



Cascading **S**tyle **S**heets

RULE #1

You don't know CSS.

CSS Drafts

W3C CSS Working Group Editor Drafts			Login
Specification	Last Update	By	
CSS Box Alignment 3	2020-12-17 16:03:34 PST	fantasai	↺ 🟢
css-animations			
CSS Animations 1 (Current Work)	2020-09-18 14:48:54 PDT	SebastianZ	↺ 🟡
CSS Animations 2	2020-07-16 03:10:20 PDT	anders_hartvoll_ruud	↺ 🟢
css-backgrounds			
CSS Backgrounds 3 (Current Work)	2020-12-21 10:03:32 PST	chrisl	☰ ↺ 🟡
CSS Backgrounds 4	2020-03-10 08:57:47 PDT	AmeliaBR	↺ 🟢
css-box			
CSS Box 3 (Current Work)	2020-12-16 00:26:25 PST	fantasai	↺ 🟢
CSS Box 4	2020-09-18 14:48:54 PDT	SebastianZ	↺ 🟢
css-break			
CSS Fragmentation 3 (Current Work)	2020-12-07 16:50:16 PST	g_rard_talbot	☰ ↺ 🟢
CSS Fragmentation 4	2020-09-18 14:48:54 PDT	SebastianZ	↺ 🟡
css-cascade			
CSS Cascading 3	2020-12-21 13:15:52 PST	fantasai	☰ ↺ 🟢
CSS Cascading 4 (Current Work)	2020-12-21 13:15:52 PST	fantasai	☰ ↺ 🟢
CSS Cascading 5	2021-01-06 16:42:45 PST	mirisuzanne	↺ 🟢
css-color			
CSS Color 3	2020-07-22 16:26:32 PDT	tabatkins	☰ ↺
CSS Color 4 (Current Work)	2021-01-06 01:40:57 PST	chrisl	↺ 🟡
CSS Color 5	2021-01-05 02:22:23 PST	chrisl	↺ 🟡
CSS Color Adjust	2020-12-16 12:40:33 PST	fantasai	↺ 🟡
css-conditional			
CSS Conditional 3 (Current Work)	2020-12-09 08:21:34 PST	fantasai	☰ ↺ 🟢
CSS Conditional 4	2020-03-04 08:53:46 PST	chrisl	↺ 🟡
css-contain			
CSS Containment 1	2020-12-20 21:51:52 PST	florian	☰ ↺ 🟢
CSS Containment 2 (Current Work)	2020-12-16 09:48:44 PST	florian	☰ ↺ 🟡
CSS Generated Content 3	2020-12-15 15:01:24 PST	tabatkins	↺ 🟡
CSS Counter Styles 3	2020-03-10 08:57:47 PDT	AmeliaBR	☰ ↺ 🟡
CSS Device Adaptation 1	2020-06-03 17:44:21 PDT	florian	↺ 🟡
CSS Display 3	2020-12-21 13:03:03 PST	fantasai	☰ ↺ 🟢
CSS Easing 1	2020-04-19 22:40:53 PDT	brian_birtles__via_travi	↺ 🟢
CSS Expressive Generalizations and Gadgetry 1	2020-10-26 02:17:46 PDT	manish_goregaokar	
CSS Environment Variables 1	2018-08-03 13:58:03 PDT	tabatkins	


























Please send comments, questions, and error reports to www-style@w3.org

<https://drafts.csswg.org/>

CSS Specification

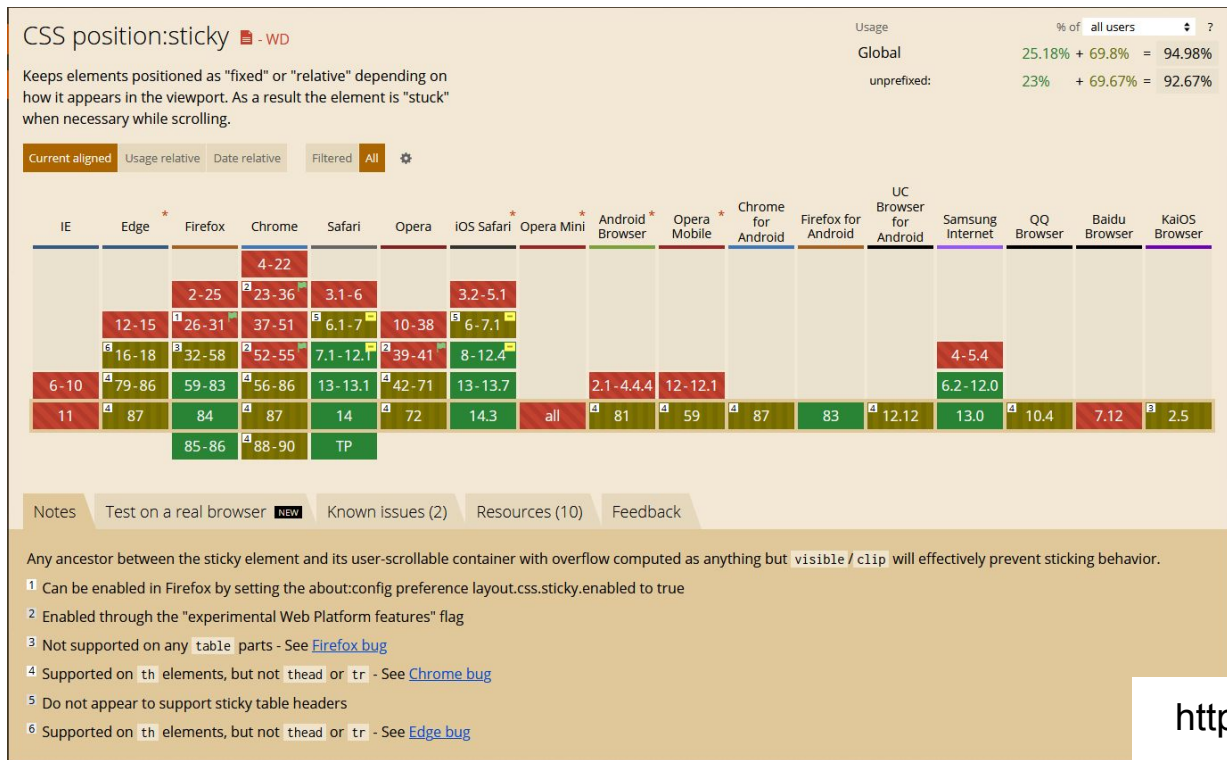
TABLE OF SPECIFICATIONS

Ordered from most to least stable:

Completed	Current	Upcoming	Notes	
CSS Snapshot 2020	NOTE		Latest stable CSS	
CSS Snapshot 2018	NOTE			
CSS Snapshot 2017	NOTE			
CSS Snapshot 2015	NOTE			
CSS Snapshot 2010	NOTE			
CSS Snapshot 2007	NOTE			
CSS Color Level 3	REC	REC		
CSS Namespaces	REC	REC		
Selectors Level 3	REC	REC		
CSS Level 2 Revision 1	REC	REC	See Errata	
Media Queries	REC	REC		
CSS Style Attributes	REC	REC		
CSS Fonts Level 3	REC	REC		
CSS Writing Modes Level 3	REC	REC		
CSS Basic User Interface Level 3	REC	REC		
CSS Containment Level 1	REC	REC		
Stable	Current	Upcoming	Notes	
CSS Backgrounds and Borders Level 3	CR	PR		
CSS Conditional Rules Level 3	CR	CR		
CSS Multi-column Layout Level 1	WD	CR		
CSS Values and Units Level 3	CR	PR		
CSS Flexible Box Layout Level 1	CR	PR		
CSS Cascading and Inheritance Level 3	CR	PR		
CSS Counter Styles Level 3	CR	PR		
Testing	Current	Upcoming	Notes	
CSS Images Level 3	CR	CR		
CSS Speech	CR	CR		

<https://www.w3.org/Style/CSS/current-work>

Can I Use?



<https://caniuse.com>

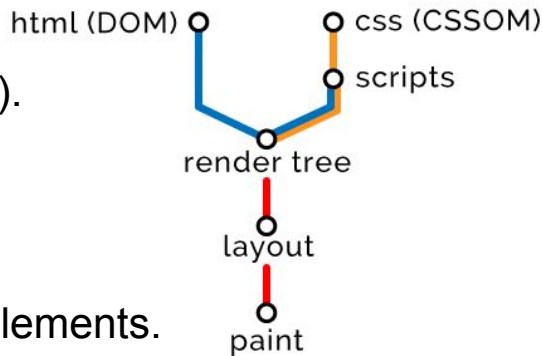
CSS Object Model (CSSOM)

The CSS Object Model is a set of APIs allowing the manipulation of CSS from JavaScript. It is much like the DOM, but for the CSS rather than the HTML. It allows users to read and modify CSS style dynamically.

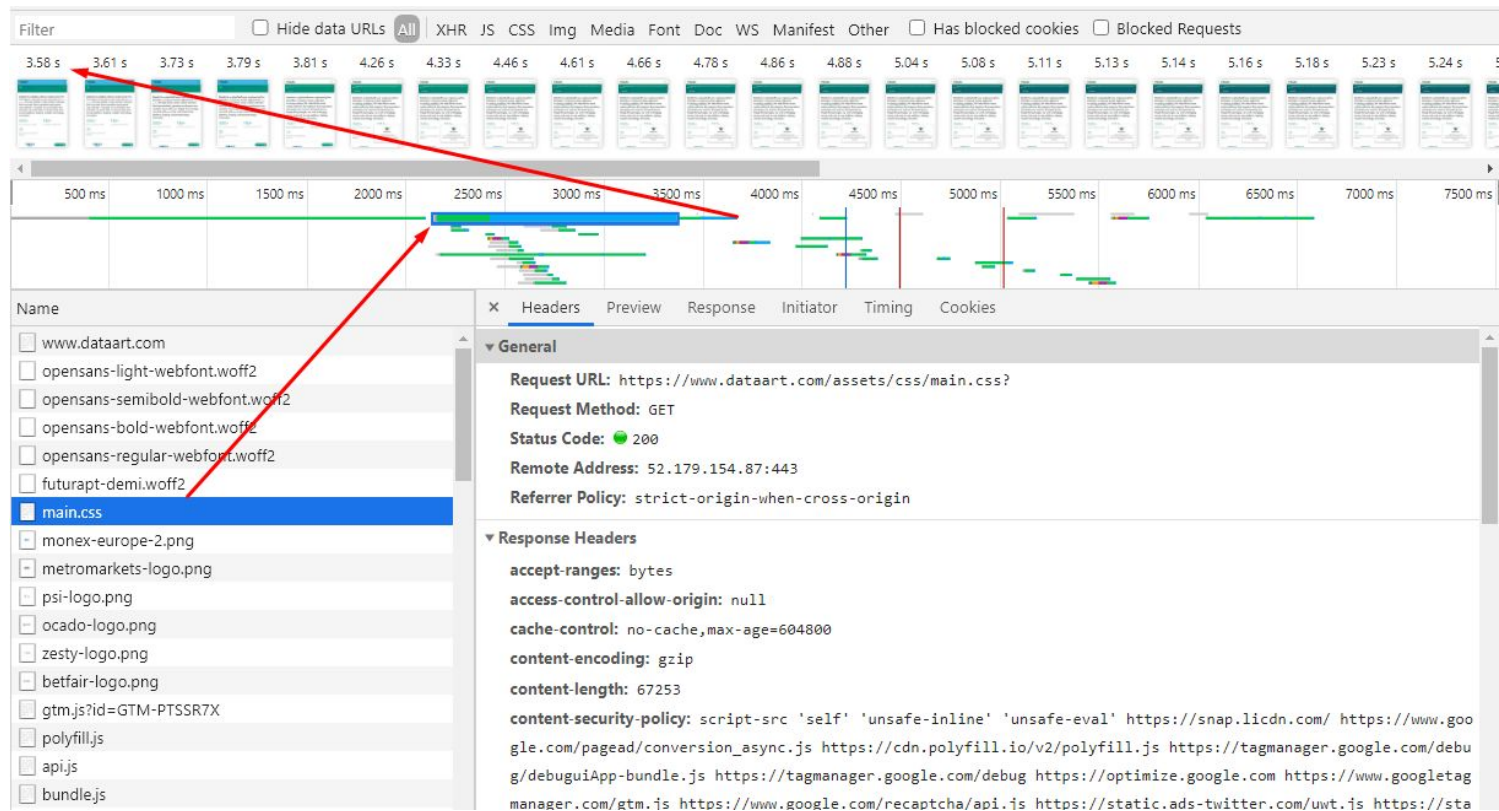
How the browser renders the page?

When the browser receives our HTML it parses it, breaking it down in to a vocabulary it understands, which is kept consistent in all browsers thanks to the HTML5 DOM Specification. It then runs through a series of steps to construct and render the page. Here's the very high level overview.

