CSS

What is CSS?

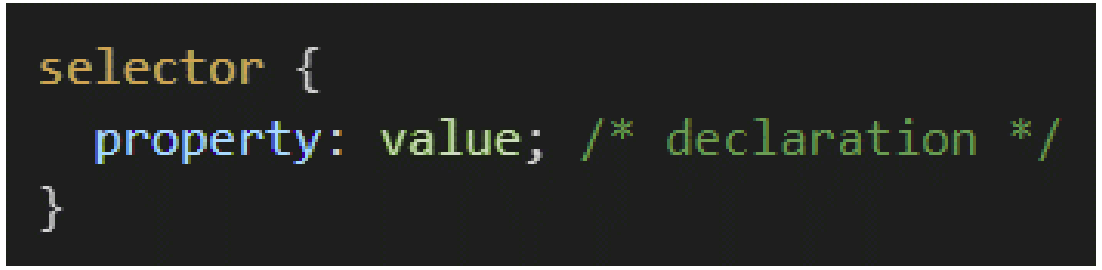
- **CSS** is a **style sheet language** that's focused entirely on improving the presentation of HTML elements. While CSS is a static programming language, it can be used to make your website appear visually pleasing and modern.

Focus points:

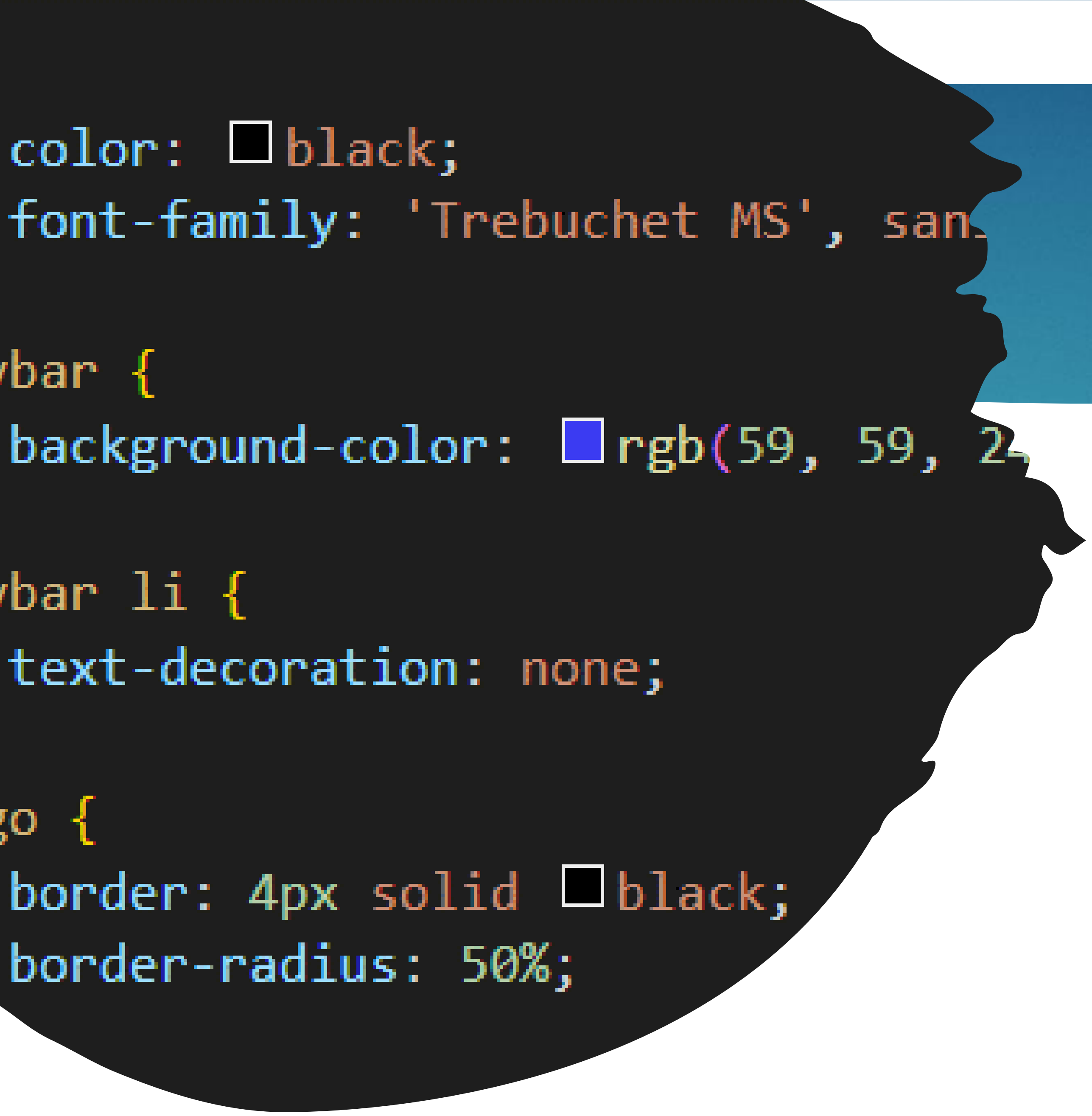
- Selectors
- Properties
- Cascade of CSS
- Adding CSS to HTML
- Box Model
- Display property
- Flexbox

How does CSS work?

- At the most basic level, **CSS** is just a set of rules you declare that represents how the browser should interpret the visual appearance of **HTML** elements. Each rule is made up of selectors and declarations where each declaration is made up of a *property:value* pair.



```
selector {
  property: value; /* declaration */
}
```



```
color: black;
font-family: 'Trebuchet MS', san

bar {
background-color: rgb(59, 59, 2

bar li {
text-decoration: none;

o {
border: 4px solid black;
border-radius: 50%;
```

```
ody {
    background-color:
    color: rgb(24, 24,
}
.container {
    border-radius: 4px;
    border: 4px solid g
```

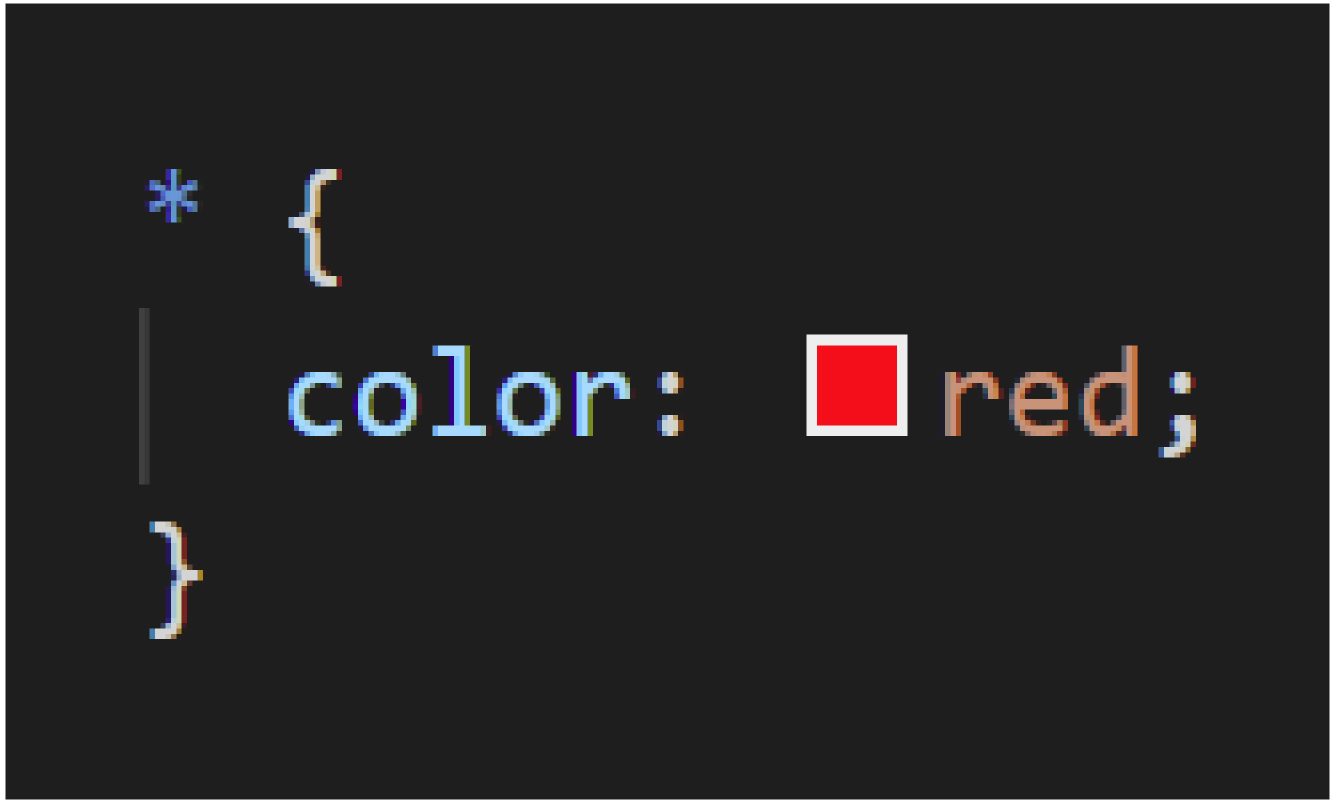
Selectors

CSS

Selectors

- **Selectors** simply refer to which **HTML elements** the given rules apply. There are many types of selectors that we will cover so that we could find the easiest way to **select HTML elements** that we want to:
  - **Universal selectors**
  - **Type selectors**
  - **Class selectors**
  - **ID selectors**
  - **Grouping selectors**
  - **Chaining selectors**
  - **Descendant combinator**

Universal Selectors

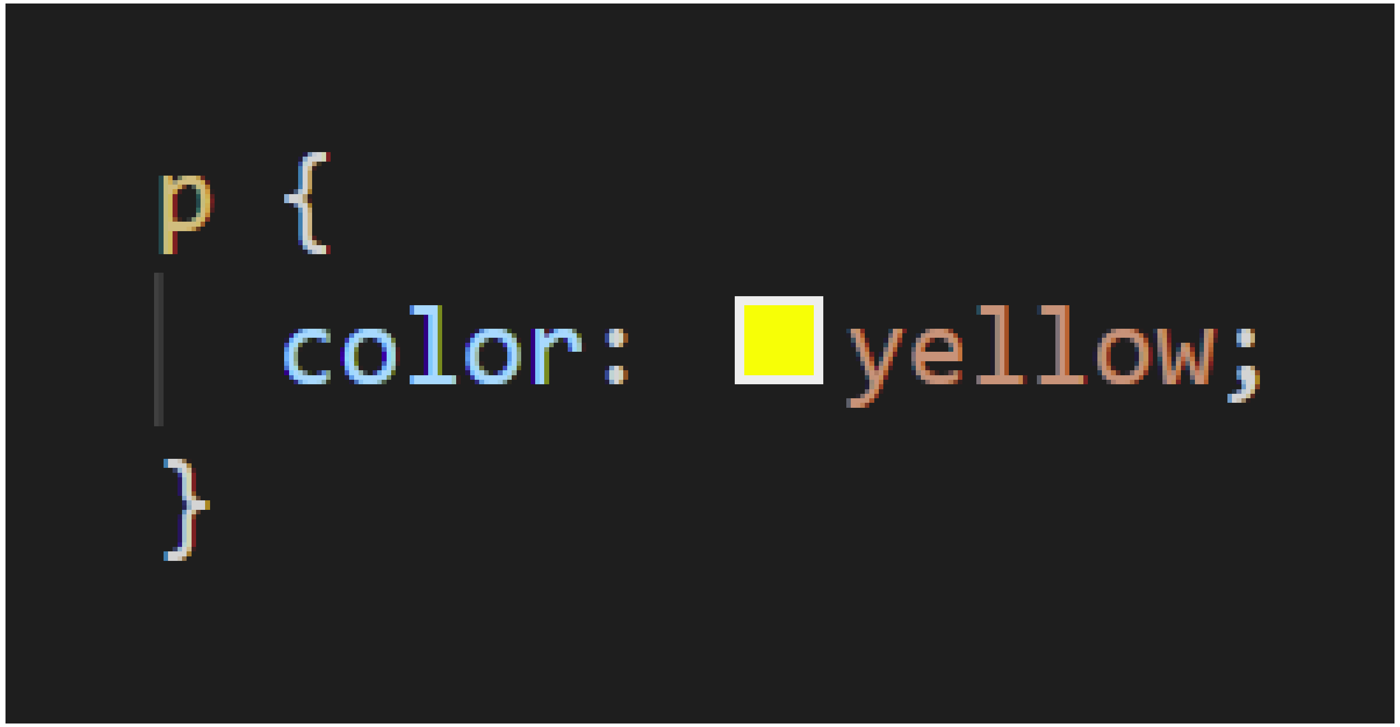


```
* {
    color: 🟥 red;
}
```

The **universal selector** selects elements of any type, hence its name. The syntax for it is the asterix sign ‘*’.

The example in the image would give all elements in the HTML page the color red.

Type Selectors

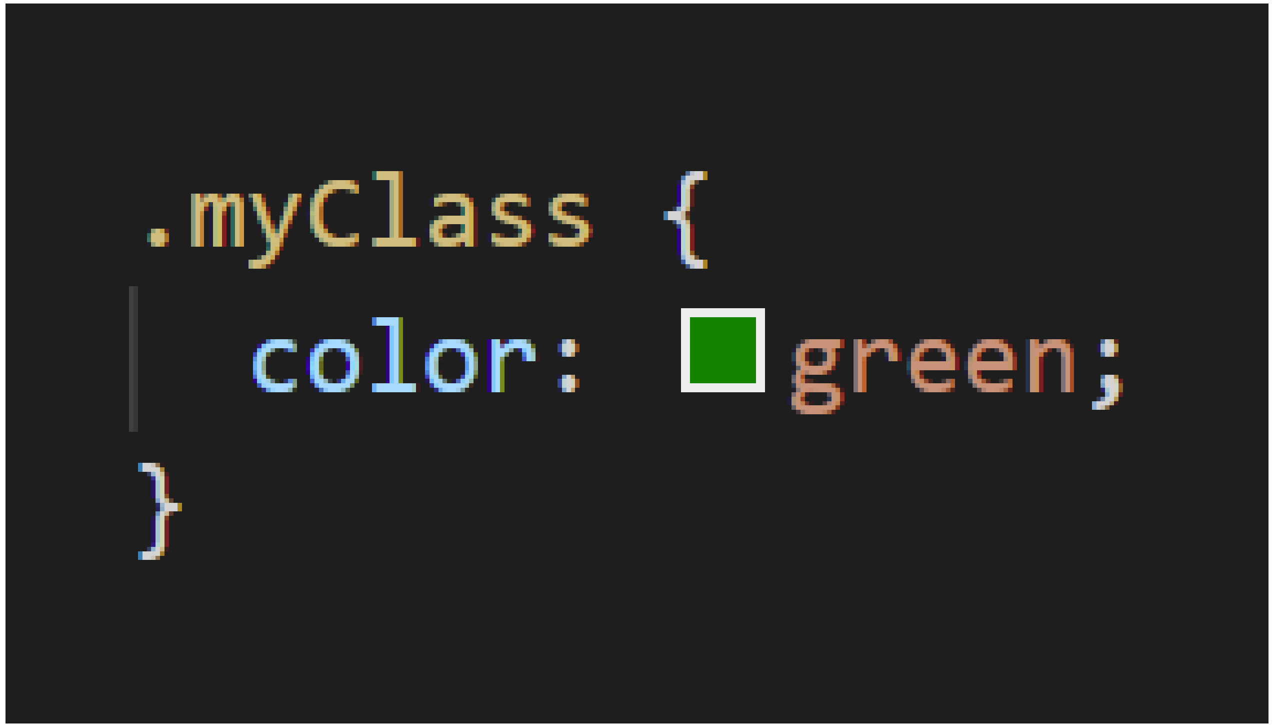


```
p {
  color: 🟨yellow;
}
```

The **type selector** (also known as 'element selector') will select all the elements of the given type. The syntax for it is the name of the element without the angle brackets.

