# Anatomy of CSS

 Estimated Time: **1-2 hours**

CSS (Cascading Style Sheets) is the language used to control the presentation layer. Whereas HTML is about structure and content (aka, the "content layer"), CSS is about style and appearance. We use it to control the visual aspects of the content on a page: from fonts to color to size to animations and more.

We already encountered some simple CSS rulesets in Lesson 1. In the instructions for getting to know Sublime Text, we created a rule that targeted paragraph elements, setting the text color to red:

p {

color: red;

}

When we previewed our index.html page (which linked to the file containing this CSS) in the browser, we saw that the paragraph text was red.

Although there's plenty about CSS that can be tricky (especially positioningand layout which we'll deal with in depth later in this unit), the syntax itself is straightforward. Target a set of elements with a CSS selector (p, above), and set properties and values as required by your presentational goal (that the paragraph text be red).

In the remainder of this reading, we'll give you the background knowledge about how CSS works that you'll need to reason about styling and presentation. Like the Anatomy of HTML assignment in the previous lesson, this too will be reading-heavy, without a ton of doing, but it will give you the knowledge you need to understand and communicate with others about CSS.

Here's what we're going to cover:

* The parts of a CSS rule and the terminology used to describe CSS
* How to target document elements using CSS selectors
* CSS specificity
* the "cascade"
* How to activate CSS in your HTML code
* The box model
* Browser defaults, CSS resets, and CSS normalization

### Anatomy of CSS

The key vocabulary words for CSS are: ruleset, selector, declaration block, declarations, property, and value.

input {

display: block;

font-family: 'Proxima Nova W01', sans-serif;

font-weight: 400;

font-style: normal;

height: 45px;

width: 420px;

min-width: 210px;

max-width: 100%;

padding: 6px 1em 5px;

border: 1px solid #d0d2d5;

border-radius: 3px;

font-size: 15px;

line-height: 30px;

color: #404853;

box-shadow: inset 0 3px 7px #f6f7f7;

}

This snippet is from a page on Thinkful's website. It is a ruleset that describes how input elements should look. It consists of a selector (input, in this case), which is the element or elements that will be targeted by the declarations that follow.

The declaration block follows the selector. A declaration block is a set of declarations contained in curly brackets ({, }). Within the block, each line is a separate declaration.

A declaration consists of a property and the value it is to be set to (for instance, box-shadow: inset 0 3px 7px #f6f7f7;, above). With CSS3, there are nearly 200 supported properties that can be set. **Don't worry, you don't need to memorize them all!**

Each property has a range of valid values (for instance, width could be set in terms of pixels or percentage, among other options, but setting width to "foo" would not be valid).

*/\*invalid\*/*

.foo {

width: 'foo';

}

*/\* valid \*/*

.foo {

width: 100px;

}

.bar {

width: 50%;

}

The property name is followed by a colon, and the value is followed by a semicolon.

In the first line of the snippet above (/\*invalid\*/), notice that we've used a code comment. As with HTML, you can use comments in CSS to document your code or temporarily disable (comment out) a block of code.

With real web pages, the final presentation of a given element will usually be the result of several rulesets interacting. We'll learn more about this in a moment when we discuss CSS specificity, but for the moment, know that this:

p {

font-family: Arial;

}

p {

font-size: 20px;

}

is valid (though not the most concise) CSS. In practice though, the computed CSS that gets applied to an element by the browser is often the result of a set of rulesets, possibly spanning more than one stylesheet.

### CSS Selectors

CSS provides a rich set of selectors that give you precise control over the elements targeted by a declaration block. Later in this lesson, you'll complete drills on CSS selectors to build up your working knowledge of the most commonly used selectors.

