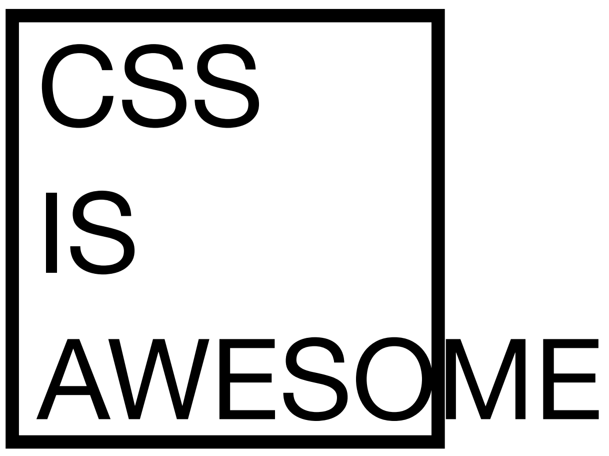
Modern CSS

Sample Chapter: CSS Selectors



Joe Attardi

CSS selectors

One of the core concepts in CSS is that of selectors. A selector determines which element(s) a CSS rule applies to. There are several ways an element can be targeted with a selector, which we will cover in this chapter.

CSS selectors can target multiple elements on the page. That is, a single CSS rule can apply to multiple elements. An element or class selector can select multiple different elements that have that class or element name.

Similarly, a single HTML element can be affected by multiple CSS rules.

Basic selector types

The universal selector

The universal selector, specified simply as an asterisk (*), matches all elements. This can be specified as a single selector, to select all elements in the document, or in combination with combinators (discussed below).

Element selectors

An element selector targets an HTML element by its element or tag name. The syntax of the selector is simply the name of the element.

CSS

```css
p {
  margin: 25px;
}
```

In this example, all p elements on the page will have a margin of 25 pixels applied to them.

ID selectors

An HTML element can have an `id` attribute. As a general rule, there should only be one element with a given `id`. If there are multiple elements with the same `id`, only one of them will be targeted with an ID selector.

An ID selector is specified with the # character followed by the `id` value.

CSS

```css
#header {
  padding: 25px;
}
```

The above selector will target the element with an `id` of header and give it a padding of 25 pixels.

HTML

```
<div id="header">...</div>
```

In general, the use of ID selectors should be avoided. Due to their tight coupling to a specific HTML element, they are not reusable. The CSSLint tool has a good explanation as to why you should avoid ID selectors, which can be read in full here: https://github.com/CSSLint/csslint/wiki/Disallow-ids-in-selectors

The explanation reads, in part:

"For years, developers have treated IDs as the way to say "that thing!" However, IDs have a downside: they are completely unique and therefore cannot be reused. You could potentially style every element on your page with an ID selector but you would lose a lot of the benefit of CSS in the process.

One of CSS's benefits is the ability to reuse style rules in multiple places. When you start out with an ID selector, you're automatically limiting that style to a single element."

Class selectors

While only a single element can be targeted by an ID selector, any number of HTML elements can have the same `class` attribute. A class selector will match every element in the document with the given class. Class selectors are specified with a dot, followed by the name of the class.

CSS

```
.nav-link {
  color: darkcyan;
}
```

This rule will match every element with a class of `nav-link` and give it a color of `darkcyan`.

Attribute selectors

HTML elements can also be selected by their attribute values, or by the presence of an attribute. The attribute is specified inside square brackets, and the attribute selector can take several forms.

```
[attributeName]
```

Selects all elements that have the given attribute, regardless of its value.

```
[attributeName="value"]
```

Selects all elements that have the given attribute, whose value is the string `value`.

```
[attributeName*="value"]
```

Selects all elements that have the given attribute, whose value contains the string `value`.

```
[attributeName~="value"]
```

Selects all elements that have the given attribute, whose value contains the string `value` separated by whitespace.

```
[attributeName^="value"]
```

Selects all elements that have the given attribute, whose value begins with `value`.

```
[attributeName$="value"]
```

Selects all elements that have the given attribute, whose value ends with `value`.

Combining selectors

Any of the above selectors (with the exception of the universal selector) can be used alone or in conjunction with other selectors to make the selector more specific. This is best illustrated with some examples:

```
div.my-class
```

Matches all `div` elements with a class of `my-class`.

```
span.class-one.class-two
```

Matches all span elements with a class of *both* `class-one` and `class-two`.

```
a.nav-link[href*="example.com"]
```

Matches all a elements with a class of `nav-link` that have an `href` attribute that contains the string `example.com`.