The example in the image would color all the paragraphs yellow.

Class Selectors



```
.myClass {
  color: 🟩 green;
}
```

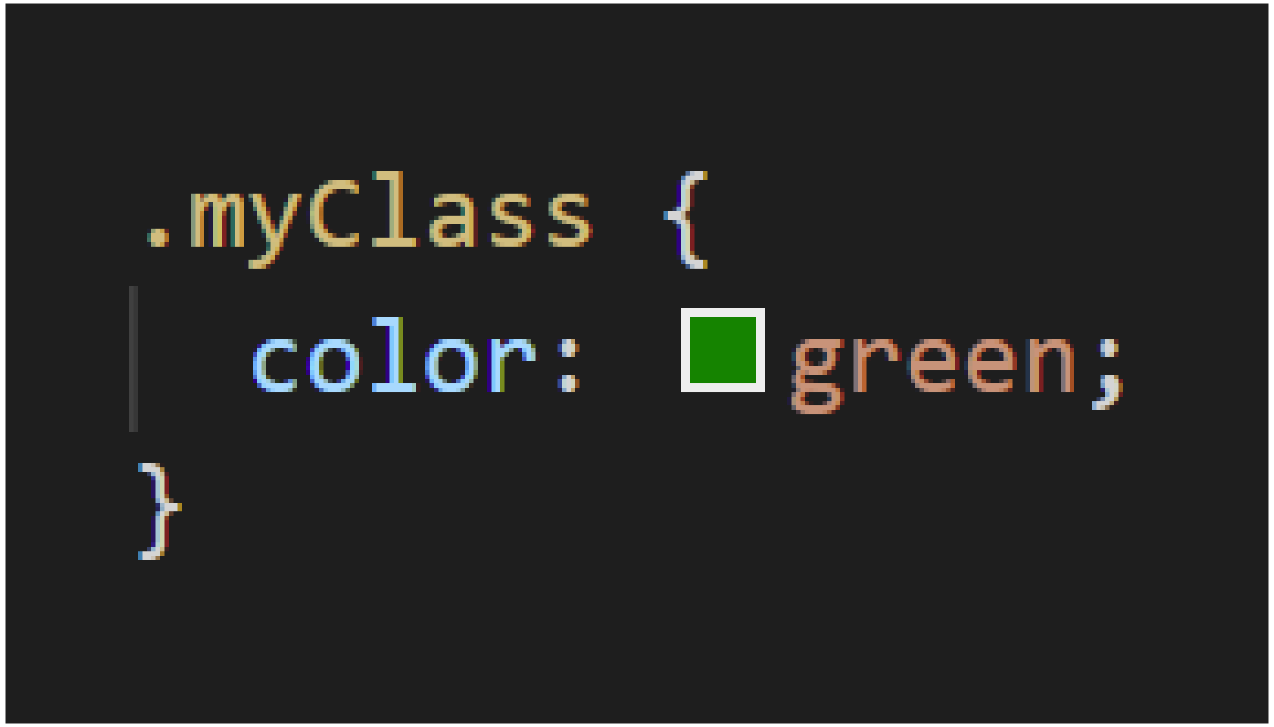
The **class selector** will select all elements with the given class. That given class is just an attribute you place in the opening tag of the HTML element.

(ex.: <p class="myClass">)

The syntax for class selectors is a period '**.**' followed by a case-sensitive value of the class attribute.

The example in the image would color all elements with the class name 'myClass' green.

Class Selectors

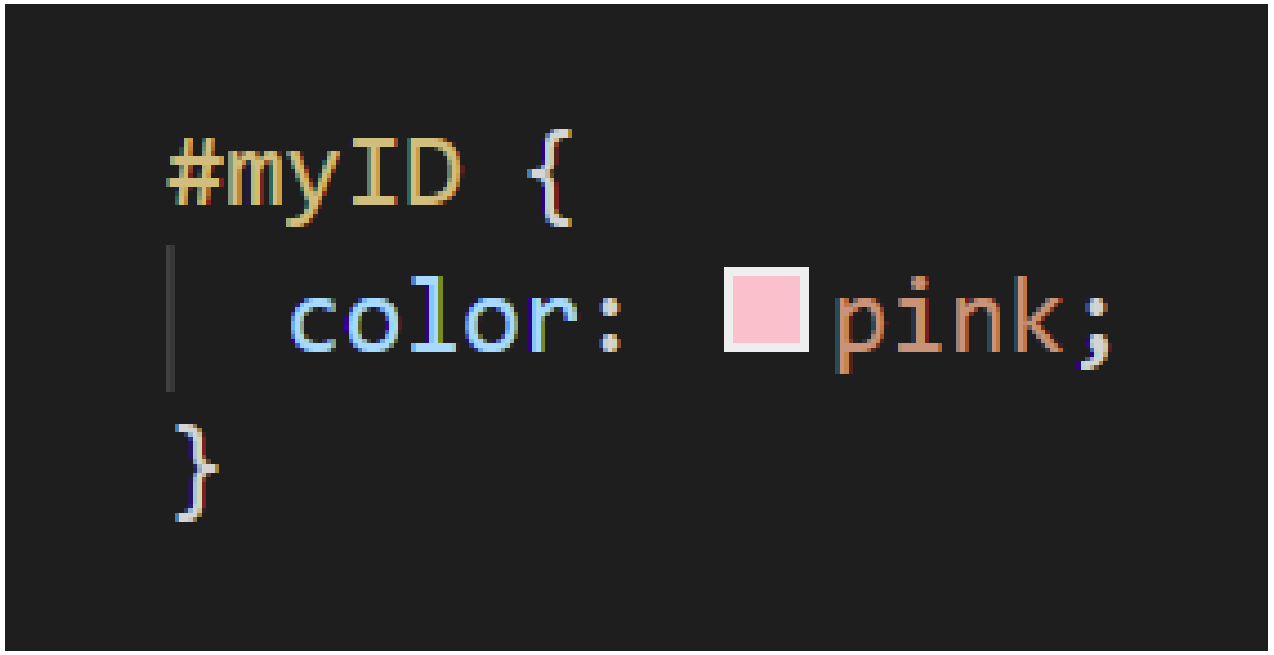


```
.myClass {
  color: 🟩green;
}
```

Classes aren’t meant to be unique! Which means that you can apply the same class on as many elements as you want, each of them will be affected by the same CSS ruleset you have given.

You can also apply more classes to a single element.

ID Selectors

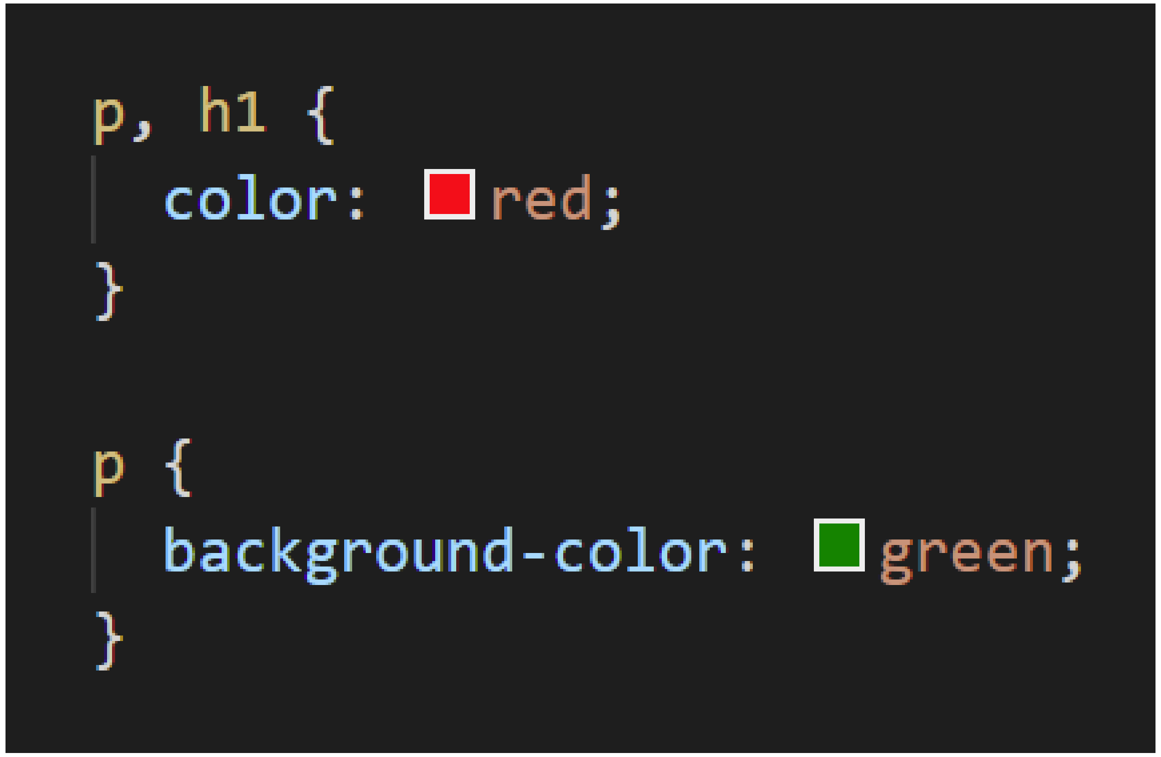


```
#myID {
  color: pink;
}
```

ID selectors are **similar** to **class selectors**. They select an element with the given **ID**, which is another attribute you place on an HTML element ***(ex.: <p id="myID">)***. The difference between IDs and classes is that an element can have only one **ID**, while elements can have many classes. The syntax for ID selectors is a hashtag sign **'#'** followed by a case-sensitive value of the **id** attribute.

The example in the image would color only the one element that has the given ID attribute.

Grouping Selectors

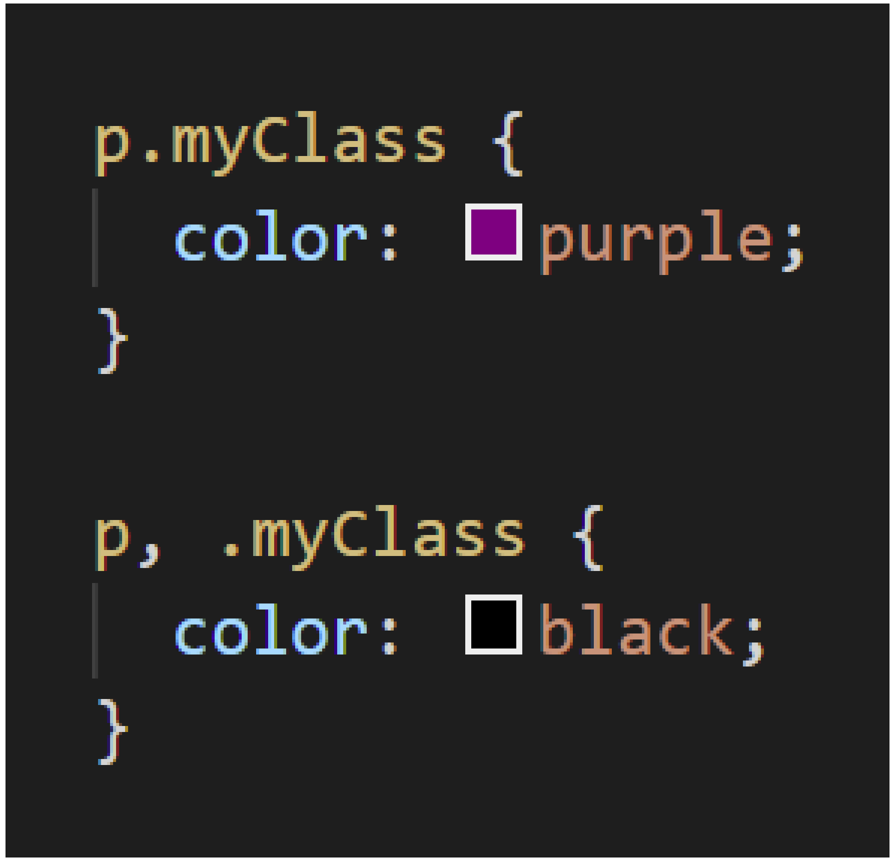


```
p, h1 {
    color: 🟥red;
}

p {
    background-color: 🟩green;
}
```

Grouping selectors will select multiple groups of elements which share the same style declarations. They are very useful as they shorten the amount of code you need to write, modify or read. The syntax for grouping selectors is simply a **comma** between each of the selectors you wish to apply the ruleset to.

The example in the image would color paragraphs and headers red, but the paragraphs would have a green background color.

Chaining Selectors

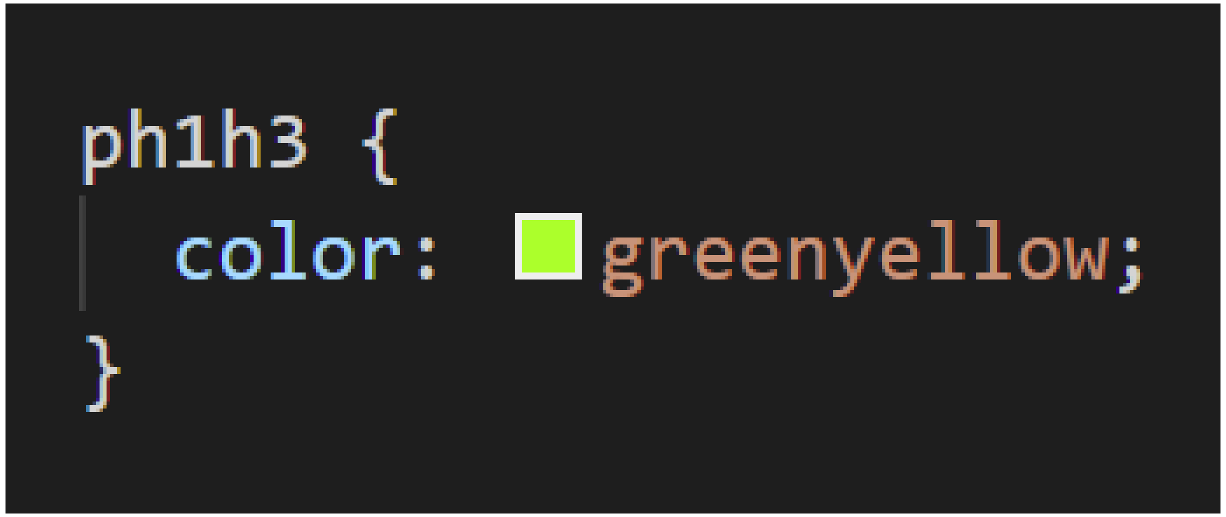


```css
p.myClass {
  color: purple;
}

p, .myClass {
  color: black;
}
```

Chaining selectors is a way to select elements that correspond to the multiple selectors given. The syntax for it is just combining the multiple selectors with no whitespace between them.

The example below would color purple, only paragraphs with the class ‘myClass’, all other paragraphs or elements with the class name myClass will be colored black.