The following examples will give you a sense of some of the ways you can target elements for style rules:

*/\* universal selector (applies to everything) \*/*

\* {

*/\* set stuff \*/*

}

*/\* targeting a single element type \*/*

p {

*/\* set stuff \*/*

}

*/\* targeting two different elements \*/*

p, input {

*/\* set stuff \*/*

}

*/\* targeting a class \*/*

.foo {

*/\* set stuff \*/*

}

*/\* targeting an id \*/*

*/\* avoid these, but know how to recognize*

*them in the wild. It's usually better to*

*use a class selector instead.*

*\*/*

**#bar** {

*/\* set stuff \*/*

}

*/\* targeting an element with a class \*/*

*/\* try to avoid this, in favor of simple class declaration \*/*

p.foo {

*/\* set stuff \*/*

}

*/\* targeting descendants \*/*

ul.foo li {

*/\* any `li` within `ul.foo` will get targeted \*/*

}

*/\* targeting direct children \*/*

ul > li {

*/\* only `li`s that are direct children of ul targeted \*/*

}

*/\* targeting submit buttons \*/*

button[type="submit"] {

*/\* any button with a type of "submit" \*/*

}

All of these selectors can be combined to get more specific targeting.

When you're writing a ruleset, try to keep your selectors as non-specific as possible. For instance, p.foo would target any paragraph with the foo class. While that's valid CSS, it's almost always better to just create a ruleset for only .foo. That way the settings can be reused on other targets if we decide they should have the same style rules.

In addition to element, class, and id selectors, there are also pseudo-classesand pseudo-elements. A pseudo-class is used to specify a special state of the element. A pseudo-element is used to style specified parts of an element.

The following CodePen gives us an example of both:

In this example, we have a ruleset for the first letter of every paragraph. The selector here (p::first-letter - note the double colon) is a pseudo-element. It targets specific parts of the document. Here we're setting font properties for the first letter of every paragraph. You'll encounter other pseudo-elements later in this lesson.

This example also uses a pseudo-class to generate hover behavior (div.foo:hover {...}). If you hover over the gray container box, the background color switches to orange. Pseudo-classes target specific states of the document (such as the user hovering over, or clicking on an element).

### Getting CSS into HTML

Typically, to utilize CSS, we link to one or more external stylesheets that contain style rules that get applied to the linking HTML page.

<head>

<link rel="stylesheet" type="text/css" href="./css/main.css">

<link rel="stylesheet" type="text/css" href="./css/nav-bar.css">

</head>

You already got experience with linking to CSS files when you ported Sally Student's "About me" page with Sublime Text.

Although you are discouraged from using the next two strategies unless you have a specific, compelling reason, it's worth knowing that there are two other ways of affecting the presentation layer.

Most HTML elements allow a style property to be set. So <p style="color: blue; font-family: Arial">Lorem ipsum</p> would result in blue Arial text. This is an example of inline styling and it is avoided. It's best practice to use HTML only for describing structure and content, and using CSS to describe presentation. Engineers refer to this as maintaining the separation of concerns between HTML and CSS. Inline styles encode information about how an element should look directly into the HTML. This means you cannot reuse these styles on other elements. One of the advantages of CSS is reusable classes (say an .error class with red font color that we can apply wherever we need to give a user an error message), and classes are not possible with inline styles. By isolating presentation into our CSS, our code will be more maintainable and extensible (that is, easier to apply presentation rules we've set up in one place to new use cases).

It is also possible to put CSS within a [style element](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style). Here's an example:

*<!-- somewhere in <head> -->*

<style>

p {

color: red;

}

</style>

*<!-- somewhere in body -->*

<p>Hi there how's it going?</p>

At this early point in your learning, avoid internal styles. There are some reasons to use them (for instance, [improving page load speed](https://developers.google.com/speed/docs/insights/OptimizeCSSDelivery#example)), but that's something you can focus on when you're further along in your learning.

### CSS Specificity

You just learned that a given HTML element's appearance can be determined by numerous CSS rulesets. In the example we gave above, there were no conflicting property/value pairs, but in real web pages -- especially for bigger sites with more contributors -- you often end up with style rules from multiple places targeting a specific element. If there are conflicts for a given property, the browser will choose the rule with higher specificity.

Let's look at an example to see how this works.

In this example, according to the second ruleset selector (.foo p), the paragraph color should be red, because the selector says to target all paragraphs in an element with .foo class. But the previous selector has higher specificity because it targets paragraphs in any div whose class is foo, so our end result is green.

