Cascade and inheritance

Previous Overview: Introduction to CSS Next

In a previous article, we got into the various CSS selectors. At some point in your work, you'll find yourself in the situation where multiple CSS rules will have selectors matching the same element. In such cases, which CSS rule "wins", and ends up being the one that is finally applied to the element? This is controlled by a mechanism called the Cascade; this is also related to inheritance (elements will take some property values from their parents, but not others). In this article we will define what the CSS cascade is, what specificity is, what importance is, and how properties inherit from different rules.

Prerequisites: Basic computer literacy, basic software installed, basic

	knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of How CSS works (study the previous articles in this module.)
Objective:	To learn about the cascade and specificity, and how inheritance works in CSS.

The final style for an element can be specified in many different places, which can interact in complex ways. This complex interaction makes CSS powerful, but it can also make it confusing and difficult to debug. This article aims to clear up some of that complexity; if you don't understand it immediately, don't worry — this is one of the hardest parts of CSS theory to comprehend. You are advised to give it a try now, but then keep it nearby as a handy guide to return to when questions about the cascade and inheritance come up.

The cascade

CSS is an abbreviation for *Cascading Style Sheets*, which indicates that the notion of the cascade is important. At its most basic level, it indicates that the order of CSS rules matter, but it's more complex than that. What selectors win out in the cascade depends on three factors (these are listed in order of weight — earlier ones will overrule later ones):

1. Importance
2. Specificity
3. Source order

Importance

In CSS, there is a special piece of syntax you can use to make sure that a certain declaration will *always* win over all others: `!important`.

Let's look at an example:

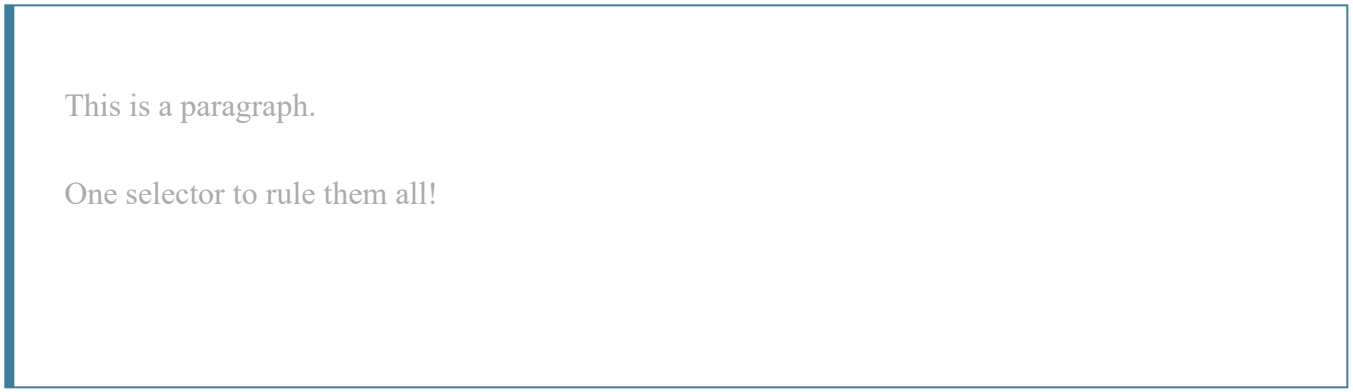
```
<p class="better">This is a paragraph.</p>
<p class="better" id="winning">One selector to rule them all!</p>
```

```
#winning {
  background-color: red;
  border: 1px solid black;
}

.better {
  background-color: gray;
  border: none !important;
}

p {
  background-color: blue;
  color: white;
  padding: 5px;
}
```

This produces the following:



Let's walk through this to see what's happening:

1. You'll see that the third rule's `color` and `padding` values have been applied, but the `background-color` hasn't. Why? Really all three should surely apply, because rules later in the source order generally override earlier rules.
2. However, The rules above it win, because IDs/class selectors have higher specificity than element selectors (you'll learn more about this in the next section).
3. Both elements have a `class` of `better`, but the 2nd element also has an `id` of `winning`. Since IDs have an *even higher* specificity than classes (you can only have one element with each unique ID on a page, but many elements with the same class — ID selectors are *very specific* in what they target), the red background color and the 1 pixel black border should both be applied to the 2nd element, with the first element getting the gray background color, and no border, as specified by the class.
4. The 2nd element *does* get the red background color, but no border. Why? Because of the `!important` declaration in the second rule — including this after `border: none` means that this declaration will win over the border value in the previous rule, even though the ID has higher specificity.

