Chapter 17: Cascading and Specificity

Section 17.1: Calculating Selector Specificity

Each individual CSS Selector has its own specificity value. Every selector in a sequence increases the sequence's overall specificity. Selectors fall into one of three different specificity groups: *A*, *B* and *c*. When multiple selector sequences select a given element, the browser uses the styles applied by the sequence with the highest overall specificity.

Group	Comprised of	Examples
A	id selectors	`#foo`
B	class selectors	`.bar`
	attribute selectors	`[title]`, `[colspan="2"]`
	pseudo-classes	`:hover`, `:nth-child(2)`
c	type selectors	`div`, `li`
	pseudo-elements	`::before`, `::first-letter`

Group *A* is the most specific, followed by Group *B*, then finally Group *c*.

The universal selector (`*`) and combinators (like `>` and `~`) have no specificity.

Example 1: Specificity of various selector sequences

```
#foo #baz {}        /* a=2, b=0, c=0 */

#foo.bar {}         /* a=1, b=1, c=0 */

#foo {}             /* a=1, b=0, c=0 */

.bar:hover {}       /* a=0, b=2, c=0 */

div.bar {}          /* a=0, b=1, c=1 */

:hover {}           /* a=0, b=1, c=0 */

[title]             /* a=0, b=1, c=0 */

.bar {}             /* a=0, b=1, c=0 */

div ul + li {}      /* a=0, b=0, c=3 */

p::after {}         /* a=0, b=0, c=2 */

*::before {}        /* a=0, b=0, c=1 */

::before {}         /* a=0, b=0, c=1 */

div {}              /* a=0, b=0, c=1 */

*                   /* a=0, b=0, c=0 */
```

Example 2: How specificity is used by the browser

Imagine the following CSS implementation:

```
#foo {
  color: blue;
}
```

```
.bar {
  color: red;
  background: black;
}
```

Here we have an ID selector which declares `color` as *blue*, and a class selector which declares `color` as *red* and `background` as *black*.

An element with an ID of `#foo` and a class of `.bar` will be selected by both declarations. ID selectors have a Group *A* specificity and class selectors have a Group *B* specificity. An ID selector outweighs any number of class selectors. Because of this, `color:blue;` from the `#foo` selector and the `background:black;` from the `.bar` selector will be applied to the element. The higher specificity of the ID selector will cause the browser to ignore the `.bar` selector's `color` declaration.

Now imagine a different CSS implementation:

```
.bar {
  color: red;
  background: black;
}

.baz {
  background: white;
}
```

Here we have two class selectors; one of which declares `color` as *red* and `background` as *black*, and the other declares `background` as *white*.

An element with both the `.bar` and `.baz` classes will be affected by both of these declarations, however the problem we have now is that both `.bar` and `.baz` have an identical Group *B* specificity. The cascading nature of CSS resolves this for us: as `.baz` is defined *after* `.bar`, our element ends up with the *red* `color` from `.bar` but the *white* `background` from `.baz`.

Example 3: How to manipulate specificity

The last snippet from Example 2 above can be manipulated to ensure our `.bar` class selector's `color` declaration is used instead of that of the `.baz` class selector.

```
.bar {}          /* a=0, b=1, c=0 */
.baz {}          /* a=0, b=1, c=0 */
```

The most common way to achieve this would be to find out what other selectors can be applied to the `.bar` selector sequence. For example, if the `.bar` class was only ever applied to `span` elements, we could modify the `.bar` selector to `span.bar`. This would give it a new Group *C* specificity, which would override the `.baz` selector's lack thereof:

```
span.bar {}      /* a=0, b=1, c=1 */
.baz {}          /* a=0, b=1, c=0 */
```

However it may not always possible to find another common selector which is shared between any element which uses the `.bar` class. Because of this, CSS allows us to duplicate selectors to increase specificity. Instead of just `.bar`, we can use `.bar.bar` instead (See The grammar of Selectors, W3C Recommendation). This still selects any element with a class of `.bar`, but now has double the Group *B* specificity:

```
.bar.bar {}      /* a=0, b=2, c=0 */
.baz {}          /* a=0, b=1, c=0 */
```

`!important` and inline style declarations

The `!important` flag on a style declaration and styles declared by the HTML `style` attribute are considered to have a greater specificity than any selector. If these exist, the style declaration they affect will overrule other declarations regardless of their specificity. That is, unless you have more than one declaration that contains an `!important` flag for the same property that apply to the same element. Then, normal specificity rules will apply to those properties in reference to each other.

Because they completely override specificity, the use of `!important` is frowned upon in most use cases. One should use it as little as possible. To keep CSS code efficient and maintainable in the long run, it's almost always better to increase the specificity of the surrounding selector than to use `!important`.

One of those rare exceptions where `!important` is not frowned upon, is when implementing generic helper classes like a `.hidden` or `.background-yellow` class that are supposed to always override one or more properties wherever they are encountered. And even then, you need to know what you're doing. The last thing you want, when writing maintainable CSS, is to have `!important` flags throughout your CSS.

A final note

A common misconception about CSS specificity is that the Group *A*, *B* and *c* values should be combined with each other (a=1, b=5, c=1 => 151). This is **not** the case. If this were the case, having 20 of a Group *B* or *c* selector would be enough to override a single Group *A* or *B* selector respectively. The three groups should be regarded as individual levels of specificity. Specificity cannot be represented by a single value.