1. Use the HTML to create the Document Object Model (DOM).
2. Use the CSS to create the CSS Object Model (CSSOM).
3. Execute the Scripts on the DOM and CSSOM.
4. Combine the DOM and CSSOM to form the Render Tree.
5. Use the Render Tree to Layout the size and position of all elements.
6. Paint in all the pixels.



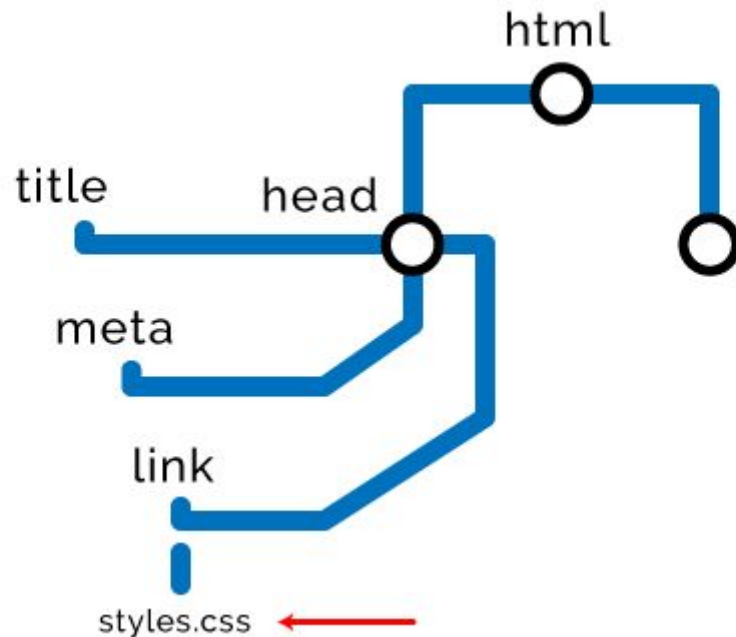
dataart.com



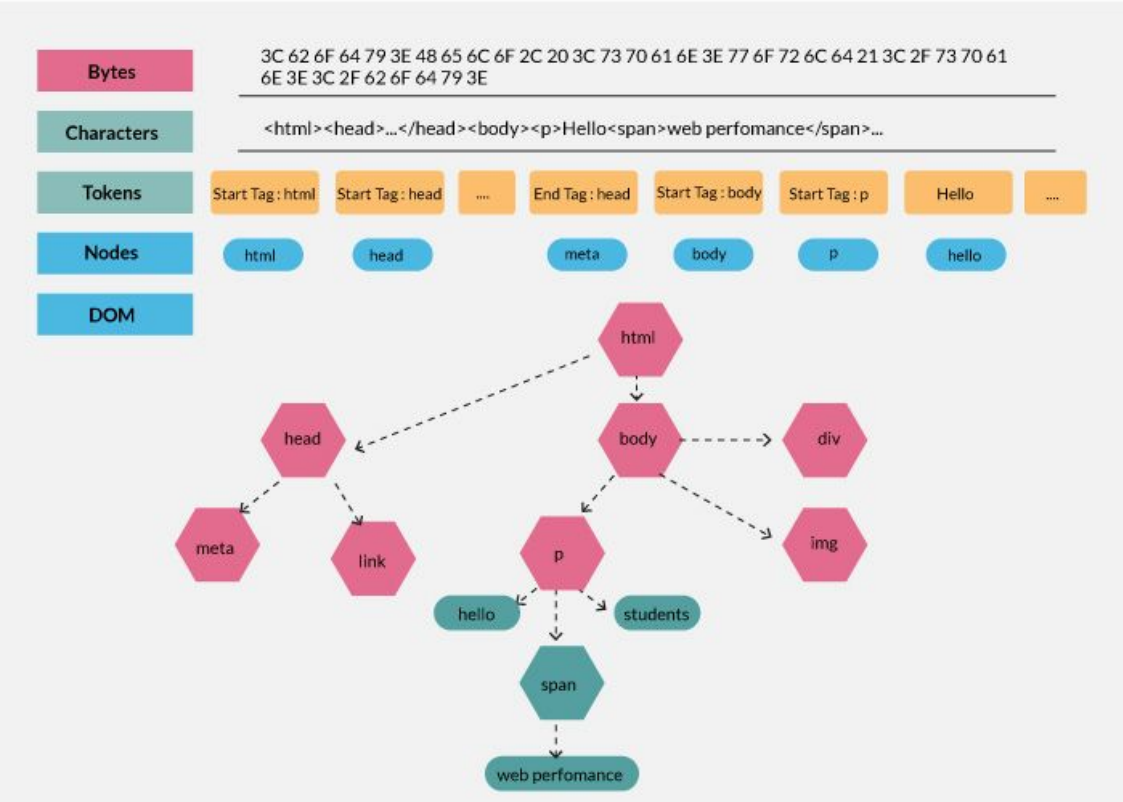
Step one—HTML

The browser starts reading the markup from top to bottom and uses it to create the DOM by breaking it down into Nodes.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>The "Click the button" page</title>
5      <meta charset="UTF-8">
6      <link rel="stylesheet" href="styles.css" />
7    </head>
8
9    <body>
10     <h1>
11       Click the button.
12     </h1>
```



HTML Parsing Process



RULE #2

Styles at the top and scripts at the bottom.

While there are exceptions and nuances to this rule, the general idea is to load styles as early as possible and scripts as late as possible. The reason for this is that scripts require the HTML and CSS to have finished parsing before they execute, therefore we put styles up high so they have ample time to compute before we compile and execute our scripts at the bottom.

RULE #3

Minification and compression.

This applies to all content we're delivering, including HTML, CSS, JavaScript, images and other assets.

Minification removes any redundant characters, including whitespace, comments, extra semicolons, etc.

RULE #4

Preload, Prefetch, Prerender, Preconnect.

These four html attributes allow us to preemptively load resources or make network connections that our user may need in the future.

- preload resources for the current page navigation.
- prefetch resources for future navigations.
- prerender for fully renders a page in the background.
- preconnect for future network requests and conducts.

RULE #5

Accessibility.

While this won't make your page download any faster it will drastically increase the satisfaction of impaired users. Make sure to provide for everyone! Use **aria** labels on elements, provide **alt** text on images and all other accessibility features.

Step two—CSS

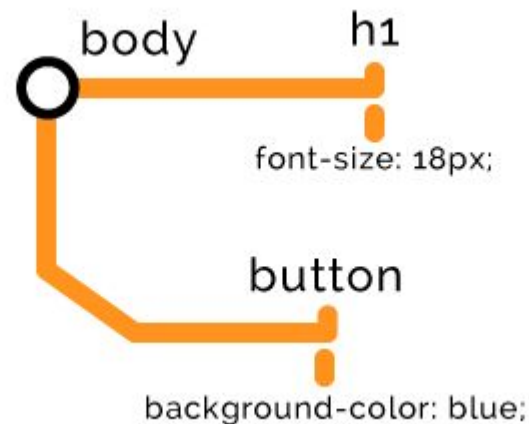
When it finds any style related Nodes, i.e. external, internal or inline styles, it stops **rendering** the DOM and uses these Nodes to create the CSSOM. That's why they call CSS “**Render Blocking**”.

Construction of the CSSOM blocks the rendering of the page so we want to load styles as early as possible in the tree, make them as lightweight as possible, and defer loading of them where efficient.

Types of styles

```
1  // External stylesheet
2  <link rel="stylesheet" href="styles.css">
3
4  // Internal styles
5  <style>
6    h1 {
7      font-size: 18px;
8    }
9  </style>
10
11 // Inline styles
12 <button style="background-color: blue;">Click me</button>
```

CSSOM



Inline styles

Advantages:

1. Testing.
2. Quick-fixes.
3. Small Websites.
4. Lower the HTTP Requests.

Disadvantages

1. Overriding.
2. Every Element.
3. Pseudo-elements.

Internal CSS

Advantages:

1. Cache Problem.
2. Pseudo-elements.
3. One style of same element.
4. No additional downloads.

Disadvantages

1. Multiple Documents.
2. Slow Page Loading.
3. Large File Size.

External CSS

Advantages:

1. Full Control of page structure.
2. Reduced file-size.
3. Less load time.
4. Higher page rankin.

Disadvantages

1. No disadvantages.

RULE #6

Less specificity.

There's one obvious drawback in there being physically more data to transfer with more elements chained together, thus enlarging the CSS file, but there's also a client-side computational drain on calculating styles with higher specificity.

```
1 // More specific selectors == bad
2 .header .nav .menu .link a.navItem {
3     font-size: 18px;
4 }
5
6 // Less specific selectors == good
7 a.navItem {
8     font-size: 18px;
9 }
```

RULE #7

Only deliver what you need.

This may sound silly or patronising, but if you've worked on the front end for any length of time you'll know one of the big problems with CSS is the unpredictability of deleting stuff. By design it's cursed to keep growing and growing.

To trim CSS down as much as possible use tools like the uncss package or alternative, there's a lot of choice.

RULE #8

Use media attributes.

```
1 // This css will download and block rendering of the page in all circumstances.
2 // media="all" is the default value and the same as not declaring any media attribute.
3 <link rel="stylesheet" href="mobile-styles.css" media="all">
4
5 // On mobile this css will download in the background and not disrupt the page load.
6 <link rel="stylesheet" href="desktop-styles.css" media="min-width: 590px">
```

Step three—JavaScript

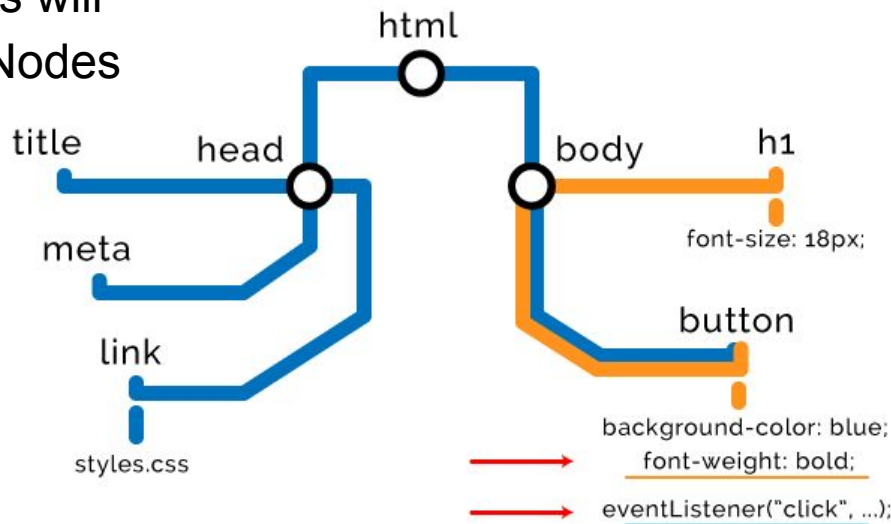
The browser carries on building DOM/CSSOM nodes until ... it finds any JavaScript Nodes, i.e. external or inline scripts.

Therefore the browser has to stop parsing nodes, finish building the CSSOM, execute the script, then carry on. That's why they call JavaScript **"Parser Blocking"**.

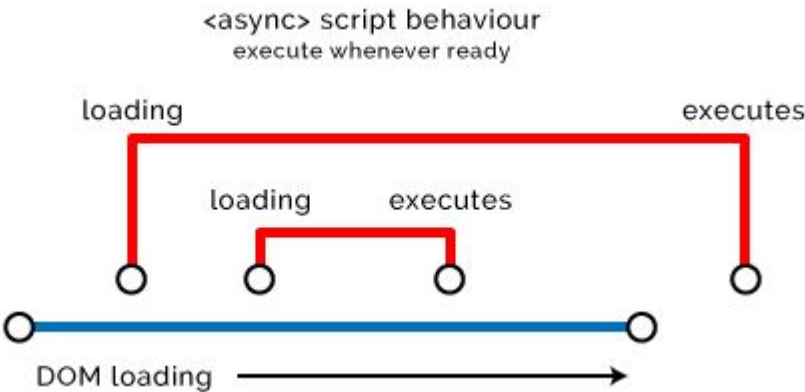
```
1 // An external script
2 <script src="app.js"></script>
3
4 // An internal script
5 <script>
6     alert("Oh, hello");
7 </script>
```

JavaScript, DOM and CSSOM

Browsers have something called a “**Preload Scanner**” that will scan the DOM for scripts and begin pre-loading them, but scripts will execute in order only after prior CSS Nodes have been constructed.

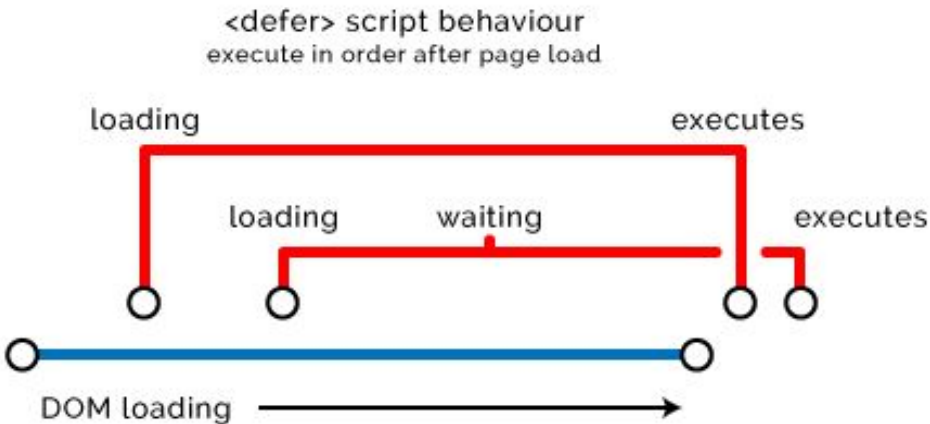


Load scripts asynchronously



