**What is CSS?**

As we have mentioned before, CSS is a language for specifying how documents are presented to users — how they are styled, laid out, etc.

### **External stylesheet**

An external stylesheet is when you have your CSS written in a separate file with a .css extension, and you reference it from an HTML [<link>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link) element.

This method is arguably the best, as you can use one stylesheet to style multiple documents, and would only need to update the CSS in one place if changes were needed.

### **Internal stylesheet**

An **internal stylesheet** is where you don't have an external CSS file, but instead place your CSS inside a [<style>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style) element, contained inside the [HTML head](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_HTML)

This can be useful in some circumstances (maybe you're working with a [content management system](https://developer.mozilla.org/en-US/docs/Glossary/CMS) where you can't modify the CSS files directly), but it isn't quite as efficient as external stylesheets — in a website, the CSS would need to be repeated across every page, and updated in multiple places if changes were required.

### **Inline styles**

**Inline styles** are CSS declarations that affect one element only, contained within a [style](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#attr-style) attribute.

Please don't do this, unless you really have to! It is really bad for maintenance (you might have to update the same information multiple times per document), and it also mixes your presentational CSS information with your HTML structural information, making the CSS harder to read and understand. Keeping your different types of code separated and pure makes for a much easier job for all who work on the code.

The only time you might have to resort to using inline styles is when your working environment is really restrictive (perhaps your CMS only allows you to edit the HTML body.)

# CSS syntax

**Note**: CSS is a declarative language, which makes its syntax fairly easy and straightforward to understand. In addition, it also has a very nice error recovery system that allows you to make mistakes without breaking everything — for example declarations that aren't understood are generally just ignored. The downside is that it can be harder to understand where errors are coming from. Read on, and all will become clear eventually.

**Important**: If a property is unknown or if a value is not valid for a given property, the declaration is deemed invalid and is wholly ignored by the browser's CSS engine.

### **CSS statements**

* **At-rules** are used in CSS to convey **metadata, conditional information, or other descriptive information.** They start with an at sign (@), followed by an identifier to say what kind of rule it is, then a syntax block of some kind, ending with a semi-colon (;). Each type of [at-rule](https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule), defined by the identifier, will have its own internal syntax and semantics. Examples include:
  + [@charset](https://developer.mozilla.org/en-US/docs/Web/CSS/@charset) and [@import](https://developer.mozilla.org/en-US/docs/Web/CSS/@import) (metadata)
  + [@media](https://developer.mozilla.org/en-US/docs/Web/CSS/@media) or [@document](https://developer.mozilla.org/en-US/docs/Web/CSS/@document) (conditional information, also called nested statements, see below.)
  + [@font-face](https://developer.mozilla.org/en-US/docs/Web/CSS/@font-face) (descriptive information)

Specific  syntax example:

@import 'custom.css';

This at-rule imports another CSS file into the current CSS.

* **Nested statements** are a specific subset of at-rule, the syntax of which is a nested block of CSS rules that will only be applied to the document if a specific condition is matched:
  + The [@media](https://developer.mozilla.org/en-US/docs/Web/CSS/@media) at-rule content is applied only if the device which runs the browser matches the expressed condition;
  + the [@supports](https://developer.mozilla.org/en-US/docs/Web/CSS/@supports) at-rule content is applied only if the browser actually supports the tested feature;
  + the [@document](https://developer.mozilla.org/en-US/docs/Web/CSS/@document) at-rule content is applied only if the current page matches some conditions.

Specific syntax example

@media (min-width: 801px) {

body {

margin: 0 auto;

width: 800px;

}

}

The above nested statement only applies the nested rule when the page's width exceeds 800 pixels.

## Beyond syntax: make CSS readable

### **White space**

White space means actual spaces, tabs and new lines. You can add white space to make your style sheets more readable.

### **Comments**

Comments in CSS begin with /\* and end with \*/.

### **Shorthand**

Some properties like [font](https://developer.mozilla.org/en-US/docs/Web/CSS/font), [background](https://developer.mozilla.org/en-US/docs/Web/CSS/background), [padding](https://developer.mozilla.org/en-US/docs/Web/CSS/padding), [border](https://developer.mozilla.org/en-US/docs/Web/CSS/border), and [margin](https://developer.mozilla.org/en-US/docs/Web/CSS/margin) are called **shorthand properties** — this is because they allow you to set several property values in a single line, saving time and making your code neater in the process.

* ***in shorthand like padding and margin, the values are applied in the order top, right, bottom, left (the same order as an analog clock). There are also other***
* ***shorthand types, for example two values, which set for example, the padding for top/bottom, then left/right***

For example, this line:

/\* in shorthand like padding and margin, the values are applied

in the order top, right, bottom, left (the same order as an analog clock). There are also other

shorthand types, for example two values, which set for example

the padding for top/bottom, then left/right \*/

padding: 10px 15px 15px 5px;

Does the same thing as all these:

padding-top: 10px;

padding-right: 15px;

padding-bottom: 15px;

padding-left: 5px;

### **Different types of selectors**

* **Simple selectors**: Match one or more elements based on element type, [class](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#attr-class), or [id](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#attr-id).
* **Attribute selectors**: Match one or more elements based on their attributes/attribute values.
* **Pseudo-classes**: Match one or more elements that exist in a certain state, such as an element that is being hovered over by the mouse pointer, or a checkbox that is currently disabled or checked, or an element that is the first child of its parent in the DOM tree.
* **Pseudo-elements**: Match one or more parts of content that are in a certain position in relation to an element, for example the first word of each paragraph, or generated content appearing just before an element.
* **Combinators**: These are not exactly selectors themselves, but ways of combining two or more selectors in useful ways for very specific selections. So for example, you could select only paragraphs that are direct descendants of divs, or paragraphs that come directly after headings.
* **Multiple selectors**: Again, these are not separate selectors; the idea is that you can put multiple selectors on the same CSS rule, separated by commas, to apply a single set of declarations to all the elements selected by those selectors.

**Important**: An ID name must be unique in the document. Behaviors regarding duplicated IDs are unpredictable. For example, in some browsers, only the first instance is counted, and the rest are ignored.

## Universal selector

The universal selector (\*) is the ultimate joker. It allows selecting all elements in a page. As it is rarely useful to apply a style to every element on a page, it is often used in combination with other selectors (see [Combinators](https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Simple_selectors#Combinators) below.)

**Important**: Take care when using the universal selector. As it applies to all elements, using it in large web pages can have a perceptible impact on performance: web pages display slower than expected. There are not many instances where you'd use this selector.