Chaining Selectors

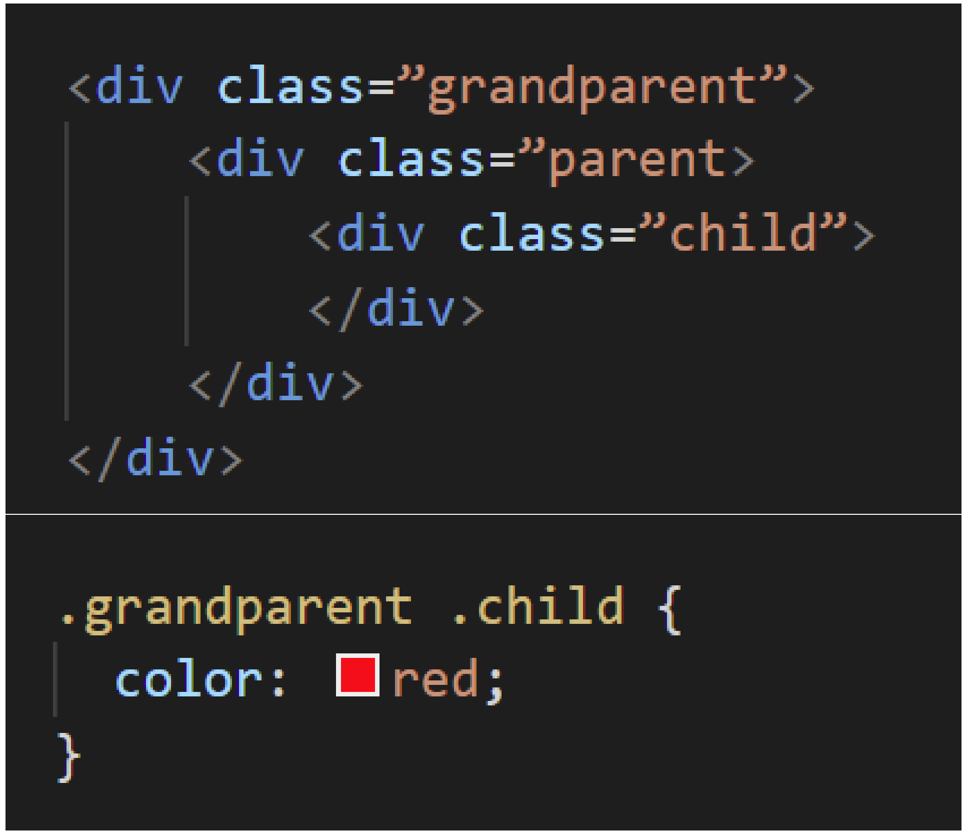


```
ph1h3 {
    color: 🟩greenyellow;
}
```

You **can’t** chain more than one type selector as elements can’t be more than one type as you can see in the example. However, you can chain an infinitely long chain of other selector types.

Note that if it’s infinitely long, there’s probably a better way to select the elements you need.

Descendant Combinator



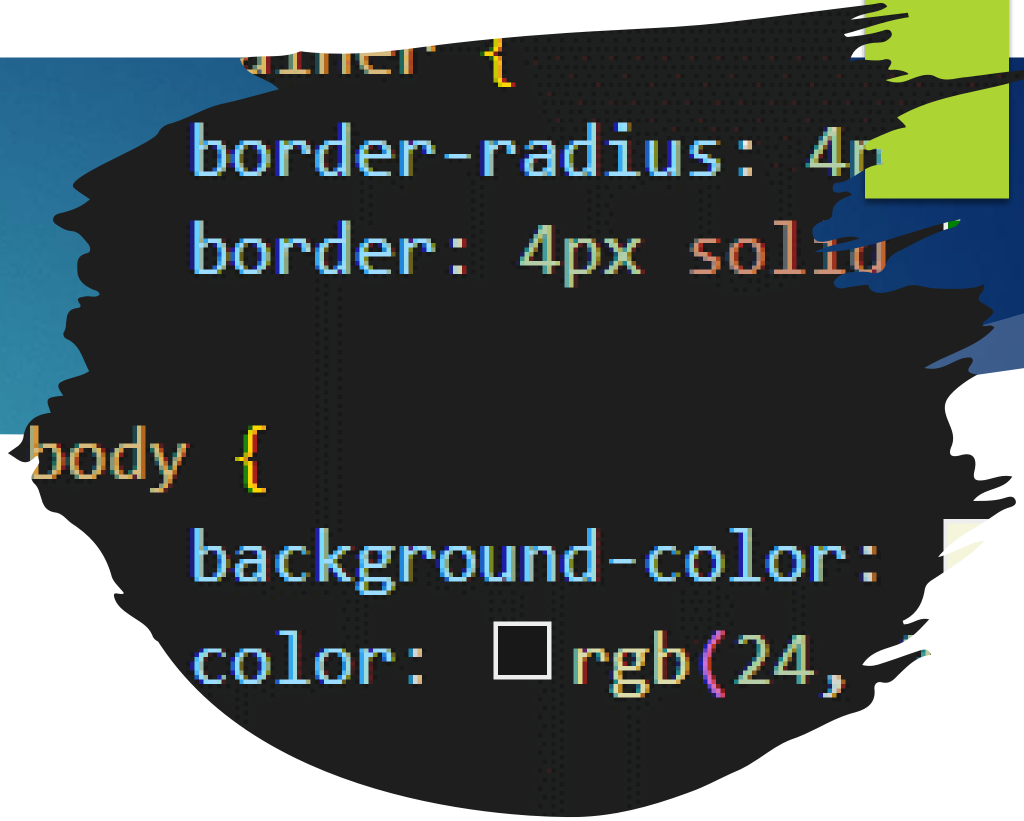
```html
<div class=”grandparent”>
    <div class=”parent>
        <div class=”child”>
        </div>
    </div>
</div>
```

```css
.grandparent .child {
  color: 🟥red;
}
```

Combinators allow us to combine multiple selectors differently than either grouping or chaining them, as they show a relationship between the selectors. Descendant combinators are represented by a single space between selectors. A descendant combinator will only cause elements that match the last selector to be selected, if they also have an ancestor (parent, grandparent, etc.) that matches the previous selector.

For example, if our selector is **.ancestor .child** then our **.child** will be affected by given CSS rules only if it's nested inside of .ancestor, no matter how deep it's nested.

```
er: 4px solid black;
er-radius: 50%;

r: black;
-family: 'Trebuchet MS', sans-s

i {
-decoration: none;

{
round-color: rgb(
```



```
ainer {
  border-radius: 4p
  border: 4px solid

body {
  background-color:
  color: rgb(24,
```

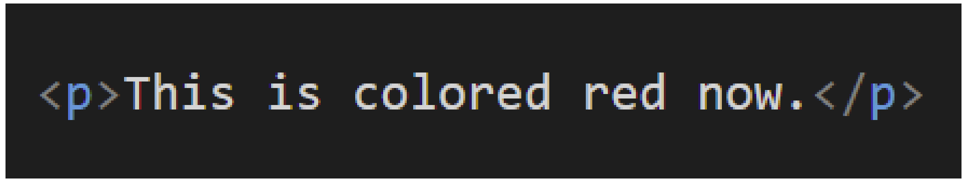
Properties

CSS

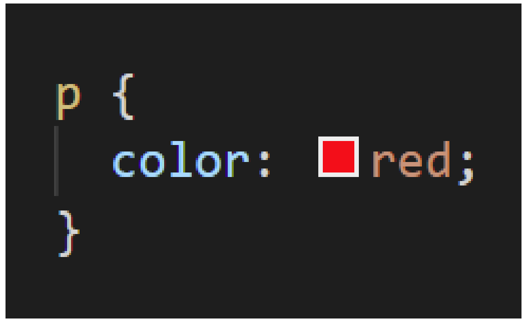
Properties

- Properties are the styling rules that we apply to the elements that we have selected. They are in the shape of ***property:value***. Here are a few commonly used properties which we will cover:
  - **color** and **background-color**
  - **font-family**, **font-size**, **font-weight** and **text-align**
  - **height**, **width**, **border-radius**

Color property



```html
<p>This is colored red now.</p>
```

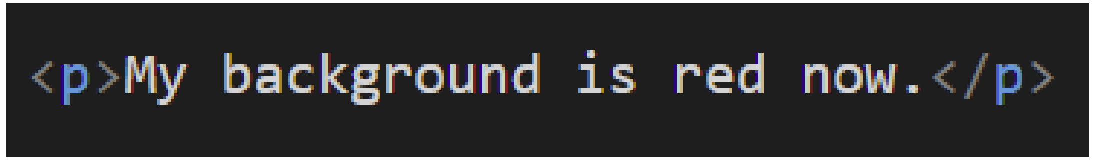


```css
p {
  color: 🟥red;
}
```

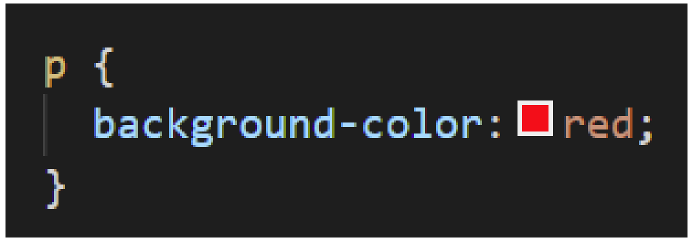
This is colored red now.

The **color** CSS property sets the foreground color value of an element's text and text decorations.

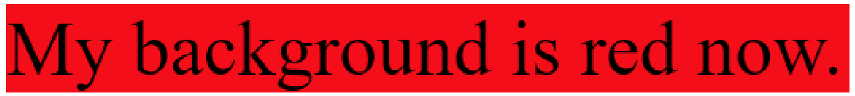
Background-color property



```
<p>My background is red now.</p>
```



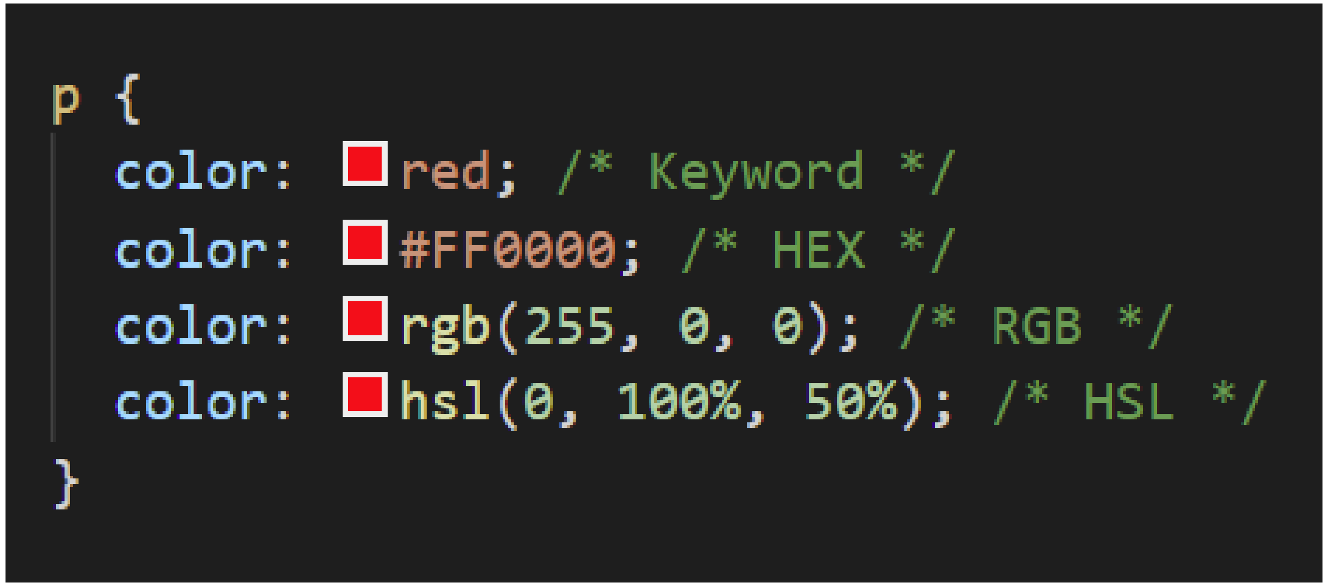
```
p {
  background-color: 🟥red;
}
```



My background is red now.

The **background-color** CSS property sets the background color value of an element.

Color values

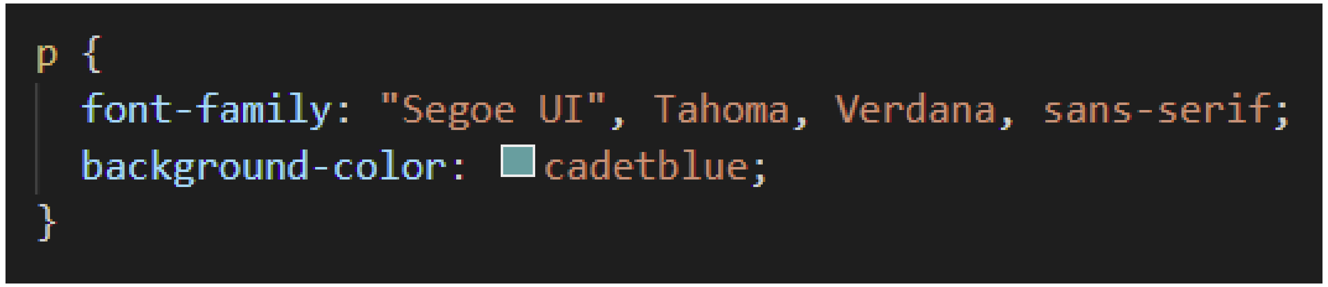


```
p {
  color: 🟥red; /* Keyword */
  color: 🟥#FF0000; /* HEX */
  color: 🟥rgb(255, 0, 0); /* RGB */
  color: 🟥hsl(0, 100%, 50%); /* HSL */
}
```

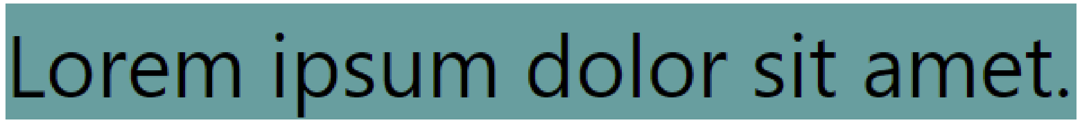
Both **color** and **background-color** can accept different kinds of values, a common type that we’ve shown is using a keyword for an actual color name (red, green, yellow). Other types are HEX, RGB and HSL values.

In the example you can see 4 ways to display the color red.