When creating your CSS style sheet, you should maintain the lowest specificity as possible. If you need to make the specificity a little higher to overwrite another method, make it higher but as low as possible to make it higher. You shouldn't need to have a selector like this:

```
body.page header.container nav div#main-nav li a {}
```

This makes future changes harder and pollutes that css page.

You can calculate the specificity of your selector here

Section 17.2: The !important declaration

The `!important` declaration is used to override the usual specificity in a style sheet by giving a higher priority to a rule. Its usage is: `property : value !important;`

```
#mydiv {
    font-weight: bold !important;          /* This property won't be overridden
                                              by the rule below */
}

#outerdiv #mydiv {
    font-weight: normal;                   /* #mydiv font-weight won't be set to normal
                                              even if it has a higher specificity because of
                                              the !important declaration above */
}
```

Avoiding the usage of `!important` is strongly recommended (unless absolutely necessary), because it will disturb the natural flow of css rules which can bring uncertainty in your style sheet. Also it is important to note that when multiple `!important` declarations are applied to the same rule on a certain element, the one with the higher specificity will be the ona applied.

Here are some examples where using `!important` declaration can be justified:

- If your rules shouldn't be overridden by any inline style of the element which is written inside `style` attribute of the html element.
- To give the user more control over the web accessibility, like increasing or decreasing size of the font-size, by overriding the author style using `!important`.
- For testing and debugging using inspect element.

See also:

- **W3C - 6 Assigning property values, Cascading, and Inheritance -- 6.4.2 !important rules**

Section 17.3: Cascading

Cascading and specificity are used together to determine the final value of a CSS styling property. They also define the mechanisms for resolving conflicts in CSS rule sets.

CSS Loading order

Styles are read from the following sources, in this order:

1. User Agent stylesheet (The styles supplied by the browser vendor)
2. User stylesheet (The additional styling a user has set on his/her browser)
3. Author stylesheet (Author here means the creator of the webpage/website)
    - Maybe one or more `.css` files
    - In the `<style>` element of the HTML document
4. Inline styles (In the `style` attribute on an HTML element)

The browser will lookup the corresponding style(s) when rendering an element.

How are conflicts resolved?

When only one CSS rule set is trying to set a style for an element, then there is no conflict, and that rule set is used.

When multiple rule sets are found with conflicting settings, first the Specificty rules, and then the Cascading rules are used to determine what style to use.

Example 1 - Specificity rules

```
.mystyle { color: blue; }     /* specificity: 0, 0, 1, 0 */
div { color: red; }           /* specificity: 0, 0, 0, 1 */
```

```
<div class="mystyle">Hello World</div>
```

What color will the text be? (hover to see the answer)

> blue

First the specificity rules are applied, and the one with the highest specificity "wins".

Example 2 - Cascade rules with identical selectors

External css file

```
.class {
```

```
  background: #FFF;
}
```

Internal css (in HTML file)

```
<style>
.class { background:
  #000;
}
<style>
```

In this case, where you have identical selectors, the cascade kicks in, and determines that the last one loaded "wins".

Example 3 - Cascade rules after Specificity rules

```
body > .mystyle { background-color: blue; }      /* specificity: 0, 0, 1, 1 */
.otherstyle > div { background-color: red; }     /* specificity: 0, 0, 1, 1 */
```

```
<body class="otherstyle">
  <div class="mystyle">Hello World</div>
</body>
```

What color will the background be?

> red

After applying the specificity rules, there's still a conflict between blue and red, so the cascading rules are applied on top of the specificity rules. Cascading looks at the load order of the rules, whether inside the same `.css` file or in the collection of style sources. The last one loaded overrides any earlier ones. In this case, the `.otherstyle > div` rule "wins".

A final note

- Selector specificity always take precedence.
- Stylesheet order break ties.
- Inline styles trump everything.

Section 17.4: More complex specificity example

```
div {
    font-size: 7px;
    border: 3px dotted pink;
    background-color: yellow;
    color: purple;
}

body.mystyle > div.myotherstyle {
    font-size: 11px;
    background-color: green;
}

#elmnt1 {
    font-size: 24px;
    border-color: red;
}
```

```
.mystyle .myotherstyle {
    font-size: 16px;
    background-color: black;
    color: red;
}
```

```
<body class="mystyle">
    <div id="elmnt1" class="myotherstyle"> Hello,
        world!
    </div>
</body>
```

What borders, colors, and font-sizes will the text be?

font-size:

> `font-size: 24;`, since `#elmnt1` rule set has the highest specificity for the `<div>` in question, every property here is set.

border:

> `border: 3px dotted red;`. The border-color `red` is taken from `#elmnt1` rule set, since it has the highest specificity. The other properties of the border, border-thickness, and border-style are from the `div` rule set.

background-color:

> `background-color: green;`. The `background-color` is set in the `div`, `body.mystyle > div.myotherstyle`, and `.mystyle .myotherstyle` rule sets. The specificities are (0, 0, 1) vs. (0, 2, 2) vs. (0, 2, 0), so the middle one "wins".

color:

> `color: red;`. The color is set in both the `div` and `.mystyle .myotherstyle` rule sets. The latter has the higher specificity of (0, 2, 0) and "wins".