**Presence and value attribute selectors**

These attribute selectors try to match an exact attribute value:

* [attr] : This selector will select all elements with the attribute attr, whatever its value.
* [attr=val] : This selector will select all elements with the attribute attr, but only if its value is val.
* [attr~=val]: This selector will select all elements with the attribute attr, but only if the value val is one of a space-separated list of values contained in attr's value, for example a single class in a space-separated list of classes.
* [attr|=val] : This selector will select all elements with the attribute attr for which the value is exactly val or starts with val- (careful, the dash here isn't a mistake, this is to handle language codes.)
* [attr^=val] : This selector will select all elements with the attribute attr for which the value starts with val.
* [attr$=val] : This selector will select all elements with the attribute attr for which the value ends with val.
* [attr\*=val] : This selector will select all elements with the attribute attr for which the value contains the string val (unlike [attr~=val], this selector doesn't treat spaces as value separators but as part of the attribute value.)

## Pseudo-classes

A CSS [pseudo-class](https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes) is a keyword, preceded by a colon (:), added to the end of selectors to specify you want to style the selected elements, and only when they are in certain state. For example, you might want to style an element only when it is being hovered over by the mouse pointer, or a checkbox when it is disabled or checked, or an element that is the first child of its parent in the DOM tree.

* [:active](https://developer.mozilla.org/en-US/docs/Web/CSS/:active)
* [:any](https://developer.mozilla.org/en-US/docs/Web/CSS/:any)
* [:checked](https://developer.mozilla.org/en-US/docs/Web/CSS/:checked)
* [:default](https://developer.mozilla.org/en-US/docs/Web/CSS/:default)
* [:dir()](https://developer.mozilla.org/en-US/docs/Web/CSS/:dir)
* [:disabled](https://developer.mozilla.org/en-US/docs/Web/CSS/:disabled)
* [:empty](https://developer.mozilla.org/en-US/docs/Web/CSS/:empty)
* [:enabled](https://developer.mozilla.org/en-US/docs/Web/CSS/:enabled)
* [:first](https://developer.mozilla.org/en-US/docs/Web/CSS/:first)
* [:first-child](https://developer.mozilla.org/en-US/docs/Web/CSS/:first-child)
* [:first-of-type](https://developer.mozilla.org/en-US/docs/Web/CSS/:first-of-type)
* [:fullscreen](https://developer.mozilla.org/en-US/docs/Web/CSS/:fullscreen)
* [:focus](https://developer.mozilla.org/en-US/docs/Web/CSS/:focus)
* [:focus-within](https://developer.mozilla.org/en-US/docs/Web/CSS/:focus-within)
* [:hover](https://developer.mozilla.org/en-US/docs/Web/CSS/:hover)
* [:indeterminate](https://developer.mozilla.org/en-US/docs/Web/CSS/:indeterminate)
* [:in-range](https://developer.mozilla.org/en-US/docs/Web/CSS/:in-range)
* [:invalid](https://developer.mozilla.org/en-US/docs/Web/CSS/:invalid)
* [:lang()](https://developer.mozilla.org/en-US/docs/Web/CSS/:lang)
* [:last-child](https://developer.mozilla.org/en-US/docs/Web/CSS/:last-child)
* [:last-of-type](https://developer.mozilla.org/en-US/docs/Web/CSS/:last-of-type)
* [:left](https://developer.mozilla.org/en-US/docs/Web/CSS/:left)
* [:link](https://developer.mozilla.org/en-US/docs/Web/CSS/:link)
* [:not()](https://developer.mozilla.org/en-US/docs/Web/CSS/:not)
* [:nth-child()](https://developer.mozilla.org/en-US/docs/Web/CSS/:nth-child)
* [:nth-last-child()](https://developer.mozilla.org/en-US/docs/Web/CSS/:nth-last-child)
* [:nth-last-of-type()](https://developer.mozilla.org/en-US/docs/Web/CSS/:nth-last-of-type)
* [:nth-of-type()](https://developer.mozilla.org/en-US/docs/Web/CSS/:nth-of-type)
* [:only-child](https://developer.mozilla.org/en-US/docs/Web/CSS/:only-child)
* [:only-of-type](https://developer.mozilla.org/en-US/docs/Web/CSS/:only-of-type)
* [:optional](https://developer.mozilla.org/en-US/docs/Web/CSS/:optional)
* [:out-of-range](https://developer.mozilla.org/en-US/docs/Web/CSS/:out-of-range)
* [:read-only](https://developer.mozilla.org/en-US/docs/Web/CSS/:read-only)
* [:read-write](https://developer.mozilla.org/en-US/docs/Web/CSS/:read-write)
* [:required](https://developer.mozilla.org/en-US/docs/Web/CSS/:required)
* [:right](https://developer.mozilla.org/en-US/docs/Web/CSS/:right)
* [:root](https://developer.mozilla.org/en-US/docs/Web/CSS/:root)
* [:scope](https://developer.mozilla.org/en-US/docs/Web/CSS/:scope)
* [:target](https://developer.mozilla.org/en-US/docs/Web/CSS/:target)
* [:valid](https://developer.mozilla.org/en-US/docs/Web/CSS/:valid)
* [:visited](https://developer.mozilla.org/en-US/docs/Web/CSS/:visited)

## Pseudo-elements

[Pseudo-elements](https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements) are very much like pseudo-classes, but they have differences. They are keywords, this time preceded by two colons (::), that can be added to the end of selectors to select a certain part of an element.

* [::after](https://developer.mozilla.org/en-US/docs/Web/CSS/::after)
* [::before](https://developer.mozilla.org/en-US/docs/Web/CSS/::before)
* [::first-letter](https://developer.mozilla.org/en-US/docs/Web/CSS/::first-letter)
* [::first-line](https://developer.mozilla.org/en-US/docs/Web/CSS/::first-line)
* [::selection](https://developer.mozilla.org/en-US/docs/Web/CSS/::selection)
* [::backdrop](https://developer.mozilla.org/en-US/docs/Web/CSS/::backdrop)

## Combinators and groups of selectors

| **Name** | **Syntax** | **Selects** |
| --- | --- | --- |
| Group of selectors | A, B | Any element matching A and/or B (see [Group of selectors on one rule](https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Combinators_and_multiple_selectors#Group_of_selectors_on_one_rule), below - **Group of Selectors** is not considered to be a combinator). |
| Descendant selector | A B | Any element matching B that is a *descendant* of an element matching A (that is, a child, or a child of a child, etc.). |
| Child selector | A > B | Any element matching B that is a *direct child* of an element matching A. |
| Adjacent sibling selector | A + B | Any element matching B that is the next *sibling* of an element matching A (that is, the next child of the same parent). |
| General sibling selector | A ~ B | Any element matching B that is one of the next *siblings* of an element matching A (that is, one of the next children of the same parent). |

# CSS values and units

* Numeric values: Length values for specifying e.g. element width, border thickness, or font size, and unitless integers for specifying e.g. relative line width or number of times to run an animation.
* Percentages: Can also be used to specify size or length — relative to a parent container's width or height for example, or the default font-size.
* Colors: For specifying background colors, text colors, etc.
* Coordinate positions: e.g. for specifying the position of a positioned element relative to the top left of the screen.
* Functions: For specifying e.g. background images or background image gradients.