Font-family



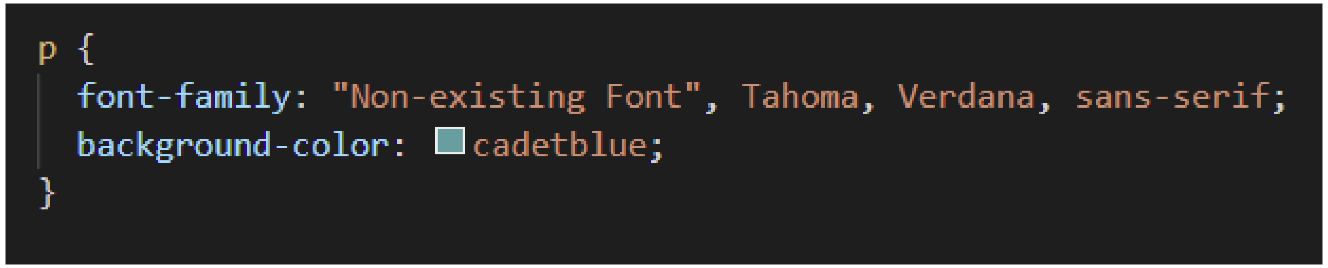
```
p {
  font-family: "Segoe UI", Tahoma, Verdana, sans-serif;
  background-color: cadetblue;
}
```



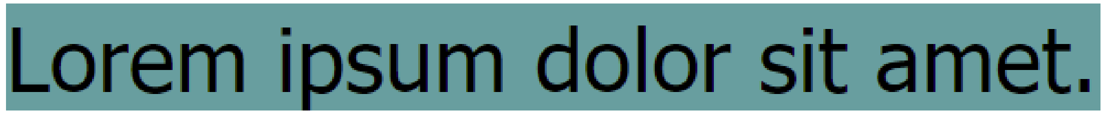
Lorem ipsum dolor sit amet.

font-family property can be a single value or a comma-separated list of values that determine what font an element uses. Each font will fall into one of two categories, either a “font family name” like "Times New Roman" (we use quotes due to the whitespace between words) or a “generic family name” like sans-serif (generic family names never use quotes).

Font-family



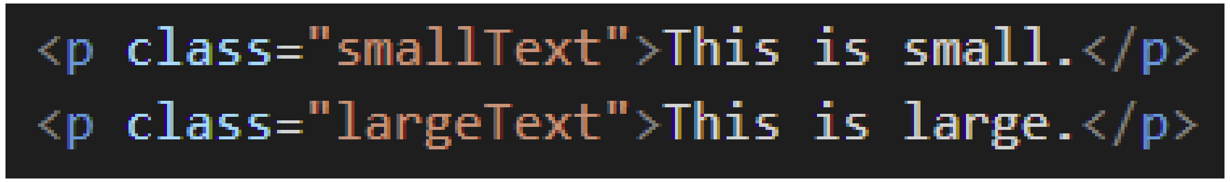
```
p {
  font-family: "Non-existing Font", Tahoma, Verdana, sans-serif;
  background-color: cadetblue;
}
```



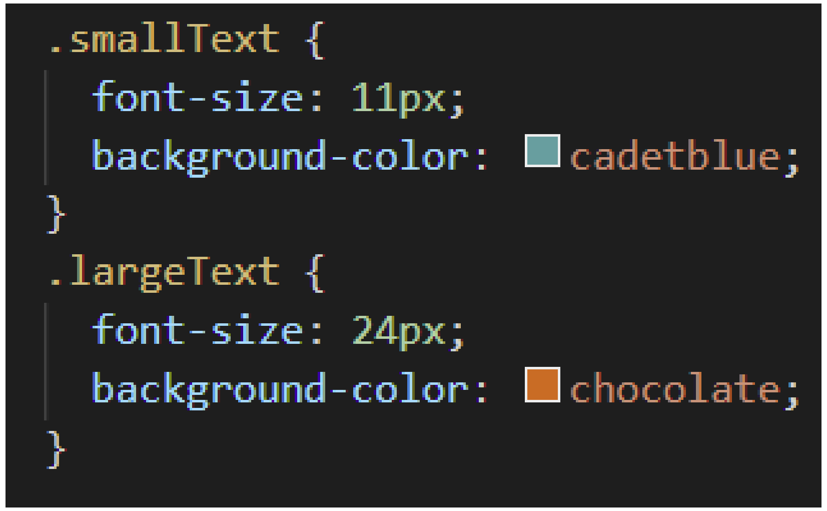
Lorem ipsum dolor sit amet.

If a browser cannot find or does not support the first font in the list, it will try to use the next one, and the next one until it finds a specified font that it can support. That is why it’s good practice to include a list of fonts rather than just one font, starting with the font you want to be used most, and ending with a generic font in case all previous fail.

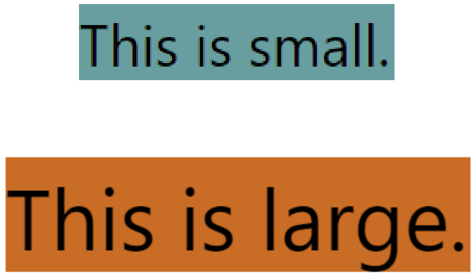
Font-size



```html
<p class="smallText">This is small.</p>
<p class="largeText">This is large.</p>
```



```css
.smallText {
  font-size: 11px;
  background-color: cadetblue;
}
.largeText {
  font-size: 24px;
  background-color: chocolate;
}
```



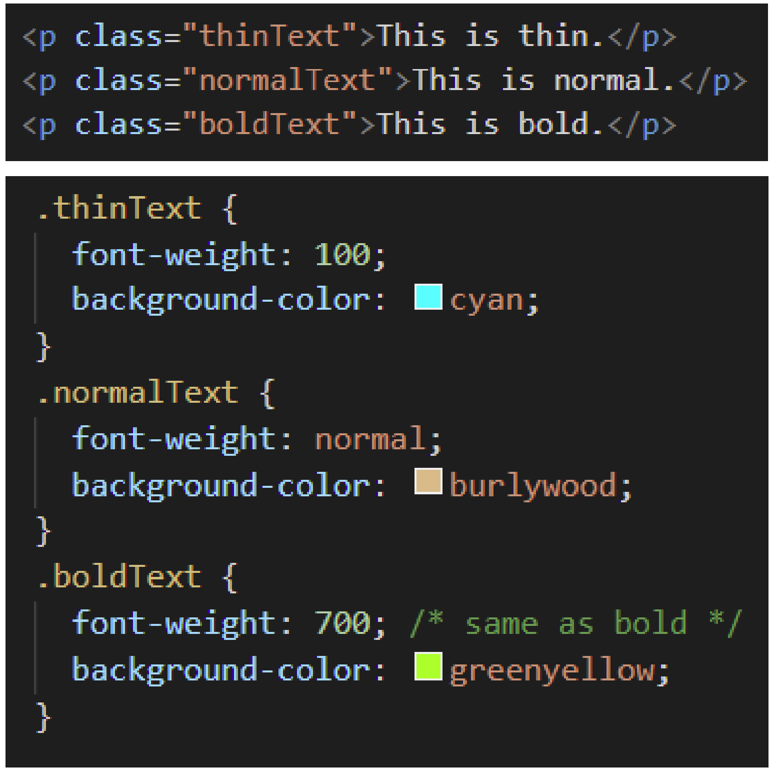
This is small.

This is large.

font-size property affects the size of the font.

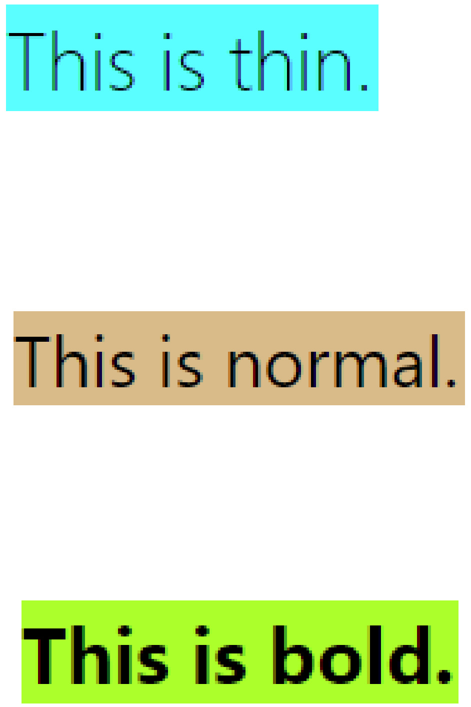
When setting the value to this property, there should be no whitespace between the amount and the unit (16px instead of 16 px).

Font-weight



```html
<p class="thinText">This is thin.</p>
<p class="normalText">This is normal.</p>
<p class="boldText">This is bold.</p>
```

```css
.thinText {
  font-weight: 100;
  background-color: cyan;
}
.normalText {
  font-weight: normal;
  background-color: burlywood;
}
.boldText {
  font-weight: 700; /* same as bold */
  background-color: greenyellow;
}
```



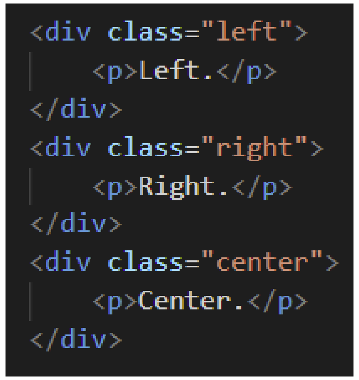
This is thin.

This is normal.

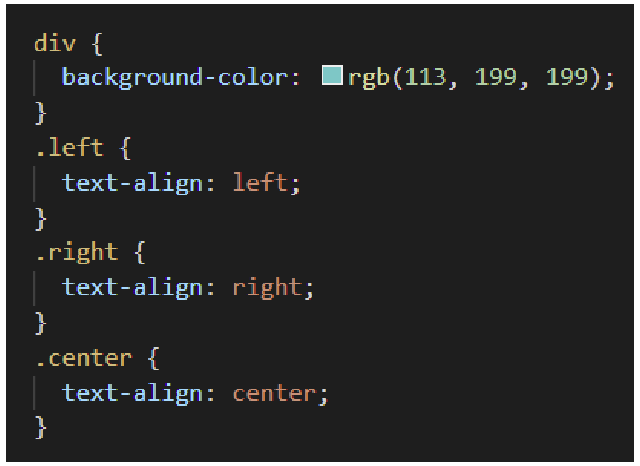
This is bold.

font-weight property affects the boldness of text if the fonts support the specified weight. Its value can be a keyword (**thin**, **normal**, **bold**) or a number between **1** and **1000**.

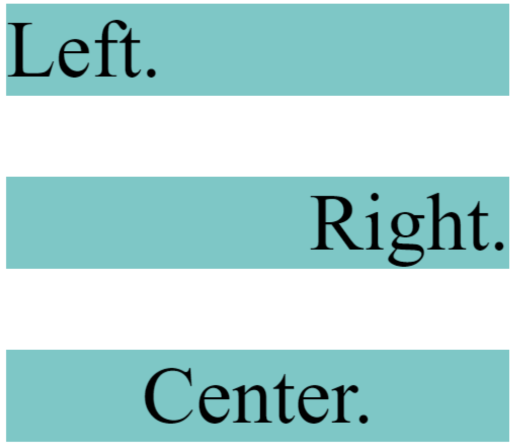
Text-align



```
<div class="left">
    <p>Left.</p>
</div>
<div class="right">
    <p>Right.</p>
</div>
<div class="center">
    <p>Center.</p>
</div>
```



```
div {
  background-color: 🟦rgb(113, 199, 199);
}
.left {
  text-align: left;
}
.right {
  text-align: right;
}
.center {
  text-align: center;
}
```



text-align property will align text horizontally within an element, and its value one of the following common keywords:

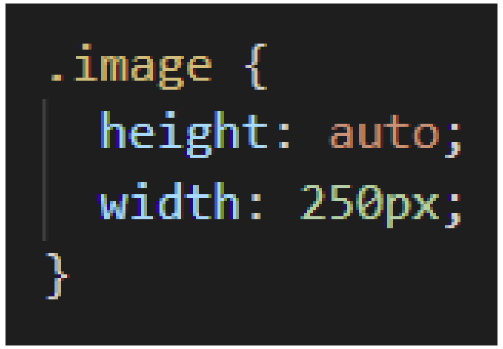
- **left** (default value, content aligns along the left side)
- **right** (content aligns along the right side)
- **center** (content aligns in the center, whitespace should be equal on left and right side)
- **justify** (distribute your text evenly between the edges)
- **inherit** (its value will be whatever its parent elements is)

Image properties
height & width

By default, an **<img>** element's width and height values will be the same as the actual image file's width and height. If you wanted to adjust the size of the image without causing it to lose its proportions, you would use a value of "auto" for the height property and adjust the width value.

It's best to include both of these properties for **<img>** elements, even if you don't plan on adjusting the values from the image file's original ones. When these values aren't included, if an image takes longer to load than the rest of the page contents, the image won't take up any space on the page at first, but will suddenly cause a drastic shift of the other page contents once it does load in. Explicitly stating a height and width prevents this from happening, as space will be '*reserved*' on the page and will just appear as a blank space until the image loads.

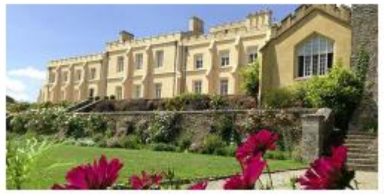
Image properties
height & width



```
.image {
  height: auto;
  width: 250px;
}
```

Original image

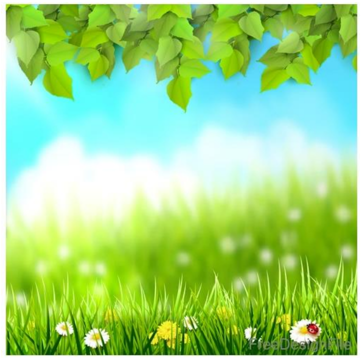
Edited height & width



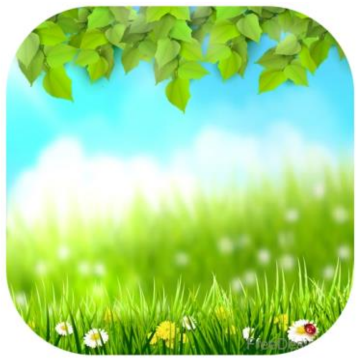
The first image (original) has no CSS applied to it and has its original dimensions set *(500px by 250px)*.

In the second image we have edited the width and set its height to auto. By doing so, it scaled down proportionally, meaning its height is now 125px.

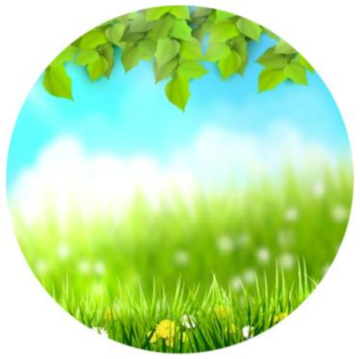
Image properties
border-radius



```
.image {
  border-radius: 0%;
}
```

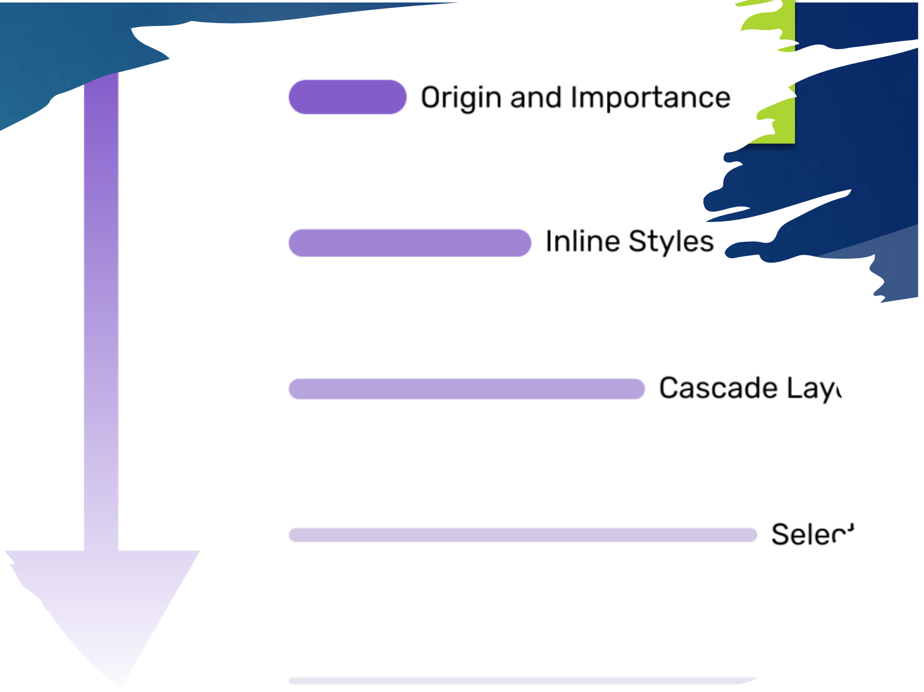


```
.image {
  border-radius: 25%;
}
```



```
.image {
  border-radius: 50%;
}
```

There are other fun properties such as **border-radius** (it does not apply to images only) which we can use to make our image have rounded corners or be in the shape of a circle.



