# CSS

## So, Bootstrap was pretty cool right? Now let's learn about CSS.

## Wait, wait, I forgot about last week. What is CSS?

CSS stands for Cascading Style Sheets. CSS is the standard language for defining styles on web pages. Although CSS is more widely known for its application in HTML documents, it can be used for defining styles for any structured document format (such as XML for example).

Styles are set using CSS properties. For example, you can set font properties (size, colors, style etc), background images, border styles, and much more.

## CSS History

CSS has gone through some major changes since CSS level 1 became a W3C recommendation in December 1996. CSS1 describes the CSS language as well as a simple visual formatting model. CSS2, which became a W3C recommendation in May 1998, builds on CSS1 and adds support for media-specific style sheets (e.g. printers and aural devices), downloadable fonts, element positioning and tables.

As of this writing, CSS3 is currently under development, and includes more advanced features such as animations, background gradients, multi-column layouts, border images, and more.

## HTML vs CSS

It's not really "HTML vs CSS". In fact HTML and CSS are the best of friends! CSS is a supplementary extension to HTML. Here's what each does:

### HTML

The purpose of HTML is to provide document structure and meaning. This is particularly true with the introduction of HTML5. You use HTML to specify what elements go on the page (eg, headings, paragraphs, tables, images, etc). Each of these elements (as well as their attributes and attribute values) have a certain meaning.

### CSS

You use CSS to specify how those HTML elements look. But not just how they look, how they are presented. After all, you can use CSS to provide styles for speech output (for example, for users using a screen reader). So, you can write an HTML document without being concerned with its presentation, then use CSS to specify how it will be presented within any given context. Not only this, but you can change the CSS without having to change the HTML. In other words, you can "plug" a style sheet into an HTML document and the HTML document will immediately take on the styles of that style sheet.

## Class and ID Selectors

For the CSS Beginner Tutorial we looked solely at HTML selectors — those that represent an HTML tag.

You can also define your own selectors in the form of class and ID selectors.

The benefit of this is that you can have the same HTML element, but present it differently depending on its class or ID.

In the CSS, a class selector is a name preceded by a full stop (“.”) and an ID selector is a name preceded by a hash character (“#”).

So the CSS might look something like:

```
#top {
    background-color: #ccc;
    padding: 20px
}

.intro {
    color: red;
    font-weight: bold;
}
```

The HTML refers to the CSS by using the attributes id and class. It could look something like this:

```
<div id="top">

<h1>Chocolate curry</h1>

<p class="intro">This is my recipe for making curry purely with chocolate</p>

<p class="intro">Mmm mm mmmmm</p>

</div>
```

The difference between an ID and a class is that an ID can be used to identify one element, whereas a class can be used to identify more than one.

You can also apply a selector to a specific HTML element by simply stating the HTML selector first, so p.jam { /* whatever */ } will only be applied to paragraph elements that have the class “jam”.

## Grouping and Nesting

Two ways that you can simplify your code — both HTML and CSS — and make it easier to manage.

### Grouping

You can give the same properties to a number of selectors without having to repeat them.

For example, if you have something like:

```
h2 {
    color: red;
}

.thisOtherClass {
    color: red;
}

.yetAnotherClass {
    color: red;
}
```

You can simply separate selectors with commas in one line and apply the same properties to them all so, making the above:

```
h2, .thisOtherClass, .yetAnotherClass {
    color: red;
}
```

### Nesting

If the CSS is structured well, there shouldn’t be a need to use many class or ID selectors. This is because you can specify properties to selectors within other selectors.

For example:

```
#top {
    background-color: #ccc;
    padding: 1em
}

#top h1 {
    color: #ff0;
}

#top p {
    color: red;
    font-weight: bold;
}
```

This removes the need for classes or IDs on the p and h1 tags if it is applied to HTML that looks something like this:

```
<div id="top">
    <h1>Chocolate curry</h1>
    <p>This is my recipe for making curry purely with chocolate</p>
    <p>Mmm mm mmmmm</p>
</div>
```

This is because, by separating selectors with spaces, we are saying “h1 inside ID top is `color #ff0”` and “p inside ID top is red and bold”.

