CSS Handbook

Flavio Copes

Updated July 31, 2026

Contents

Preface

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

This book was first published 2019.

Last updated in 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 it should render the elements on the page.

A CSS file contains several CSS rules.

Each rule is composed of 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 of 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 block**. The declaration block contains various **declarations**, each composed of a **property** and a **value**.

In this example, `p` is the selector, and applies one declaration 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

CSS was created to separate presentation from HTML structure. Before CSS (introduced in 1996), web pages used presentational HTML tags like `<font>` and `<center>`, which mixed content with styling. CSS evolved from Level 1 to Level 2, and now follows a modular approach with new features continuously added. Today, CSS has excellent cross-browser support and is more powerful than ever.

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;
```

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.

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

Pseudo class	Targets
`: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.

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 of 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 significantly by device pixel density.

There is a convention that makes 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);
}
```

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`.

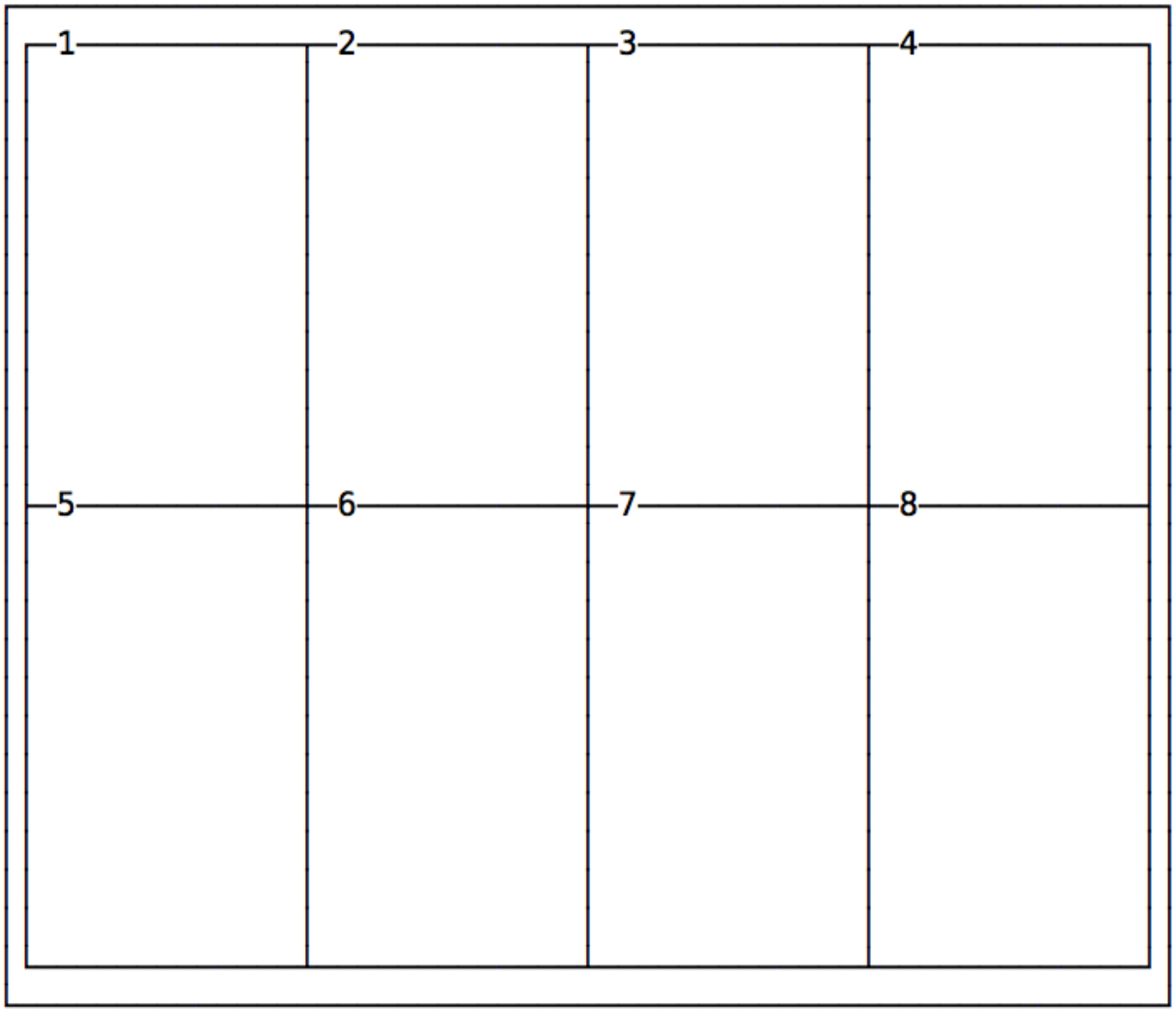
Then we'll talk about how to load custom fonts, using `@import` or `@font-face`, or by loading a font stylesheet.

`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> <fo
```

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

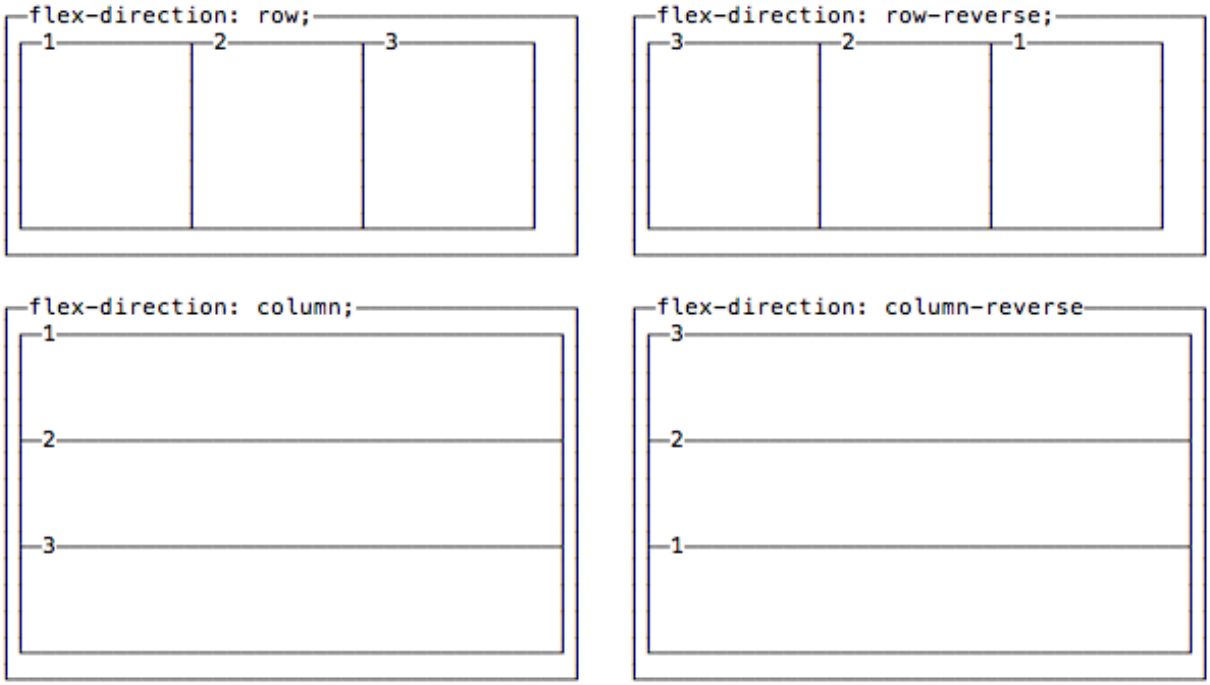
Every CSS element is essentially a box. Every element is a generic box.

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`

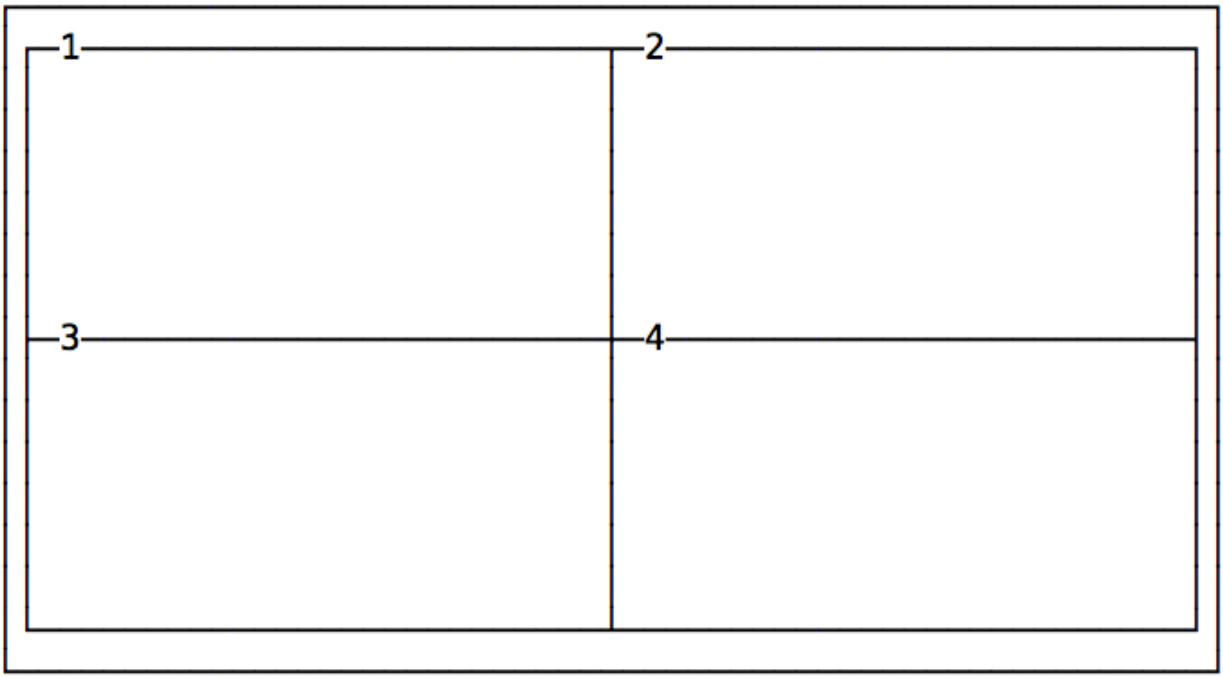
- `border-image-repeat`

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.

The `margin: 0 auto;` technique remains useful for simple centering needs or when working with legacy codebases.

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;
}
```

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.