The Cascade of CSS

CSS

The Cascade of CSS

The **cascade** is an algorithm that defines how to combine property values originating from different sources. It lies at the core of CSS, as emphasized by the name: Cascading Style Sheets. Basically, it defines which rules get applied to our HTML. By understanding these rules, you will avoid many situations in which you don’t understand why your CSS isn’t working the way you intended it to work. We will cover three factors that the cascade uses to determine which rules get applied:

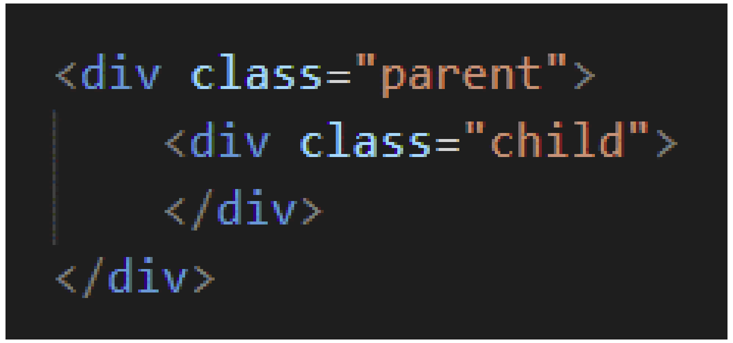
1. **Specificity**
2. **Inheritance**
3. **Rule Order**

Specificity

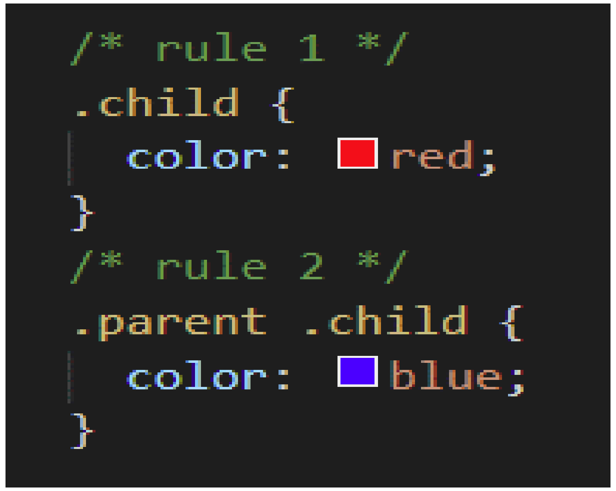
A CSS declaration that is more specific will take precedence over less specific ones. Inline styles, which we will go over more later in the lesson, have the highest specificity compared to selectors, while each type of selector has its own specificity level that contributes to how specific a declaration is.

Specificity will only be considered when an element has multiple, conflicting declarations targeting it. An ID selector will always beat any number of class selectors, a class selector will always beat any number of type selectors, and a type selector will always beat any number of anything less specific than it.

Specificity



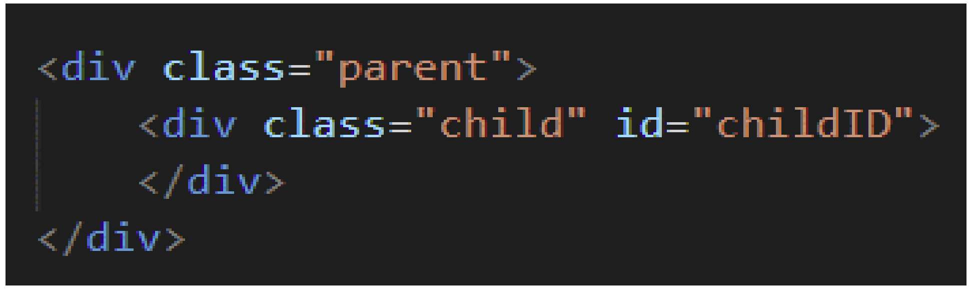
```
<div class="parent">
    <div class="child">
    </div>
</div>
```



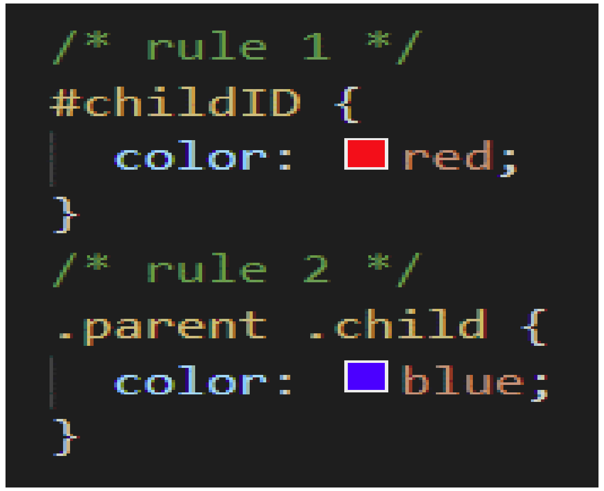
```
/* rule 1 */
.child {
  color: 🟥 red;
}
/* rule 2 */
.parent .child {
  color: 🟦 blue;
}
```

In the example, we see both rules using only class selectors, however, **rule 2** is more specific because it is using more class selectors, so the **color: blue;** would take precedence.

Specificity



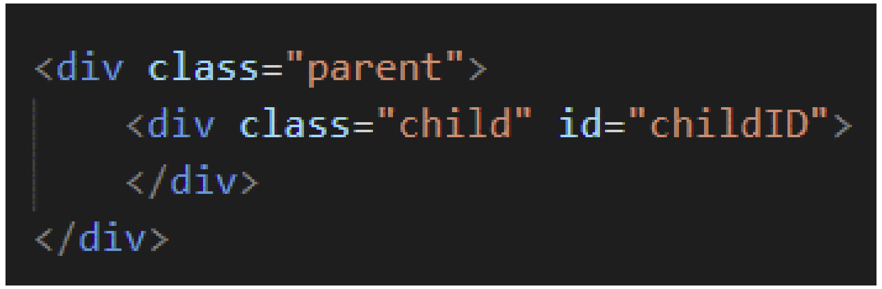
```
<div class="parent">
    <div class="child" id="childID">
    </div>
</div>
```



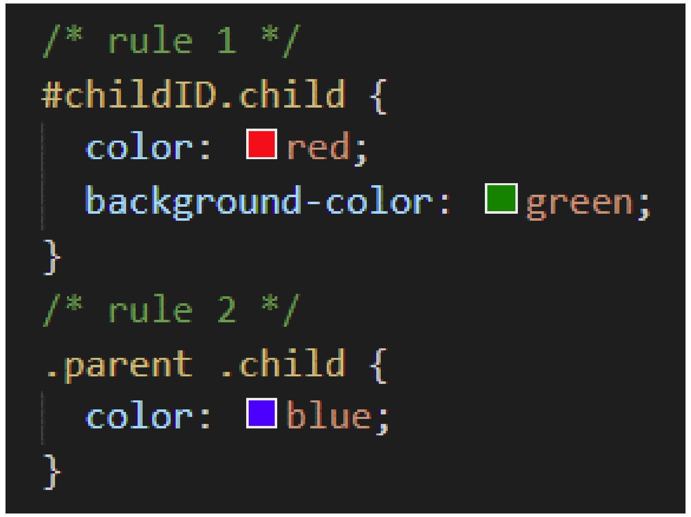
```
/* rule 1 */
#childID {
  color: red;
}
/* rule 2 */
.parent .child {
  color: blue;
}
```

In the example, even though **rule 2** is using more selectors, **rule 1** is more specific because IDs are more specific than classes, so, we would see **color: red;** taking precedence.

Specificity



```html
<div class="parent">
    <div class="child" id="childID">
    </div>
</div>
```

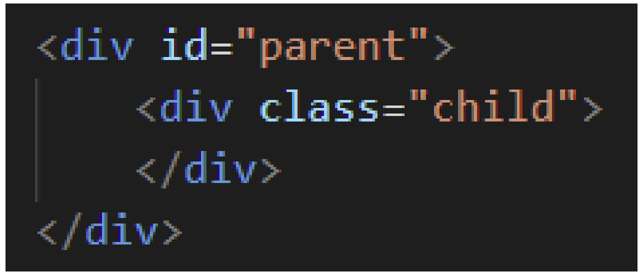


```css
/* rule 1 */
#childID.child {
  color: 🟥red;
  background-color: 🟩green;
}
/* rule 2 */
.parent .child {
  color: 🟦blue;
}
```

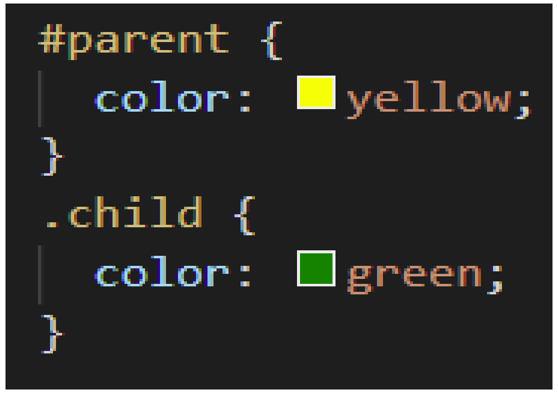
What do you think about this one?

Inheritance

Inheritance refers to certain CSS properties that, when applied to an element, are inherited by that element's descendants, even if we don't explicitly write a rule for those descendants. Typography based properties (color, font-size, font-family, etc.) are usually inherited, while most other properties aren't.



```html
<div id="parent">
    <div class="child">
    </div>
</div>
```

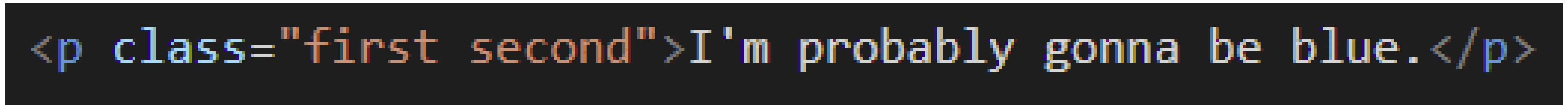


```css
#parent {
  color: 🟨yellow;
}
.child {
  color: 🟩green;
}
```

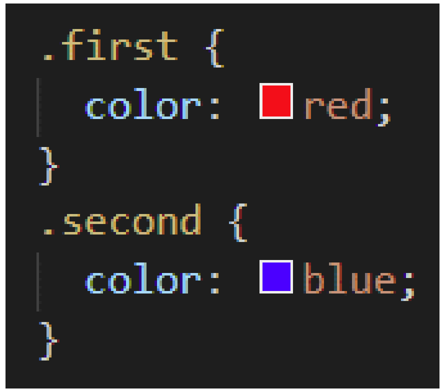
In the example, despite the parent having an ID and therefore a higher specificity, the child elements **color: green;** rule would apply as that declaration directly targets it, while **color: yellow;** from the parent is only inherited.

Rule Order

The third and final factor, the one that is considered after every other factor has been taken into account and there are still multiple conflicting declarations - rule order. The one which was **last declared** is the one that will be applied.

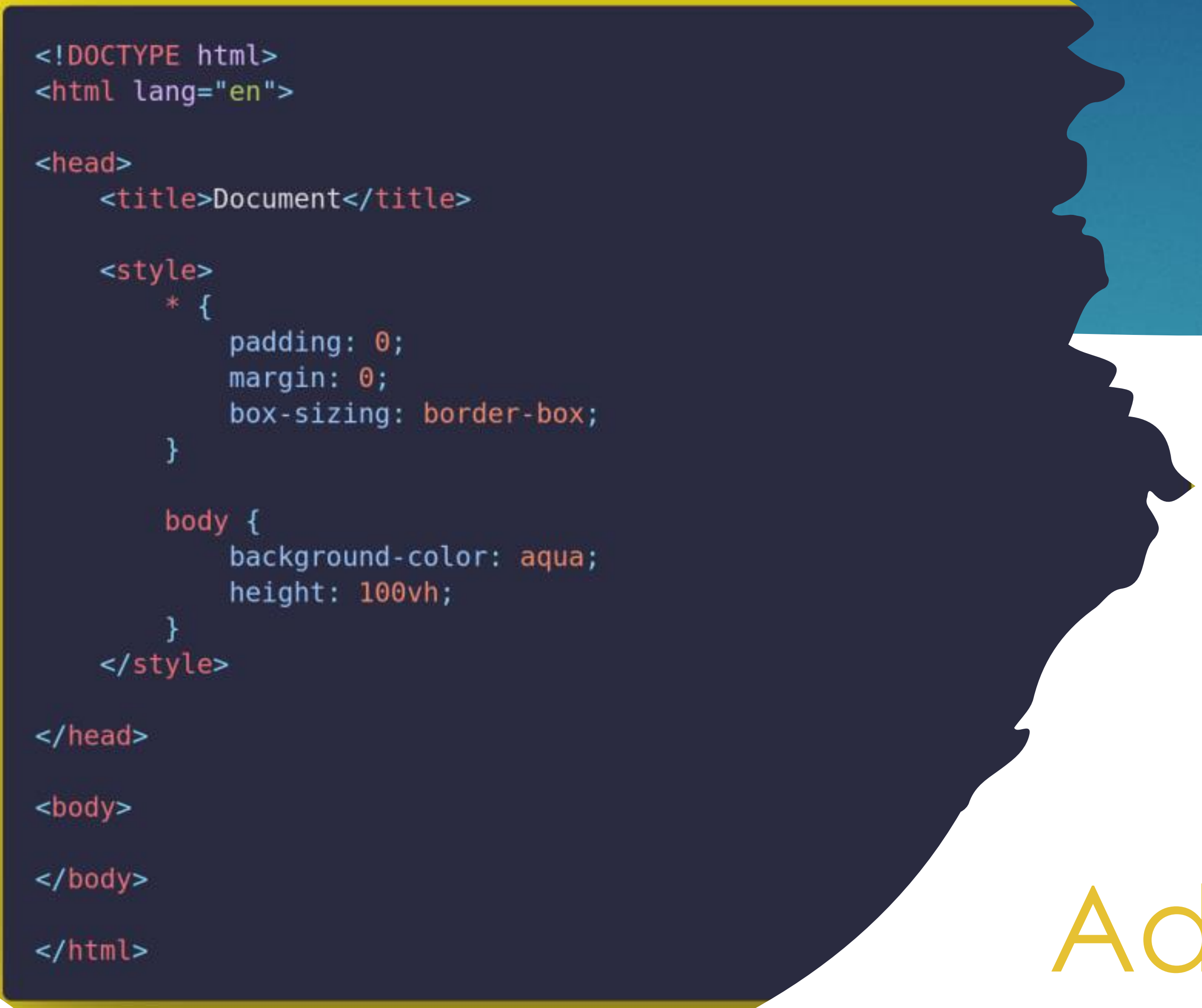


```html
<p class="first second">I'm probably gonna be blue.</p>
```



```css
.first {
  color: 🟥red;
}
.second {
  color: 🟦blue;
}
```