The key take away for now is that style rules with higher specificity will trump those with lower specificity if there are conflicting values being set for some property. Store this bit of knowledge away in your toolkit -- at some point in the future, you'll find yourself debugging a style setting, certain that a rule you wrote should be causing an element to look a certain way, but finding that in reality that element looks a different way. When that happens, it's often a sign that a higher priority rule is over-riding the one you're working with at the moment.

The following post from Medium is an excellent starting point to learn more about specificity: [Understanding CSS: Selector Specificity](https://medium.com/@dte/understanding-css-selector-specificity-a02238a02a59#edc9)

## The Cascade

We're now in a position to understand the cascade from Cascading Style Sheets. The cascade is a process browsers follow to determine which CSS values get applied for all the properties on a given element. Above, we saw an example of conflicting CSS values. In one case the later one overwrote the earlier one, and in another case, an earlier rule with higher specificity trumped a later rule in the same file.

To determine which property-value pairs to apply to a particular element, the browser

* Determines which rules apply to the element
* Takes all the relevant rulesets and sorts them according to their origin (for instance, inline styles vs. external stylesheets -- inline styles win over external) and importance (more on importance in a moment).
* Takes all rulesets that have same origin and importance, and sort them by selector specificity.
* If there are still conflicting values for rulesets with same importance, origin, and specificity, applies the last to be declared ruleset.

Don't worry too much about committing the above to memory at the moment. It should become part of your working knowledge, but is not essential for you to memorize at the moment.

Finally, we glossed over the word importance in our definition of the cascadeabove. CSS allows you to supply the keyword !important in order to make a rule that might otherwise be lower in the cascade override others. Here's an example:

p {

color: red !important;

}

*/\* later in the file, or in a different file\*/*

p {

*/\* the above rule marked important*

*has precedence\*/*

color: blue;

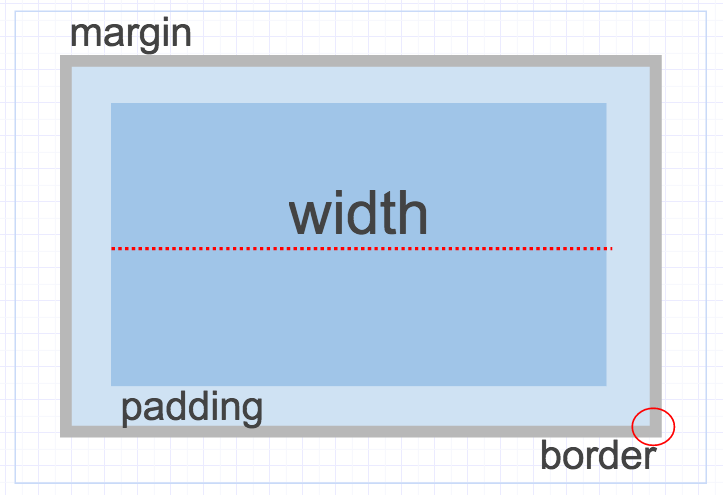
}

You should know about !important, but try to avoid using it in your CSS. There are rare occasions where it's the right move, but usually if you have to use !important, it's a sign that there are problems with the application of your style rules (for instance, you may just need to use a more specific selector).

### The box model

The box model is crucial to understanding CSS, especially when we move on to positioning and layout later in this lesson. The basic idea is that most elements on a web page are really rectangular boxes, even if they appear to be a different shape.

In the default model (box-sizing: content-box), the total space taken up by an element is determined by its width, height, padding, border, and margin, as illustrated below:



To calculate the measurements for an element, we add width, plus 2x padding, plus 2x border, plus 2x margin. div.foo below would have a total width of 200 (width) + 20 (padding left and padding right, each 10) + 2 (left and right border) + 60 (left and right margin) = 282px.

div.foo {

width: 200px;

height: 100px;

padding: 10px;

border: 1px solid green;

margin: 30px;

}

Have a look at the following example, and compute what you think the width of the div.foo and div.bar elements would be, based on the values you see in the CSS (and using the formula we just described above).

If you answered that the total width of div.foo is 222px and div.bar is 120px, then you were right.

Now you might be thinking, "Wait a minute -- div.foo has a directly set width of 200px. Why on earth is its total width greater than its explicitly set width? That's counterintuitive!"