This can get quite complicated (because it can go for more than two levels, such as this inside this inside this inside this etc.) and may take a bit of practice.

## Pseudo Classes

Pseudo classes are bolted on to selectors to specify a state or relation to the selector. They take the form of `selector:pseudo_class { property: value; }`, simply with a colon in between the selector and the pseudo class.

Links link, targeting unvisited links, and visited, targeting, you guessed it, visited links, are the most basic pseudo classes.

The following will apply colors to all links in a page depending on whether the user has visited that page before or not:

```
a:link {
    color: blue;
}

a:visited {
    color: purple;
}
```

## Dynamic Pseudo Classes

Also commonly used for links, the dynamic pseudo classes can be used to apply styles when something happens to something.

- active is for when something activated by the user, such as when a link is clicked on.
- hover is for a when something is passed over by an input from the user, such as when a cursor moves over a link.
- focus is for when something gains focus, that is when it is selected by, or is ready for, keyboard input.

**focus** is most often used on form elements but can be used for links. Although most users will navigate around and between pages using a pointing device such as a mouse those who choose note to, or are unable to do so, such as those with motor disabilities, may navigate using a keyboard or similar device. Links can be jumped between using a tab key and they will gain focus one at a time.

```
a:active {
    color: red;
}

a:hover {
    text-decoration: none;
    color: blue;
    background-color: yellow;
}

input:focus, textarea:focus {
    background: #eee;
}
```

## First Children

Finally (for this article, at least), there is the first-child pseudo class. This will target something only if it is the very first descendant of an element. So, in the following HTML…

```
<body>
    <p>I’m the body’s first paragraph child. I rule. Obey me.</p>
    <p>I resent you.</p>
...
```

...if you only want to style the first paragraph, you could use the following CSS:

```
p:first-child {
    font-weight: bold;
    font-size: 40px;
}
```

**note**

CSS3 has also delivered a whole new set of pseudo classes: last-child, target, first-of-type, and more.

## Shorthand Properties

Some CSS properties allow a string of values, replacing the need for a number of properties. These are represented by values separated by spaces.

Margins and Padding margin allows you to amalgamate margin-top, margin-right, margin-bottom, and margin-left in the form of property: top right bottom left;

So:

```
p {
    margin-top: 1px;
    margin-right: 5px;
    margin-bottom: 10px;
    margin-left: 20px;
}
```

Can be summed up as:

```
p {
    margin: 1px 5px 10px 20px;
}
```

padding can be used in exactly the same way.

By stating just two values (such as padding: 1em 10em;), the first value will be the top and bottom and the second value will be the right and left.

## Borders

border-width can be used in the same way as margin and padding, too. You can also combine border-width, border-color, and border-style with the border property. So:

```
p {
    border-width: 1px;
    border-color: red;
    border-style: solid;
}
```

Can be simplified to be:

```
p {
    border: 1px red solid;
}
```

The width/color/style combination can also be applied to border-top, border-right etc.

## Fonts

Font-related properties can also be gathered together with the font property:

```
p {
    font: italic bold 12px/2 courier;
}
```

This combines font-style, font-weight, font-size, line-height, and font-family.

So, to put it all together, try this code:

```
p {
    font: 14px/1.5 "Times New Roman", times, serif;
    padding: 30px 10px;
    border: 1px black solid;
    border-width: 1px 5px 5px 1px;
    border-color: red green blue yellow;
    margin: 10px 50px;
}
```

Lovely.

## Background Images

Used in a very different way to the `img` HTML element, CSS background images are a powerful way to add detailed presentation to a page.