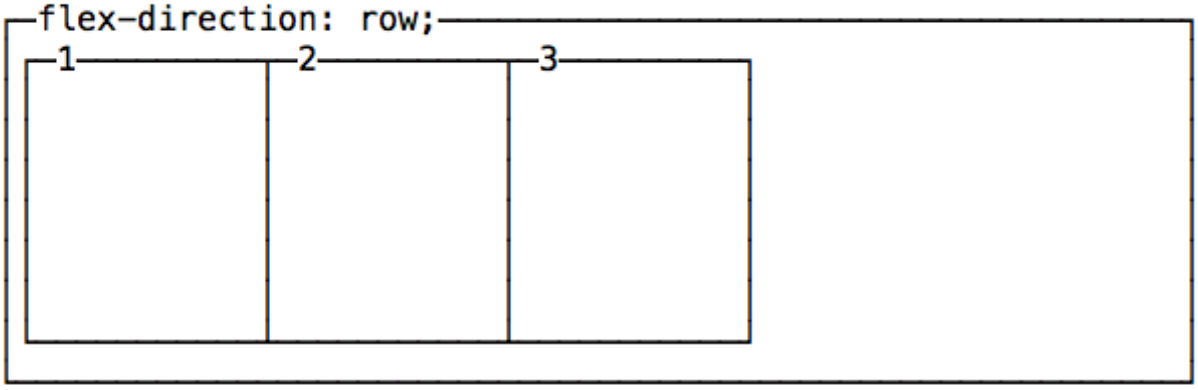
Take this example I made on Codepen. I create a parent container, a child container, and an inner box with some text:

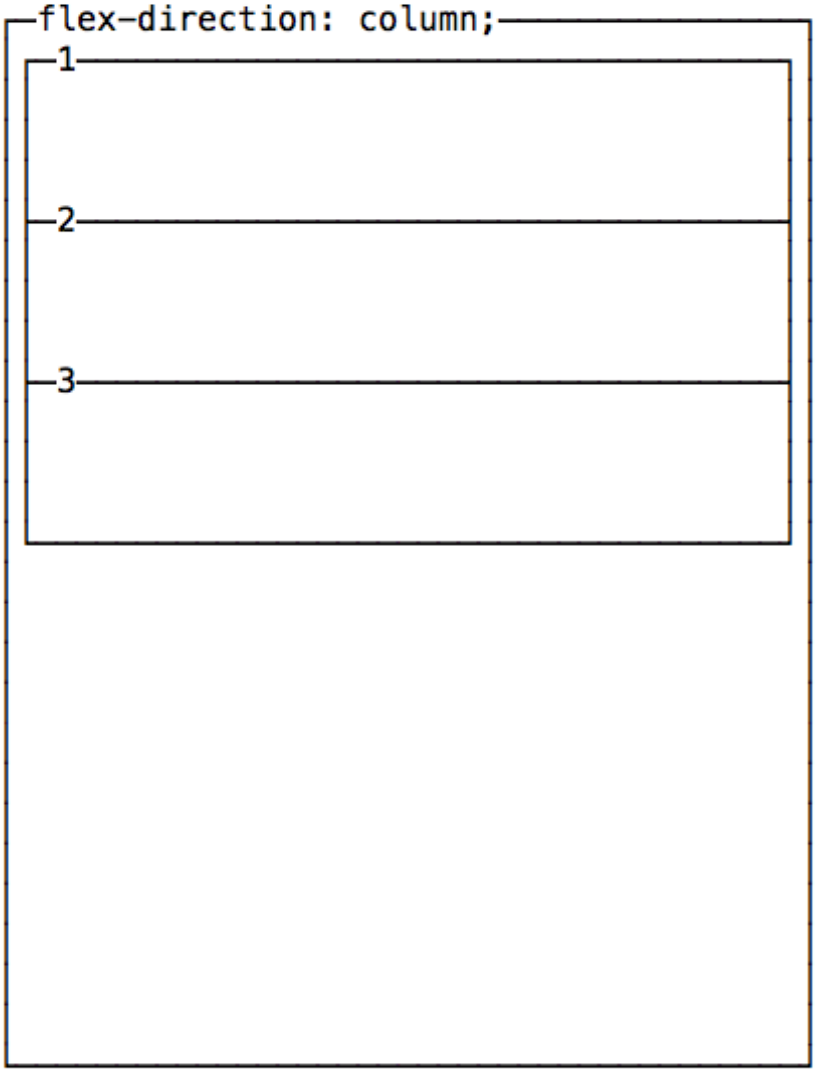
```
<div class="parent">
  <div class="child">
    <div class="box">
      <p>Test</p>
    </div>
  </div>
</div>
```