## Numeric values

### **Length and size**

Pixels (px) are referred to as **absolute units** because they will always be the same size regardless of any other related settings. Other absolute units are as follows:

* ***mm, cm, in***: Millimeters, centimeters, or inches.
* ***pt, pc***: Points (1/72 of an inch) or picas (12 points.)

You probably won't use any of these very often except pixels.

*There are also relative units, which are relative to the current element's font-size or*[*viewport*](https://developer.mozilla.org/en-US/docs/Glossary/viewport)*size:*

* ***em***: 1em is the same as the font-size of the current element (more specifically, the width of a capital letter M.) The default base font-size given to web pages by web browsers before CSS styling is applied is 16 pixels, which means the computed value of 1em is 16 pixels for an element by default. But beware — font sizes are inherited by elements from their parents, so if different font sizes have been set on parent elements, the pixel equivalent of an em can start to become complicated. Don't worry too much about this for now — we'll cover inheritance and font-sizing in more detail in later articles and modules. **ems are the most common relative unit you'll use in web development**.
* ***ex, ch***: Respectively these are the height of a lower case x, and the width of the number 0. These are not as commonly used or well-supported as ems.
* ***rem***: The rem (root em) works in exactly the same way as the em, except that it will always equal the size of the default base font-size; inherited font sizes will have no effect, so this sounds like a much better option than ems, although rems don't work in older versions of Internet Explorer (see more about cross-browser support in [Debugging CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Debugging_CSS).)
* ***vw, vh***: Respectively these are 1/100th of the width of the viewport, and 1/100th of the height of the viewport.

Using relative units is quite useful — you can size your HTML elements relative to your font or viewport size, meaning that the layout will stay looking correct if for example the text size is doubled across the whole website by a visually impaired user.

### **Unitless values**

You'll sometimes come across unitless numeric values in CSS — this is not always an error, in fact, it is perfectly allowed in some circumstances. For example, if you want to completely remove the margin or padding from an element, you can just use unitless 0 — 0 is 0, no matter what units were set before!

margin: 0;

#### Unitless line height

Another example is [line-height](https://developer.mozilla.org/en-US/docs/Web/CSS/line-height), which sets how high each line of text in an element is. You can use units to set a specific line height, but it is often easier to use a unitless value, which acts as a simple multiplying factor.

#### Number of animations

[CSS Animations](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations) allow you to animate HTML elements on the page. Let's present a simple example that causes a paragraph to rotate when it is moused over. The HTML for this example is pretty simple:

<p>Hello</p>

The CSS is a little more complex:

@keyframes rotate {

0% {

transform: rotate(0deg);

}

100% {

transform: rotate(360deg);

}

}

p {

color: red;

width: 100px;

font-size: 40px;

transform-origin: center;

}

p:hover {

animation-name: rotate;

animation-duration: 0.6s;

animation-timing-function: linear;

animation-iteration-count: 5;

}

Here you can see a number of interesting units that we don't talk about explicitly in this article ([<angle>](https://developer.mozilla.org/en-US/docs/Web/CSS/angle)s, [<time>](https://developer.mozilla.org/en-US/docs/Web/CSS/time)s, [<timing-function>](https://developer.mozilla.org/en-US/docs/Web/CSS/timing-function)s, [<string>](https://developer.mozilla.org/en-US/docs/Web/CSS/string)s...), but the one we are interested in here is in the line animation-iteration-count: 5; — this controls how many times the animation occurs when it is set off (in this case, when the paragraph is moused over,) and is a simple unitless whole number (integer, in computer speak.)

## Percentages

You can also use percentage values to specify most things that can be specified by specific numeric values. This allows us to create, for example, boxes whose width will always shift to be a certain percentage of their parent container's width. This can be compared to boxes that have their width set to a certain unit value (like px or ems), which will always stay the same length, even if their parent container's width changes.

## Colors

There are many ways to specify color in CSS, some of which are more recently implemented than others. The same color values can be used everywhere in CSS, whether you are specifying text color, background color, or whatever else.

### **Keywords**

The simplest, oldest color types in CSS are the color keywords. These are specific strings representing particular color values. For example, the following code:

<p>This paragraph has a red background</p>

p {

background-color: red;

}

### **Hexadecimal values**

The next ubiquitous color system is hexadecimal colors, or hex codes. Each hex value consists of a hash/pound symbol (#) followed by six hexadecimal numbers, each of which can take a value between 0 and f (which represents 15) — so 0123456789abcdef. Each pair of values represents one of the channels — red, green and blue — and allows us to specify any of the 256 available values for each (16 x 16 = 256.)

So, for example, this code:

<p>This paragraph has a red background</p>

<p>This paragraph has a blue background</p>

<p>This paragraph has a kind of pinky lilac background</p>

/\* equivalent to the red keyword \*/

p:nth-child(1) {

background-color: #ff0000;

}

/\* equivalent to the blue keyword \*/

p:nth-child(2) {

background-color: #0000ff;

}

/\* has no exact keyword equivalent \*/

p:nth-child(3) {

background-color: #e0b0ff;

}

These values are a bit more complex and less easy to understand, but they are a lot more versatile than keywords — you can use hex values to represent any color you want to use in your color scheme.

### **RGB**

The third scheme we'll talk about here is RGB. An RGB value is a function — rgb() — which is given three parameters that represent the **red**, **green** and **blue** channel values of the colors, in much the same way as hex values. The difference with RGB is that each channel is represented not by two hex digits, but by a decimal number between 0 and 255.

Let's rewrite our last example to use RGB colors:

<p>This paragraph has a red background</p>

<p>This paragraph has a blue background</p>

<p>This paragraph has a kind of pinky lilac background</p>

/\* equivalent to the red keyword \*/

p:nth-child(1) {

background-color: rgb(255,0,0);

}

/\* equivalent to the blue keyword \*/

p:nth-child(2) {

background-color: rgb(0,0,255);

}

/\* has no exact keyword equivalent \*/

p:nth-child(3) {

background-color: rgb(224,176,255);

}

### **HSL**

1. **hue**: the base shade of the color. This takes a value between 0 and 360, presenting the angles round a color wheel.
2. **saturation**: how saturated is the color? This takes a value from 0-100%, where 0 is no color (it will appear as a shade of grey), and 100% is full color saturation
3. **lightness**: how light or bright is the color? This takes a value from 0-100%, where 0 is no light (it will appear completely black) and 100% is full light (it will appear completely white).