```
1 <script src="async-script.js" async></script>
1 <script src="defer-script.js" defer></script>
```

Unfortunately async and defer do not work on inline scripts since browsers by default will compile and execute them as soon as it has them.



Let's repeat again

CSS blocks rendering

JavaScript blocks parser...

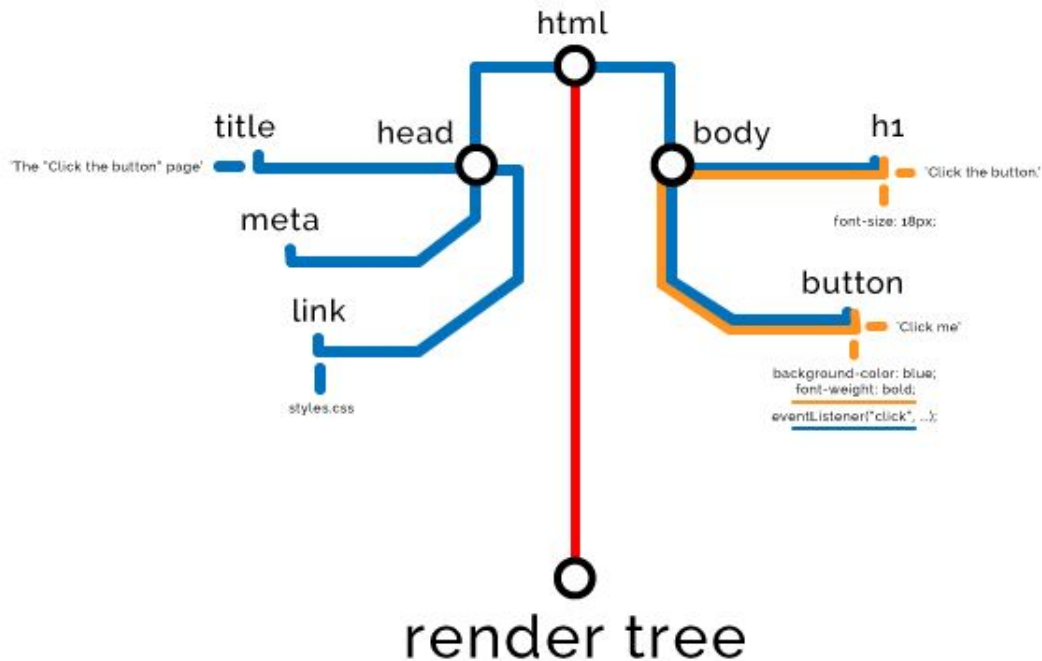
unless!

defer waits until parser is finished

async executes as soon as it loads

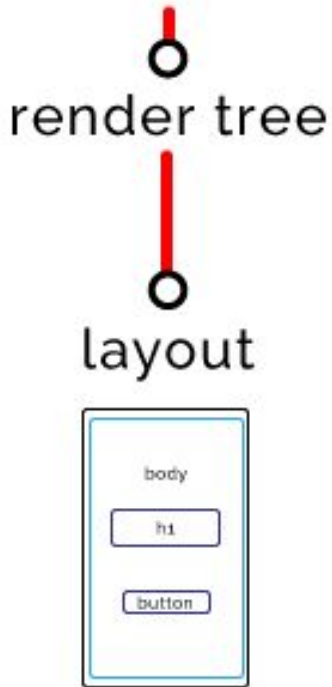
Step four—Render Tree

Once all Nodes have been read and the DOM and CSSOM are ready to be combined, the browser constructs the Render Tree. If we think of Nodes as words, and the Object Models as sentences, then the Render Tree is a full page. Now the browser has everything it needs to render the page.



Step five—Layout

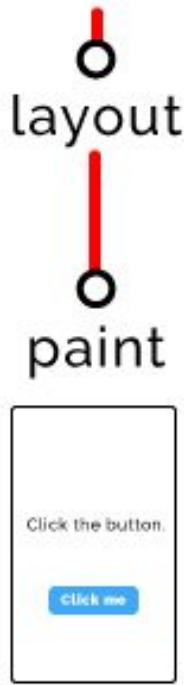
The Layout phase, determining the size and position of all elements on the page.



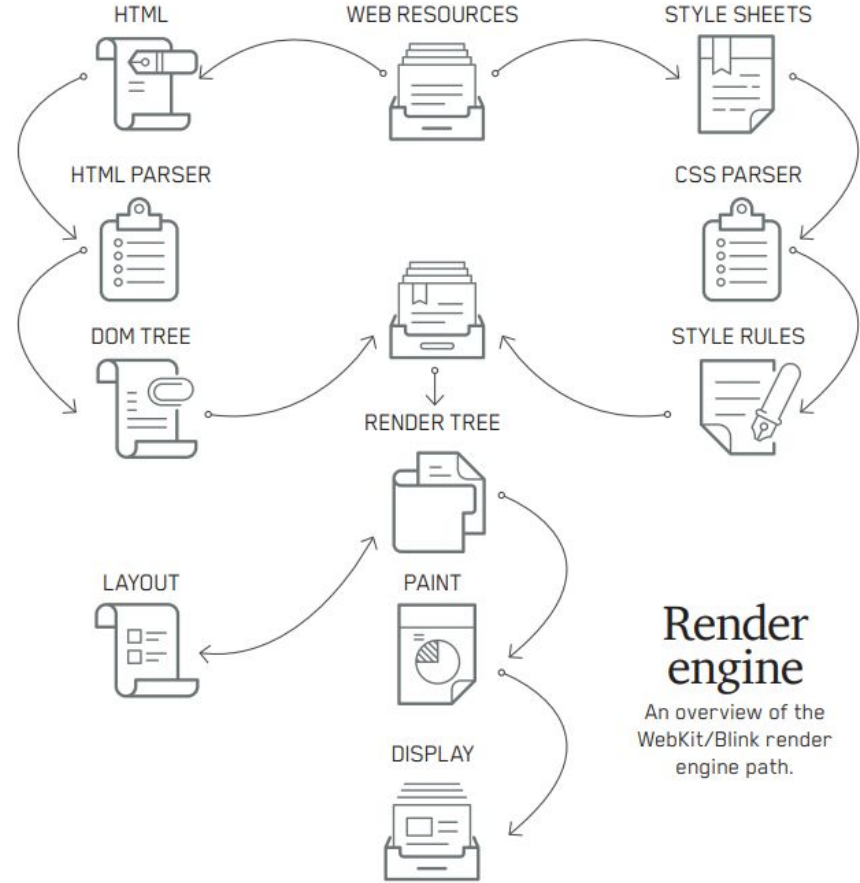
Step six—Paint

And finally we enter the Paint phase where we actually rasterize pixels on the screen, ‘painting’ the page for our user.

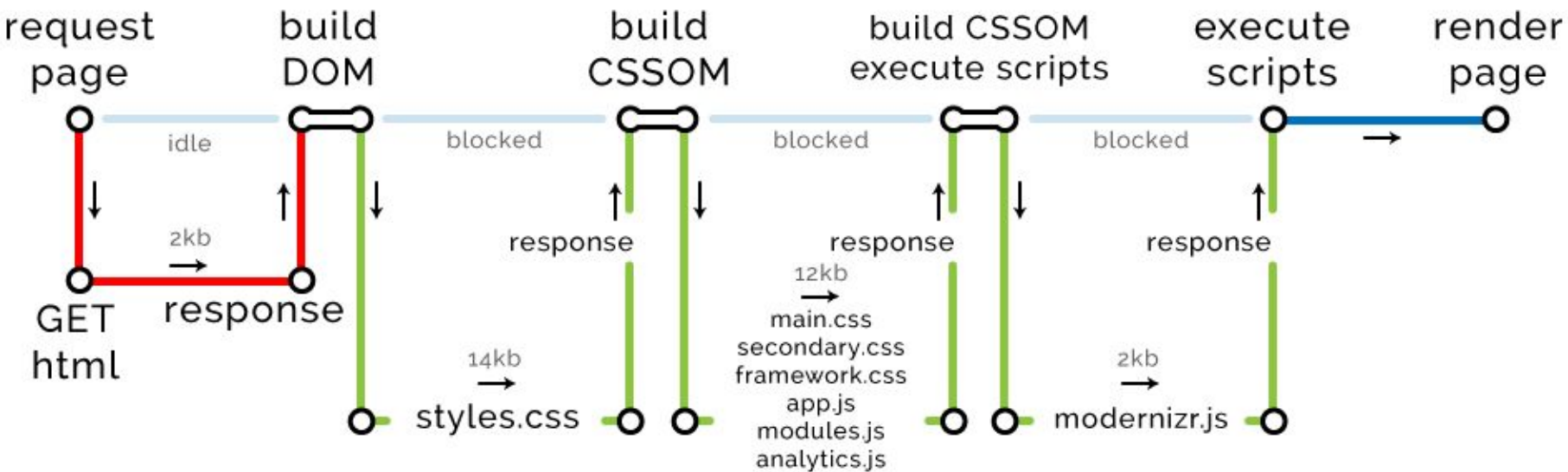
If JavaScript events change any part of the page it causes a redraw of the Render Tree (4) and forces us to go through Layout (5) and Paint (6) again. Modern browsers are smart enough to only conduct a partial redraw but we can’t rely on this being efficient or performant.



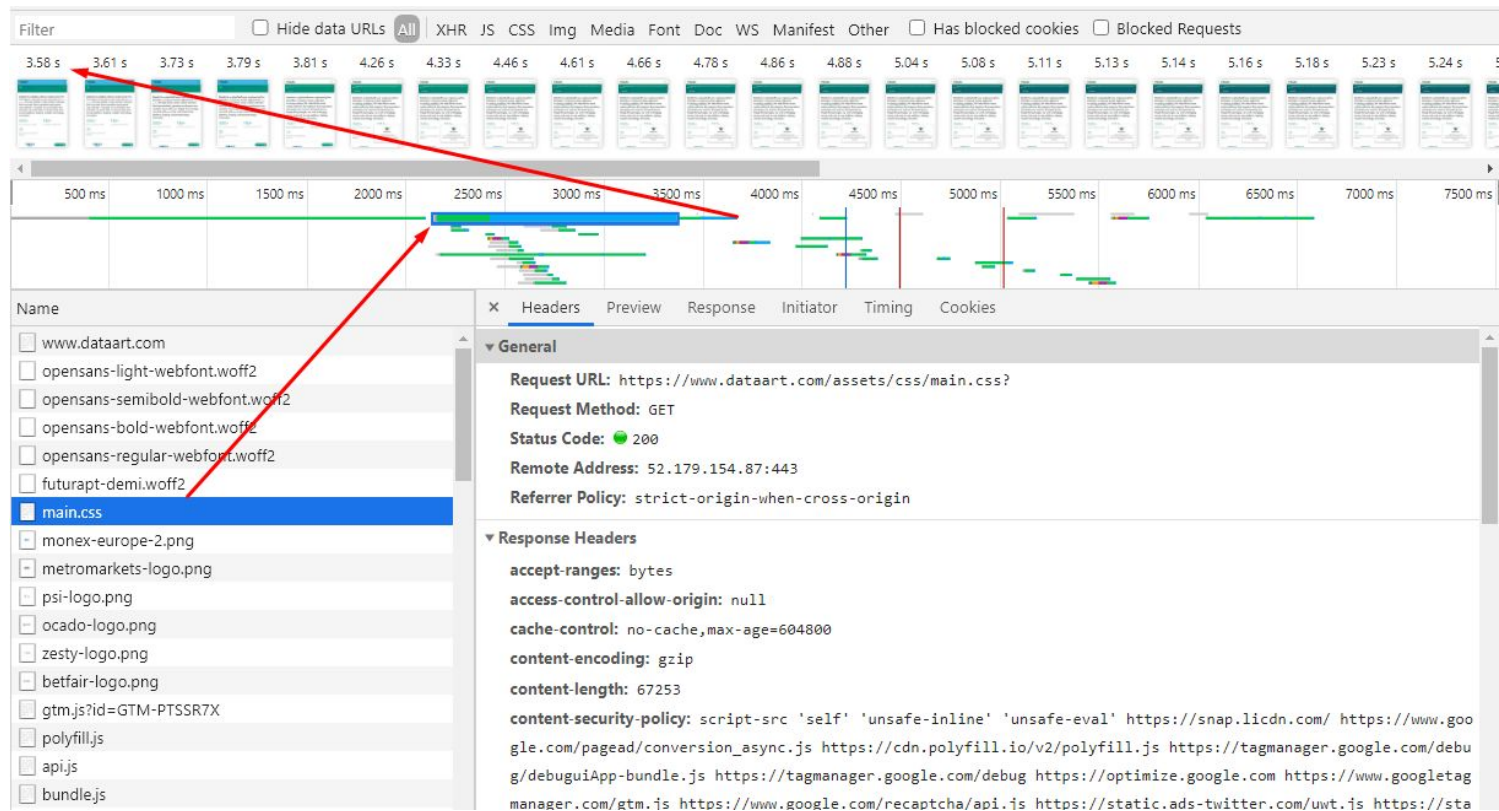
All together



Critical Path Length



dataart.com



Google's current standard

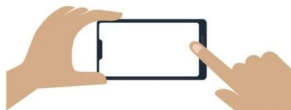
Core Web Vitals



(Loading)

LCP

Largest Contentful Paint



(Interactivity)

FID

First Input Delay



(Visual Stability)

CLS

Cumulative Layout Shift



RULE #9

Get good at Chrome Developer Tools.

DevTools are sooo amazing.

<https://developers.google.com/web/tools/chrome-devtools>

Q&A

CSS Specificity

If there are two or more conflicting CSS rules that point to the same element, the browser follows some rules to determine which one is most specific and therefore wins out.

Think of specificity as a score/rank that determines which style declarations are ultimately applied to an element.

If you have the same rule names, the browser chooses the latter.

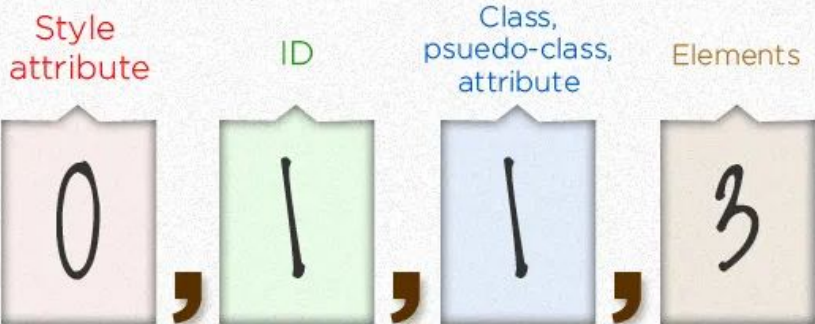
Specificity Hierarchy

Every selector has its place in the specificity hierarchy. There are four categories which define the specificity level of a selector:

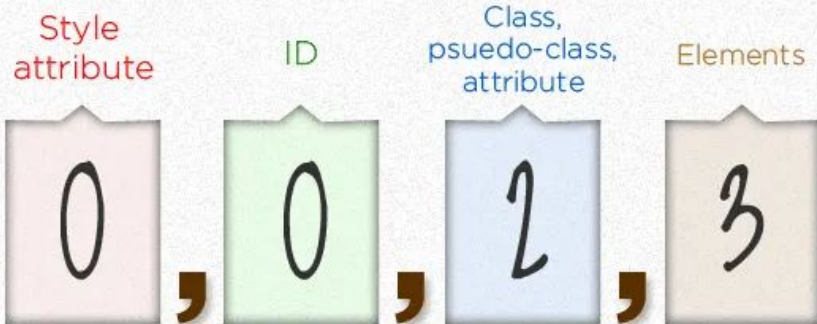
- **Inline styles** - An inline style is attached directly to the element to be styled.
Example: `<h1 style="color: #ffffff;">`.
- **IDs** - An ID is a unique identifier for the page elements, such as `#navbar`.
- **Classes, attributes and pseudo-classes** - This category includes `.classes`, `[attributes]` and pseudo-classes such as `:hover`, `:focus` etc.
- **Elements and pseudo-elements** - This category includes element names and pseudo-elements, such as `h1`, `div`, `:before` and `:after`.

How to Calculate Specificity?

ul#nav li.active a



body.ie7 .col_3 h2 ~ h2



How to Calculate Specificity?

#footer *:not(nav) li

Style attribute	ID	Class, psuedo-class, attribute	Elements
0	1	0	2

ul > li ul li ol li:first-letter

Style attribute	ID	Class, psuedo-class, attribute	Elements
0	0	0	7

Important Notes

- The universal selector (*) has no specificity value (0,0,0,0)
- Pseudo-elements (e.g. :first-line) get 0,0,0,1 unlike their psuedo-class brethren which get 0,0,1,0
- The pseudo-class :not() adds no specificity by itself, only what's inside it's parentheses.
- The **!important** value appended a CSS property value is an automatic win. It overrides even inline styles from the markup. The only way an !important value can be overridden is with another !important rule declared later in the CSS and with equal or great specificity value otherwise. You could think of it as adding **1,0,0,0,0** to the specificity value.

RULE #10

Don't forget about RULE #6.

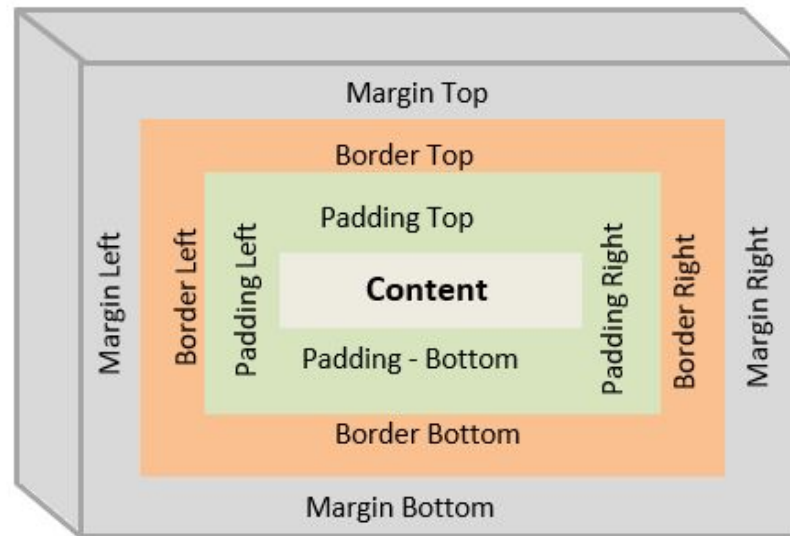
Less specificity.