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 New', monospace;
  text-align: center;
  font-size: 2rem
}
```

here's the result:



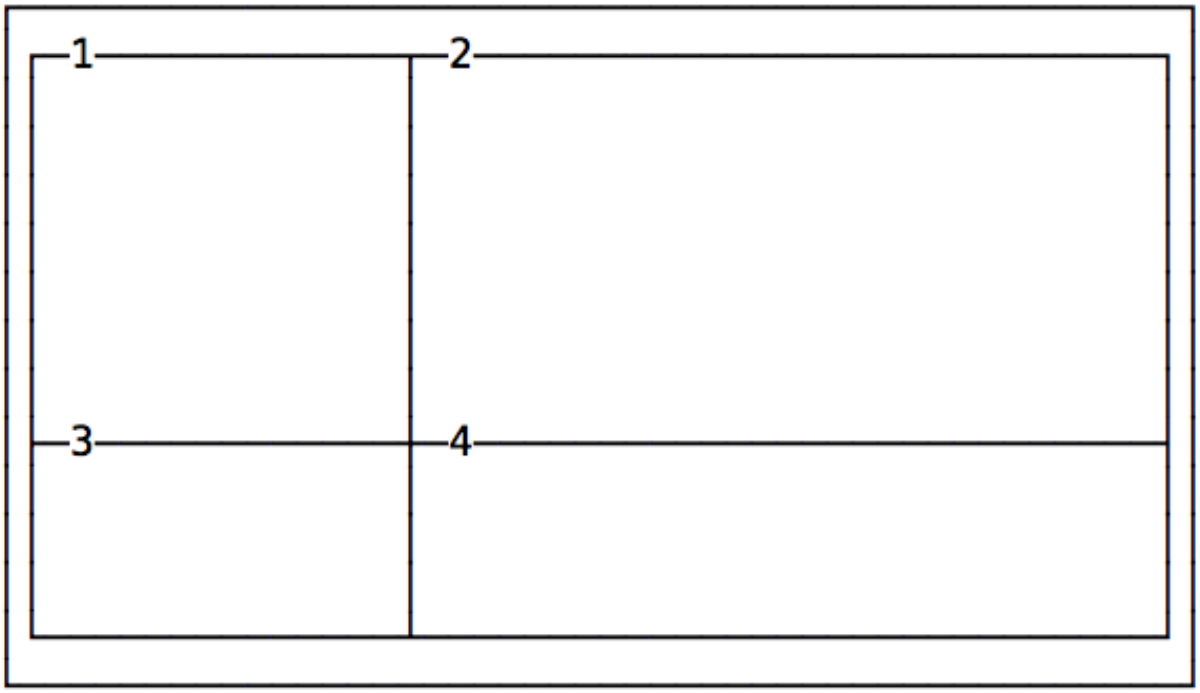


You can try and add any of the properties I mentioned before (`top`, `right`, `bottom`, `left`) to `.box`, and nothing will happen. The position is `static`.

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:

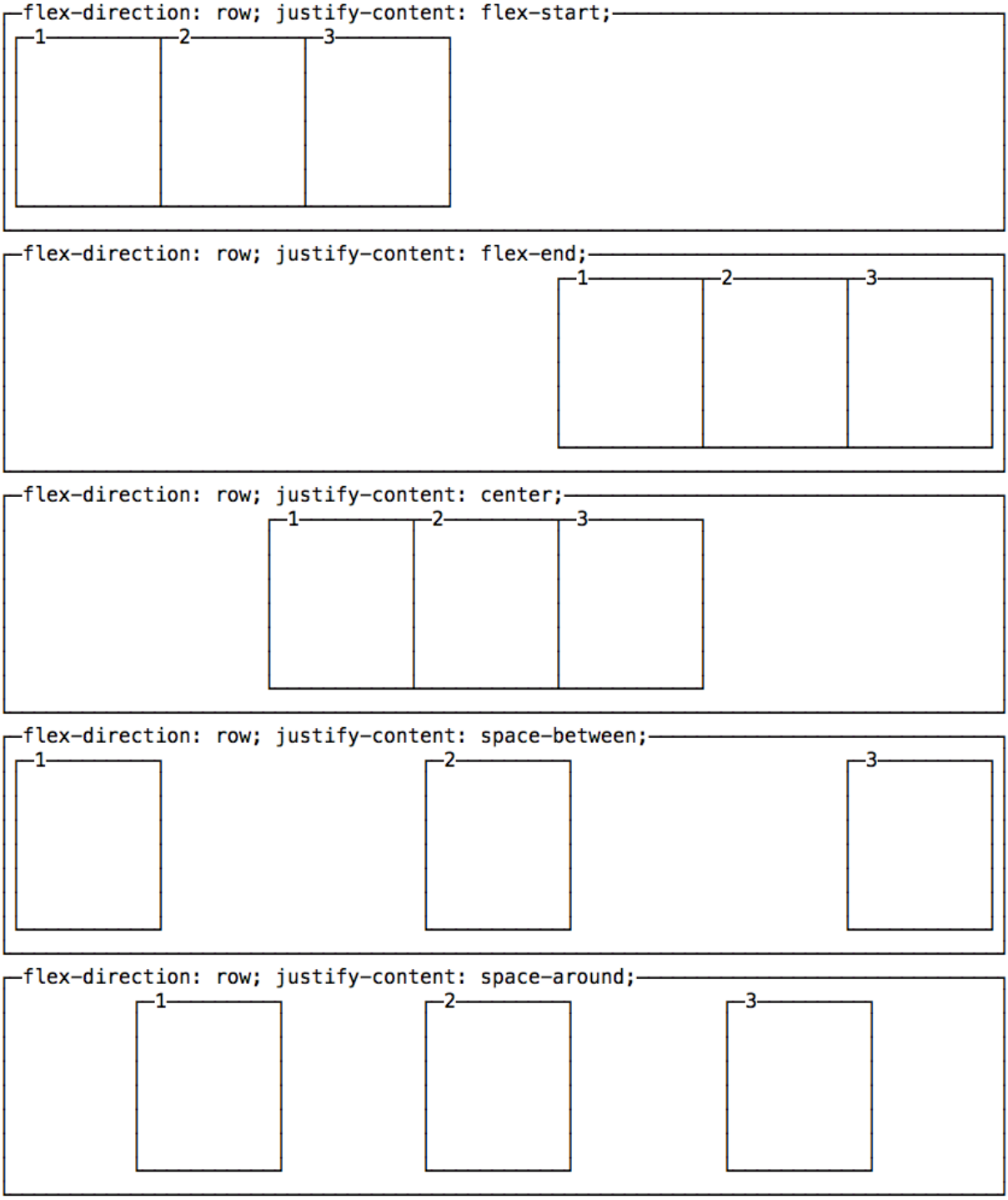
```
.box {
  /* ... */
  position: relative;
  top: -60px;
}
```



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

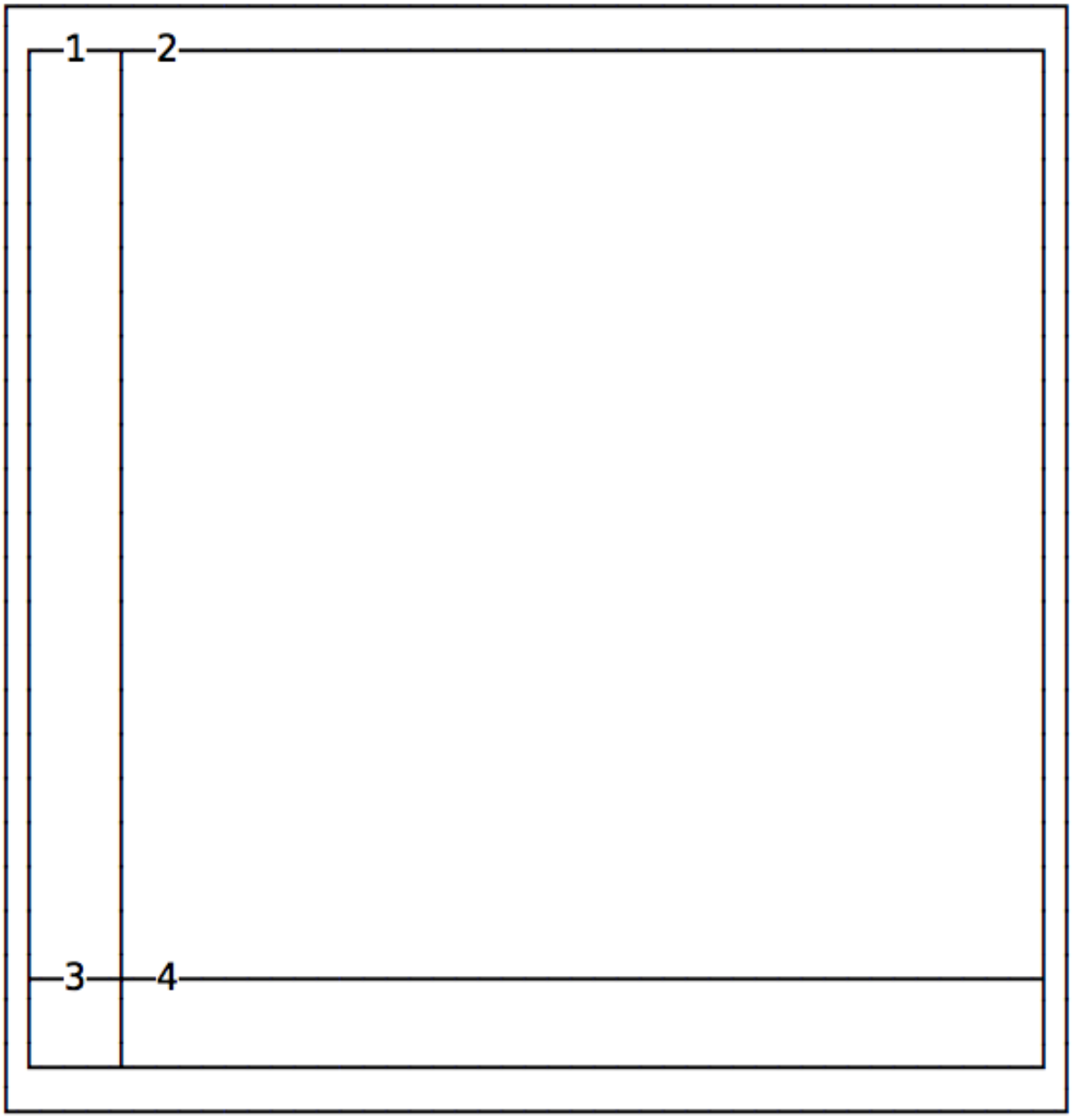
Setting `position: absolute` on an element will remove it from the document's flow, and it will not longer follow the original page positioning flow.

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.

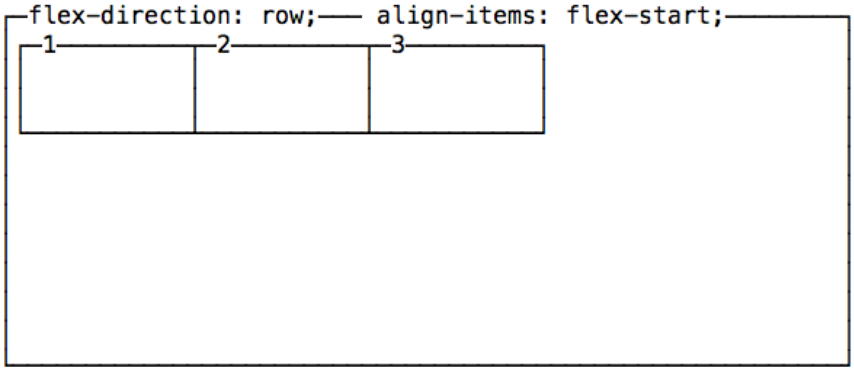
```
.box {
  /* ... */
  position: absolute;
}
```

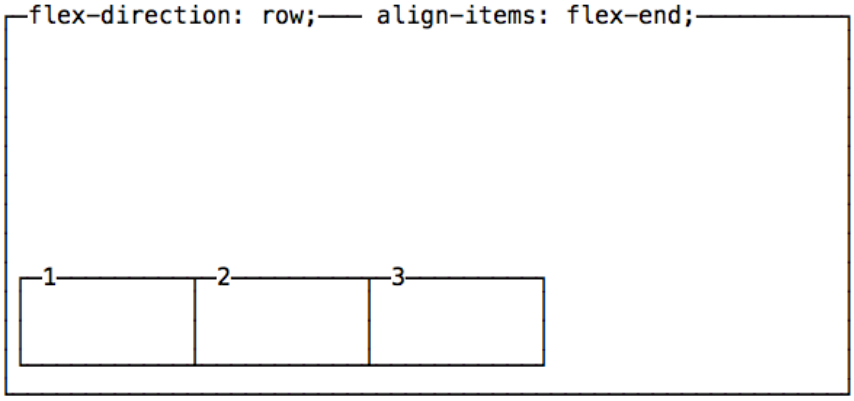


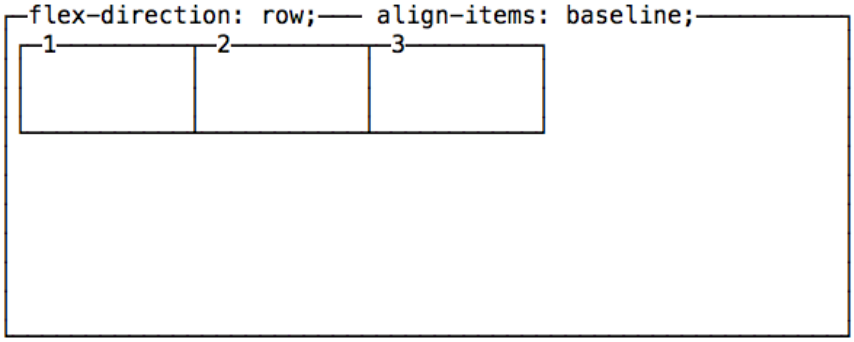
We can now move the box around as we please, using the `top`, `right`, `bottom`, `left` properties:

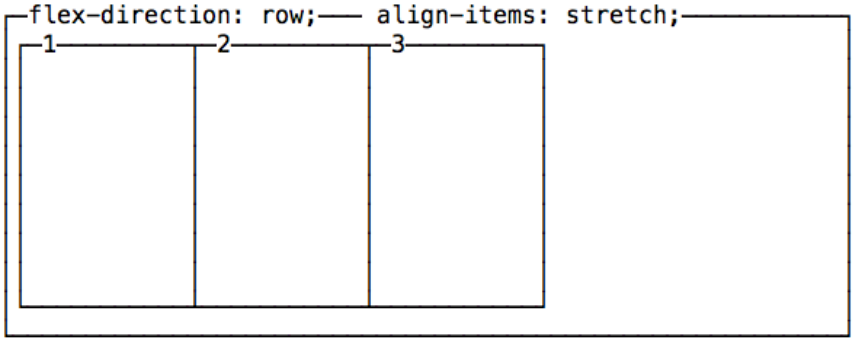
```
.box {
  /* ... */
  position: absolute;
```

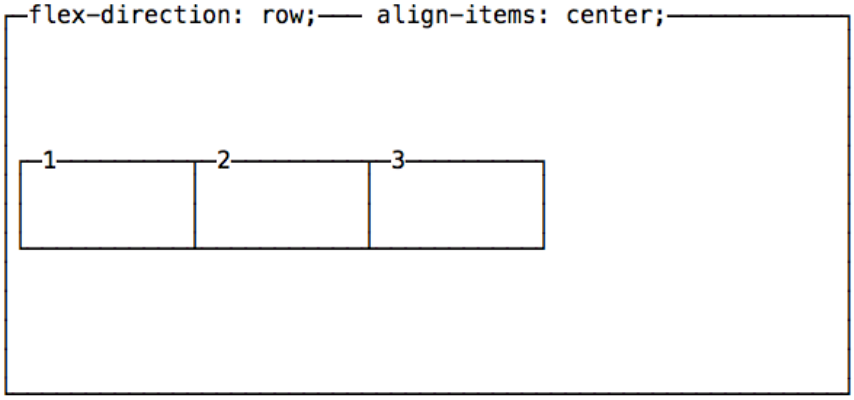
```
  top: 0px;
  left: 0px;
}
```


flex-direction: row; align-items: flex-start;
1
2
3


flex-direction: row; align-items: flex-end;
1
2
3

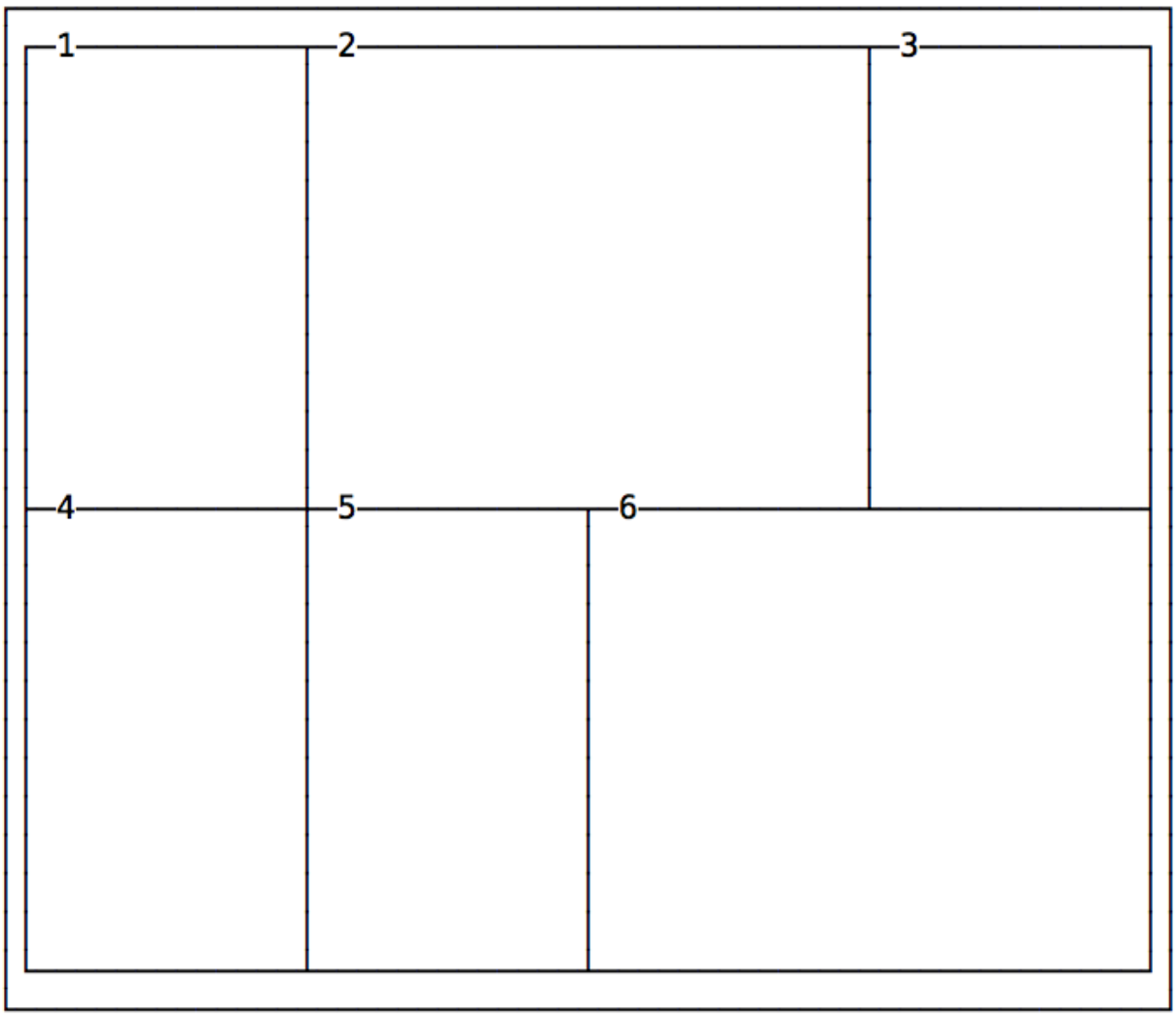

flex-direction: row; align-items: baseline;
1
2
3


flex-direction: row; align-items: stretch;
1
2
3


flex-direction: row; align-items: center;
1
2
3

or

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



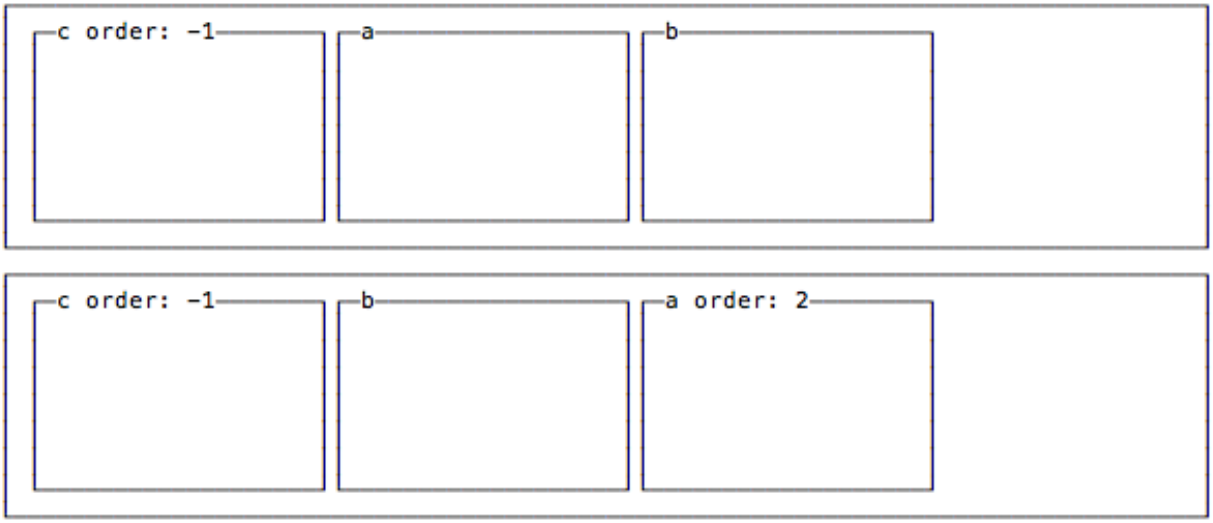
The coordinates are relative to the closest container that is not `static`.

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 {
```