Now we'll rewrite our example to use HSL colors:

1. <p>This paragraph has a red background</p>
2. <p>This paragraph has a blue background</p>
3. <p>This paragraph has a kind of pinky lilac background</p>
4. /\* equivalent to the red keyword \*/
5. p:nth-child(1) {
6. background-color: hsl(0,100%,50%);
7. }
8. /\* equivalent to the blue keyword \*/
9. p:nth-child(2) {
10. background-color: hsl(240,100%,50%);
11. }
12. /\* has no exact keyword equivalent \*/
13. p:nth-child(3) {
14. background-color: hsl(276,100%,85%);
15. }

### **RGBA and HSLA**

RGB and HSL both have corresponding modes — RGBA and HSLA — that allow you to set not only what color you want to display, but also what **transparency** you want that color to have. Their corresponding functions take the same parameters, plus a fourth value in the range 0–1 — which sets the transparency, or **alpha channel**. 0 is completely transparent, and 1 is completely opaque.

/\* Transparent red \*/

p:nth-child(1) {

background-color: rgba(255,0,0,0.5);

position: relative;

top: 30px;

left: 50px;

}

### **Opacity**

There is another way to specify transparency via CSS — the [opacity](https://developer.mozilla.org/en-US/docs/Web/CSS/opacity) property. Instead of setting the transparency of a particular color, this sets the transparency of all selected elements and their children. Again, let's study an example so we can see the difference.

<p>This paragraph is using RGBA for transparency</p>

<p>This paragraph is using opacity for transparency</p>

Now the CSS:

/\* Red with RGBA \*/

p:nth-child(1) {

background-color: rgba(255,0,0,0.5);

}

/\* Red with opacity \*/

p:nth-child(2) {

background-color: rgb(255,0,0);

opacity: 0.5;

}

## Functions

In programming, a [function](https://developer.mozilla.org/en-US/docs/Glossary/function) is a reusable section of code that can be run multiple times to complete a repetitive task with minimum effort on the part of both the developer and the computer

But you'll see functions in other places too — anytime you see a name with parenthesis after it, containing one or more values separated by commas, you are dealing with a function. For example:

/\* calculate the new position of an element after it has been rotated by 45 degress \*/

transform: rotate(45deg);

/\* calculate the new position of an element after it has been moved across 50px and down 60px \*/

transform: translate(50px, 60px);

/\* calculate the computed value of 90% of the current width minus 15px \*/

width: calc(90%-15px);

/\* fetch an image from the network to be used as a background image \*/

background-image: url('myimage.png');

## The cascade --- Very Important

CSS is an acronym for Cascading Style Sheets, which indicates that the notion of the cascade is important. At its most basic level it indicates that the order of CSS rules matter, but it's more complex than that. What selectors win out in the cascade depends on three factors (these are listed in order of weight — earlier ones will overrule later ones):

1. Importance
2. Specificity
3. Source order

### **Importance**

In CSS, there is a special piece of syntax you can use to make sure that a certain declaration will always win over all others: !important.

Let's look at an example:

<p class="better">This is a paragraph.</p>

<p class="better" id="winning">One selector to rule them all!</p>

#winning {

background-color: red;

border: 1px solid black;

}

.better {

background-color: gray;

border: none **!important**;

}

p {

background-color: blue;

color: white;

padding: 5px;

}

Let's walk through this to see what's happening.

1. You'll see that the third rule's [color](https://developer.mozilla.org/en-US/docs/Web/CSS/color) and [padding](https://developer.mozilla.org/en-US/docs/Web/CSS/padding) values have been applied, but the [background-color](https://developer.mozilla.org/en-US/docs/Web/CSS/background-color) hasn't. Why? Really all three should surely apply, because rules later in the source order generally override earlier rules.
2. However, The rules above it win, because IDs/class selectors have higher specificity than element selectors (you'll learn more about this in the next section).
3. Both elements have a [class](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#attr-class) of better, but the 2nd one has an [id](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#attr-id) of winning too. Since IDs have an *even higher* specificity than classes (you can only have one element with each unique ID on a page, but many elements with the same class — ID selectors are *very specific* in what they target), the red background color and the 1 pixel black border should both be applied to the 2nd element, with the first element getting the gray background color, and no border, as specified by the class.
4. The 2nd element *does* get the red background color, but no border. Why? Because of the !important declaration in the second rule — including this after border: nonemeans that this declaration will win over the border value in the previous rule, even though the ID has higher specificity.

**Note**: The only way to override this !important declaration would be to include another !important declaration of the same specificity later in the source order, or one with a higher specificity.

It is useful to know that !important exists so that you know what it is when you come across it in other people's code. **However**, we strongly recommend that you never use it unless you absolutely have to.

### **Specificity**

**Specificity** is basically a measure of how specific a selector is — how many elements it could match. As shown in the example seen above, element selectors have low specificity. Class selectors have a higher specificity, so will win against element selectors. ID selectors have an even higher specificity, so will win against class selectors. The only way to win against an ID selector is to use !important.

The amount of specificity a selector has is measured using four different values (or components), which can be thought of as thousands, hundreds, tens and ones — four single digits in four columns:

1. Thousands: Score one in this column if  the declaration is inside a [style](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#attr-style) attribute (such declarations don't have selectors, so their specificity is always simply 1000.) Otherwise 0.
2. Hundreds: Score one in this column for each ID selector contained inside the overall selector.
3. Tens: Score one in this column for each class selector, attribute selector, or pseudo-class contained inside the overall selector.
4. Ones: Score one in this column for each element selector or pseudo-element contained inside the overall selector.

**Note**: Universal selector (\*), combinators (+, >, ~, ' ') and negation pseudo-class (:not) have no effect on specificity.

| **Selector** | **Thousands** | **Hundreds** | **Tens** | **Ones** | **Total specificity** |
| --- | --- | --- | --- | --- | --- |
| h1 | 0 | 0 | 0 | 1 | 0001 |
| #important | 0 | 1 | 0 | 0 | 0100 |
| h1 + p::first-letter | 0 | 0 | 0 | 3 | 0003 |
| li > a[href\*="en-US"] > .inline-warning | 0 | 0 | 2 | 2 | 0022 |
| No selector, with a rule inside an element's [style](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#attr-style) attribute | 1 | 0 | 0 | 0 | 1000 |

**Note**: If multiple selectors have the same importance *and* specificity, which selector wins is decided by which comes later in the [Source order](https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Cascade_and_inheritance#Source_order).

### **Source order**

As mentioned above, if multiple competing selectors have the same importance and specificity, the third factor that comes into play to help decide which rule wins is source order — later rules will win over earlier rules. For example:

p {

color: blue;

}

/\* This rule will win over the first one \*/

p {

color: red;

}

Whereas in this example the first rule wins because source order is overruled by specificity:

/\* This rule will win \*/

.footnote {

color: blue;

}

p {

color: red;

}

### **A note on rule mixing**

When several CSS rules match the same element, they are all applied to that element. Only after that are any conflicting properties evaluated to see which individual styles will win over others.