A rule can also have multiple selectors separated by a comma:

```
.class-one, .class-two
```

Matches all elements with a class of `class-one` *as well as* all elements with a class of `class-two`.

Selector combinators

There's even more you can do with selectors. Combinators are used to select more specific elements. Combinators are used in conjunction with the basic selectors discussed above. For a given rule, multiple basic selectors can be used, joined by a combinator.

Descendant combinator

The descendant combinator matches an element that is a descendant of the element on the left hand side. *Descendant* means that the element exists somewhere within the child hierarchy - it does not have to be a direct child.

The descendant combinator is specified with a space character.

```
.header div
```

Matches all `div` elements that are direct or indirect children of an element with a class of `header`. If any of these `div`s have children that are also `div`s, those `div`s will also be matched by the selector.

Child combinator

The child combinator matches an element that is a *direct child* of the element on the left hand side. It is specified with a > character.

```
.header > div
```

Matches all `div` elements that are direct children of an element with a class of header. If those `div`s have children that are also `div`s, those divs will *not* be matched by the selector.

General sibling combinator

The general sibling combinator matches an element that is a sibling, but not necessarily an *immediate* sibling, of the element on the left hand side. It is specified with a ~ character.

Consider the following HTML:

HTML

```
<div>
  <div class="header"></div>
  <div class="body"></div>
  <div class="footer"></div>
</div>
```

The following selector would select both of the `div`s, with the class `body` or `footer`:

```
.header ~ div
```

Adjacent sibling combinator

The adjacent sibling combinator is similar to the general sibling combinator, except it only matches elements that are an *immediate* sibling. It is specified with a + character.

Looking again at the above HTML structure, the following selector would *only* select the `div` with the class `body`:

```
.header + div
```

Combining combinators

These combinators can be combined to form even more specific selectors, for example:

```
div.header > div + button
```

This selector matches a `button` element that is an immediate sibling of a `div` element, which in turn is an immediate child of a `div` with the class `header`.

Pseudo-classes

Another tool in the CSS selector toolbox is the pseudo-class. A pseudo-class allows you to select elements based on some special state of the element, in addition to all the selectors discussed above.

Some pseudo-classes let you select elements based on UI state, while others let you select elements based on their position in the document (with more precision than the combinators).

There are many pseudo-classes, but here are some of the more commonly used ones.

UI state

```
:hover
```

Matches an element that the mouse cursor is currently hovering over. This is typically used for buttons and links, but can be applied to any type of element.

```
:active
```

Matches an element that is currently being activated. For buttons and links, this usually means the mouse button has been pressed, but not yet released.

```
:visited
```

Matches a link whose URL has already been visited by the user.

```
:focus
```

Matches an element that currently has the focus. This is typically used for buttons, links, and text fields.

Document structure

```
:first-child
```

Matches an element that is the first child of its parent. Consider the following example HTML:

HTML

```
<ul class="my-list">
  <li>Item one</li>
  <li>Item two</li>
</ul>
```

The selector `.my-list > li:first-child` will match the first list item only.

```
:last-child
```

Matches an element that is the last child of its parent. In the above example, the selector `.my-list > li:last-child` will match the last list item only.

```
:nth-child()
```

Matches an element that is the *n*th child of its parent. The value of *n* is passed as a parameter to the pseudo-class. The index of the first child is 1. Going back to the example HTML above, we can also select "Item two" with the selector `.my-list > li:nth-child(2)`.

This pseudo-class can even select children at a certain interval. For example, in a longer list, we could select every other list item with the selector `.my-list > li:nth-child(2n)`. Or, we could select every four items with the selector `.my-list > li:nth-child(4n)`. We can even select all odd-numbered children with the selector `.my-list > li:nth-child(odd)`.

A complete list of pseudo-classes, including some that are still experimental, can be found at https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes.

Pseudo-elements

A pseudo-element lets you select only part of a matched element. With modern browsers, pseudo-elements are specified with a double colon (`::`) followed by the pseudo-element name.

It should be noted that some pseudo-elements only work on *block* elements. We will discuss block vs. inline elements a little later.

```
::first-line
```

Applies the styles only to the first line of an element.

```
::first-letter
```

Applies the styles only to the first letter of the first line of an element.