To jump in at the deep end, the shorthand property background can deal with all of the basic background image manipulation aspects.

```
body {
    background: white url(http://www.htmldog.com/images/bg.gif) no-repeat top
right;
}
```

This amalgamates these properties:

- background-color, which we have come across before.
- background-image, which is the location of the image itself.
- background-repeat, which is how the image repeats itself. Its value can be:
- repeat, the equivalent of a “tile” effect across the whole background,
- repeat-y, repeating on the y-axis, above and below,
- repeat-x (repeating on the x-axis, side-by-side), or
- no-repeat (which shows just one instance of the image).
- background-position, which can be top, center, bottom, left, right, a length, or a percentage, or any sensible combination, such as top right.

It can actually be used to specify a few other background features too, notably attachment, clip, origin, and size (see the background property reference for the nitty-gritty), but let’s not get carried away just yet — color, image, repeat, and position are by far the most fundamental aspects that you would want to manipulate most often.

Background-images can be used in most HTML elements - not just for the whole page (body) and can be used for simple but effective results. As an example, background images are used on this web site as the bullets in lists, as the magnifying glass in the search box, and as the icons in the top left corner of some notes, such as this one.

## Specificity

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

It may not seem like something that important, and in most cases you won’t come across any conflicts at all, but the larger and more complex your CSS files become, or the more CSS files you start to juggle with, the greater likelihood there is of conflicts arising.

**More Specific = Greater Precedence**

If the selectors are the same, then the last one will always take precedence. For example, if you had:

```
p { color: red }
p { color: blue }
```

The text in the box of p elements would be colored blue because that rule came last.

However, you won’t usually have identical selectors with conflicting declarations on purpose (because there’s not much point). Conflicts quite legitimately come up, though, when you have nested selectors.

```
div p { color: red }
p { color: blue }
```

In this example it might seem that a `p`within a `div` would be colored blue, seeing as a rule to color `p` boxes blue comes last, but they would actually be colored red due to the specificity of the first selector. Basically, the more specific a selector, the more preference it will be given when it comes to conflicting styles.

## Calculating Specificity

The actual specificity of a group of nested selectors takes some calculating. You can give every ID selector (“#whatever”) a value of 100, every class selector (“.whatever”) a value of 10 and every HTML selector (“whatever”) a value of 1. When you add them all up, hey presto, you have a specificity value.

- `p` has a specificity of 1 (1 HTML selector)
- `div p` has a specificity of 2 (2 HTML selectors, 1+1)
- `.tree` has a specificity of 10 (1 class selector)
- `div p.tree` has a specificity of 12 (2 HTML selectors + a class selector, 1+1+10)
- `#baobab` has a specificity of 100 (1 id selector)
- body `#content .alternative p` has a specificity of 112 (HTML selector + id selector + class selector + HTML selector, 1+100+10+1)

So if all of these examples were used, div p.tree (with a specificity of 12) would win out over div p (with a specificity of 2) and `body #content .alternative p` would win out over all of them, regardless of the order.

## Display

A key trick to the manipulation of HTML elements is understanding that there’s nothing at all special about how most of them work. Most pages could be made up from just a few tags that can be styled any which way you choose. The browser’s default visual representation of most HTML elements consist of varying font styles, margins, padding and, essentially, display types.

The most fundamental types of display are inline, block and none and they can be manipulated with the display property and the shockingly surprising values inline, block and none.

## Inline

inline does just what it says — boxes that are displayed inline follow the flow of a line. Anchor (links) and emphasis are examples of elements that are displayed inline by default.

The following code, for example, will cause all list items in a list to appear next to each other in one continuous line rather than each one having its own line:

```
li { display: inline }
```

## Block

block makes a box standalone, fitting the entire width of its containing box, with an effective line break before and after it. Unlike inline boxes, block boxes allow greater manipulation of height, margins, and padding. Heading and paragraph elements are examples of elements that are displayed this way by default in browsers.

The next example will make all links in “nav” large clickable blocks:

```
#navigation a {
    display: block;
    padding: 20px 10px;
}
```

display: inline-block will keep a box inline but lend the greater formatting flexibility of block boxes, allowing margin to the right and left of the box, for example.

## None

none, well, doesn’t display a box at all, which may sound pretty useless but can be used to good effect with dynamic effects, such as switching extended information on and off at the click of a link, or in alternative stylesheets.