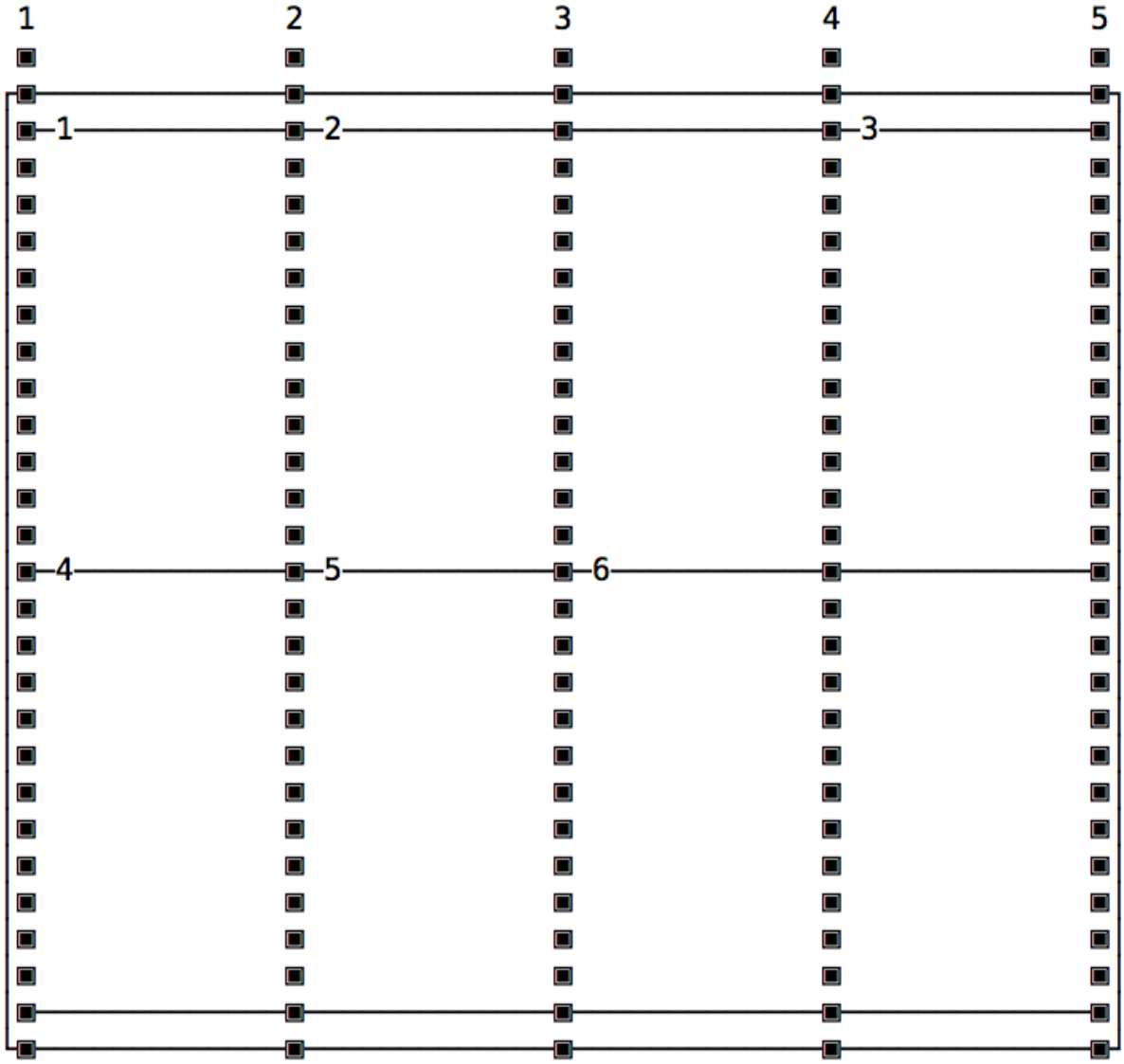
```
  /* ... */
  position: absolute;
  top: 0px;
  left: 0px;
}
```