It is counterintuitive that a declaration like width: 200px does not guarantee a 200px-wide element. The content-box model makes us do math to get even numbers when we add padding, border and, and margin to elements, and this can make designing our pages much more complicated than it needs to be.

Enter box-sizing: border-box. With it, we can set width: 200px and be sure that the element will be exactly that wide, padding, border, and margin included. We want to be sure that every element on our page has the same box model, so we use the wildcard (\*), which selects every element in the document at once:

\* {

box-sizing: border-box;

}

you can make border-box the default for all elements.

Here's a CodePen example that demonstrates what happens when we use box-sizing: border-box. Have a look at the computed dimensions of the .foo div in Developer Tools; you'll see that its total width is now the 200px we explicitly set.

If you'd like to read more about box-sizing in CSS, [this CSS Tricks post](https://css-tricks.com/box-sizing/) is an excellent starting point, as is this [Mozilla developer article](https://developer.mozilla.org/en-US/docs/Web/CSS/box-sizing).

### Browser Defaults vs. resetting vs. normalizing CSS

The final topic we want to briefly touch on in this reading is the issue of browser style defaults, and how developers use CSS resets or CSS normalization to ensure a consistent experience across browsers.

As we saw earlier in this unit, your browser has default styling for many elements. For instance, if you use <h1> and <h2> elements on a page but don't add any custom CSS, your browser will still render the <h1> as larger than the <h2>. This is a good thing in that you can view HTML without writing CSS, but it's a bad thing in that you may not want the defaults. Furthermore, browsers differ in their defaults, which means your end users might get a different experience when they're using Chrome than when they're using Firefox.

There are two common solutions to this problem. The first strategy is to use a CSS reset, the [most popular of which is the Meyer reset](http://meyerweb.com/eric/tools/css/reset/reset.css). By linking to this file in your HTML (above your other style sheets, so it sets a base that they add to), you can guarantee cross-browser consistency for default styles.

Another option is [normalize.css](https://necolas.github.io/normalize.css/), which is a CSS library that normalizes a subset of browser elements to be consistent between browsers.

If you're confused on this topic, be sure to ask your mentor about it.

# Target Practice: CSS selector drills

 Estimated Time: **1-2 hours**

To become efficient at writing CSS, you need to memorize a number of CSS selectors. In this assignment, you'll practice eight common selector types. We'll give you the low-down on each selector type, and then you'll complete a drill that asks you to use each.

Here's what we'll cover:

* Element selectors (e.g., p {...})
* Combination selectors (e.g., .foo.bar {...})
* Multi selectors (e.g., .foo, .bar {...})
* Descendant selectors (e.g., .foo li {...})
* Direct child selectors (e.g., .foo > li {...})
* Before and after pseudo-elements selectors (e.g., li::before {...})
* Anchor pseudo-classes (e.g., a:hover {...})
* Attribute selectors (e.g., input[type="text"] {...})

### Element selectors

When you want to create styles for specific element types, element selectors are your friend. Here's an example that sets styling for all paragraph elements:

p {

padding: 30px;

font-size: 16px;

}

The element selector just consists of the element you want to target. Straightforward stuff.

To complete the drill for element selectors below, you'll need to target section elements and h1 elements. Specifically:

* Write one ruleset for sections that gives them a bottom margin of 90px
* Write one ruleset for h1 elements that sets font-family to Helvetica.

### Combinations

Sometimes to implement a design, you'll need to target elements that have two or more classes. Let's say we want to target all elements that have boththe .foo and .bar classes, we could do:

.foo.bar {

*/\* make some rules \*/*

}

And if you needed to target only paragraphs that had both .foo and .bar, you could do:

p.foo.bar {

*/\* make some rules \*/*

}

When you need to target a combination like this, the rule is, if an element is part of the target, put it first. Then, chain together the classes you want to target.

To complete the drill for element selectors below, you'll need to target elements that have both the .box and .green classes. These elements should get a red border.