`::before` and `::after`

Two special pseudo-elements are `::before` and `::after`. These pseudo-elements don't select part of the element; rather, they actually create a new element as either the first child or the last child of the matched element.

These are typically used to add effects to elements.

Suppose we want to add an indicator next to all external links on our website. We can tag these external links using a class, say, `external-link`.

We can specify an external link as follows:

HTML

```html
<a class="external-link" href="https://google.com">Google</a>
```

Then we can add the indicator with the following CSS rule:

CSS

```css
.external-link::after {
  content: " (external)"
  color: green;
}
```

The result looks like this:

Google (external)

The `::after` pseudo-element added the content "(external)" and made it green.

You can find a full list of pseudo-elements (including those that are experimental) here:

https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements

Specificity

CSS

```css
.profile {
  background-color: green;
}

div {
  background-color: red;
}
```

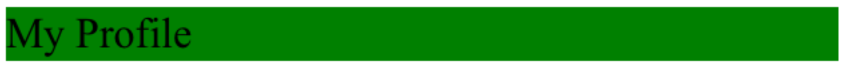
HTML

```html
<div class="profile">My Profile</div>
```

We have a conflict. Our HTML element matches both selectors - it is indeed a `div` element, and it also has the `profile` class. Each rule specifies a different `background-color`. So which background color will this `div` element have?

You might think that since the element selector rule comes after the class selector rule, the element selector rule will override the previous rule. This is how it usually works in programming, right?

Let's try it out.


My Profile

Looks like the class selector rule was applied. Why is that? The answer is *specificity*. When there is a conflict of CSS rules, the rule with the *most specific selector* will be chosen. In this case, a class selector is more specific than an element selector. The element selector applies to all `div` elements in the document, whereas the class selector applies only to elements with a class of `profile`.

Specificity rankings

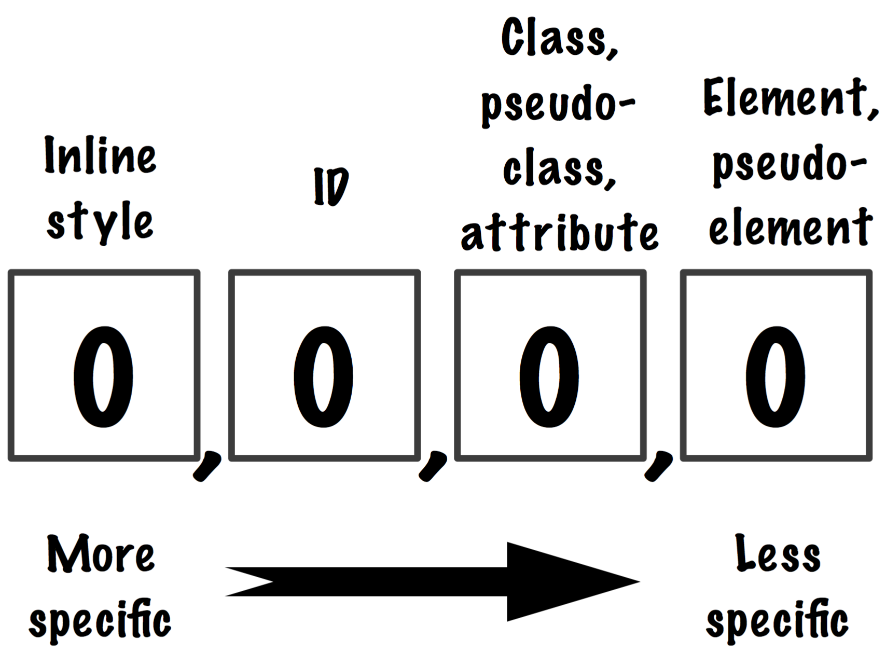
The specificity rankings of CSS rules is as follows, from most specific to least specific:

1. Inline styles via a `style` attribute
2. ID selectors
3. Class selectors, attribute selectors, and pseudo-classes
4. Element selectors and pseudo-elements

Neither the universal selector nor combinators factor into specificity.