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

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

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



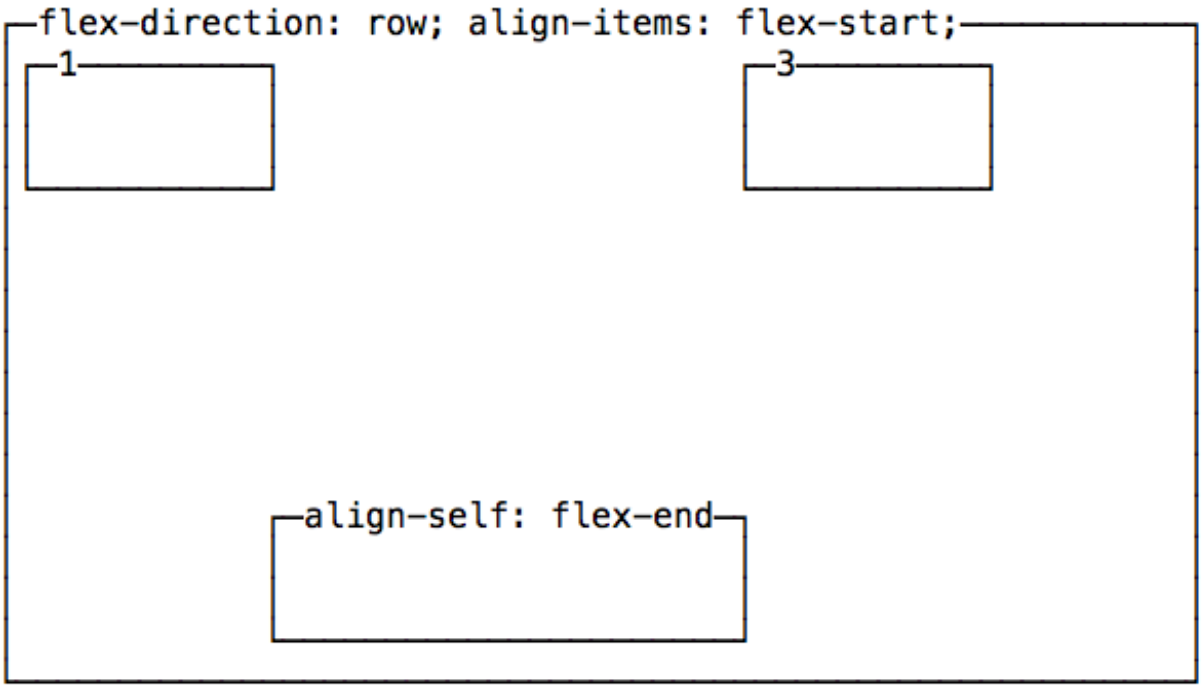
Like for relative positioning, you can use `z-index` to alter the z-axis placement.

Fixed positioning

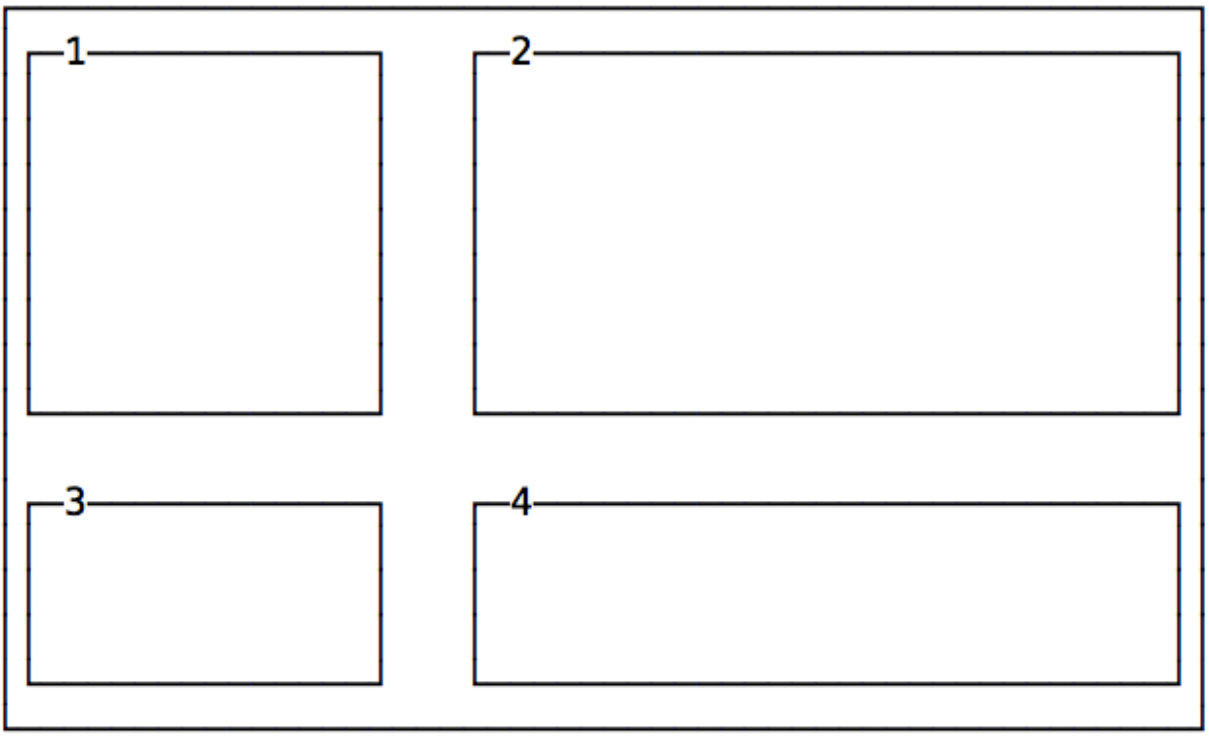
Like with absolute positioning, when an element is assigned `position: fixed` it's removed from the flow of the page.

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, `position: sticky` is a newer addition that now has excellent browser support.

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 stuck 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.

Floating and clearing

The `float` property was historically used for layouts but is now primarily used for its original purpose: making content flow around elements like images in text.

The `float` property accepts these values:

- `left` - element floats to the left
- `right` - element floats to the right
- `none` - default, no floating

Basic example:

```
img {
  float: left;
  margin: 0 20px 20px 0;
}
```

A floated element is removed from normal document flow, allowing other content to wrap around it. For modern layouts, use Flexbox or Grid instead.

Clearing

The `clear` property controls how an element behaves next to floated elements:

- `left` - clears left floats
- `right` - clears right floats
- `both` - clears both left and right floats
- `none` - default, no clearing

```
.clearfix {
  clear: both;
}
```

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.