In this example, the **<p>** element has both the **.first** and **.second** class, so, there is no inheritance and there is no specificity, only the **rule order** is left, and as **.second** is declared last, it will be applied to the page.



```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Document</title>

    <style>
        * {
            padding: 0;
            margin: 0;
            box-sizing: border-box;
        }

        body {
            background-color: aqua;
            height: 100vh;
        }
    </style>

</head>

<body>

</body>

</html>
```

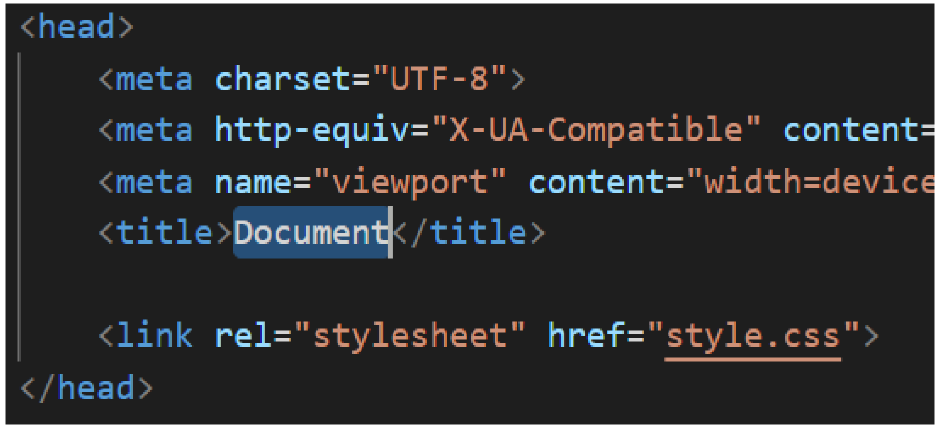
```
 html>
 lang="en" dir="ltr">

ead>
<meta charset="utf-8">
title>Angela's Personal Site</title>
<link rel="stylesheet" href="css/styles.
</head>


 cellspacing="20">
```

Adding CSS to HTML

CSS

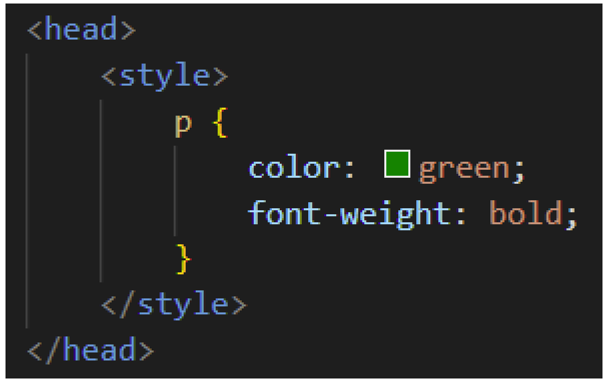
Adding CSS to HTML

We've gone over a lot of CSS, however, it is all pointless if we do not connect our CSS with our HTML, there are three ways to do it:



```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content=
    <meta name="viewport" content="width=device
    <title>Document</title>

    <link rel="stylesheet" href="style.css">
</head>
```

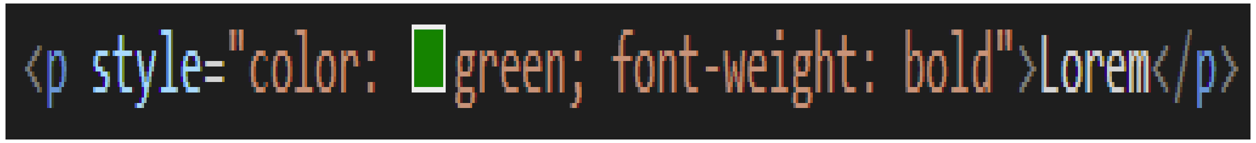
External CSS



```
<head>
    <style>
        p {
            color: 🟩green;
            font-weight: bold;
        }
    </style>
</head>
```

Internal CSS

Inline CSS

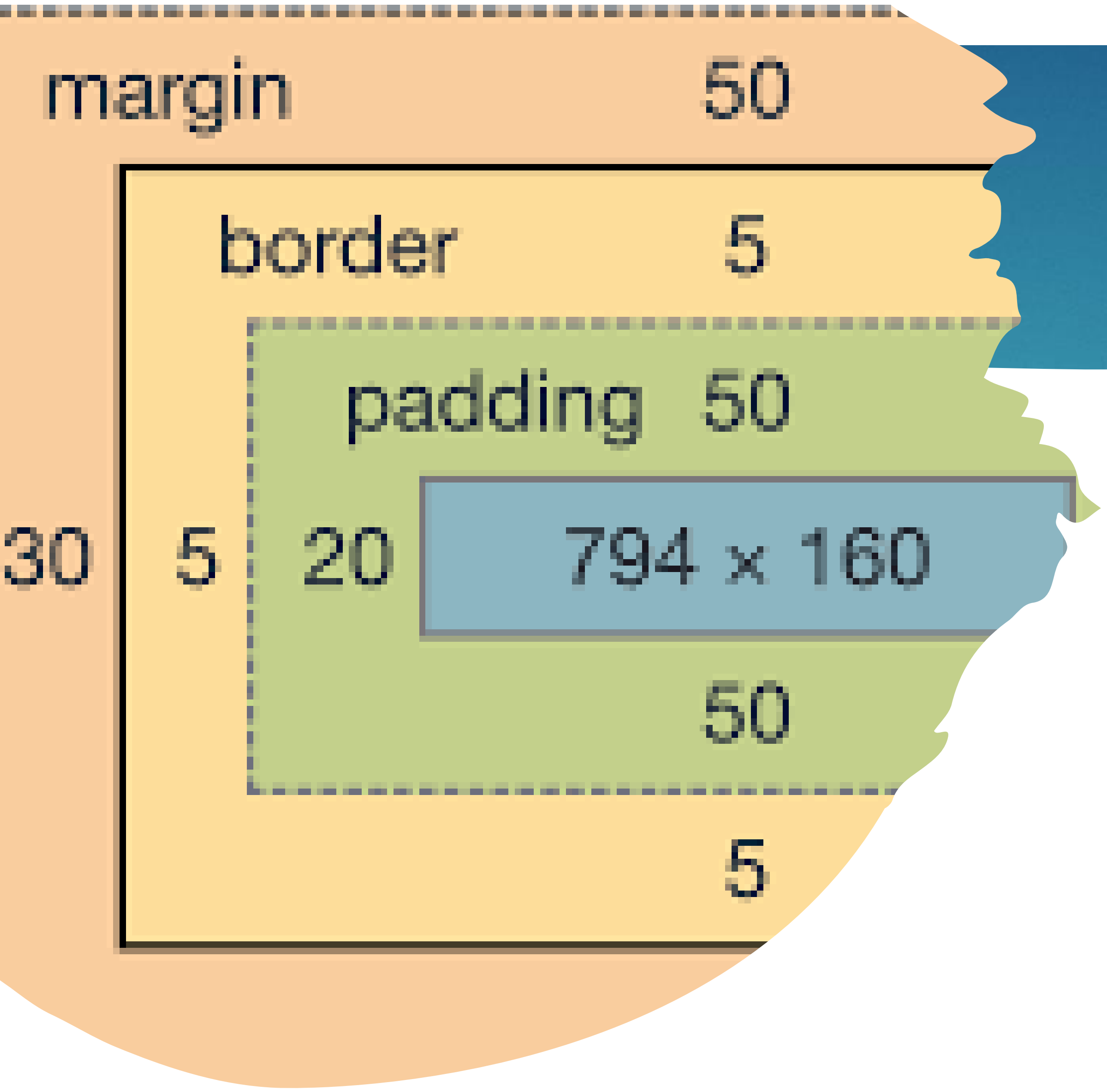


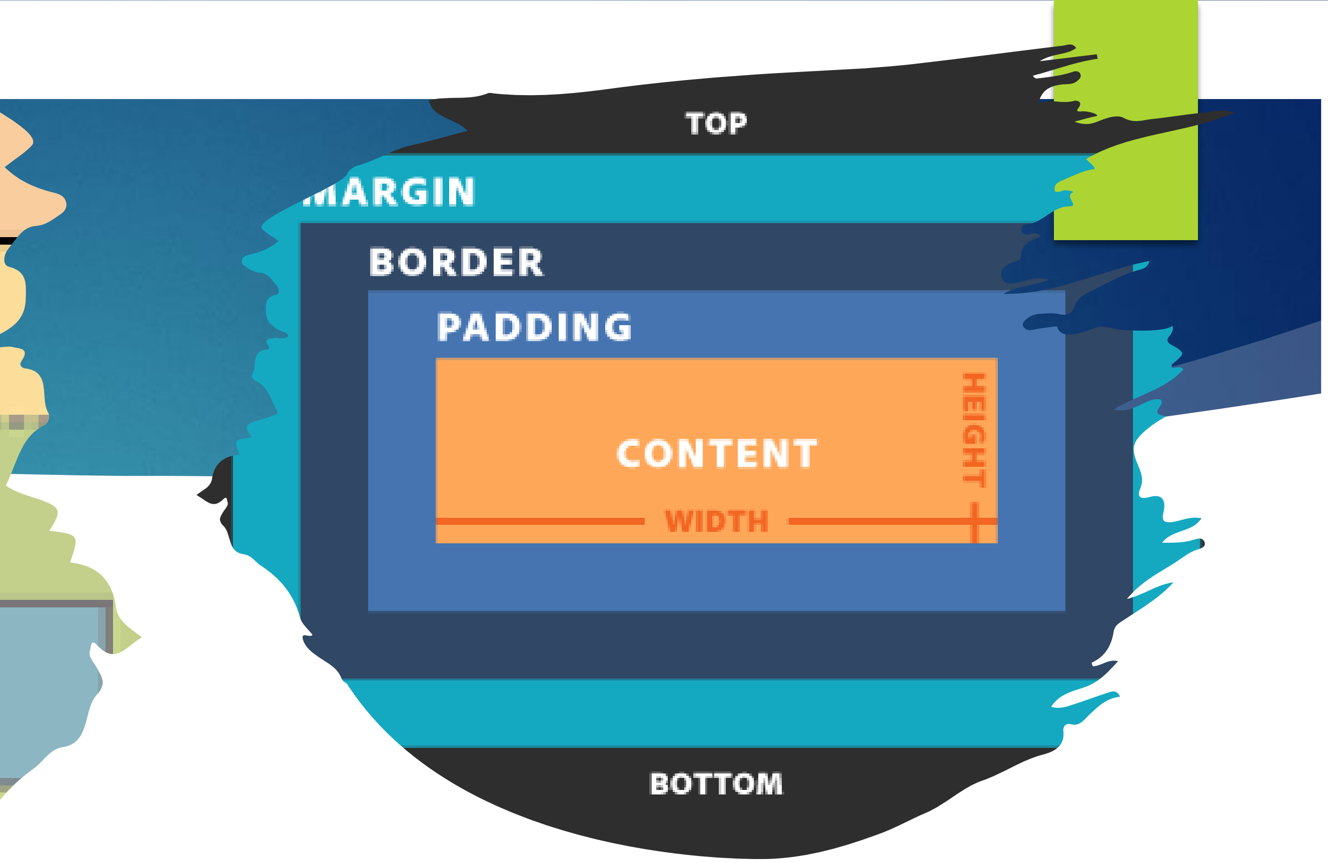
```
<p style="color: 🟩green; font-weight: bold">Lorem</p>
```

External CSS

Note that we do **not** need to name the CSS file **style.css**, you can name it however you want to as long as it's a **.css** file format. The most common names for CSS files are **style.css** and **styles.css**.

The reason why this is the most common method is because it keeps our HTML and CSS separated, which results in the HTML file looking smaller and cleaner, and it is also appliable easily to any other HTML pages.





The Box Model

CSS

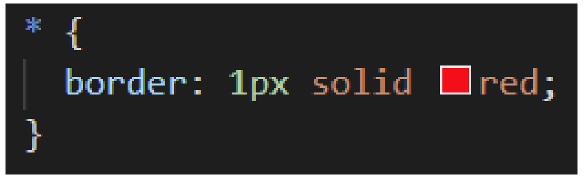
Inspect Element

Inspect Element is a developer tool offered by browsers to view and temporarily change the source code of any webpage. By using this tool, developers and designers can check and modify the front-end of any website. As you change the code, the browser keeps updating the webpage in real-time. Here are some examples of what you can do with Inspect Element:

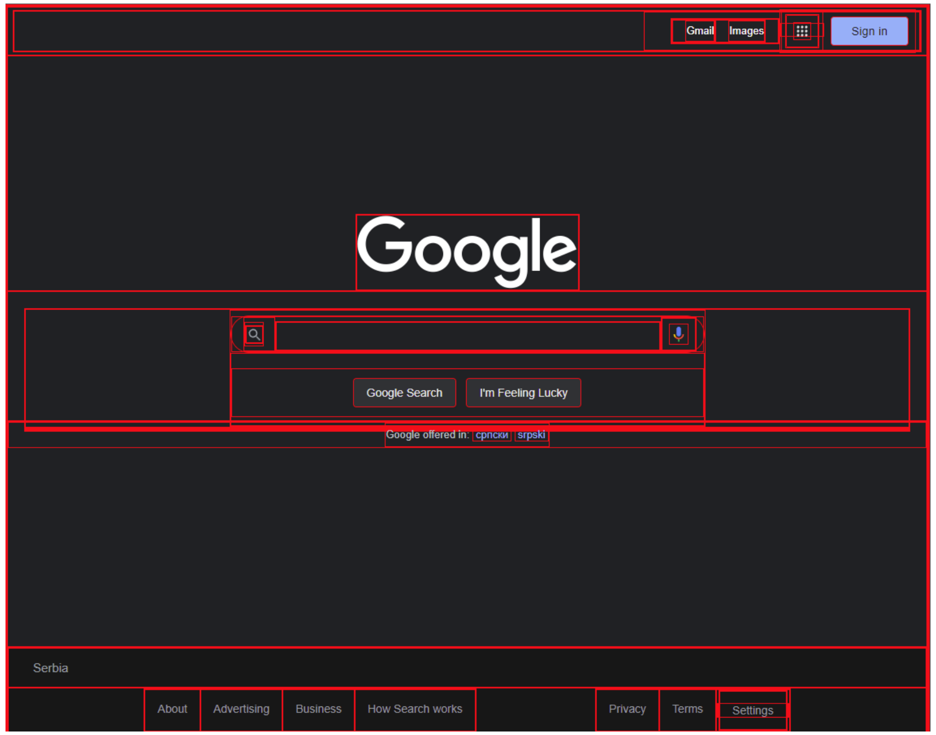
- Test out different resolutions
- Perform live edits on a webpage
- Use colorpicker to find value of a color you like
- Change text to prank friends

The Box Model

The Box Model states that every single element on the page is a rectangular box. These boxes can have more boxes in them or alongside them. The best way to see this is by giving each element a border via CSS.