> **Note**: The only way to override this `!important` declaration would be to include another `!important` declaration of the *same specificity* later in the source order, or one with a higher specificity.

It is useful to know that `!important` exists so that you know what it is when you come across it in other people's code. **However**, we strongly recommend that you never use it unless you absolutely have to. One situation in which you may have to use it is when you are working on a CMS where you can't edit the core CSS modules, and you really want to override a style that can't be overridden in any other way. But really, don't use it if you can avoid it, because `!important` changes the way the cascade normally works, so it can make debugging CSS problems really hard to work out, especially in a large stylesheet.

It is also useful to note that the importance of a CSS declaration depends on what stylesheet it is specified in — it is possible for users to set custom stylesheets to override the developer's styles, for example the user might be visually impaired, and want to set the font size on all web pages they visit to be double the normal size to allow for easier reading.

Conflicting declarations will be applied in the following order, with later ones overriding earlier ones:

1. Declarations in user agent style sheets (e.g. the browser's default styles, used when no other styling is set).
2. Normal declarations in user style sheets (custom styles set by a user).
3. Normal declarations in author style sheets (these are the styles set by us, the web developers).
4. Important declarations in author style sheets
5. Important declarations in user style sheets

It makes sense for web developers' stylesheets to override user stylesheets, so the design can be kept as intended, but sometimes users have good reasons to override web developer styles, as mentioned above — this can be achieved by using `!important` in their rules.

Specificity

Specificity is basically a measure of how specific a selector is — how many elements it *could* match. As shown in the example seen above, element selectors have low specificity. Class selectors have a higher specificity, so will win against element selectors. ID selectors have an even higher specificity, so will win against class selectors. A sure way to win against an ID selector is to use `!important`.

The amount of specificity a selector has is measured using four different values (or components), which can be thought of as thousands, hundreds, tens and ones — four single digits in four columns:

1. Thousands: Score one in this column if the declaration is inside a `style` attribute, aka inline styles. Such declarations don't have selectors, so their specificity is always simply 1000.
2. Hundreds: Score one in this column for each ID selector contained inside the overall selector.
3. Tens: Score one in this column for each class selector, attribute selector, or pseudo-class contained inside the overall selector.
4. Ones: Score one in this column for each element selector or pseudo-element contained inside the overall selector.

> **Note**: Universal selector (`*`), combinators (`+`, `>`, `~`, `' '`) and negation pseudo-class (`:not`) have no effect on specificity.

The following table shows a few isolated examples to get you in the mood. Try going through these, and making sure you understand why they have the specificity that we have given them.

Selector	Thousands	Hundreds	Tens	Ones	Total specificity
`h1`	0	0	0	1	0001
`h1 + p::first-letter`	0	0	0	3	0003
`li > a[href*="en-US"] > .inline-warning`	0	0	2	2	0022
`#identifier`	0	1	0	0	0100
No selector, with a rule inside an element's `style` attribute	1	0	0	0	1000

Note: If multiple selectors have the same importance *and* specificity, which selector wins is decided by which comes later in the Source order.

Before we move on, let's look at an example in action. Here is the HTML we are going to use:

```
<div id="outer" class="container">
  <div id="inner" class="container">
    <ul>
      <li class="nav"><a href="#">One</a></li>
      <li class="nav"><a href="#">Two</a></li>
    </ul>
  </div>
</div>
```

And here is the CSS for the example:

```
/* specificity: 0101 */
#outer a {
  background-color: red;
}

/* specificity: 0201 */
#outer #inner a {
  background-color: blue;
}

/* specificity: 0104 */
#outer div ul li a {
  color: yellow;
}

/* specificity: 0113 */
#outer div ul .nav a {
  color: white;
}

/* specificity: 0024 */
div div li:nth-child(2) a:hover {
  border: 10px solid black;
}

/* specificity: 0023 */
div li:nth-child(2) a:hover {
  border: 10px dashed black;
}