Let's see an example. First, some HTML:

<p>I'm <strong>important</strong></p>

And now some CSS to style it with:

/\* specificity: 0002 \*/

p strong {

background-color: khaki;

color: green;

}

/\* specificity: 0001 \*/

strong {

text-decoration: underline;

color: red;

}

Result:

I'm **important**

## Inheritance

The idea is that some property values applied to an element will be inherited by that element's children, and some won't.

* For example, it makes sense for [font-family](https://developer.mozilla.org/en-US/docs/Web/CSS/font-family) and [color](https://developer.mozilla.org/en-US/docs/Web/CSS/color) to be inherited, as that makes it easy for you to set a site-wide base font by applying a font-family to the [<html>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/html) element;
* As another example, it makes sense for [margin](https://developer.mozilla.org/en-US/docs/Web/CSS/margin), [padding](https://developer.mozilla.org/en-US/docs/Web/CSS/padding), [border](https://developer.mozilla.org/en-US/docs/Web/CSS/border), and [background-image](https://developer.mozilla.org/en-US/docs/Web/CSS/background-image) to NOT be inherited.

Which properties are inherited by default and which aren't is largely down to common sense.

### **Controlling inheritance**

[**inherit**](https://developer.mozilla.org/en-US/docs/Web/CSS/inherit)

Sets the property value applied to a selected element to be the same as that of its parent element.

[**initial**](https://developer.mozilla.org/en-US/docs/Web/CSS/initial)

Sets the property value applied to a selected element to be the same as the value set for that element in the browser's default style sheet.

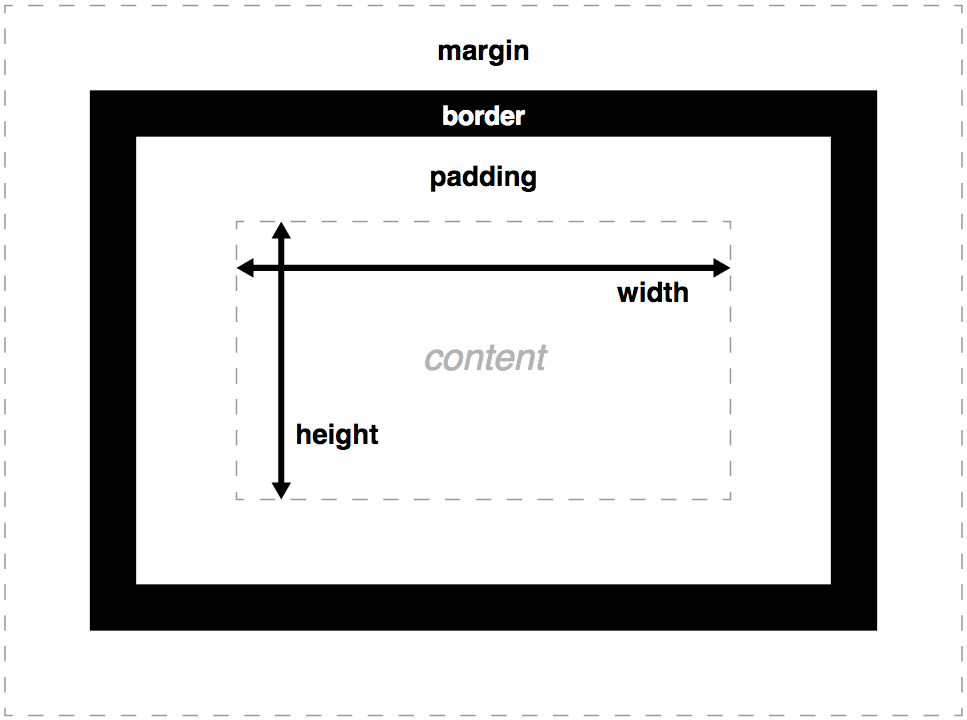
[**unset**](https://developer.mozilla.org/en-US/docs/Web/CSS/unset)

Resets the property to its natural value, which means that if the property is naturally inherited it acts like inherit, otherwise it acts like initial.

[**revert**](https://developer.mozilla.org/en-US/docs/Web/CSS/revert)

Reverts the property to the value it would have had if the current origin had not applied any styles to it.

## Box properties



[**width**](https://developer.mozilla.org/en-US/docs/Web/CSS/width)**and**[**height**](https://developer.mozilla.org/en-US/docs/Web/CSS/height)

The [width](https://developer.mozilla.org/en-US/docs/Web/CSS/width) and [height](https://developer.mozilla.org/en-US/docs/Web/CSS/height) properties set the width and height of the **content box**, which is the area in which the content of the box is displayed — this content includes both text content set inside the box, and other boxes representing nested child elements.

[**padding**](https://developer.mozilla.org/en-US/docs/Web/CSS/padding)

**Padding** refers to the *inner* margin of a CSS box — between the outer edge of the content box and the inner edge of the border. The size of this layer can be set on all four sides at once with the [padding](https://developer.mozilla.org/en-US/docs/Web/CSS/padding) shorthand property, or one side at a time with the [padding-top](https://developer.mozilla.org/en-US/docs/Web/CSS/padding-top), [padding-right](https://developer.mozilla.org/en-US/docs/Web/CSS/padding-right), [padding-bottom](https://developer.mozilla.org/en-US/docs/Web/CSS/padding-bottom) and [padding-left](https://developer.mozilla.org/en-US/docs/Web/CSS/padding-left) properties.

[**border**](https://developer.mozilla.org/en-US/docs/Web/CSS/border)

The **border** of a CSS box sits between the outer edge of the padding and the inner edge of the margin. By default the border has a size of 0 — making it invisible — but you can set the thickness, style and color of the border to make it appear. The [border](https://developer.mozilla.org/en-US/docs/Web/CSS/border) shorthand property allows you to set all of these on all four sides at once, for example border: 1px solid black. This can be broken down into numerous different longhand properties for more specific styling needs:

* [border-top](https://developer.mozilla.org/en-US/docs/Web/CSS/border-top), [border-right](https://developer.mozilla.org/en-US/docs/Web/CSS/border-right), [border-bottom](https://developer.mozilla.org/en-US/docs/Web/CSS/border-bottom), [border-left](https://developer.mozilla.org/en-US/docs/Web/CSS/border-left): Set the thickness, style and color of one side of the border.
* [border-width](https://developer.mozilla.org/en-US/docs/Web/CSS/border-width), [border-style](https://developer.mozilla.org/en-US/docs/Web/CSS/border-style), [border-color](https://developer.mozilla.org/en-US/docs/Web/CSS/border-color): Set only the thickness, style, or color individually, but for all four sides of the border.
* You can also set one of the three properties of a single side of the border individually, using [border-top-width](https://developer.mozilla.org/en-US/docs/Web/CSS/border-top-width), [border-top-style](https://developer.mozilla.org/en-US/docs/Web/CSS/border-top-style), [border-top-color](https://developer.mozilla.org/en-US/docs/Web/CSS/border-top-color), etc.