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

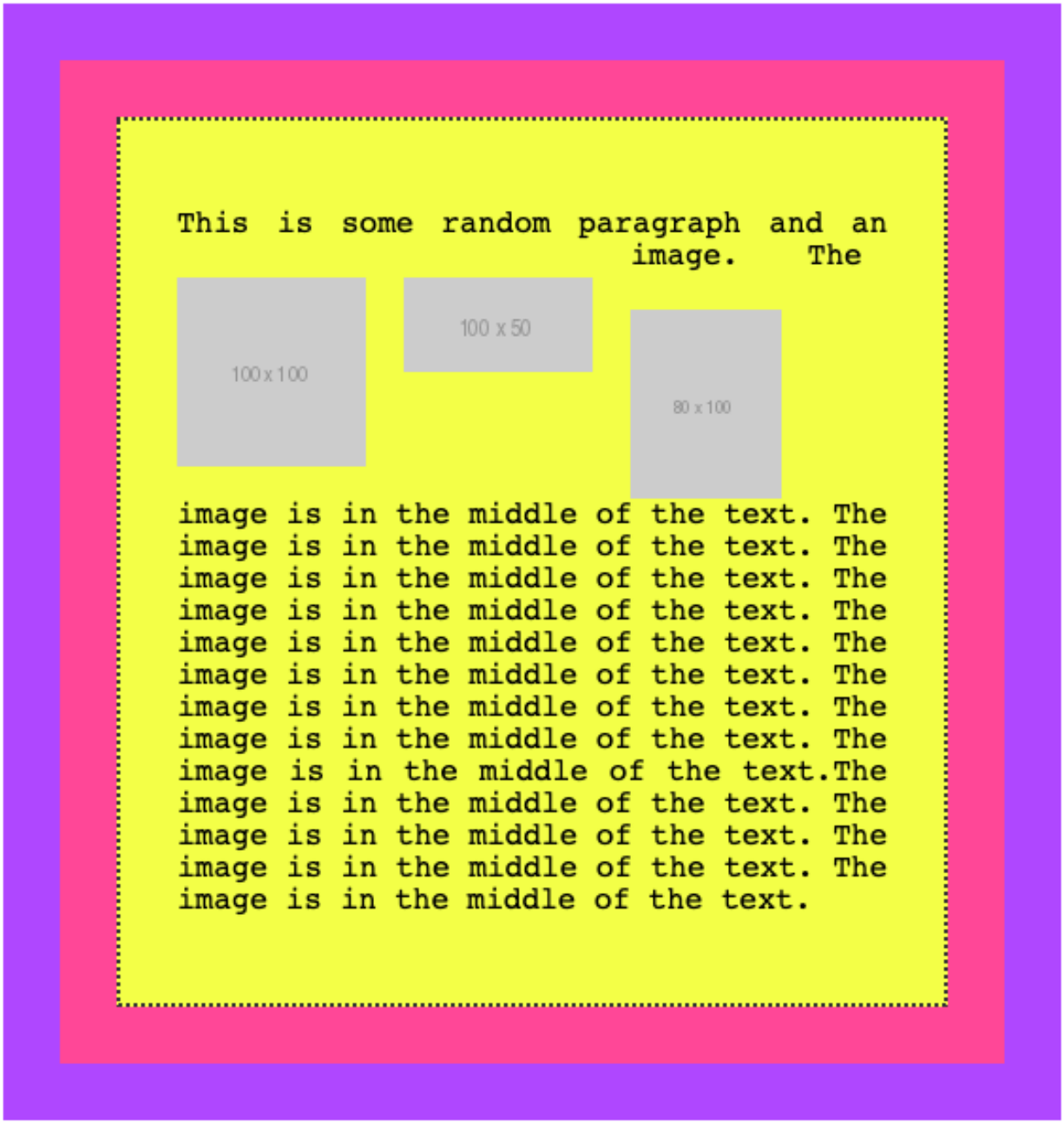
Some flexbox properties apply to the container, which sets the general rules for its items. They are

- `flex-direction`
- `justify-content`
- `align-items`
- `flex-wrap`
- `flex-flow`

Align rows or columns

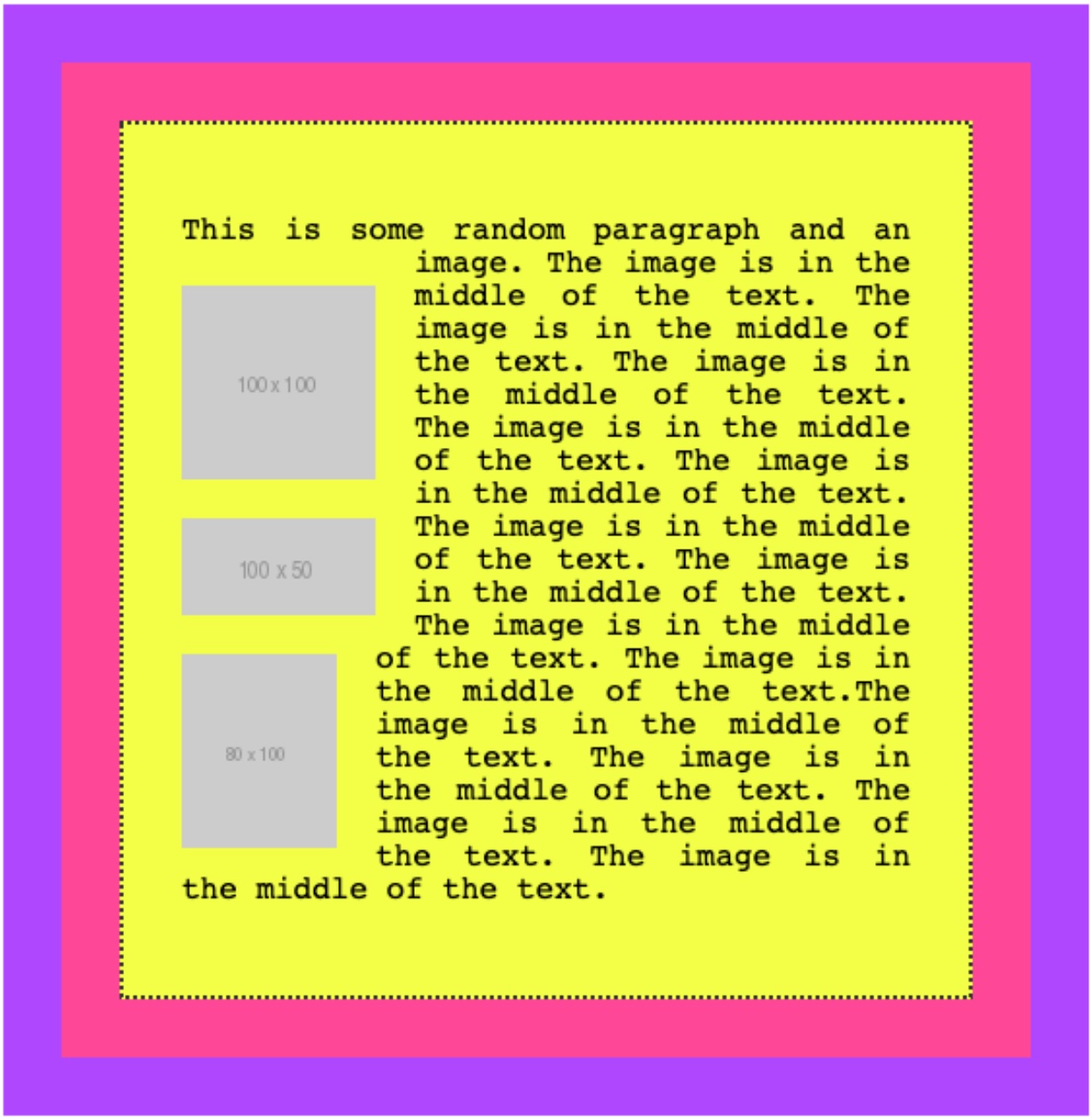
The first property we see, **`flex-direction`**, determines if the container should align its items as rows, or as columns:

- `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`.

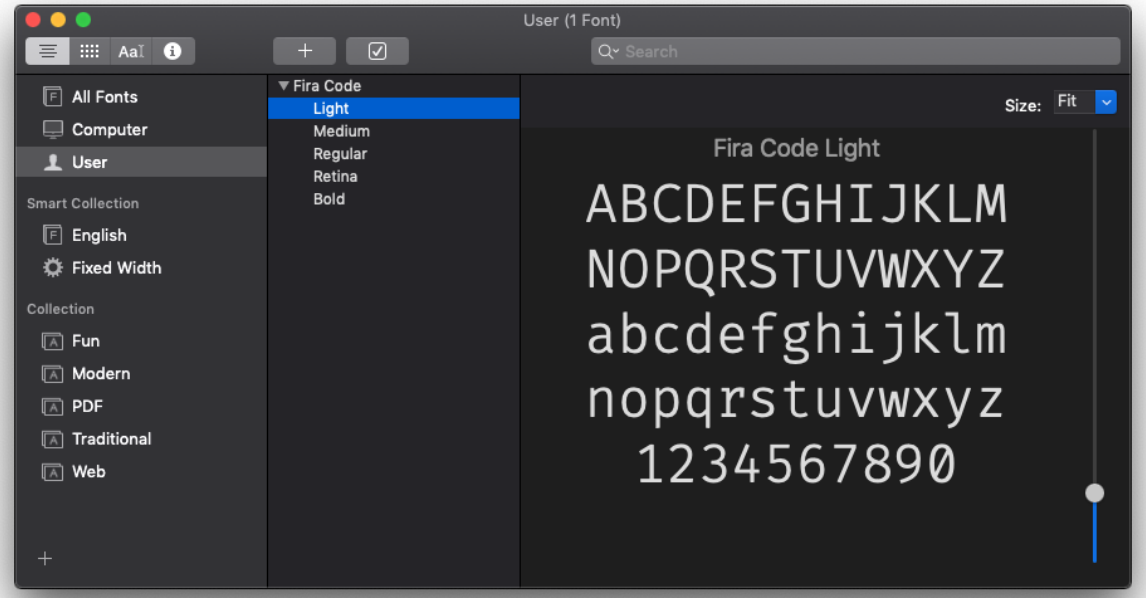


You can change this behavior using `justify-content` to change the horizontal alignment, and `align-items` to change the vertical alignment.