```
1 // More specific selectors == bad
2 .header .nav .menu .link a.navItem {
3     font-size: 18px;
4 }
5
6 // Less specific selectors == good
7 a.navItem {
8     font-size: 18px;
9 }
```

Q&A

The CSS Box Model

1. **Margin** - is the area outside of the element. Margin always transparent. (may have a negative thickness)
2. **Border** - appear around HTML elements, on the outer edge of any padding.
3. **Padding** - clears an area around the content and inherits the background color of the content area.
4. **Content** - content area (text, images and etc).



Block vs Inline Elements

HTML elements come in two different varieties: inline and block elements. They have some striking differences that also have an impact on the usage of the CSS box model.

One of the main features of block elements is that they take up the entire space of the container they are placed in. Unless otherwise instructed (meaning via CSS), they will stretch out to occupy however much space is available, moving any other elements below them.

In addition to that, block elements can contain other block or inline elements and will automatically adjust their height to fit their content.

Block vs Inline Elements

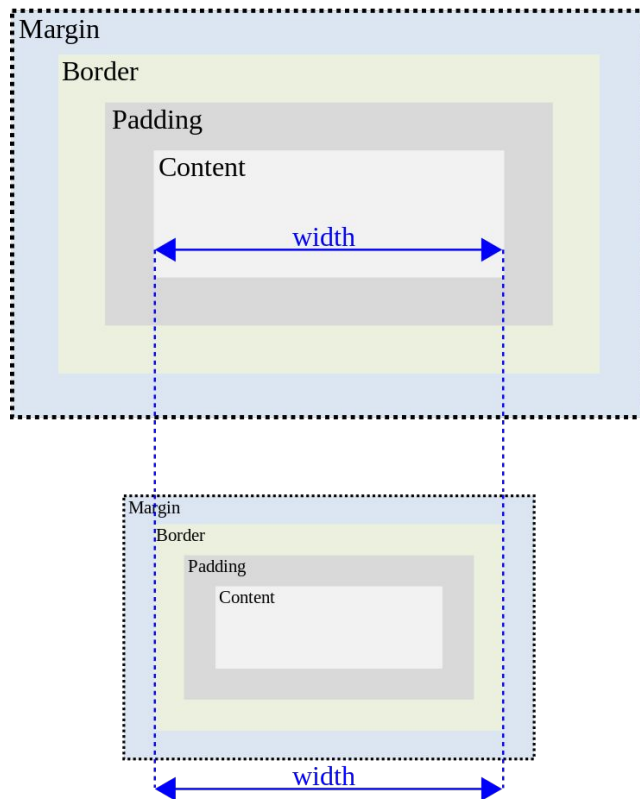
Inline elements are actually something else. In contrast, to block elements, inline elements take up only as much space as they need. They also don't move to a new line or push other elements below them.

However, because of these behaviors, much of the CSS box model only applies to block elements.

The First Question: Did you set CSS **box-sizing** Property?

Value	Description
content-box	Default. The width and height properties (and min/max properties) includes only the content. Border and padding are not included
border-box	The width and height properties (and min/max properties) includes content, padding and border
initial	Sets this property to its default value.
inherit	Inherits this property from its parent element.

The CSS Box Model & box-sizing Property



A Quick Note on Element Sizes

Without the CSS box-sizing Property

$\text{width} + \text{padding-left} + \text{padding-right} + \text{border-left} + \text{border-right} = \text{total width}$

$\text{height} + \text{padding-top} + \text{padding-bottom} + \text{border-top} + \text{border-bottom} = \text{total height}$

This means: When you set the width/height of an element, the element often appears bigger than you have set (because the element's border and padding are added to the element's specified width/height).

With the CSS box-sizing Property

The box-sizing property allows us to include the padding and border in an element's total width and height.

Since the result of using the **box-sizing: border-box;** is so much better, many developers want all elements on their pages to work this way.

What if CSS Box Model values are undeclared?

If padding or borders are undeclared, they are either zero (likely if you are using a **css reset**) or the browser default value.

If the width of a box is undeclared (and the box is a block level element), things get a little weirder. Let's start with that, and then move on to some other good-to-know stuff about the box model.

The Default Width of Block Level Boxes

If you don't declare a width, and the box has **static** or **relative** positioning, the width will remain 100% in width and the padding and border will push inwards instead of outward. But if you explicitly set the width of the box to be 100%, the padding will push the box outward as normal.

The lesson here being that the default width of a box isn't really 100%.

Parent is 300px

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet quam egestas semper.

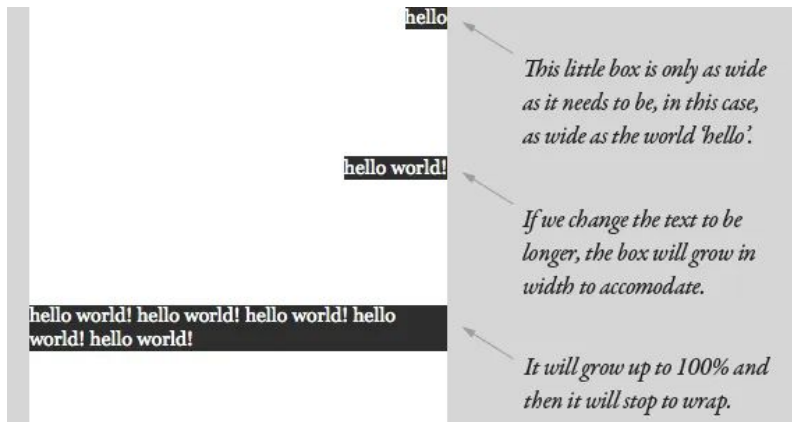
This box has 20px of padding but no set width.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet quam egestas semper.

This box has 20px of padding and width set to 100%

Absolute Boxes with No Width

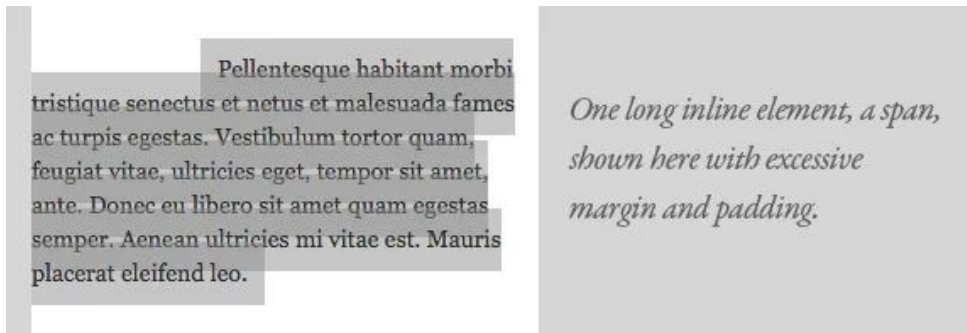
Absolutely positioned boxes that have no width set on them behave a bit strangely. Their width is only as wide as it needs to be to hold the content. So if the box contains a single word, the box is only as wide as that word renders. If it grows to two words, it'll grow that wide. This should continue until the box is 100% of the parent's width (the nearest parent with **relative** positioning, or **browser window**) and then begin to wrap. It feels natural and normal for boxes to expand vertically to accommodate content, but it just feels strange when it happens horizontally. That strange feeling is warranted, as there are plenty of quirks in how different browsers handle this, not to mention just the fact that text renders differently across platforms.



Inline Elements are Boxes Too

We've been kind of focusing on boxes as block-level elements here. It's easy to think of block-level elements as boxes, but inline elements are boxes too. **Think of them as really really long and skinny rectangles**, that just so happen to wrap at every line. They are able to have margin, padding, borders just like any other box.

Inline Elements are Boxes Too



The wrapping is what makes it confusing. A left margin as shown above pushes the box to the right as you would suspect, but only the first line, as that is the beginning of the box. Padding is applied above and below the text like it should be, and when it wraps it ignores the line above its padding and begins where the line-height dictates it should begin. The background was applied with transparency to see how it works more clearly.

The Rules of Margin Collapse

In CSS, adjacent margins can sometimes overlap. This is known as “margin collapse”, and it has a reputation for being quite dastardly.

The good news is that once we understand the rules behind this notoriously-confusing mechanism, it becomes a lot clearer, and a lot less surprising.

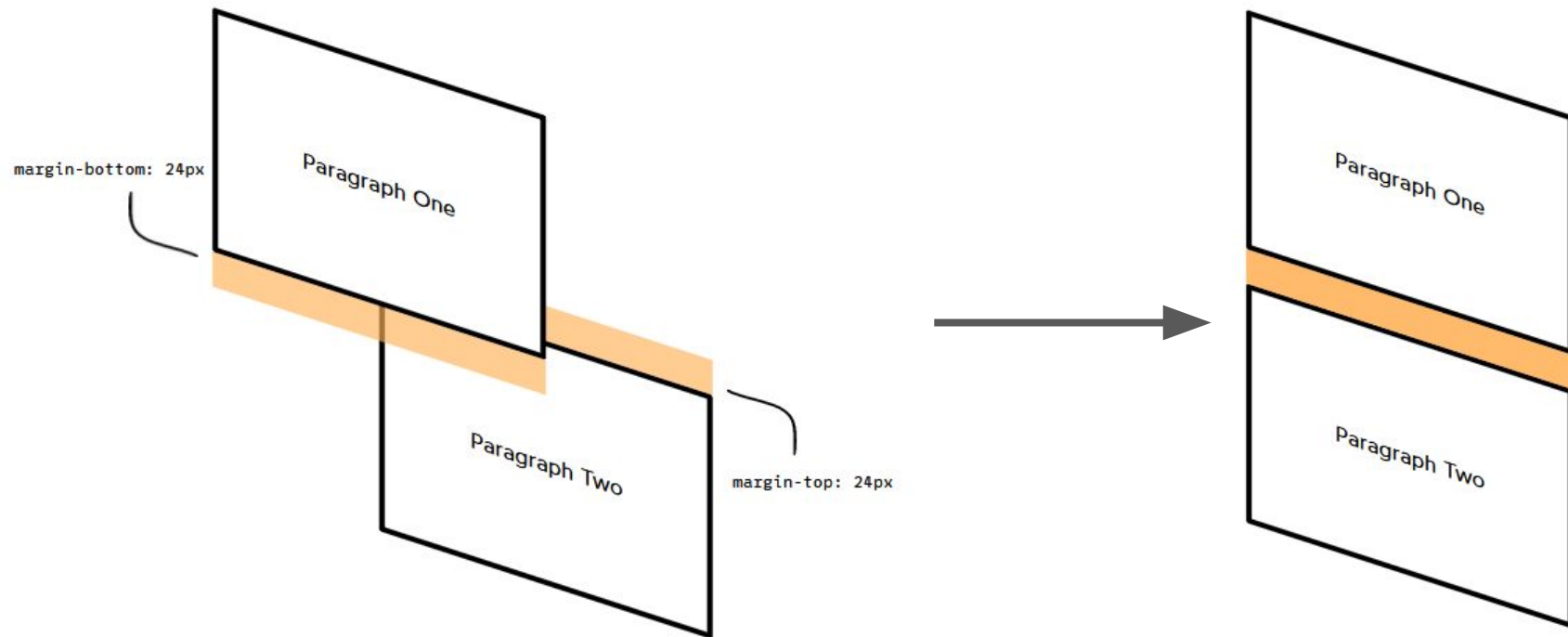
RULE #11

Only vertical margins collapse

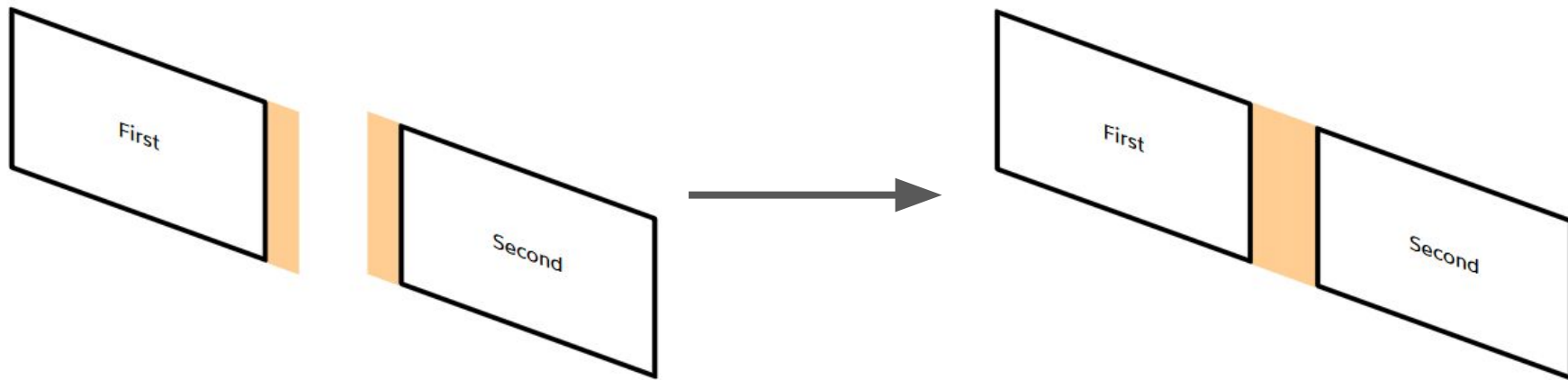
When margin-collapse was added to the CSS specification, the language designers made a curious choice: horizontal margins shouldn't collapse.

In the early days, CSS wasn't intended to be used for layouts. The people writing the spec were imagining headings and paragraphs, not columns and sidebars.

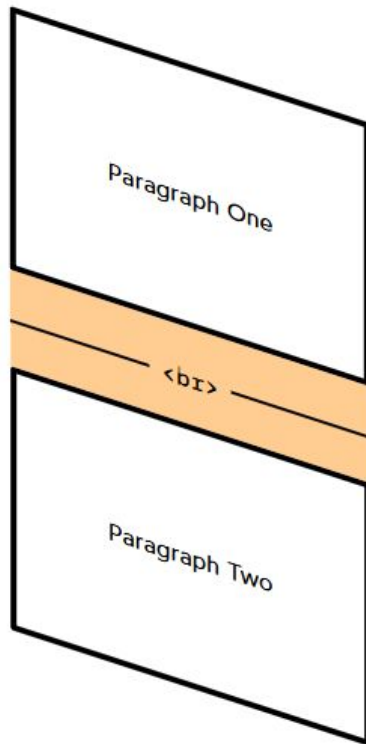
Vertical margins



Horizontal margins



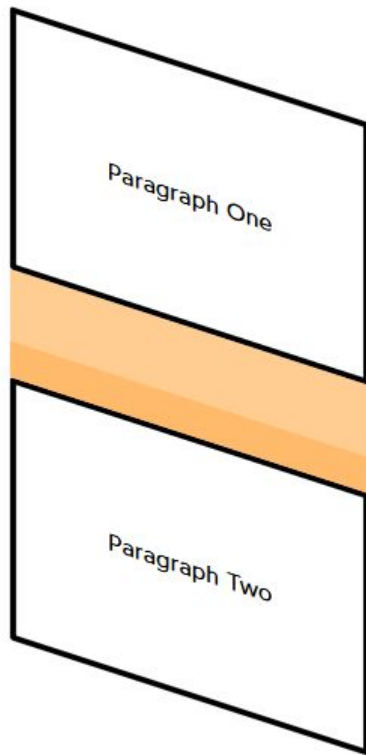
How to fix? Only adjacent elements collapse.



The bigger margin wins

What about when the margins are asymmetrical? Say, the top element wants 72px of space below, while the bottom element only needs 24px?

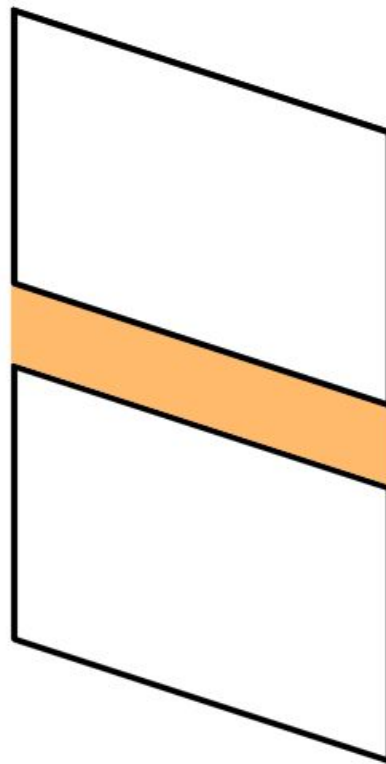
The bigger number wins.



Nesting doesn't prevent collapsing