[**margin**](https://developer.mozilla.org/en-US/docs/Web/CSS/margin)

* The margin surrounds a CSS box, and pushes up against other CSS boxes in the layout. It behaves rather like padding; the shorthand property is [margin](https://developer.mozilla.org/en-US/docs/Web/CSS/margin) and the individual properties are [margin-top](https://developer.mozilla.org/en-US/docs/Web/CSS/margin-top), [margin-right](https://developer.mozilla.org/en-US/docs/Web/CSS/margin-right), [margin-bottom](https://developer.mozilla.org/en-US/docs/Web/CSS/margin-bottom), and [margin-left](https://developer.mozilla.org/en-US/docs/Web/CSS/margin-left).
* **Note**: Margins have a specific behavior called [margin collapsing](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Mastering_margin_collapsing): When two boxes touch against one another, the distance between them is the value of the largest of the two touching margins, and not their sum.

## Advanced box manipulation

### **Overflow**

When you set the size of a box with absolute values (e.g. a fixed pixel width/height), the content may not fit within the allowed size, in which case the content overflows the box.

* auto: If there is too much content, the overflowing content is hidden and scroll bars are shown to let the user scroll to see all the content.
* hidden: If there is too much content, the overflowing content is hidden.
* visible: If there is too much content, the overflowing content is shown outside of the box (this is usually the default behavior.)

### **Outline**

to be specific, the outline is drawn outside the border box, inside the margin area.

## Types of CSS boxes

The type of box applied to an element is specified by the [display](https://developer.mozilla.org/en-US/docs/Web/CSS/display) property.

* A block box is defined as a box that's stacked upon other boxes (i.e. content before and after the box appears on a separate line), and can have width and height set on it. The whole box model as described above applies to block boxes.
* An inline box is the opposite of a block box: it flows with the document's text (i.e. it will appear on the same line as surrounding text and other inline elements, and its content will break with the flow of the text, like lines of text in a paragraph.) Width and height settings have no effect on inline boxes; any padding, margin and border set on inline boxes will update the position of surrounding text, but will not affect the position of surrounding block boxes.
* An inline-block box is something in between the first two: It flows with surrounding text and other inline elements without creating line breaks before and after it unlike a block box, but it can be sized using width and height and maintains its block integrity like a block box. It won't be broken across paragraph lines like an inline box.  *inline box*does*break on multiple lines if there is not enough space*

## Normal layout flow

HTML is displayed in the exact order in which it appears in the source code, with elements stacked up on top of one another.

Layout techniques tend to override this default behavior, using:

* The [position](https://developer.mozilla.org/en-US/docs/Web/CSS/position) property — static is the default in normal flow, but you can cause elements to be laid out differently using other values, for example always fixed to the top left of the browser viewport.
* Floats — applying a [float](https://developer.mozilla.org/en-US/docs/Web/CSS/float) value such as left can cause block level elements to line up alongside one another rather than sit on top of one another
* The [display](https://developer.mozilla.org/en-US/docs/Web/CSS/display) property — standard values such as block, inline or inline-block can change how elements behave in normal flow (see [Types of CSS boxes](https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Box_model#Types_of_CSS_boxes) for more information), whereas uncommon or specialized values allow us to lay out elements in completely different ways using tools like Flexbox.

## Floats

Floats is a technique that allows the elements to float to the left or right of one another, rather than the default of sitting on top of one another. The main uses of floats are to lay out columns and float text around an image.

The float property has four possible values:

* left — floats the element to the left.
* right — floats the element to the right.
* none — specified no floating at all. This is the default value.
* inherit — specifies that the value of the float property should be inherited from the parent element.

### **Making the columns float**

div:nth-of-type(1) {

width: 48%;

float: left;

}

div:nth-of-type(2) {

width: 48%;

float: right;

}

## Clearing floats

footer {

clear: both;

}

[clear](https://developer.mozilla.org/en-US/docs/Web/CSS/clear) can take three values:

* left: Stop any active left floats
* right: Stop any active right floats
* both: Stop any active left and right floats

## Float problems

### **The whole width can be tricky to calculate**

### **Background height of floated items**

### **Clearing floats can get complex**

## Positioning Techniques

Positioning allows you to move an element from it's original spot on the page to another spot with great accuracy.

There are four main types of positioning you should know about:

* **Static positioning** is the default that every element gets — it just means "put the element into its normal position in the document layout flow — nothing special to see here".
* **Relative positioning** allows you to modify an element's position on the page, moving it relative to its position in normal flow — including making it overlap other elements on the page. This is useful for minor layout tweaks and design pinpointing.
* **Absolute positioning** moves an element completely out of the page's normal layout flow, like it is sitting on its own separate layer. From there, you can fix it in a position relative to the edges of the page's <html> element (or its nearest positioned ancestor element). The [top](https://developer.mozilla.org/en-US/docs/Web/CSS/top) and [left](https://developer.mozilla.org/en-US/docs/Web/CSS/left) properties have a different effect on absolutely positioned elements than they do on relatively positioned elements. In this case they don't specify how much the element moves relative to its original position; instead, they specify the distance the element should sit from the top and left sides of the page's boundaries (the <html> element, to be exact).
* **Fixed positioning** is very similar to absolute positioning, except that it fixes an element relative to the browser viewport, not another element. This is useful for creating effects such as a persistent navigation menu that always stays in the same place on the screen as the rest of the content scrolls.

## CSS tables

HTML tables are fine for displaying tabular data, but many years ago — before even basic CSS was supported reliably across browsers — web developers used to also use tables for entire web page layouts — putting their headers, footers, different columns, etc. in various table rows and columns. This worked at the time, but it has many problems — table layouts are inflexible, very heavy on markup, difficult to debug, and semantically wrong (e.g., screen reader users have problems navigating table layouts).

CSS tables exist to allow you to lay out elements like they were a table, without any of the issues described above — this may sound strange, and you should use table elements for tabular data, but sometimes this can be useful. For example, you might want to lay out a form with the labels and text inputs lined up; this can be tricky, but CSS tables make it easy.