The following code, for example, could be used in a print stylesheet to basically “turn off” the display of elements such as navigation that would be useless in that situation:

```
#navigation, #related_links { display: none }
```

display: none and visibility: hidden vary in that display: none takes the element’s box completely out of play, whereas visibility: hidden keeps the box and its flow in place without visually representing its contents. For example, if the second paragraph of 3 were set to display: none, the first paragraph would run straight into the third whereas if it were set to visibility: hidden, there would be a gap where the paragraph should be.

## Tables

OK. So that was the basics. Now for something a little more advanced and rarely used…

Perhaps the best way to understand the table-related display property values is to think of HTML tables. table is the initial display and you can mimic the `tr` and `td` elements with the table-row and table-cell property values respectively.

The display property goes further by offering table-column, table-row-group, table-column-group, table-header-group, table-footer-group and table-caption as values, which are all quite self-descriptive. The immediately obvious benefit of these values is that you can construct a table by columns, rather than the row-biased method used in HTML.

Finally, the value `inline-table` basically sets the table without line breaks before and after it.

Be careful when using these values. Older browsers struggle with them and getting carried away with CSS tables can seriously damage your accessibility. HTML should be used to convey meaning, so if you have tabular data it should be arranged in HTML tables. Using CSS tables exclusively could result in a mash of data that is completely unreadable without the CSS. Bad. And not in a Michael Jackson way.

### Other display types

list-item displays a box in the way that you would usually expect an li HTML element to be displayed. To work properly then, elements displayed this way should be nested in a `ul` or `ol` element.

run-in makes a box either in-line or block depending on the display of its parent.

## Pseudo Elements

Pseudo elements suck on to selectors much like pseudo classes, taking the form of `selector:pseudoelement { property: value; }`. There are four of the suckers.

### First Letters and First Lines

The first-letter pseudo element applies to the first letter inside a box and first-line to the top-most displayed line in a box.

As an example, you could create drop caps and a bold first-line for paragraphs with something like this:

```
p {
    font-size: 12px;
}

p:first-letter {
    font-size: 24px;
    float: left;
}

p:first-line {
    font-weight: bold;
}
```

The CSS 3 specs suggest pseudo elements should include two colons, so `p::first-line` as opposed to `p:first-line`. This is to differentiate them with pseudo classes. Aiming for backwards-compatibility (whereby older web pages will still work in new browsers), browsers will still behave if they come across the single colon approach and this remains the best approach in most circumstances due to some older browsers not recognizing the double colon.

## Before and After Content

The before and after pseudo elements are used in conjunction with the content property to place content either side of a box without touching the HTML.

What?! Content in my CSS?! But I thought HTML was for content!

Well, it is. So use sparingly. Look at it like this: You are borrowing content to use solely as presentation, such as using “!” because it looks pretty. Not because you actually want to exclaim anything.

The value of the content property can be open-quote, close-quote, any string enclosed in quotation marks, or any image, using `url(imagename)`.

```
blockquote:before {
    content: open-quote;
}

blockquote:after {
    content: close-quote;
}

li:before {
    content: "POW! ";
}

p:before {
    content: url(images/jam.jpg);
}
```

The **content** property effectively creates another box to play with so you can also add styles to the "presentational content":

```
li:before {
    content: "POW! ";
    background: red;
    color: #fc0;
}
```

## Page Layout

In the olden days, pre-hominid apes used HTML tables to layout web pages. Hilarious, right?! But CSS, that 2001: A Space Odyssey monolith, soon came along and changed all of that.

Layout with CSS is easy. You just take a chunk of your page and shove it wherever you choose. You can place these chunks absolutely or relative to another chunk.

### Positioning

The **position** property is used to define whether a box is absolute, relative, static or fixed:

- static is the default value and renders a box in the normal order of things, as they appear in the HTML.
- relative is much like static but the box can be offset from its original position with the properties top, right, bottom and left.
- absolute pulls a box out of the normal flow of the HTML and delivers it to a world all of its own. In this crazy little world, the absolute box can be placed anywhere on the page using top, right, bottom and left.
- fixed behaves like absolute, but it will absolutely position a box in reference to the browser window as opposed to the web page, so fixed boxes should stay exactly where they are on the screen even when the page is scrolled.