```
<style>
  p {
    margin-top: 48px;
    margin-bottom: 48px;
  }
</style>

<div>
  <p>Paragraph One</p>
</div>
<p>Paragraph Two</p>
```

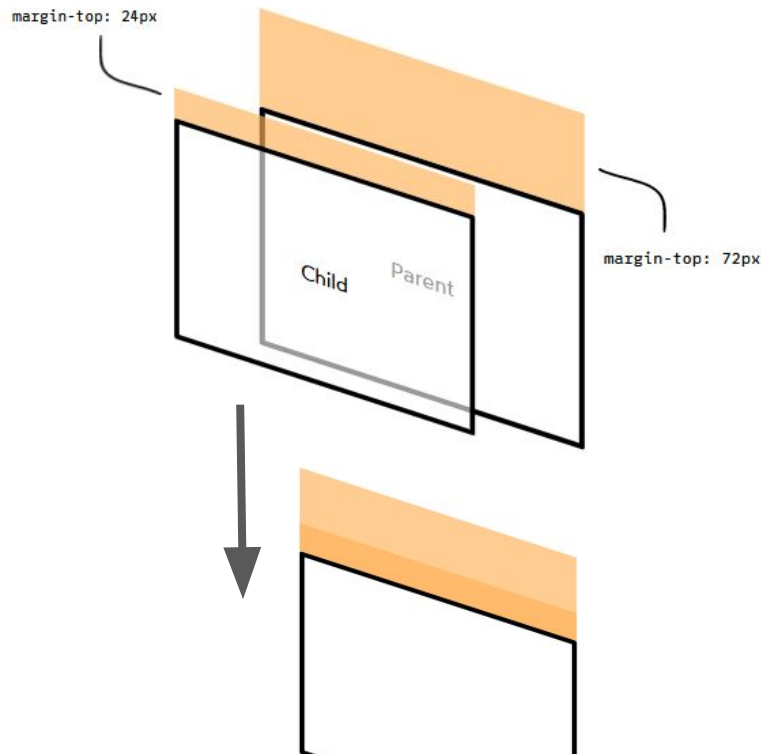


Margins can collapse in the same direction

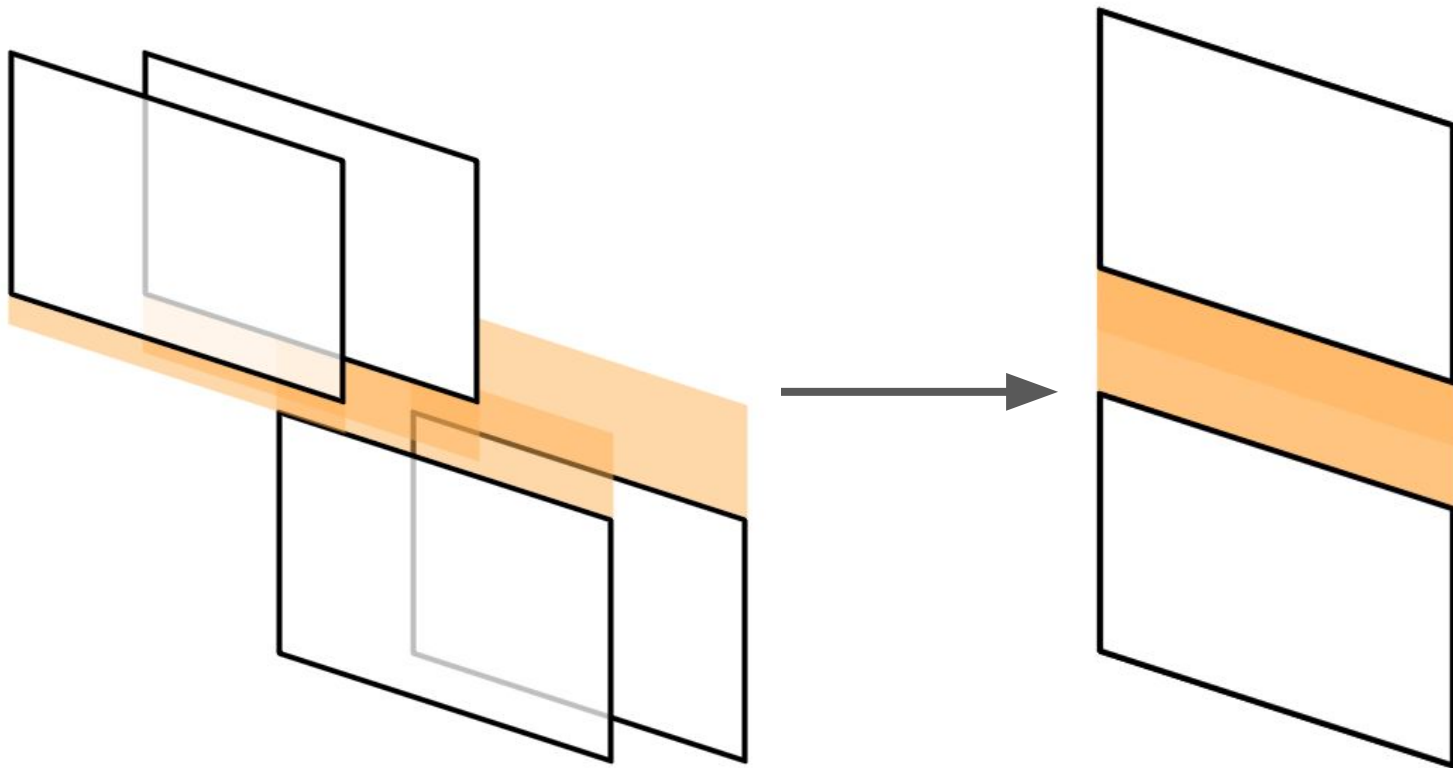
```
<style>
  .parent {
    margin-top: 72px;
  }