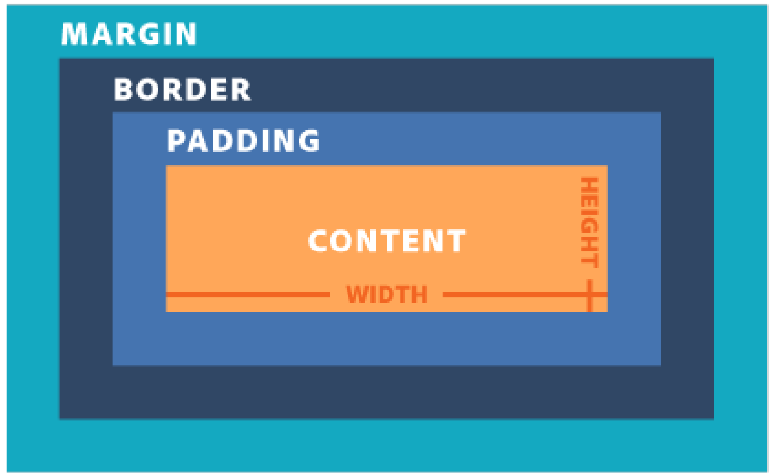
```css
* {
  border: 1px solid red;
}
```



The Box Model

There are many ways to manipulate the size of these boxes, and the space between them by using **padding, margin and border**. To sum it up briefly:

- **padding** is the space between the edge of a box and the content inside of it.
- **margin** is the space between a box and any other element that sit next to it.
- **border** adds space between the margin and the padding.



Display property

Most of the elements you have learned so far about are **block** elements, which means their default style is **display: block;**. These elements appear on the page stacked atop one another with each new element starting on a new line below the previous element.

Inline elements, unlike **block** elements, do not start on a new line. They appear in line with elements they are placed beside. One good example of an inline element is the link tag. If you put a link tag in the middle of a paragraph, it will behave like a part of the paragraph, instead of starting on a new line. Also, the padding and margins of an inline element behave differently than a **block** element's padding and margins.

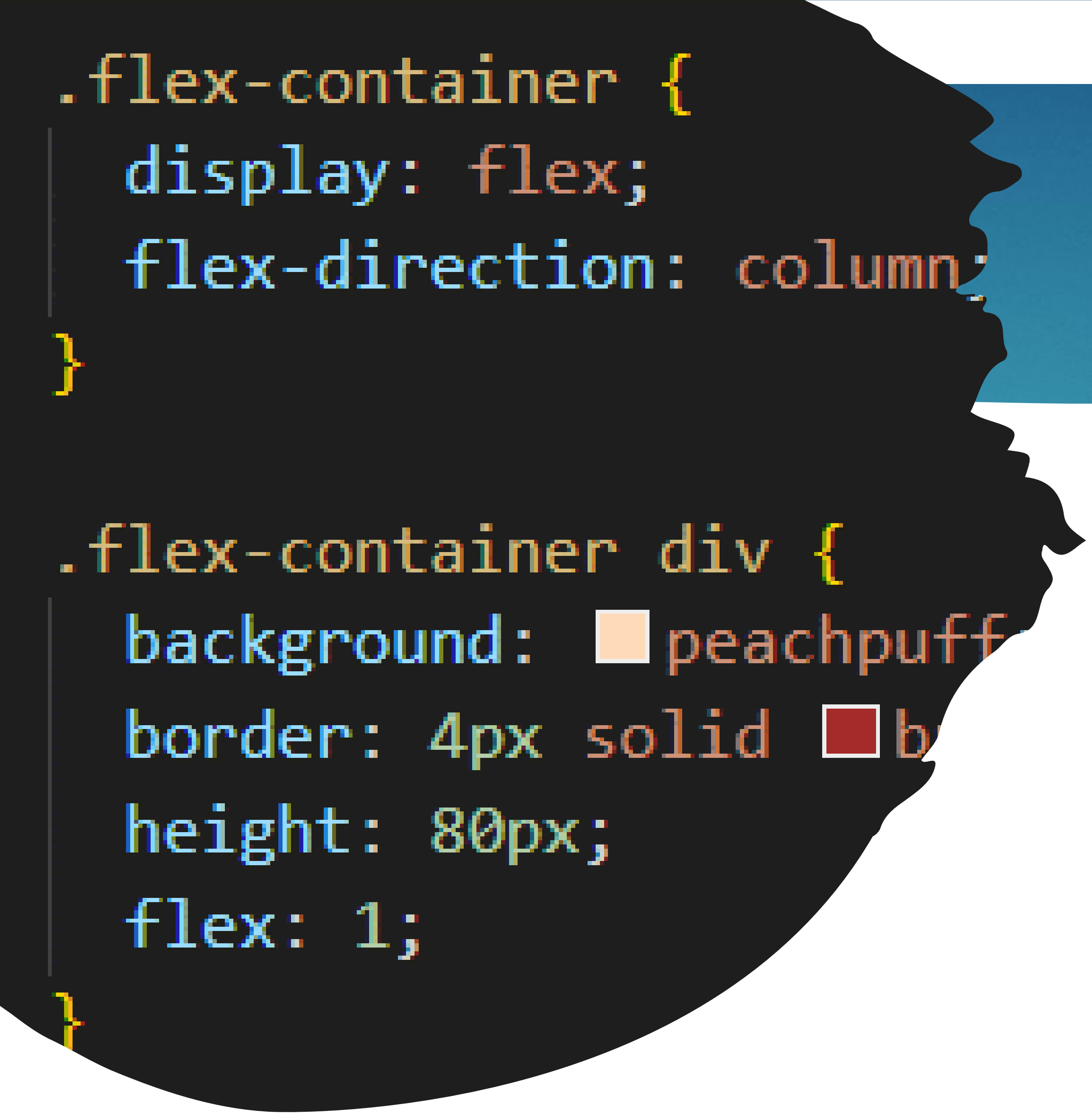
Inline-block elements behave like **inline** elements, except that they have padding and margins that behave like a **block** element’s ones. **Inline-block** is a good tool to know, but you will most likely use **Flexbox** if you want to line up a bunch of boxes, which we will cover later.

Display property

We can’t mention the display property without mentioning **div**s and **span**s. All the other HTML elements give meaning to their content - **img** element contains an image, **paragraph** element contains text, but **div**s and **span**s are just containers that can have anything inside of them. These elements are a lot more useful than they appear to be. We can use them to group elements under one parent for easier styling and positioning.

div is a **block** element by default, it is usually used as a container to group other elements. Divs allow us to divide the page into different segments and apply styling to them.

span is an **inline** element by default, it is usually used to group text content and inline HTML elements for styling when no other semantic HTML element is appropriate.



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

.flex-container div {
  background: 🟧 peachpuff
  border: 4px solid 🟥 b
  height: 80px;
  flex: 1;
}
```

```
ody {
    background-color:
    color: □ rgb(24, 24,
}

.container {
    border-radius: 4px;
    border: 4px solid 🟩 g
```

Flexbox

CSS

Flexbox

Flexbox is a layout model that allows elements to align and distribute space within a container. Using flexible widths and heights, elements can be aligned to fill a space or distribute space between elements, which makes it a great tool to use for responsive design systems.

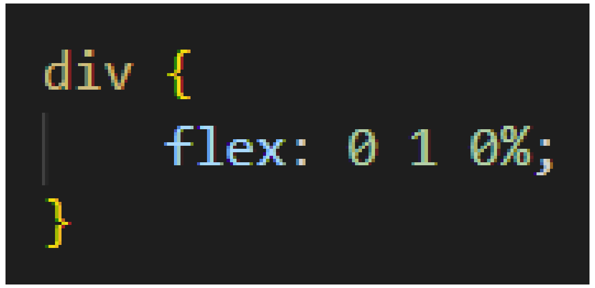
Flexbox is **not** just a single property, but an entire toolbox of properties that you can use to put things where you want them to be. Some of these properties belong on the **flex container** and some on the **flex items**.

A **flex container** is any element that has the property **display: flex;** on it, while a **flex item** is any element that's located inside the **flex container**.

Flex items can be both a flex item and a flex container, by that we mean - all flex items can also have the **display: flex;** property assigned to it, and then have its children arranged with flexbox. Creating and nesting multiple flex containers and items is the way that we will be building complex layouts.

Flexbox

The **flex** declaration is a shorthand for 3 properties that you can set on a flex item. These properties affect how flex items size themselves within their container (shorthand properties are CSS properties that let you set the values of multiple other CSS properties simultaneously). In the example below, **flex** is a shorthand for **flex-grow**, **flex-shrink** and **flex-basis**.

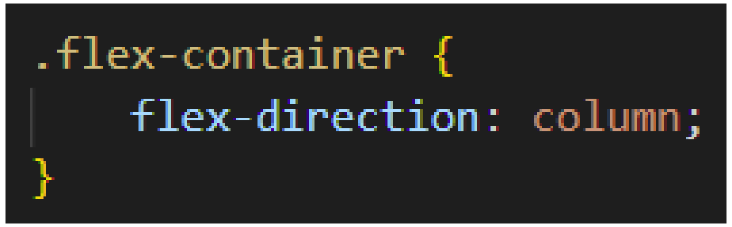


```
div {
    flex: 0 1 0%;
}
```

The default values for these 3 properties are the ones you see in the example above: **flex-grow** is 0, **flex-shrink** is 1 and **flex-basis** is 0%. Very often you see the **flex** shorthand being defined with only value (flex: 1;), in which case that value is applied to **flex-grow**, while the latter remains default.

Flexbox - Axes

The way some rules work change depending on which direction you are working with. The default direction for a flex container is **horizontal (row)**, but you can set it to **vertical (column)**. The direction can be specified via CSS like so:

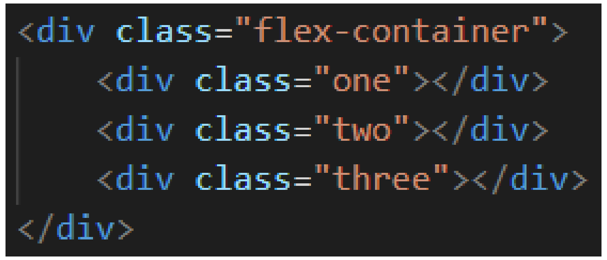


```
.flex-container {
    flex-direction: column;
}
```

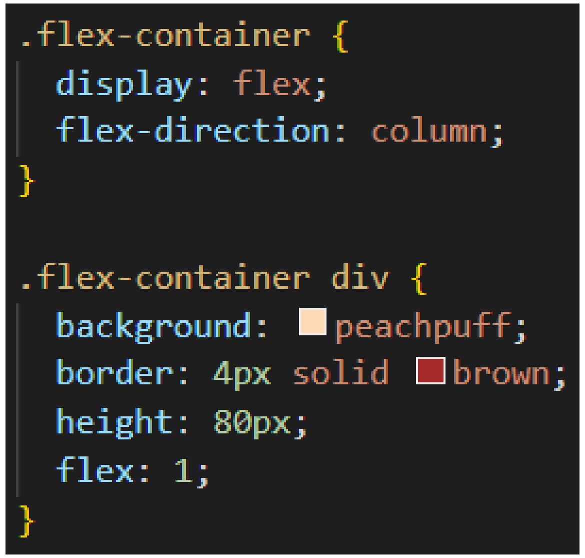
No matter which direction you’re using, you need to think of your flex-containers as having 2 axes: the main axis and the cross axis. It is the direction of these axes that changes when the flex-direction is changed. In most circumstances, flex-direction: row puts the main axis horizontal (left-to-right), and column puts the main axis vertical (top-to-bottom).

Flexbox - Axes

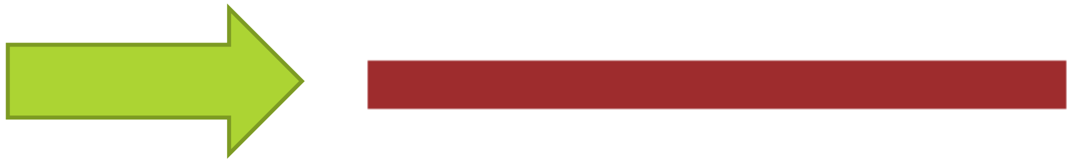
One thing to note is that in this example, **flex-direction: column** would not work as expected if we used the shorthand flex: The **div**s collapse, even though they clearly have a height defined there. The reason for this is that the flex shorthand expands flex-basis to 0, which means that all flex-growing and flex-shrinking would begin their calculations from 0. Empty divs by default have 0 height, so for our flex items to fill up the height of their container, they don’t actually need to have any height at all.



```html
<div class="flex-container">
    <div class="one"></div>
    <div class="two"></div>
    <div class="three"></div>
</div>
```

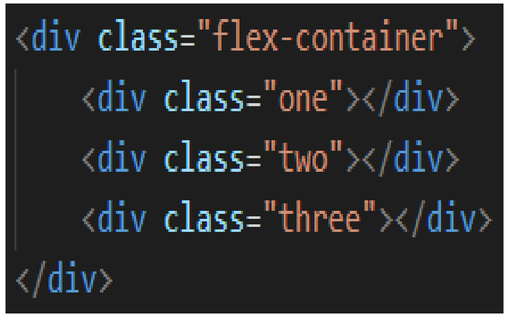


```css
.flex-container {
  display: flex;
  flex-direction: column;
}

.flex-container div {
  background: peachpuff;
  border: 4px solid brown;
  height: 80px;
  flex: 1;
}
```

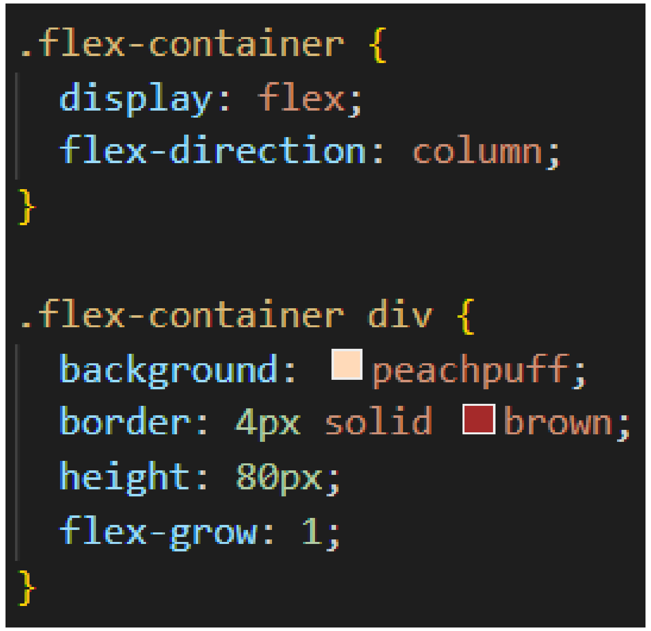


Flexbox - Axes

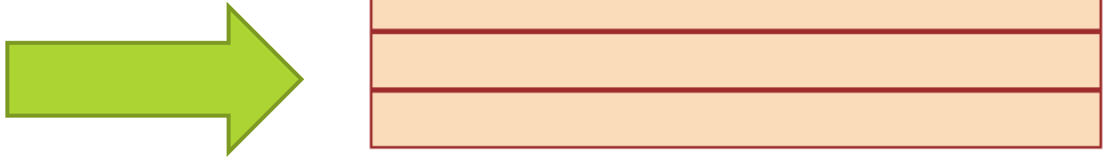
The previous example is fixed by specifying **flex: 1 1 auto**, telling the flex items to default to their given height. We could also have fixed it by putting a height on the parent .flex-container, or by using flex-grow: 1 instead of the shorthand.