```

```
/* specificity: 0033 */
div div .nav:nth-child(2) a:hover {
  border: 10px double black;
}

a {
  display: inline-block;
  line-height: 40px;
  font-size: 20px;
  text-decoration: none;
  text-align: center;
  width: 200px;
  margin-bottom: 10px;
}

ul {
  padding: 0;
}

li {
  list-style-type: none;
}
```

The result we get from this code is as follows:

One

Two

So what's going on here? First of all, we are only interested in the first seven rules of this example, and as you'll notice, we have included their specificity values in a comment before each one.

- The first two selectors are competing over the styling of the link's background color — the second one wins and makes the background color blue because it has an extra ID selector in the chain: its specificity is 201 vs. 101.
- The third and fourth selectors are competing over the styling of the link's text color — the second one wins and makes the text white because although it has one less element selector, the missing selector is swapped out for a class selector, which is worth ten rather than one. So the winning specificity is 113 vs. 104.
- Selectors 5–7 are competing over the styling of the link's border when hovered. Selector six clearly loses to five with a specificity of 23 vs. 24 — it has one fewer element selectors in the chain. Selector seven, however, beats both five and six — it has the same number of sub-selectors in the chain as five, but an element has been swapped out for a class selector. So the winning specificity is 33 vs. 23 and 24.

> **Note**: If you haven't already, review all the selectors one more time, just to make sure you understand why the specificity values have been awarded as shown.

Source order

As mentioned above, if multiple competing selectors have the same importance *and* specificity, the third factor that comes into play to help decide which rule wins is source order — later rules will win over earlier rules. For example:

```
p {
  color: blue;
}

/* This rule will win over the first one */
p {
  color: red;
}
```

Whereas in this example the first rule wins because source order is overruled by specificity:

```
/* This rule will win */
.footnote {
  color: blue;
}

p {
  color: red;
}
```

A note on rule mixing

One thing you should bear in mind when considering all this cascade theory, and what styles get applied over other styles, is that all this happens at the property level — properties override other properties, but you don't get entire rules overriding other rules. When several CSS rules match the same element, they are all applied to that element. Only after that are any conflicting properties evaluated to see which individual styles will win over others.

Let's see an example. First, some HTML:

```
<p>I'm <strong>important</strong></p>
```

And now some CSS to style it with:

```
/* specificity: 0002 */
p strong {
  background-color: khaki;
  color: green;
}

/* specificity: 0001 */
strong {
  text-decoration: underline;
  color: red;
}
```

Result:

> I'm **important**

In this example, because of its specificity, the first rule's `color` property overrides the color property of the second rule. However, both the `background-color` from the first rule and the `text-decoration` from the second rule are applied to the `<strong>` element. You'll also notice that the text of that element is bolded: this comes from the browsers' default stylesheet.

Inheritance

CSS inheritance is the last piece we need to investigate to get all the information and understand what style is applied to an element. The idea is that some property values applied to an element will be inherited by that element's children, and some won't.

- For example, it makes sense for `font-family` and `color` to be inherited, as that makes it easy for you to set a site-wide base font by applying a font-family to the `<html>` element; you can then override the fonts on individual elements where needed. It would be really annoying to have to set the base font separately on *every* element.
- As another example, it makes sense for `margin`, `padding`, `border`, and `background-image` to NOT be inherited. Imagine the styling/layout mess that would occur if you set these properties on a container element and had them inherited by every single child element, and then had to *unset* them all on each individual element!

Which properties are inherited by default and which aren't is largely down to common sense. If you want to be sure, however, you can consult the CSS Reference — each separate property page contains a summary table including various details about that element, including whether it is inherited or not.

Controlling inheritance

CSS provides four special universal property values for specifying inheritance:

`inherit`
: Sets the property value applied to a selected element to be the same as that of its parent element.

`initial`
: Sets the property value applied to a selected element to be the same as the value set for that element in the browser's default style sheet. If no value is set by the browser's default style sheet and the property is naturally inherited, then the property value is set to `inherit` instead.

`unset`
: Resets the property to its natural value, which means that if the property is naturally inherited it acts like `inherit`, otherwise it acts like `initial`.