  .child {
    margin-top: 24px;
  }
</style>

<div class="parent">
  <p class="child">Paragraph One</p>
</div>
```



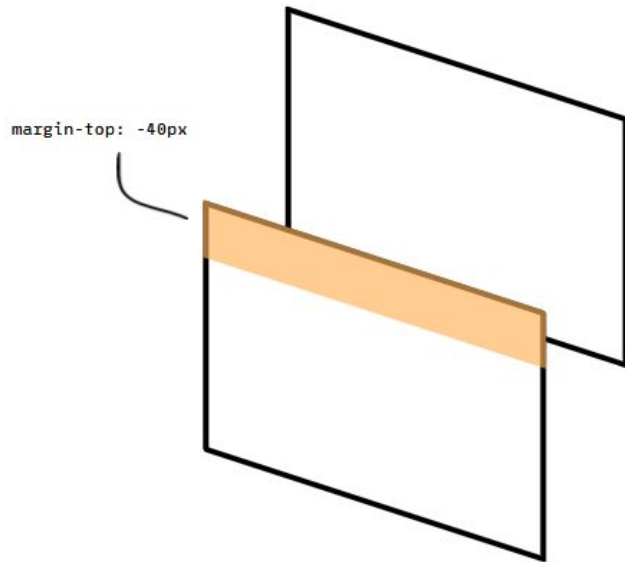
More than two margins can collapse



Negative margins

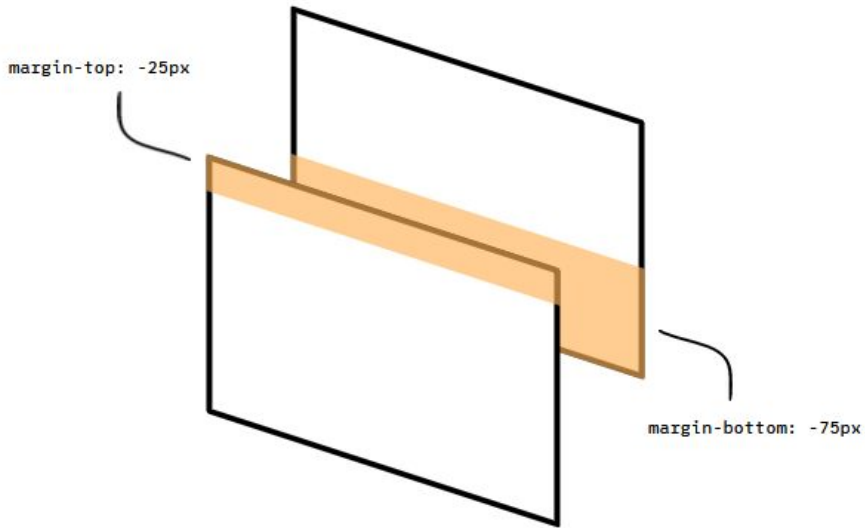
Finally, we have one more factor to consider: negative margins.

Negative margins allow us to reduce the space between two elements. It lets us pull a child outside its parent's bounding box, or reduce the space between siblings until they overlap.



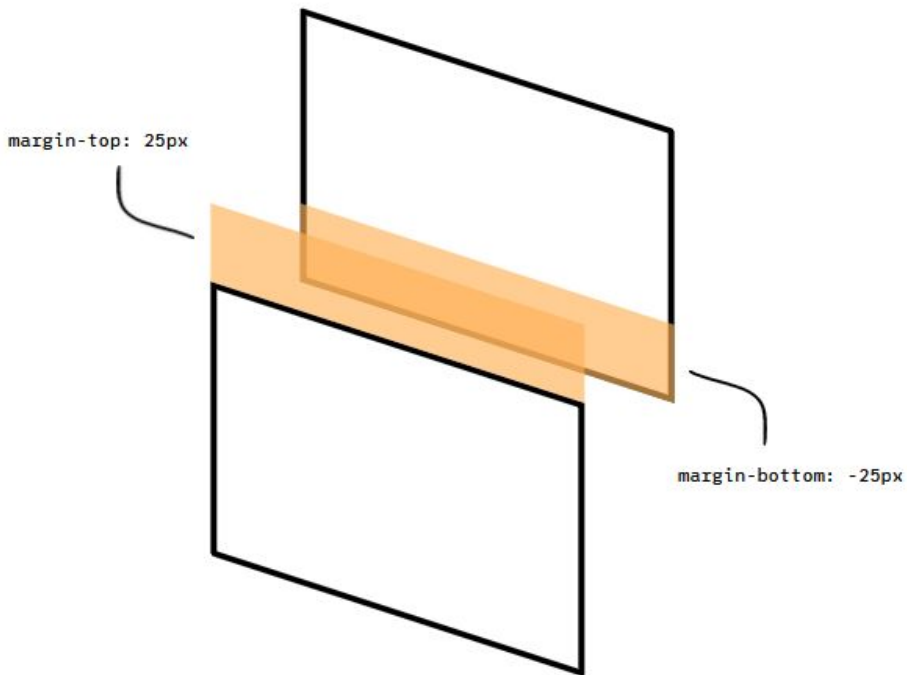
How do negative margins collapse?

Well, it's actually quite similar to positive ones! The negative margins will share a space, and the size of that space is determined by the most significant negative margin. In this example, the elements overlap by 75px, since the more-negative margin (-75px) was more significant than the other (-25px).



What about when negative and positive margins are mixed?

In this case, the numbers are added together. In this example, the -50px negative margin and the 50px positive margin cancel each other out and have no effect, since $-50\text{px} + 50\text{px}$ is 0.

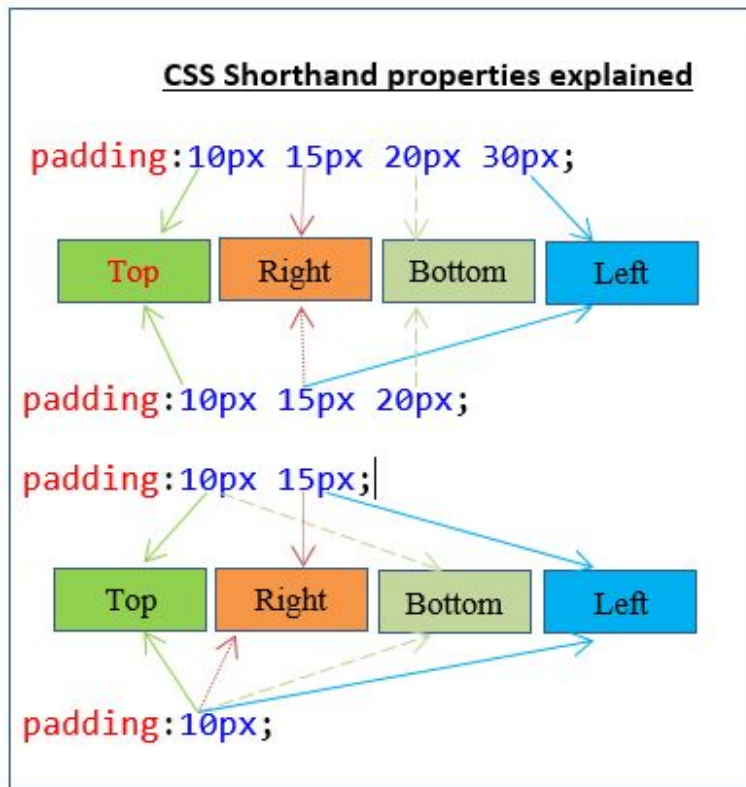


Multiple positive and negative margins

If there are more than 2 margins involved, the algorithm looks like this:

- Find the largest positive margin
- Find the largest / negative margin
- Add those two numbers together

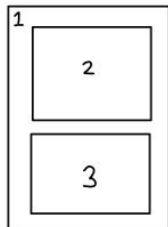
CSS Shorthand properties



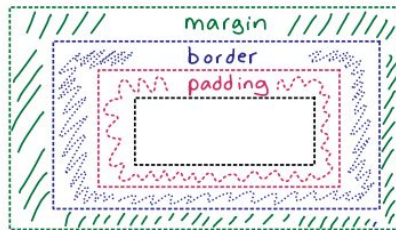
The CSS Box Model Rules

every HTML element
is in a box

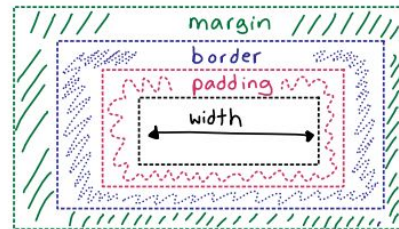
```
<div class="1">
  <div class="2" />
  <div class="3" />
</div>
```



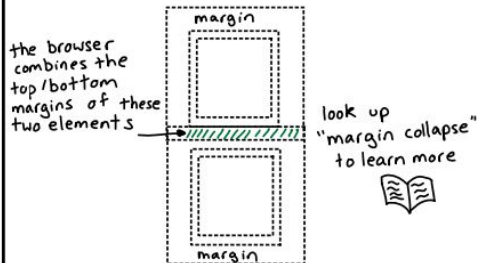
boxes have **padding**,
borders, and a **margin**



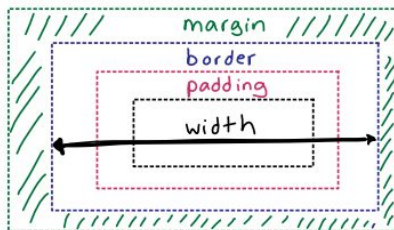
Width doesn't include
margin/border/padding
by default



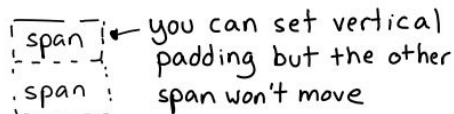
margins are allowed
to overlap sometimes



`box-sizing: border-box;`
includes **border + padding**
in the width



inline elements
ignore other inline
elements' vertical padding



Q&A

CSS **display** Property



display: inline;

Displays an element as an inline element (like ``). Any height and width properties will have no effect

Pellentesque *inline element* morbi tristique senectus
Donec eu libero sit amet quam egestas semper. Aenean

Pellentesque *inline element* morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat vitae,
ultrices eget, tempor sit amet, ante. Donec eu libero sit amet quam egestas
semper. Aenean ultricies mi vitae est. Mauris placerat eleifend leo.

display: inline-block;

Displays an element as an inline-level block container. The element itself is formatted as an inline element, but you can apply height and width values



display: block;

Displays an element as a block element (like <p>). It starts on a new line, and takes up the whole width

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat eleifend leo.

hi

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat eleifend leo.

display: flex;

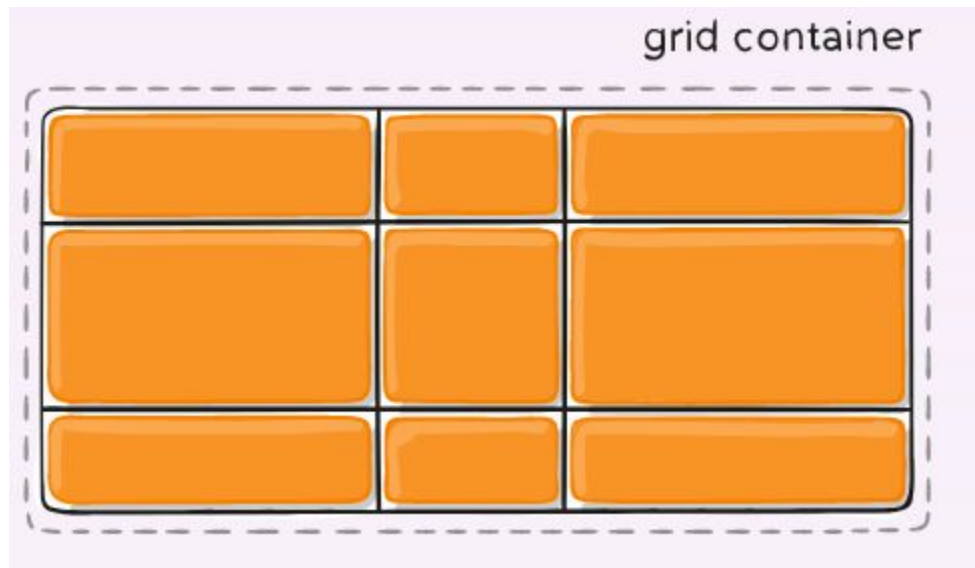
Displays an element as a block-level flex container.

container 



display: grid;

Displays an element as a block-level grid container



`display: none;`

The element is completely removed

Q&A

CSS **position** Property

The position CSS property sets how an element is positioned in a document. The top, right, bottom, and left properties determine the final location of positioned elements.

position: absolute | fixed | relative | static | sticky | inherit

position: static; ("IN-FLOW")