Calculating specificity

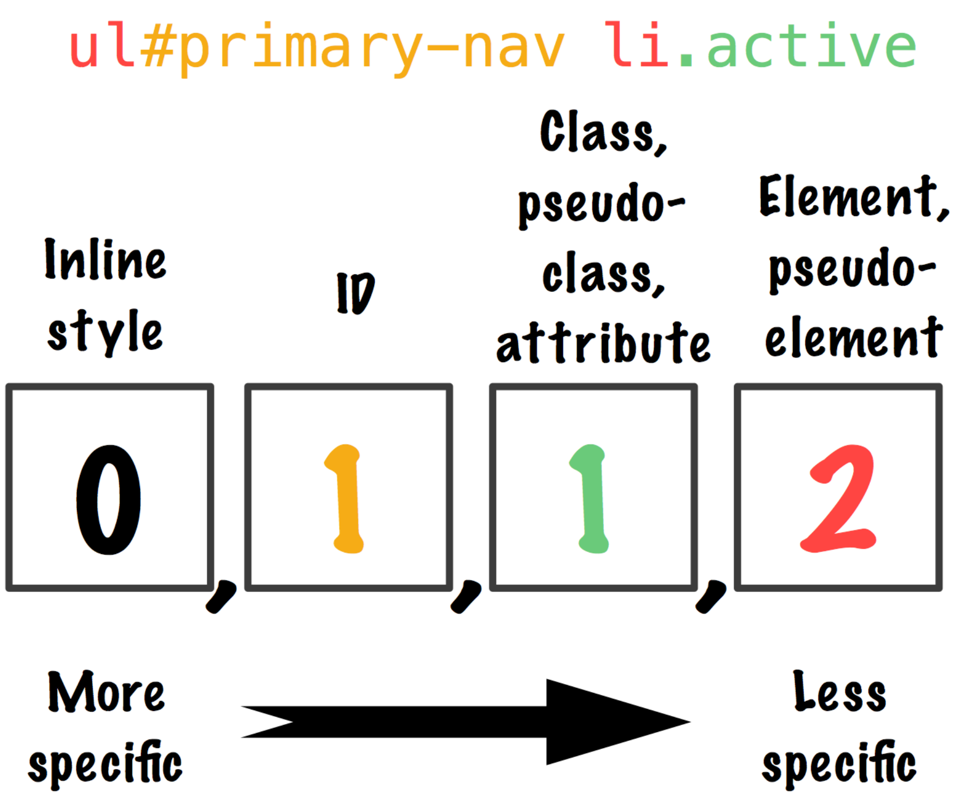
There is a general algorithm for calculating a CSS rule's specificity. To calculate the specificity of a CSS rule, imagine 4 boxes, one for each type of style rule in the list above. Initially, each box has a zero in it.



If the element has an inline style, add a 1 to the first box. In this case, the inline style automatically wins.

For each ID in the selector, add 1 to the value in the second box. For each class, pseudo-class, or attribute in the selector, add 1 to the value in the third box. Finally, for each element or pseudo-element in the selector, add 1 to the value in the last box.

Let's look at an example, the selector `ul#primary-nav li.active`. This would result in the following scores:



You can think of this value like the number 112. This score is calculated for each of the rule selectors, and the one with the highest value wins. In most cases you can think of this as such a base-10 number.

If multiple conflicting rules are calculated to have the same specificity, whatever rule appears last in the file wins.

It is important to note that specificity rules only apply to the *conflicting properties* in multiple CSS rules.

CSS

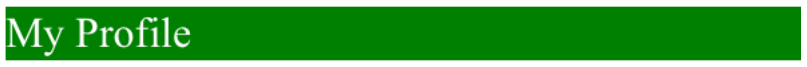
```css
.profile {
  background-color: green;
}

div {
  background-color: red;
  color: white;
}
```

HTML

```html
<div class="profile">My Profile</div>
```

The rendered output will look like this:



Notice that the element got the green background from the class selector, but also got the white color from the element selector. When there are multiple CSS rules that match a single element, the union of all the CSS properties is taken and applied to the element. If there are multiple values for a given property, it is then that the specificity rules will be applied to determine the final value of that property.

The escape hatch: `!important`

A CSS property can have the string `!important` after it. This will override any calculated specificity. A style rule with `!important` always wins over anything else.

This is generally considered a bad practice, though. In most cases, it's a better idea to figure out the specificity of the rules you're trying to apply, and make a rule more specific to make its styles apply.

Now that we've looked in detail at how to select elements, let's start to explore how to style them. The next step is to look at some of the basic building blocks of CSS rules.