<form>

<p>First of all, tell us your name and age.</p>

<div>

<label for="fname">First name:</label>

<input type="text" id="fname">

</div>

<div>

<label for="lname">Last name:</label>

<input type="text" id="lname">

</div>

<div>

<label for="age">Age:</label>

<input type="text" id="age">

</div>

</form>

html {

font-family: sans-serif;

}

form {

display: table;

margin: 0 auto;

}

form div {

display: table-row;

}

form label, form input {

display: table-cell;

margin-bottom: 10px;

}

form label {

width: 200px;

padding-right: 5%;

text-align: right;

}

form input {

width: 300px;

}

form p {

display: table-caption;

caption-side: bottom;

width: 300px;

color: #999;

font-style: italic;

}

## Flexible boxes

<section>

<div>This is a box</div>

<div>This is a box</div>

<div>This is a box</div>

</section>

<button class="create">Create box</button>

<button class="reset">Reset demo</button>

html {

font-family: sans-serif;

}

section {

width: 93%;

height: 240px;

margin: 20px auto;

background: purple;

display: flex;

}

div {

color: white;

background: orange;

flex: 1;

margin-right: 10px;

text-shadow: 1px 1px 1px black;

}

div:last-child {

margin-right: 0;

}

section, div {

border: 5px solid rgba(0,0,0,0.85);

padding: 10px;

}

* display: flex; tells the [<section>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/section) element's children to be laid out as flexible boxes — by default, they will all stretch to fill the available height of the parent, whatever that is, and be laid out in a row — with enough width to wrap their content.
* flex: 1; tells each [<div>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) element to take up an equal amount of the space available in the row, no matter how many there are.

## Why Flexbox?

For a long time, the only reliable cross browser-compatible tools available for creating CSS layouts were things like [floats](https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Floats) and [positioning](https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Positioning). These are fine and they work, but in some ways, they are also rather limiting and frustrating.

The following simple layout requirements are either difficult or impossible to achieve with such tools, in any kind of convenient, flexible way:

* Vertically centering a block of content inside its parent.
* Making all the children of a container take up an equal amount of the available width/height, regardless of how much width/height is available.
* Making all columns in a multiple column layout adopt the same height even if they contain a different amount of content.

## Columns or rows?

Flexbox provides a property called [flex-direction](https://developer.mozilla.org/en-US/docs/Web/CSS/flex-direction) that specifies what direction the main axis runs in (what direction the flexbox children are laid out in) — by default this is set to row, which causes them to be laid out in a row in the direction your browser's default language works in (left to right, in the case of an English browser).

Try adding the following declaration to your section rule:

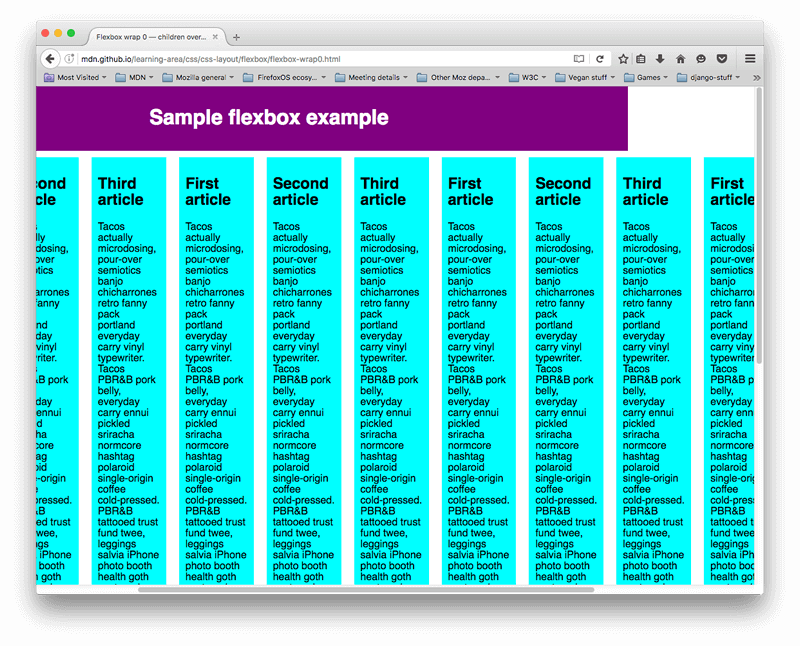
flex-direction: column;

You'll see that this puts the items back in a column layout, much like they were before we added any CSS. Before you move on, delete this declaration from your example.

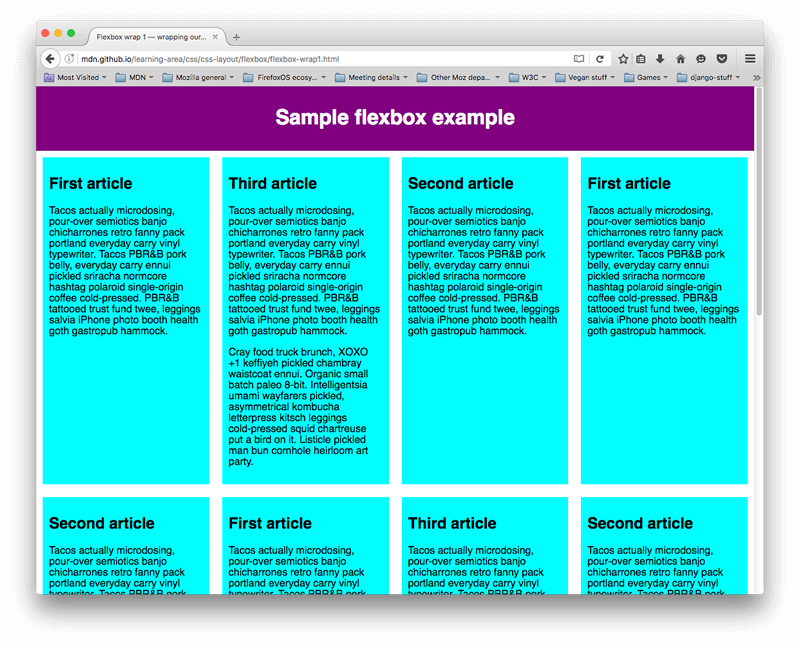
**Note**: You can also lay out flex items in a reverse direction using the row-reverse and column-reverse values. Experiment with these values too!

## Wrapping

One issue that arises when you have a fixed amount of width or height in your layout is that eventually your flexbox children will overflow their container, breaking the layout.



flex-wrap: wrap;



We now have multiple rows — as many flexbox children are fitted onto each row as makes sense, and any overflow is moved down to the next line. The flex: 200px declaration set on the articles means that each will be at least 200px wide;

## flex-flow shorthand

At this point it is worth noting that a shorthand exists for [flex-direction](https://developer.mozilla.org/en-US/docs/Web/CSS/flex-direction) and [flex-wrap](https://developer.mozilla.org/en-US/docs/Web/CSS/flex-wrap) — [flex-flow](https://developer.mozilla.org/en-US/docs/Web/CSS/flex-flow). So for example, you can replace

flex-direction: row;

flex-wrap: wrap;

with

flex-flow: row wrap;

## Flexible sizing of flex items

First, add the following rule to the bottom of your CSS:

article {

flex: 1;

}

In this case, we are giving each [<article>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/article) element a value of 1, which means they will all take up an equal amount of the spare space left after things like padding and margin have been set.