`revert`
: Reverts the property to the value it would have had if the current origin had not applied any styles to it. In other words, the property's value is set to the user stylesheet's value for the property (if one is set), otherwise, the property's value is taken from the user agent's default stylesheet.

See Origin of CSS declarations in Introducing the CSS Cascade for more information on each of these and how they work.

> **Note**: `initial` and `unset` are not supported in Internet Explorer.

Of these, `inherit` is frequently the most interesting — it allows us to explicitly make an element inherit a property value from its parent.

Let's take a look at an example. First some HTML:

```
<ul>
  <li>Default <a href="#">link</a> color</li>
  <li class="my-class-1">Inherit the <a href="#">link</a> color</li>
  <li class="my-class-2">Reset the <a href="#">link</a> color</li>
  <li class="my-class-3">Unset the <a href="#">link</a> color</li>
</ul>
```

Now some CSS for styling:

```
body {
  color: green;
}

.my-class-1 a {
  color: inherit;
}

.my-class-2 a {
```

```
  color: initial;
}

.my-class-3 a {
  color: unset;
}
```

Result:

- Default link color
- Inherit the link color
- Reset the link color
- Unset the link color

Let's explain what's going on here:

- We first set the `color` of the `<body>` to green.
- As the `color` property is naturally inherited, all child elements of body will have the same green color. It's worth noting that browsers set the color of links to blue by default instead of allowing the natural inheritance of the color property, so the first link in our list is blue.
- The second rule sets links within an element with the class `my-class-1` to inherit its color from its parent. In this case, it means that the link inherits its color from its `<li>` parent, which, by default inherits its color from its own `<ul>` parent, which ultimately inherits its color from the `<body>` element, which had its `color` set to `green` by the first rule.

- The third rule selects any links within an element with the class `my-class-2` and sets their color to `initial`. Usually, the initial value set by browsers for the text color is black, so this link is set to black.
- The last rule selects all links within an element with the class `my-class-3` and sets their color to `unset` — we unset the value. Because the color property is a naturally inherited property it acts exactly like setting the value to `inherit`. As a consequence, this link is set to the same color as the body — green.

Resetting all property values

The CSS shorthand property `all` can be used to apply one of these inheritance values to (almost) all properties at once. Its value can be any one of the inheritance values (`inherit`, `initial`, `unset`, or `revert`). It's a convenient way to undo changes made to styles so that you can get back to a known starting point before beginning new changes.

Active learning: playing with the cascade

In this active learning, we'd like you to experiment with writing a single new rule that will override the color and background color that we've applied to the links by default. Can you use one of the special values we looked at in the Controlling inheritance section to write a declaration in a new rule that will reset the background color back to white, without using an actual color value?

If you make a mistake, you can always reset it using the *Reset* button. If you get really stuck, press the *Show solution* button to see a potential answer.

HTML Input

```
<div id="outer" class="container">
  <div id="inner" class="container">
    <ul>
      <li class="nav"><a href="#">One</a></li>
      <li class="nav"><a href="#">Two</a></li>
    </ul>
  </div>
</div>
```

CSS Input

```
#outer div ul .nav a {
  background-color: blue;
  padding: 5px;
  display: inline-block;
  margin-bottom: 10px;
}

div div li a {
  color: yellow;
}
```

Output

- One
- Two

Reset Show solution

What's next

If you understood most of this article, then well done — you've started getting familiar with the fundamental mechanics of CSS. The last bit of central theory is the box model, which we'll cover next.

If you didn't fully understand the cascade, specificity, and inheritance, then don't worry! This is definitely the most complicated thing we've covered so far in the course, and is something that even professional web developers sometimes find tricky. We'd advise that you return to this article a few times as you continue through the course, and keep thinking about it. Refer back to here if you start to come across strange issues with styles not applying as expected. It could be a specificity issue.

Previous Overview: Introduction to CSS Next

In this module

- How CSS works
- CSS syntax
- Selectors
- Simple selectors
- Attribute selectors
- Pseudo-classes and pseudo-elements
- Combinators and multiple selectors
- CSS values and units
- Cascade and inheritance
- The box model
- Debugging CSS
- Fundamental CSS comprehension