### Multiple Selectors

Let's say you want to target elements that have either .foo or .bar, or both.bizz and .bang. To target multiple selectors with the same ruleset, just separate your selectors with commas:

.foo, .bar, .bizz.bang {

*/\* make the rules \*/*

}

To complete the drill, you'll need to write a single ruleset that targets elements that either have both the .box and .foo classes, or elements that have both the .circle and .bar classes. These elements should get a solid 2px black border.

You'll know you've targeted correctly when the blue and green boxes and blue and green circles have a black border.

### Descendant and Direct Child Selectors

Sometimes you need to target elements that are children of another element. For instance you might want to target all paragraphs that appear within aside elements:

aside p {

*/\* make rules \*/*

}

This snippet uses a descendant selector, which will target all children of an aside that are paragraphs. Descendant selectors target children of the parent element whether they're immediate children, or further down the hierarchy. In the following code snippet, the first paragraph and the second one (appearing inside div.alert) would be selected by aside p {...}.

<aside>

<p>The quick brown fox jumps over the lazy dog.</p>

<div class='alert'>

<h3>Pay attention!</h3>

<p>Because the quick brown fox jumps over the lazy dog.</p>

</div>

</aside>

Sometimes we need to target only elements that are direct children of an element. For that we can use the direct child selector:

aside > p {

*/\* make rules \*/*

}

This rule would target the first paragraph in the HTML snippet above, but not the paragraph inside div.alert.

To complete the drill below, you'll need to write two rulesets. The first should target all list items that are descendants of an element with the .foo class. The second should target all list items that are direct children of the.barclass.

In the end result, the first two list items on the page ("foo" and "bar") should have 1px solid red borders, but not the second two ("bizz and bang"). In the numbered list, all first level list items should have a 1px solid green border, but the second level ones (that is, the steps for washing fruits and veggies) should not.

## ::before and ::after pseudoelements

The ::before and ::after pseudoelements allow you to render content just before or after your element. This technique is great for creating smart quotes around block quotes (which you'll have to do in the drill below). More broadly, writing ::before and ::after style rules can be a good way to handle repeated visual content that surrounds an element.

Here, we use ul li::before to set a custom icon from FontAwesome for bullet points:

Inspecting the CSS, you can see that we've set the content: "\f121" and font-family: FontAwesome. \f121 is the unicode value for the FontAwesome code icon. We've set its color to green, its font-size to 20px, and given it a right margin of 5px to give the text for each bullet point some room to breath. It's also worth noticing that we've set list-style: none on the parent <ul> element, which turns off the default dots that appear for bullet points.

To complete the drill below, you'll need to:

Create CSS rules that surround <blockquote>s in smart quotes (i.e, curly quotation marks). This means that any time a <blockquote> element appears, without putting quotation characters in your HTML, the block quote should get quotes before and after it. Create a ruleset for the .author class that puts an en dash (–) before it (you can see how we're using it in the HTML in this drill).

### Anchor pseudoclasses

CSS provides four pseudoclasses for anchor elements:

a:link {

*/\* unvisited link \*/*

}

a:visited {

*/\* visited link \*/*

}

a:hover {

*/\* mouse over link \*/*

}

a:active {

*/\* selected link (i.e., you've clicked*

*but not released on the link)*

*\*/*

}

Write style declarations for each anchor tag state, in the CodePen below, following these requirements:

* Unvisited links should be green
* When the user hovers over a link, it should get a larger font-size
* Visited links should be red
* Active links should be black

### Attribute Selectors

CSS lets us target elements by attribute value. Common uses for this include targeting specific kinds of form inputs (for instance, input[type="radio"] to apply rules exclusively to radio selector inputs). We have several options of matching attribute value:

* Exact match: element[attribute=value]
* Match pattern anywhere in value: element[attribute\*=value]
* Match pattern at beginning of value: element[attribute^=value]
* Match pattern at end of value: element[attribute$=value]

To complete the drill below, you'll need to add a selector that targets attribute values. This CodePen contains images of dogs and cats. All dog images have source urls with the word "dog", and all cat images have source urls with the word "cat" in them.

Decide which you like better, dogs or cats. Activate the style rule at the bottom to select all images with source urls that have the losing animal's name. For instance, if you're a dog person, you'll write the selector so that cats get turned upside down.