Now add the following rule below the previous one:

article:nth-of-type(3) {

flex: 2;

}

The first two flex items have one each so they take 1/4 of the available space each. The third one has two units, so it takes up 2/4 of the available space (or 1/2).

You can also specify a minimum size value inside the flex value. Try updating your existing article rules like so:

article {

flex: 1 200px;

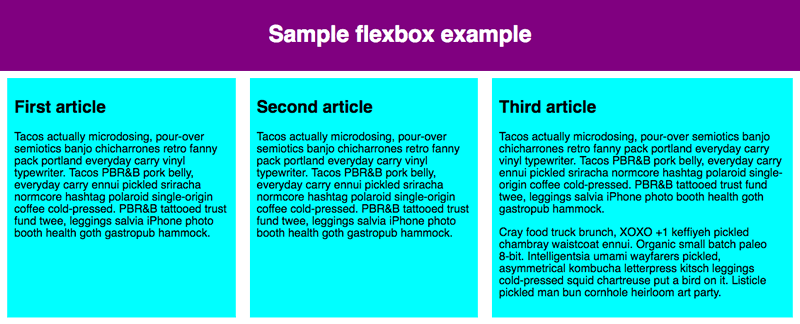
}

article:nth-of-type(3) {

flex: 2 200px;

}

This basically states "Each flex item will first be given 200px of the available space. After that, the rest of the available space will be shared out according to the proportion units." Try refreshing and you'll see a difference in how the space is shared out.



## Horizontal and vertical alignment

div {

display: flex;

align-items: center;

justify-content: space-around;

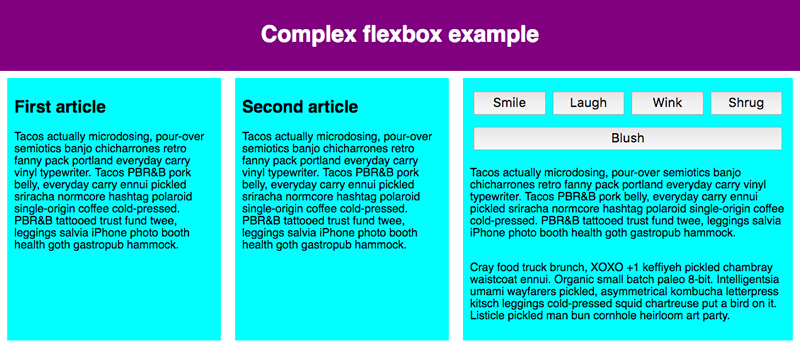
}

[align-items](https://developer.mozilla.org/en-US/docs/Web/CSS/align-items) controls where the flex items sit on the cross axis.

[justify-content](https://developer.mozilla.org/en-US/docs/Web/CSS/justify-content) controls where the flex items sit on the main axis.

* The default value is flex-start, which makes all the items sit at the start of the main axis.
* You can use flex-end to make them sit at the end.
* center is also a value for justify-content, and will make the flex items sit in the center of the main axis.
* The value we've used above, space-around, is useful — it distributes all the items evenly along the main axis, with a bit of space left at either end.
* There is another value, space-between, which is very similar to space-around except that it doesn't leave any space at either end.

## Nested flex boxes



section - article

article

article - div - button

div button

div button

button

button

section {

display: flex;

}

article {

flex: 1 200px;

}

article:nth-of-type(3) {

flex: 3 200px;

display: flex;

flex-flow: column;

}

article:nth-of-type(3) div:first-child {

flex:1 100px;

display: flex;

flex-flow: row wrap;

align-items: center;

justify-content: space-around;

}

button {

flex: 1;

margin: 5px;

font-size: 18px;

line-height: 1.5;

}

## Grid layout

## What is grid layout?

A grid is simply a collection of horizontal and vertical lines creating a pattern against which we can line up our design elements. They help us to create designs where elements don’t jump around or change width as we move from page to page, providing greater consistency on our websites.https://mdn.mozillademos.org/files/13899/grid.png

**Note**: It may seem surprising to anyone coming from a design background that CSS doesn’t have an inbuilt grid system, and instead we seem to be using a variety of suboptimal methods to create grid-like designs. As you’ll discover in the last part of this article this is set to change, however you are likely to need to know the existing methods of creating grids for some time to come.

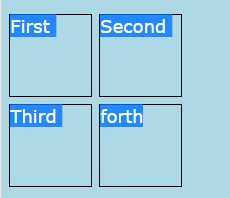
<https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Grids>

# CSS preprocessor:

A **CSS preprocessor** is a program that lets you generate [CSS](https://developer.mozilla.org/en-US/docs/Glossary/CSS) from the preprocessor's own unique [syntax](https://developer.mozilla.org/en-US/docs/Glossary/syntax). There are many CSS preprocessors to choose from, however most CSS preprocessors will add some features that don't exist in pure CSS, such as mixin, nesting selector, inheritance selector, and so on. These features make the CSS structure more readable and easier to maintain.

Interview Question:

Write CSS?



<!DOCTYPE html>

<html>

<head>

<style>

body {

background-color: lightblue;

}

h1 {

color: white;

text-align: center;

}

div{

position: relative;

}

p {

width: 90px;

height:90px;

position: absolute;

border: solid 1px black;

}

p:nth-of-type(1){

top:0px;

left:0px;

}

p:nth-of-type(2){

top:0px;

left:100px;

}

p:nth-of-type(3){

top:100px;

left:0px;

}

p:nth-of-type(4){

top:100px;

left:100px;

}

</style>

</head>

<body>

<div>

<p>First</p>

<p>Second</p>

<p>Third</p>

<p>forth</p>

</div>

</body>

</html>

Center the Header?

# **margin: auto;**

#main {

width: 600px;

margin: 0 auto;

}

<div id="main">

Setting the width of a block-level element will prevent it from stretching out to the edges of its container to the left and right. Then, you can set the left and right margins to auto to horizontally center that element within its container. The element will take up the width you specify, then the remaining space will be split evenly between the two margins.

**Vertically Centering using Position:**

<div>

<p></p>

</div>

div {

width:200px;

height:200px;

background-color:red;

position:relative;

}

div p {

width:50%;

height:50%;

background-color:blue;

position:absolute;

top:25%;

left:25%;

}

**Vertical Centering using Flex:**

<div>

<p></p>

</div>

div {

width:200px;

height:200px;

background-color:red;

display:flex;

flex-flow:column;

justify-content: center;

}

div p {

width:100%;

height:50%;

background-color:blue;

}

Interview Questions? ---VERY IMPORTANT

<https://thatjsdude.com/interview/css.html>

Vertically centering:

<http://vanseodesign.com/css/vertical-centering/>