```html
<div class="flex-container">
    <div class="one"></div>
    <div class="two"></div>
    <div class="three"></div>
</div>
```

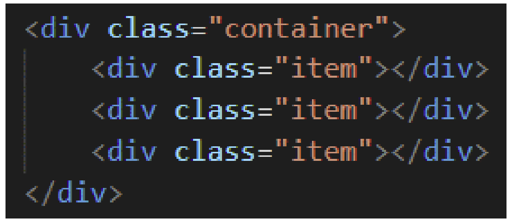


```css
.flex-container {
  display: flex;
  flex-direction: column;
}

.flex-container div {
  background: peachpuff;
  border: 4px solid brown;
  height: 80px;
  flex-grow: 1;
}
```

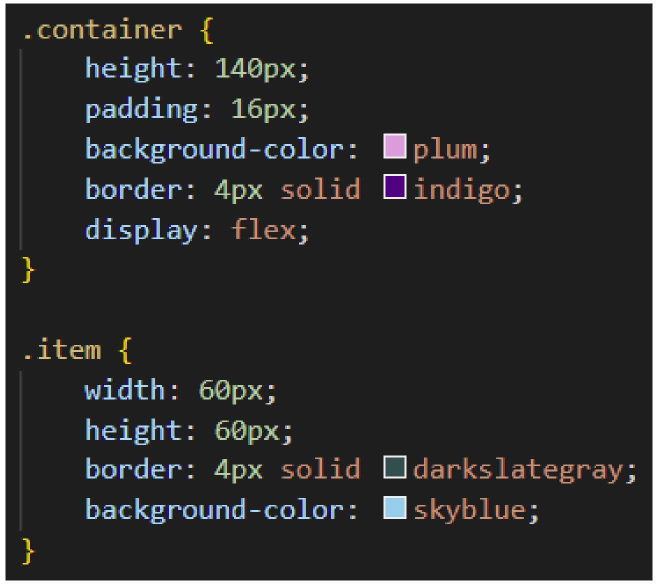


Flexbox - Alignment

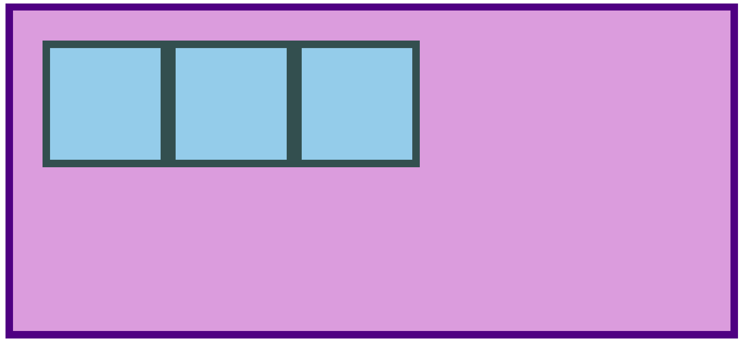
So far everything we've touched with flexbox has used the rule flex: 1 on all flex items, which makes the items grow or shrink equally to fill all of the available space. Very often, however, this is not the desired effect. Flex is also very useful for arranging items that have a specific size. Let's look at another example:



```html
<div class="container">
    <div class="item"></div>
    <div class="item"></div>
    <div class="item"></div>
</div>
```

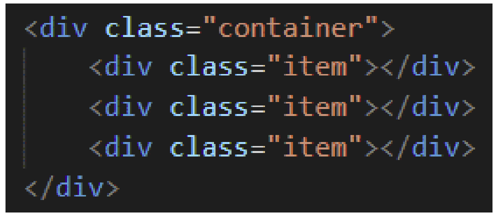


```css
.container {
    height: 140px;
    padding: 16px;
    background-color: plum;
    border: 4px solid indigo;
    display: flex;
}

.item {
    width: 60px;
    height: 60px;
    border: 4px solid darkslategray;
    background-color: skyblue;
}
```

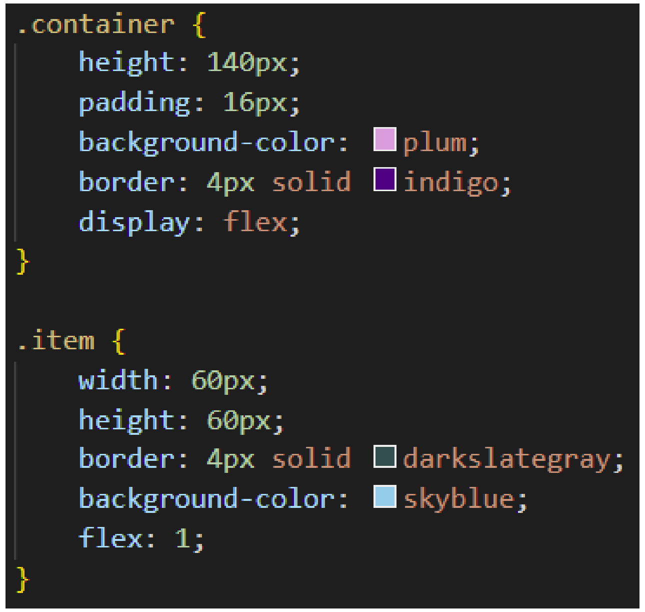


Flexbox - Alignment

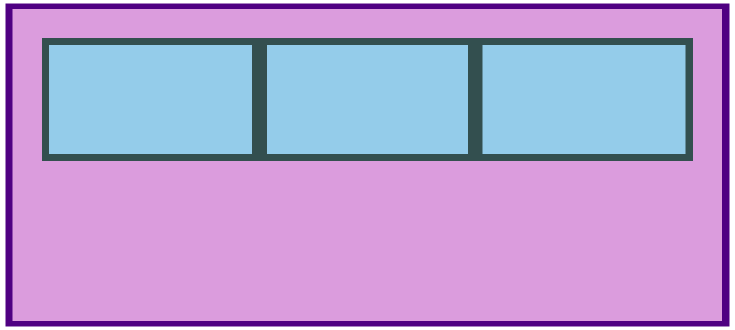
If we add **flex: 1;** to the item class, we can see that the items expand to take the full width of the parent.



```
<div class="container">
    <div class="item"></div>
    <div class="item"></div>
    <div class="item"></div>
</div>
```

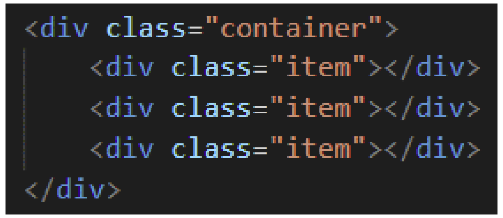


```
.container {
    height: 140px;
    padding: 16px;
    background-color: plum;
    border: 4px solid indigo;
    display: flex;
}

.item {
    width: 60px;
    height: 60px;
    border: 4px solid darkslategray;
    background-color: skyblue;
    flex: 1;
}
```

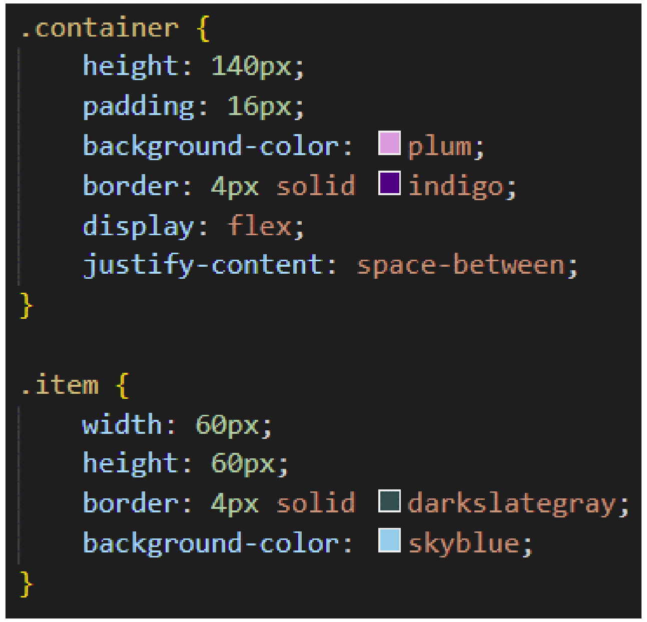


Flexbox - Alignment

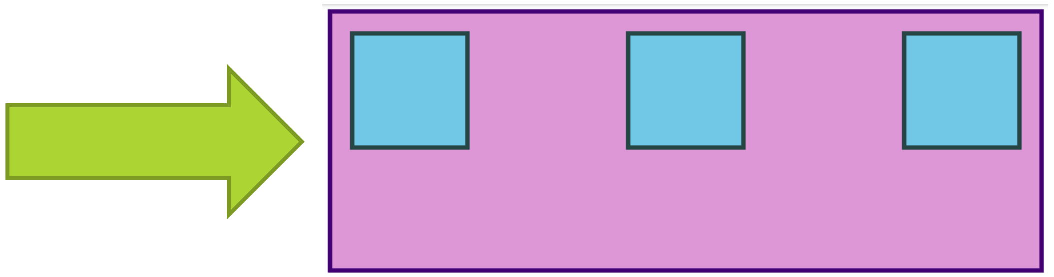
However, what if we wanted the items to stay the original width, but distribute themselves differently inside the container? First we remove **flex: 1;** from **.item** and add **justify-content: space-between** to **.container.**



```
<div class="container">
    <div class="item"></div>
    <div class="item"></div>
    <div class="item"></div>
</div>
```

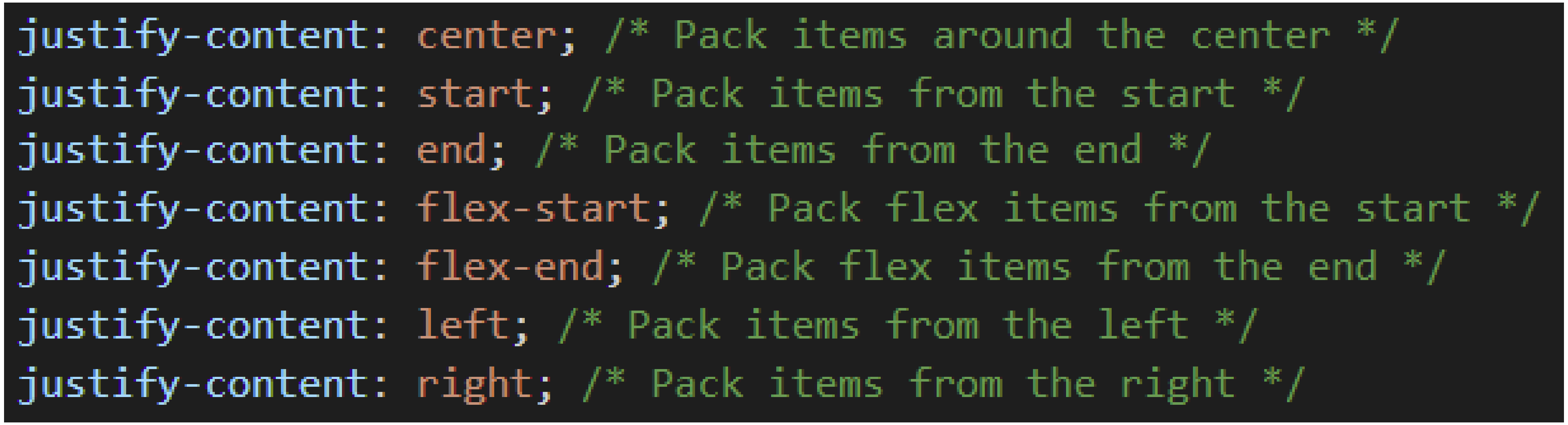


```
.container {
    height: 140px;
    padding: 16px;
    background-color: plum;
    border: 4px solid indigo;
    display: flex;
    justify-content: space-between;
}

.item {
    width: 60px;
    height: 60px;
    border: 4px solid darkslategray;
    background-color: skyblue;
}
```



Flexbox - Alignment

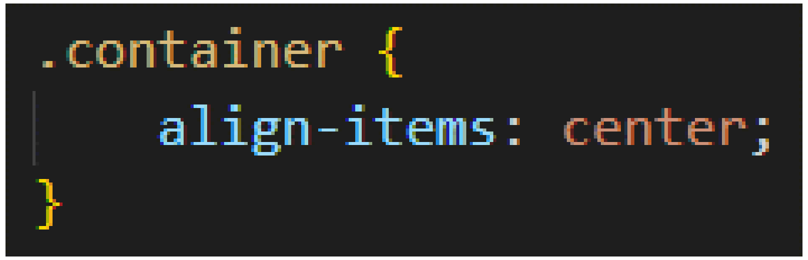
justify-content aligns items across the main axis. Let’s look at a few values that you can use here:



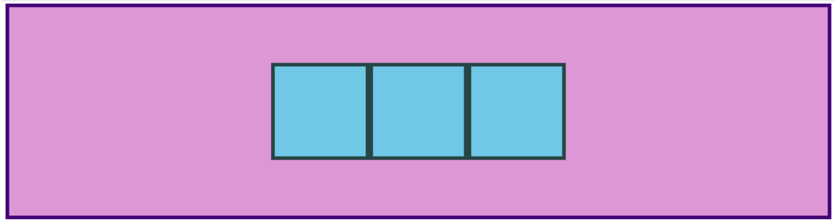
```
justify-content: center; /* Pack items around the center */
justify-content: start; /* Pack items from the start */
justify-content: end; /* Pack items from the end */
justify-content: flex-start; /* Pack flex items from the start */
justify-content: flex-end; /* Pack flex items from the end */
justify-content: left; /* Pack items from the left */
justify-content: right; /* Pack items from the right */
```

Flexbox - Alignment

To change the placement of items along the cross-axis use **align-items**. In the example below you can see how the items are placed in center along the cross axis by putting **align-items: center;** onto **.container**.



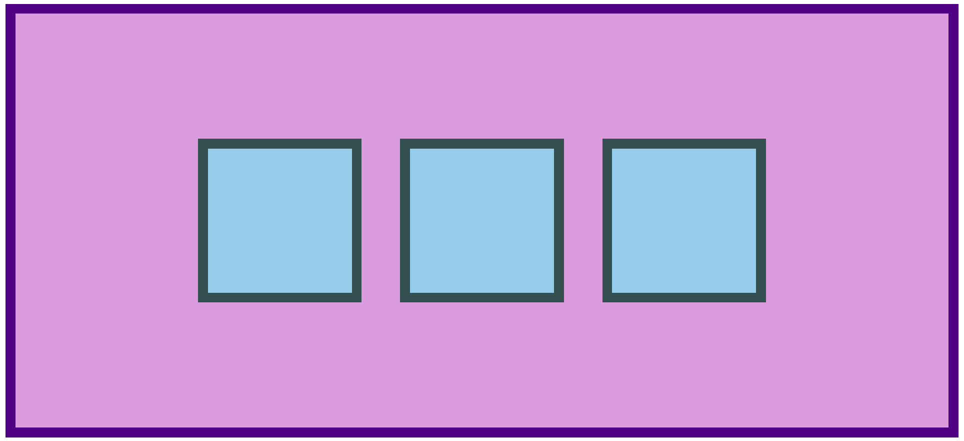
```
.container {
    align-items: center;
}
```



Because justify-content and align-items are based on the main and cross axis of your container, their behavior changes when you change the flex-direction of a flex-container. For example, when you change flex-direction to column, justify-content aligns vertically and align-items aligns horizontally.

Flexbox – Gap

One more useful feature of flex is the **gap** property which is placed on flex containers. Setting **gap** onto containers adds a specified place between its flex items. It’s similar to adding a margin to the items themselves. Adding a **gap** of 16px on previous example produces the result below:



Assignment

branko.fd.br@gmail.com

Thank you for your attention!