Every element has a static position by default, so the element will stick to the normal page flow. So if there is a **left/right/top/bottom/z-index** set then there will be **no effect** on that element.



position: relative; ("IN-FLOW" / "OUT-FLOW")

An element's original position remains in the flow of the document, just like the static value. But now left/right/top/bottom/z-index will work. The positional properties “nudge” the element from the original position in that direction.



position: absolute; ("OUT-FLOW")

The element is removed from the flow of the document and other elements will behave as if it's not even there whilst all the other positional properties will work on it.



position: fixed; (OUT-FLOW)

The element is removed from the flow of the document like absolutely positioned elements. In fact they behave almost the same, only fixed positioned elements are always relative to the document, not any particular parent, and are unaffected by scrolling.



position: sticky; ("IN-FLOW" / "OUT-FLOW")

The element is treated like a relative value until the scroll location of the viewport reaches a specified threshold, at which point the element takes a fixed position where it is told to stick.



position: inherit;

The position value doesn't cascade, so this can be used to specifically force it to, and inherit the positioning value from its parent.

Q&A

The Flexbox

The CSS flexbox can control the following aspects of the layout:

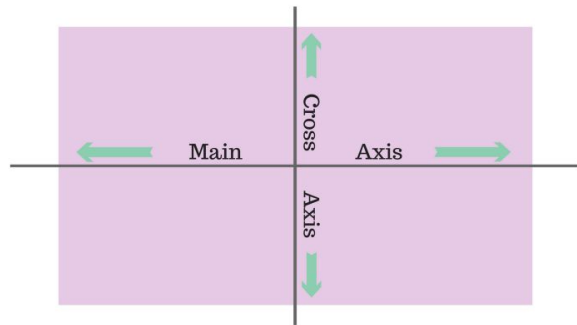
- The direction to which the items are displayed
- The wrapping of items when the window is resized
- The justification of items and space in between
- The vertical alignment of items
- The alignment of lines of items
- The order of items in a line
- The items' ability to grow or shrink when the window resizes
- The alignment of an individual item

Flexbox axis

The CSS flexbox has two axes: **the main** and **the cross**.

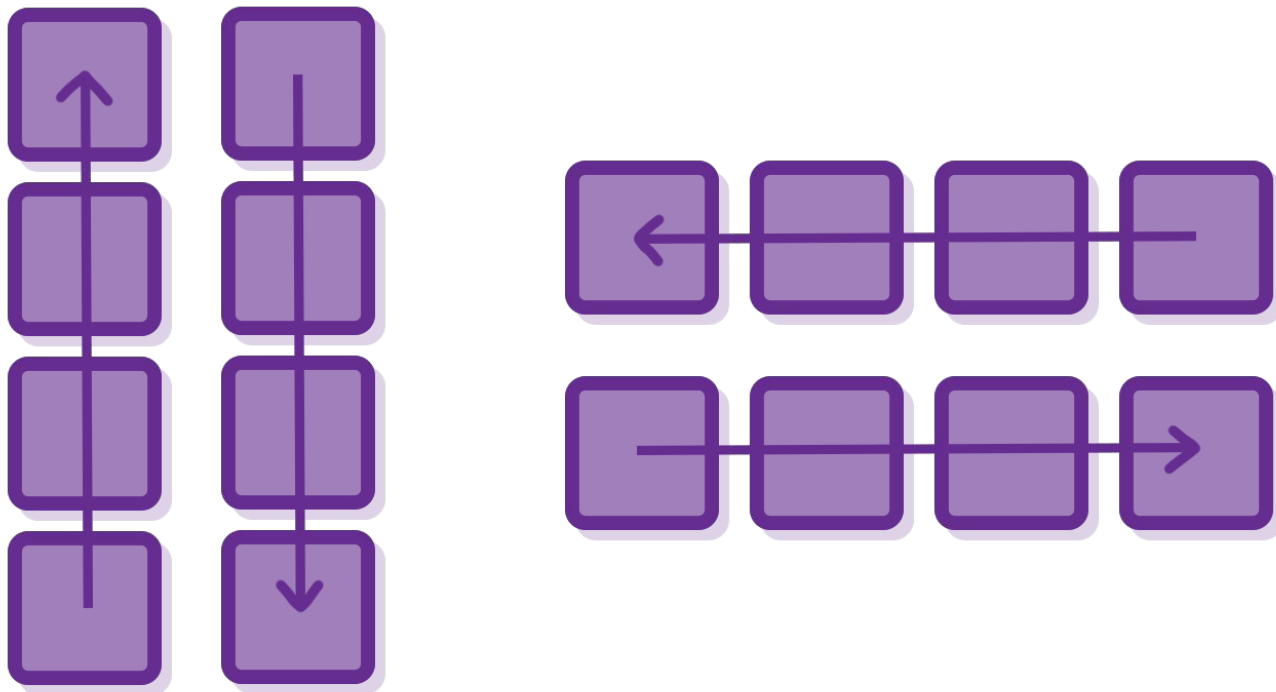
You define the main axis with the flex-direction property. It has four possible values: **row**, **row-reverse**, **column**, and **column-reverse**.

Cross axis always goes perpendicular (\perp) to the main axis. If the main axis has a row or row-reverse direction, the cross axis runs along a column or column-reverse direction (and vice versa).



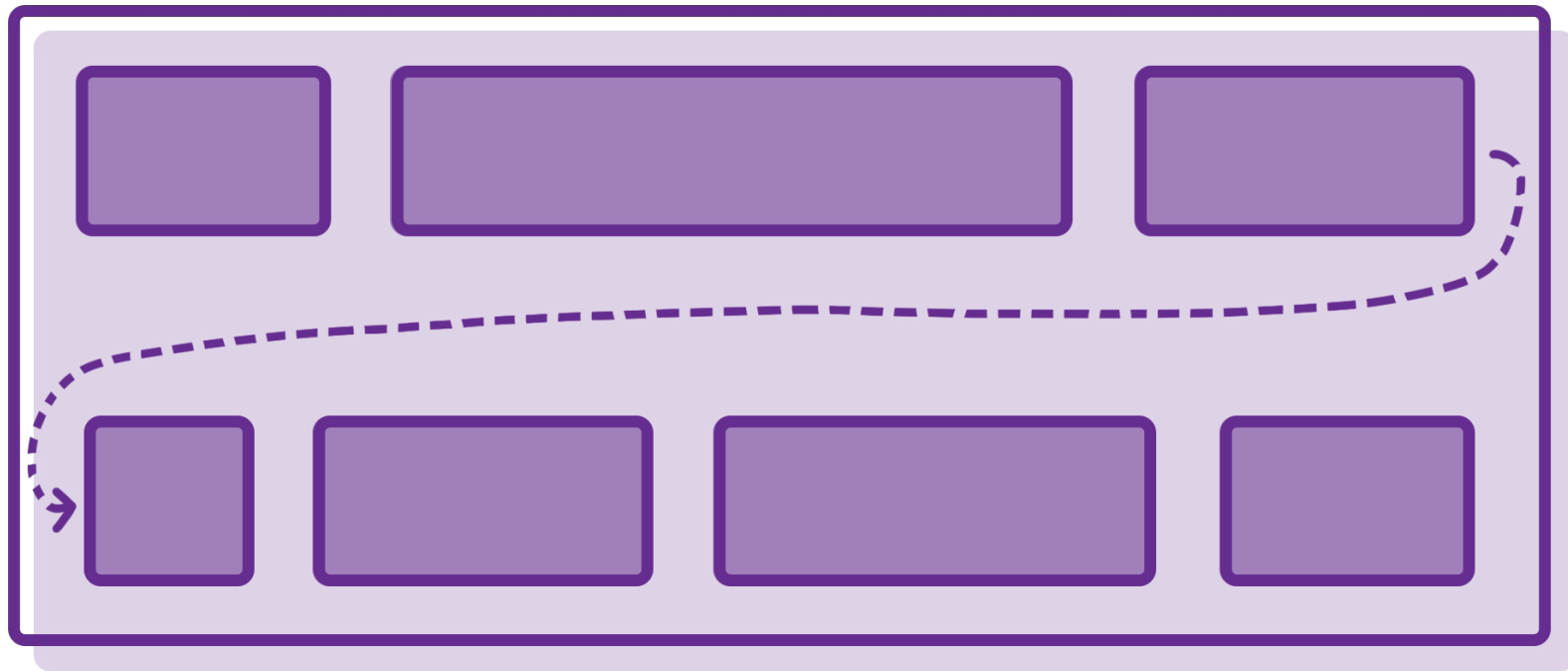
flex-direction

column, column-reverse, row, row-reverse



flex-wrap

nowrap, wrap, wrap-reverse



justify-content

flex-start



space-between



flex-end



space-around



center

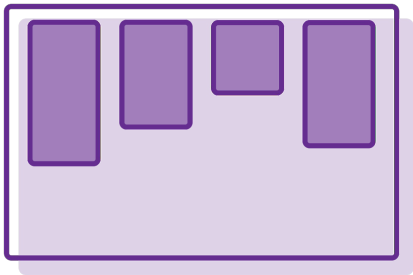


space-evenly

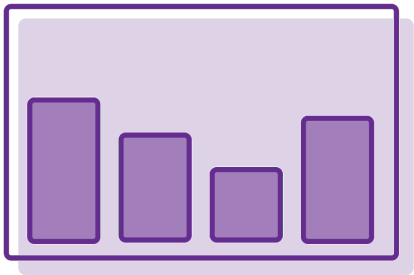


align-items

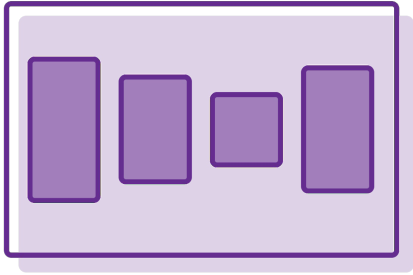
flex-start



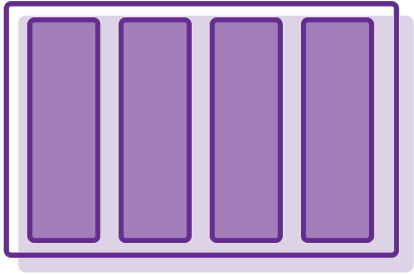
flex-end



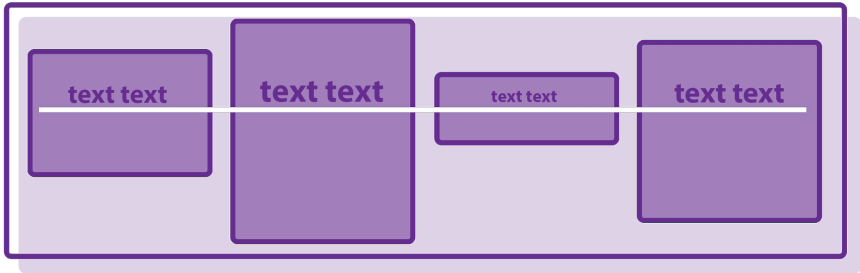
center



stretch

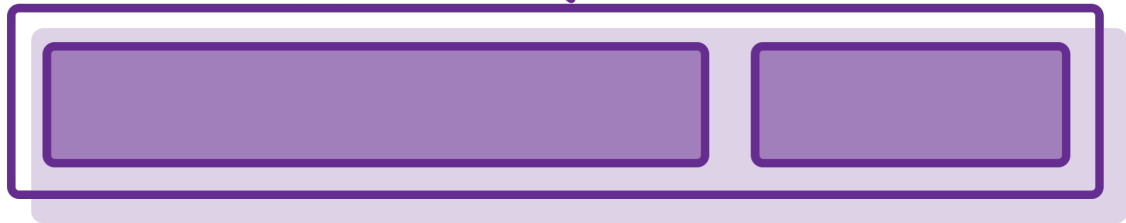


baseline



Flex containers & Flex items

container



items



Initial items values

flex-grow: 0; - property defines how much a flex item expands in relation to other items (if there is available space for that). You set its value using positive numbers without units. The number represents a proportion.

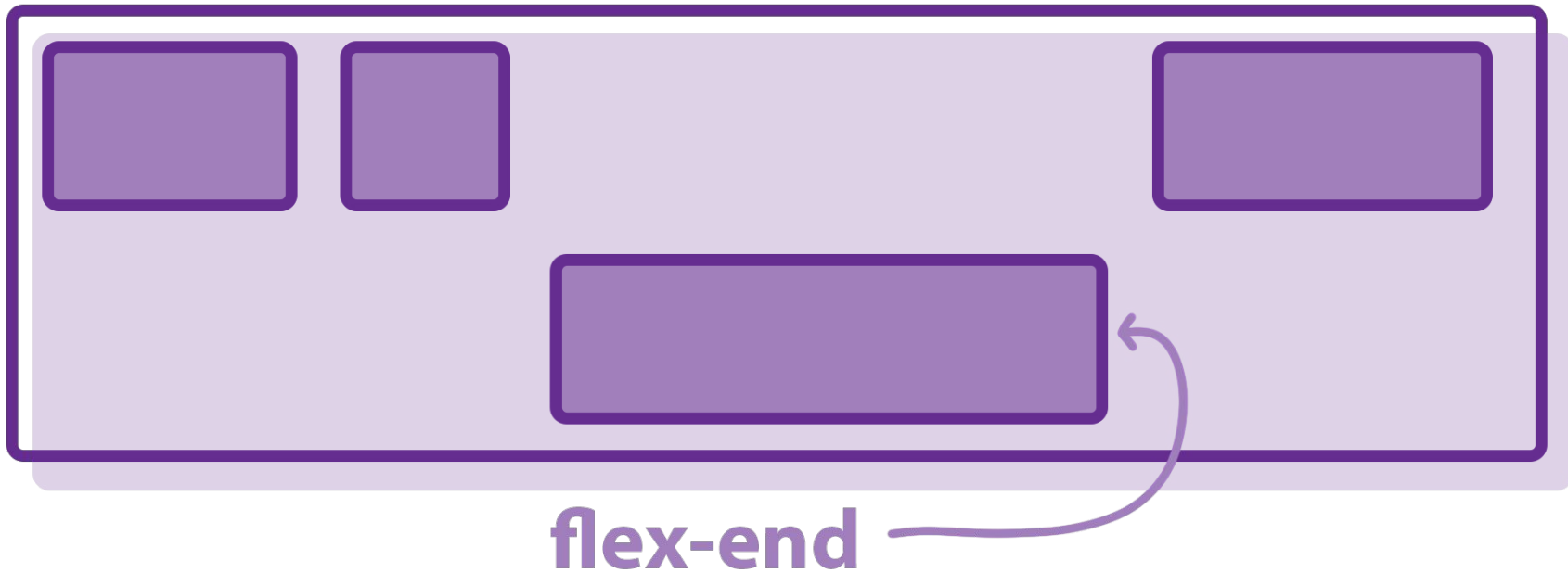
flex-shrink: 1; - property is similar to flex-grow. However, flex-shrink defines the proportion in which a flex item shrinks more than other items in that container (when there is not enough space to keep the original size). Therefore, flex-shrink decreases the size of items to make them fit into the container.

flex-basis: auto; - property defines the initial size of a flex item. It accepts length units or percentages.