Change the horizontal alignment

`justify-content` has 5 possible values:

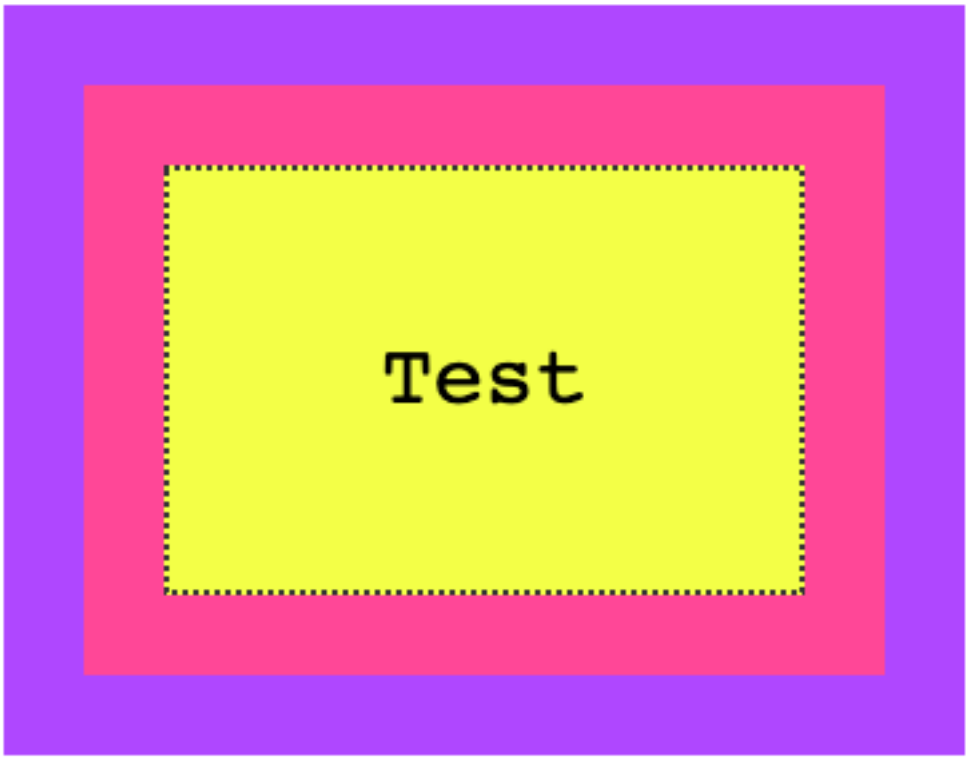
- `flex-start`: align to the left side of the container.
- `flex-end`: align to the right side of the container.
- `center`: align at the center of the container.
- `space-between`: display with equal spacing between them.
- `space-around`: display with equal spacing around them



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.



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`.

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:

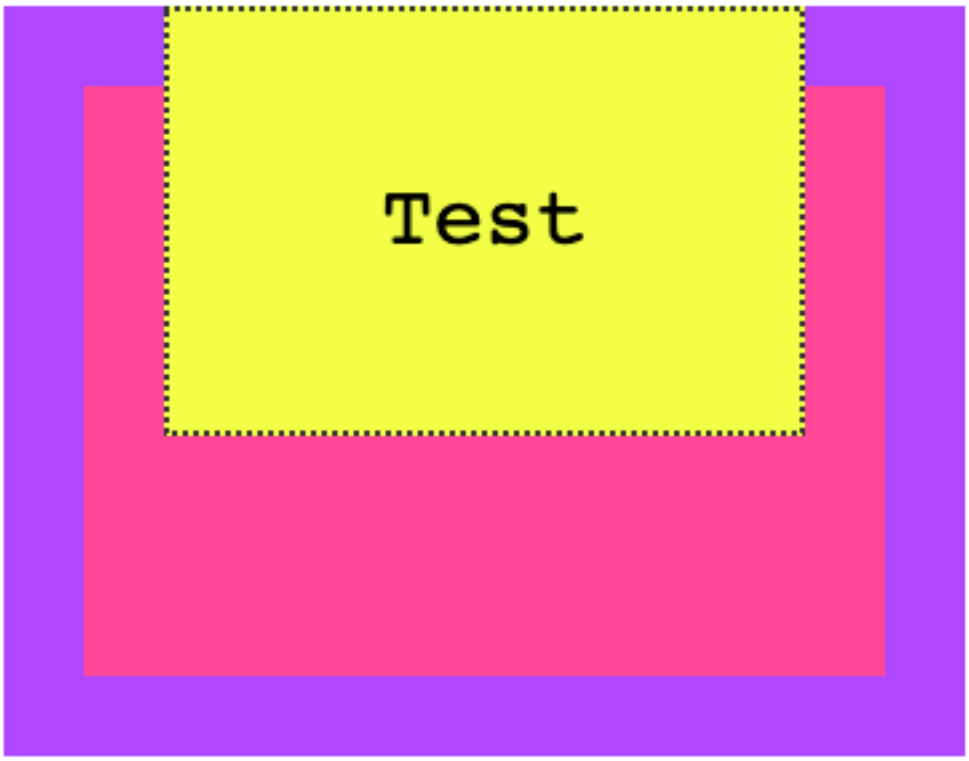
- `order`
- `align-self`
- `flex-grow`
- `flex-shrink`
- `flex-basis`
- `flex`

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.

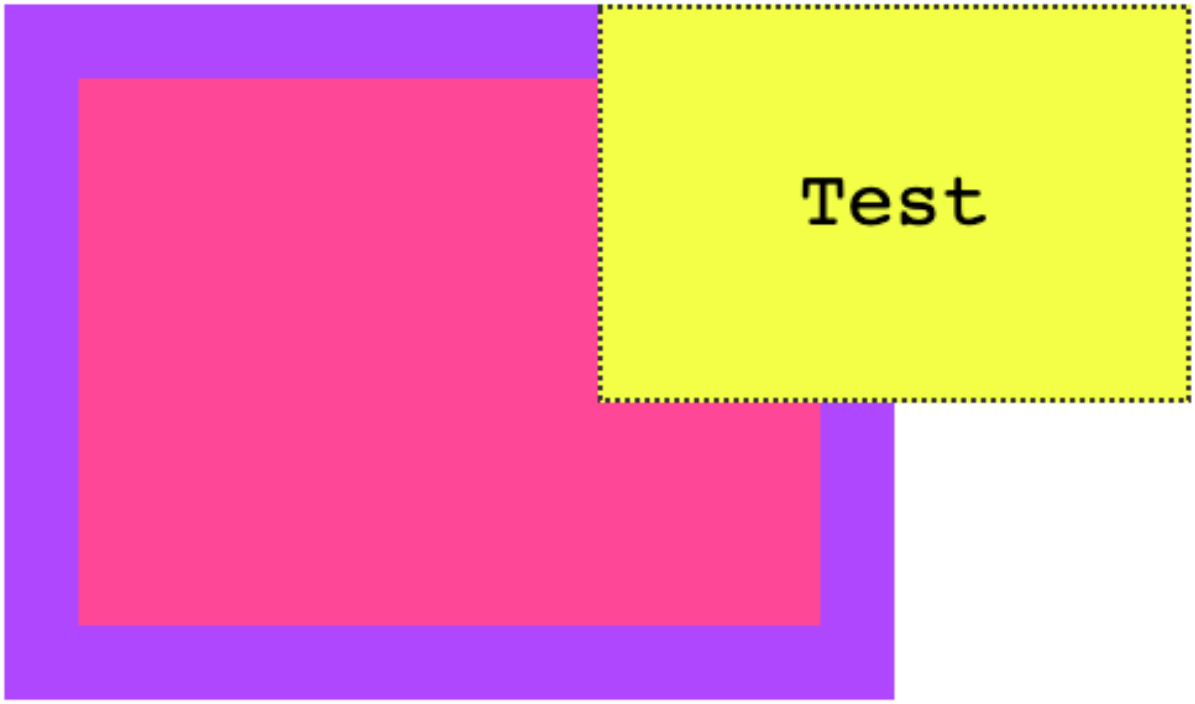
You can override this property using `order` on each separate item. This is a property you set on the item, not the container. You can make an item appear before all the others by setting a negative value.



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`

Common Flexbox Patterns

Center Content

```
.center {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
}
```

Equal Width Columns

```
.columns {
  display: flex;
}
```

```
.column {
  flex: 1;
}
```

Navigation Bar

```
.navbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
}
.nav-links {
  display: flex;
  gap: 20px;
}
```

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

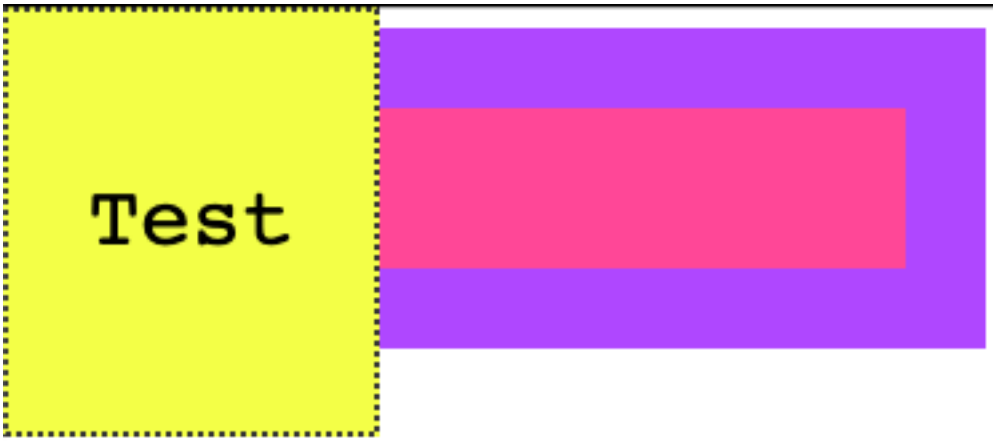
Those properties define the number of columns and rows in the grid, and they also set the width of each column/row.

