always;
}

/* Allow skipping (default) */
.skippable {
  scroll-snap-stop: normal;
}
```

Scroll Padding and Margin

Add spacing around snap points:

```
/* Padding on the container */
.container {
  scroll-snap-type: x mandatory;
  scroll-padding: 20px; /* All sides */
  /* Or individual sides */
  scroll-padding-left: 50px;
  scroll-padding-right: 50px;
}

/* Margin on items */
.item {
  scroll-snap-align: start;
  scroll-margin: 10px; /* Space before snap point */
}

/* Visual indicator space */
.with-indicator {
  scroll-snap-type: y mandatory;
  scroll-padding-top: 80px; /* Space for fixed header */
}
```

Practical Examples

Image Carousel

```
.carousel {
  display: flex;
  overflow-x: auto;
  scroll-snap-type: x mandatory;
  scroll-behavior: smooth;
  gap: 20px;
  padding: 20px;
}
```

```
.carousel img {
  flex: 0 0 auto;
  width: 300px;
  height: 200px;
  object-fit: cover;
  scroll-snap-align: center;
  border-radius: 10px;
}

/* Hide scrollbar but keep functionality */
.carousel {
  scrollbar-width: none; /* Firefox */
  -webkit-overflow-scrolling: touch; /* Smooth iOS scrolling */
}

.carousel::-webkit-scrollbar {
  display: none; /* Chrome, Safari */
}
```

Full Page Sections

```
.fullpage-container {
  height: 100vh;
  overflow-y: auto;
  scroll-snap-type: y mandatory;
  scroll-behavior: smooth;
}

.section {
  height: 100vh;
  scroll-snap-align: start;
  display: flex;
  align-items: center;
  justify-content: center;
}

/* With navigation dots */
.section:nth-child(odd) {
  background: linear-gradient(135deg, #667eea, #764ba2);
}

.section:nth-child(even) {
  background: linear-gradient(135deg, #f093fb, #f5576c);
}
```

Product Gallery

```
.product-gallery {
  display: grid;
  grid-auto-flow: column;
  grid-auto-columns: min(100%, 400px);
  overflow-x: auto;
  scroll-snap-type: x proximity;
  scroll-padding: 20px;
  gap: 20px;
}

.product-card {
  scroll-snap-align: start;
  background: white;
  border-radius: 12px;
  padding: 20px;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}
```

Mobile-Friendly Tabs

```
.tab-container {
  display: flex;
  overflow-x: auto;
  scroll-snap-type: x mandatory;
  border-bottom: 2px solid #e0e0e0;
}

.tab {
  flex: 0 0 auto;
  padding: 15px 30px;
  scroll-snap-align: start;
  white-space: nowrap;
  cursor: pointer;
  transition: background 0.3s;
}

.tab.active {
  border-bottom: 2px solid #007bff;
  background: rgba(0, 123, 255, 0.1);
}
```

Timeline

```
.timeline {
  display: flex;
  overflow-x: auto;
  scroll-snap-type: x mandatory;
```

```
  padding: 50px 0;
  position: relative;
}

.timeline::before {
  content: '';
  position: absolute;
  top: 50%;
  left: 0;
  right: 0;
  height: 2px;
  background: #ddd;
}

.timeline-item {
  flex: 0 0 auto;
  width: 250px;
  scroll-snap-align: center;
  padding: 20px;
  text-align: center;
}

.timeline-item::before {
  content: '';
  width: 20px;
  height: 20px;
  background: #007bff;
  border-radius: 50%;
  margin: 0 auto 20px;
  display: block;
}
```

Best Practices

1. **Always provide visual feedback** - Show scroll indicators or progress
2. **Test on touch devices** - Ensure smooth touch scrolling
3. **Consider accessibility** - Provide keyboard navigation alternatives
4. **Use scroll-behavior: smooth** - For programmatic scrolling
5. **Performance** - Scroll snap is hardware-accelerated and performant

```
/* Smooth scrolling for the entire page */
html {
  scroll-behavior: smooth;
}

/* Respect user preferences */
@media (prefers-reduced-motion: reduce) {
```

```
  html {
    scroll-behavior: auto;
  }

  .carousel {
    scroll-snap-type: none;
  }
}
```

Scroll snap provides a native CSS solution for creating engaging, app-like scrolling experiences without JavaScript.

Conclusion

Throughout this handbook, we've explored the fundamental principles of CSS that form the backbone of modern web design. From basic selectors and the box model to more advanced concepts like media queries, feature detection, and container queries, you now have a solid foundation to build upon.

CSS is constantly evolving, with new features being developed and implemented across browsers regularly. What makes CSS powerful is not just knowing the individual properties and values, but understanding how they work together to create coherent, responsive, and accessible web experiences.

Remember that mastering CSS is a journey rather than a destination. The best way to improve is through practice—building real projects, experimenting with different techniques, and staying curious about new developments in the field.

Thank you for exploring this CSS handbook. May your styles be clean, your selectors specific, and your layouts never break!