```
.sherthandprops: { flex: 1 1 150px; }
```

align-self

flex-start



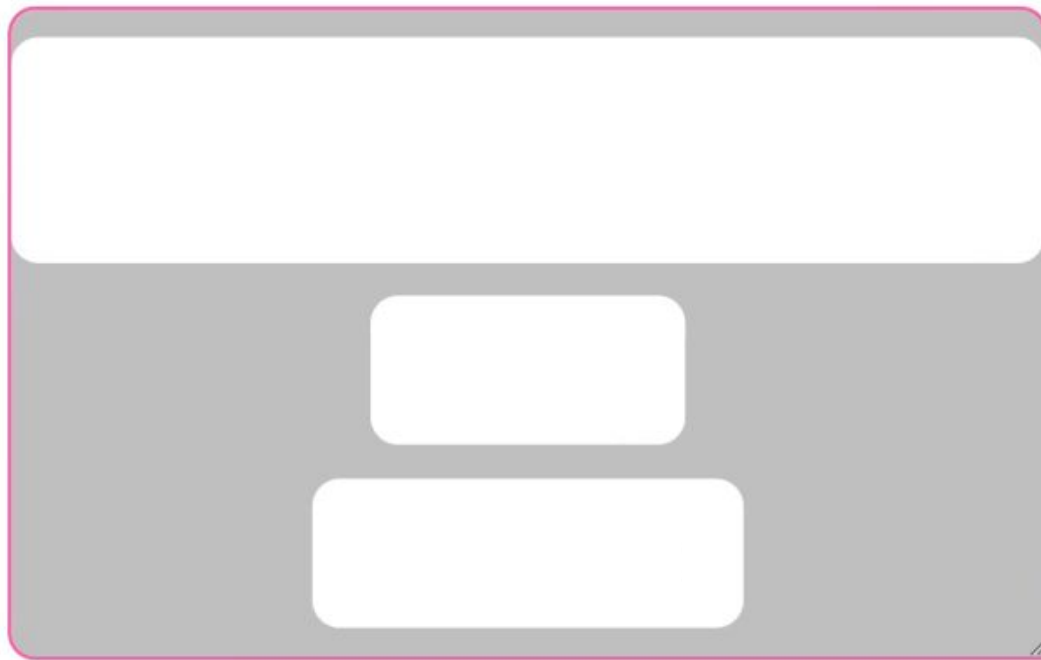
Q&A

Centering in CSS

1. centering should be able to handle changes to width
2. centering should be able to handle changes to height
3. centering should be dynamic to number of items
4. centering should be dynamic to length and language of content
5. centering should be document direction and writing mode agnostic

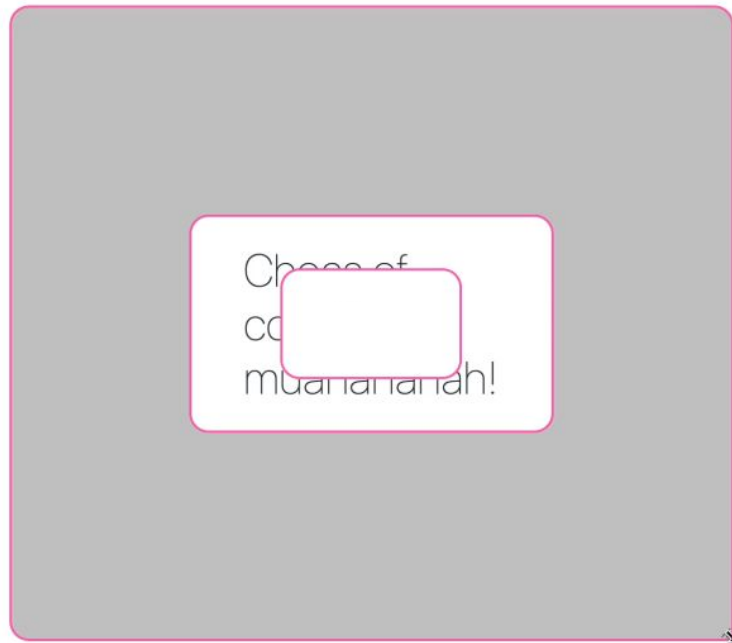
flex

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



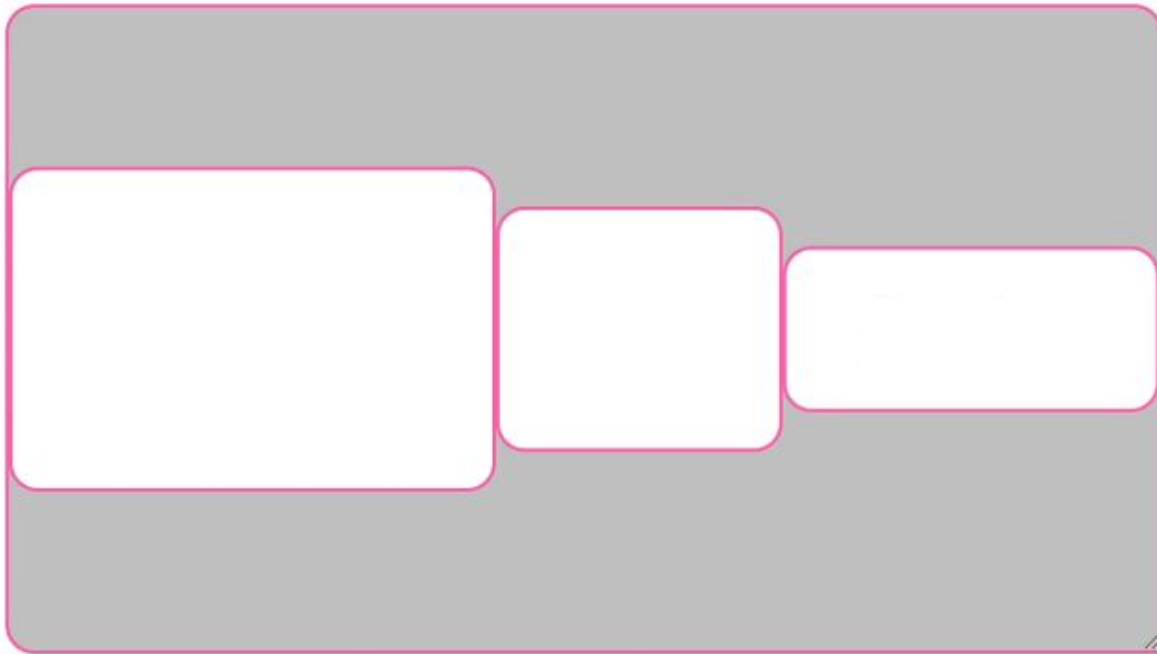
position: absolute

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



margin

```
.automargin > * {  
  margin: auto;  
}
```



RULE #12 (!important)

If you don't know
something
just **google** it!

THANK YOU!

Q&A