When we talk of absolutely positioned boxes being placed anywhere on the page, they're actually still relatively positioned from the edges of the page. And to add another backtrack, the page doesn't actually have to be the container — a box will be "absolutely" positioned in relation to any non-static positioned containing box. Just ignore that for now, though…

### Layout using absolute positioning

You can create a traditional two-column layout with absolute positioning if you have something like the following HTML:

```
<div id="navigation">
    <ul>
        <li><a href="this.html">This</a></li>
        <li><a href="that.html">That</a></li>
        <li><a href="theOther.html">The Other</a></li>
    </ul>
</div>

<div id="content">
    <h1>Ra ra banjo banjo</h1>
    <p>Welcome to the Ra ra banjo banjo page. Ra ra banjo banjo. Ra ra banjo
banjo. Ra ra banjo banjo.</p>
    <p>(Ra ra banjo banjo)</p>
</div>
```

We're being old school and using div elements so as not to introduce too many new things, but Sectioning is sexier.

And if you apply the following CSS:

```
#navigation {
    position: absolute;
    top: 0;
    left: 0;
    width: 200px;
}

#content {
    margin-left: 200px;
}
```

You will see that this will set the navigation bar to the left and set the width to 200 pixels. Because the navigation is absolutely positioned, it has nothing to do with the flow of the rest of the page so all that is needed is to set the left margin of the content area to be equal to the width of the navigation bar.

How stupidly easy! And you aren't limited to this two-column approach. With clever positioning, you can arrange as many blocks as you like. If you wanted to add a third column, for example, you could add a "navigation2" chunk to the HTML and change the CSS to:

```
#navigation {
    position: absolute;
    top: 0;
    left: 0;
    width: 200px;
}

#navigation2 {
    position: absolute;
    top: 0;
    right: 0;
    width: 200px;
}

#content {
    margin: 0 200px; /* setting top and bottom margin to 0 and right and left
margin to 200px */
}
```

The only downside to absolutely positioned boxes is that because they live in a world of their own, there is no way of accurately determining where they end. If you were to use the examples above and all of your pages had small navigation bars and large content areas, you would be okay, but, especially when using relative values for widths and sizes, you often have to abandon any hope of placing anything, such as a footer, below these boxes. If you wanted to do such a thing, one way would be to float your chunks, rather than absolutely positioning them.

## Floating

Floating a box will shift it to the right or left of a line, with surrounding content flowing around it.

Floating is normally used to shift around smaller chunks within a page, such as pushing a navigation link to the right of a container, but it can also be used with bigger chunks, such as navigation columns.

Using float, you can `float: left` or `float: right.`

Working with the same HTML, you could apply the following CSS:

```
#navigation {
    float: left;
    width: 200px;
}

#navigation2 {
    float: right;
    width: 200px;
}

#content {
    margin: 0 200px;
}
```

Then, if you do not want the next box to wrap around the floating objects, you can apply the clear property:

- clear: left will clear left floated boxes
- clear: right will clear right floated boxes
- clear: both will clear both left and right floated boxes.

So if, for example, you wanted a footer in your page, you could add a chunk of HTML…

```
<div id="footer">
    <p>Footer! Hoorah!</p>
</div>
```

…and then add the following CSS:

```
#footer {
    clear: both;
}
```

And there you have it. A footer that will appear underneath all columns, regardless of the length of any of them.

This has been a general introduction to positioning and floating, with emphasis on the larger “chunks” of a page, but remember, these methods can be applied to any box within those boxes, too. With a combination of positioning, floating, margins, padding and borders, you should be able to represent any web design your mischievous little imagination can conjure. The best way to learn? Tinker! Play! Go!

References:

1. https://www.htmldog.com/guides/css/intermediate/layout/