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

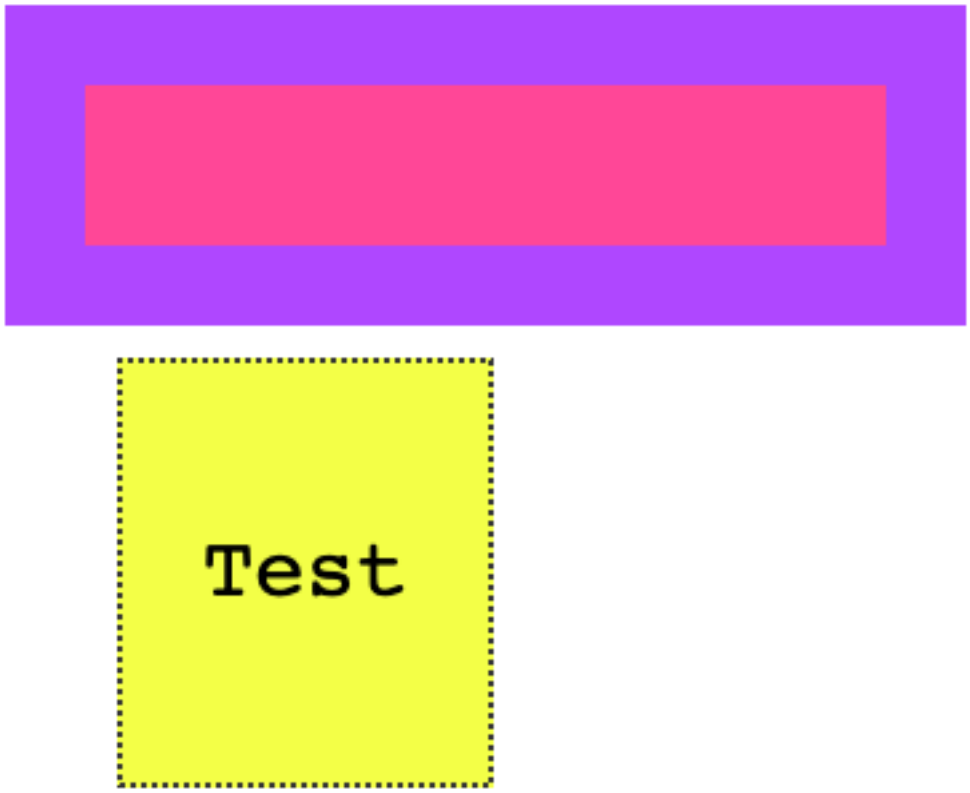
In the above examples we made pretty, regular grids by using the same values for rows and the same values for columns.

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 .";
}
```

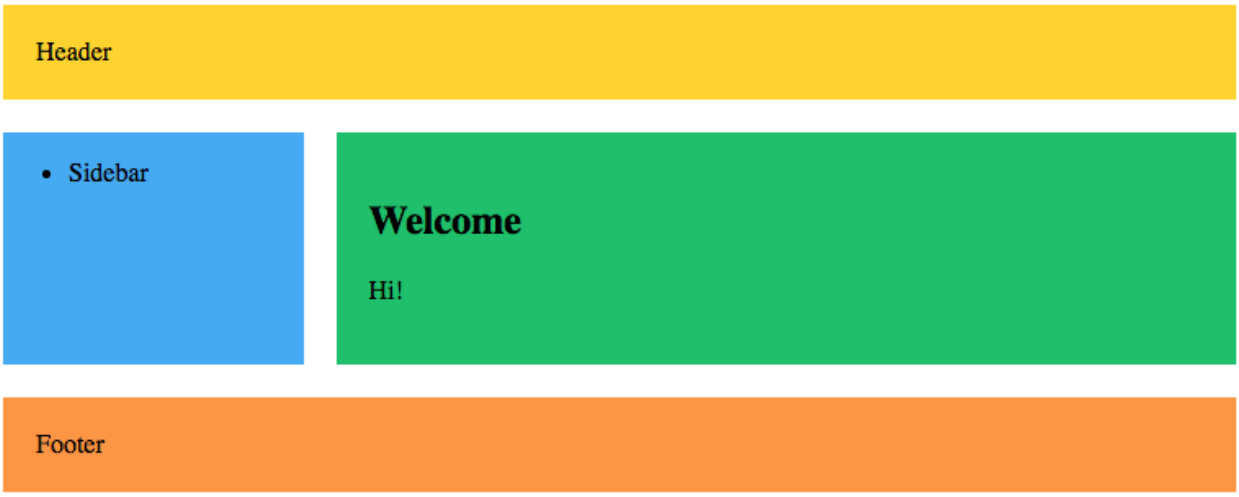
Fill a page with a grid

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";
  }
}
```

Common Grid Patterns

Card Layout

```
.cards {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
  gap: 20px;
}
```

Dashboard Layout

```
.dashboard {
  display: grid;
  grid-template-columns: 250px 1fr;
  grid-template-rows: 60px 1fr;
  grid-template-areas:
    "sidebar header"
```

```
    "sidebar main";
  height: 100vh;
}
.sidebar { grid-area: sidebar; }
.header { grid-area: header; }
.main { grid-area: main; }
```

Holy Grail Layout

```
.layout {
  display: grid;
  grid-template-rows: auto 1fr auto;
  grid-template-columns: 200px 1fr 200px;
  min-height: 100vh;
}
.header { grid-column: 1 / -1; }
.footer { grid-column: 1 / -1; }
```

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 help 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, red);
}
```

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);
}
```

Modern CSS Features

CSS Nesting

CSS now supports nesting selectors directly, similar to preprocessors like Sass:

```
.card {
  padding: 20px;
  border-radius: 8px;

  &:hover {
    transform: scale(1.02);
  }

  .title {
    font-size: 1.5rem;
    margin-bottom: 10px;
  }
}
```

CSS Logical Properties

Use logical properties for better internationalization support:

```
.element {
  margin-inline: 20px; /* instead of margin-left/right */
  padding-block: 10px; /* instead of padding-top/bottom */
  border-inline-start: 1px solid #ccc; /* instead of border-left */
}
```

Modern Color Functions

Advanced color manipulation with new functions:

```
.modern-colors {
  /* OKLCH for better color spaces */
  color: oklch(70% 0.25 180deg);

  /* Color mixing */
  background: color-mix(in srgb, blue 30%, white);
```

```
  /* Relative colors */
  border-color: rgb(from var(--primary) r g b / 0.5);
}
```

Container Queries

Style elements based on their container size, not viewport:

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

@container (min-width: 400px) {
  .card {
    display: grid;
    grid-template-columns: 1fr 2fr;
  }
}
```

CSS Cascade Layers

Control specificity with layers:

```
@layer base, components, utilities;

@layer base {
  h1 { font-size: 2rem; }
}

@layer components {
  .heading { font-size: 1.5rem; }
}
```

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: port
```

`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.

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 Best Practices

Organization & Architecture

Use Consistent Naming Conventions

```
/* BEM (Block Element Modifier) */
.card {}
.card__title {}
.card__title--large {}
.card--featured {}
```

Organize Your CSS

```
/* Reset/Base styles first */
* { box-sizing: border-box; }
```

```
/* Layout components */
.container, .grid, .flex {}

/* UI components */
.button, .card, .nav {}

/* Utilities last */
.text-center, .hidden, .mb-4 {}
```

Performance Optimization

Minimize Specificity

```
/* Good - low specificity */
.nav-link { color: blue; }

/* Avoid - high specificity */
header nav ul li a.link { color: blue; }
```

Use Efficient Selectors

```
/* Fast */
.my-class {}
#my-id {}

/* Slower */
div > ul + p {}
[data-attribute*="value"] {}
```

Optimize for Repaints

```
/* Prefer transform/opacity for animations */
.element {
  transition: transform 0.3s ease;
}
.element:hover {
  transform: translateY(-2px);
}
```

Modern CSS Patterns

CSS Custom Properties for Theming

```
:root {
  --primary-color: #007bff;
  --primary-dark: #0056b3;
  --spacing-unit: 8px;
  --border-radius: 4px;
```

```
}

.button {
  background: var(--primary-color);
  padding: calc(var(--spacing-unit) * 2);
  border-radius: var(--border-radius);
}
```

Responsive Design with Clamp

```
.title {
  /* Responsive font size without media queries */
  font-size: clamp(1.25rem, 4vw, 2.5rem);

  /* Responsive spacing */
  margin-bottom: clamp(1rem, 3vw, 2rem);
}
```

Component-Based Approach

```
/* Self-contained component */
.card {
  /* All card styles here */
  padding: 1.5rem;
  border-radius: 8px;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

.card__header {}
.card__body {}
.card__footer {}
```

Accessibility Considerations

Focus Management

```
.button:focus-visible {
  outline: 2px solid var(--primary-color);
  outline-offset: 2px;
}

/* Remove outline for mouse users */
.button:focus:not(:focus-visible) {
  outline: none;
}
```

Color Contrast & Dark Mode

```
@media (prefers-color-scheme: dark) {
  :root {
    --bg-color: #1a1a1a;
    --text-color: #ffffff;
  }
}

/* Ensure sufficient contrast */
.text {
  color: var(--text-color);
  background: var(--bg-color);
}
```

Reduced Motion Support

```
@media (prefers-reduced-motion: reduce) {
  *, *::before, *::after {
    animation-duration: 0.01ms !important;
    animation-iteration-count: 1 !important;
    transition-duration: 0.01ms !important;
  }
}
```

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.

As you continue your CSS journey, focus on:

- **Performance optimization:** Writing efficient CSS that loads quickly and renders smoothly
- **Maintainability:** Creating organized, scalable stylesheets that other developers can understand
- **Accessibility:** Ensuring your designs work for everyone, regardless of their abilities or devices
- **Progressive enhancement:** Building robust layouts that work everywhere but are enhanced for modern browsers

Thank you for exploring this CSS handbook. May your styles be clean, your selectors specific, and

your layouts never break!