

Contents

Cascade Rules	1
CSS Rule Sorting	1
Origin	2
Weight Beats Specificity	3
Styles That Stick	6
Inheritance	6
Initial Value	7
Rule Evaluation	7
Take a Deep Breath	8
Specificity Details	8
Calculating Specificity	8
Media	10
Print Media	10
Debugging Print Styles	10
Resources	11

Cascade Rules

This chapter will discuss how the browser determines which style rule(s) to apply to a particular DOM element. The details for how this is done are in the [specification](#) on the Cascade.

CSS Rule Sorting

Before we dive in to the details, let's attempt to get a broad overview of the four-step process browsers use to determine which rules to apply for a given element:

1. **Gather Qualified Declarations:** First gather all declarations that apply to the element for the current medium (example mediums might be: *screen*, *print*, *tv*, *braille*, etc. See [the media types spec](#) for all available media types). Pass all matches through to the next part of the sorting algorithm
2. **Sort by Origin:** Next, we sort by stylesheet *origin* in ascending order of precedence provided by [the spec](#) as follows:

- user agent declarations
- user normal declarations
- author normal declarations
- author important declarations
- user important declarations (intuitively we can see that, in order to provide proper accessibility, user's with special requirements need a means of having "the final say" for things like increased text size)

*The term **weight** is used to describe the priority assigned to competing style rules using the above considerations. Style rules with a higher **weight** will take precedence over those with a lower **weight**.*

3. **Sort by Specificity:** If (and only if) the above sort results in rules that have the same importance and origin, than sort those rules by selector specificity
4. **Sort by Order of Declaration:** If (and only if) the competing declarations have the same weight and specificity, than sort by order specified (where the latter specified rule wins).

The reason we're describing the sort ordering process first, is that it's important to understand the sequence in which weight, specificity, and declaration order are evaluated. First *weight*, then *specificity*, and finally *order of declaration*.

There's actually another dimension to this. If we have elements that don't have any styles applied directly to them, there's still the case that those elements may "inherit" styles from their ancestors. Further, style properties have initial default values to consider. We'll cover those details in a bit.

Now that we've discussed the browser's overall strategy for sorting style rules, let's look at **origin**, **specificity**, and **inheritance** in more detail.

Origin

There are three **types** of stylesheets that a browser must consider when sorting in ascending orders of precedence:

1. User Agent stylesheet

The user agent stylesheet sets up a browser's default styles. Note that each browser has its own stylesheet so Chrome's default style will be a bit different than say Firefox's.

2. User stylesheet

User's can define their own stylesheets to override the browser's default stylesheet. As such, the user stylesheet is given more **weight** than the browser's stylesheet.

3. Author stylesheet

Page authors get the highest precedence and their stylesheets will override both the user agent and user stylesheets (with the earlier caveat that any rule with **!important** from the user stylesheet will override all other rules providing for accessibility concerns).

The type of stylesheet that is being considered—user agent, user, or author—determines the stylesheet **origin**.

Weight Beats Specificity

As noted from the sorting process described earlier, the browser will prioritize rules with more weight before considering specificity. This is rarely an issue to page authors since an author stylesheet generally gets the most weight anyway (ignoring, of course, the user stylesheet **!important** special case). However, it's worth noting that it's possible for specificity to be completely ignored when origin and/or importance are not the same. Consider a page that includes a simple author stylesheet like so:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset=utf-8>
<title>Cascade Example</title>
<link rel="stylesheet" type="text/css" href="author.css">
</head>
<body>
  <h1 id="obnoxious">I'm an h1</h1>
</body>
</html>

/* AUTHOR STYLESHEET */
h1 {
  font-size: 72px;
  font-family: "Arial Rounded MT Bold", "Arial Narrow Bold", sans-serif;
  font-weight: normal;
  text-transform: uppercase;
  color: purple;
}
```

Now consider a user stylesheet that is added as a preference:

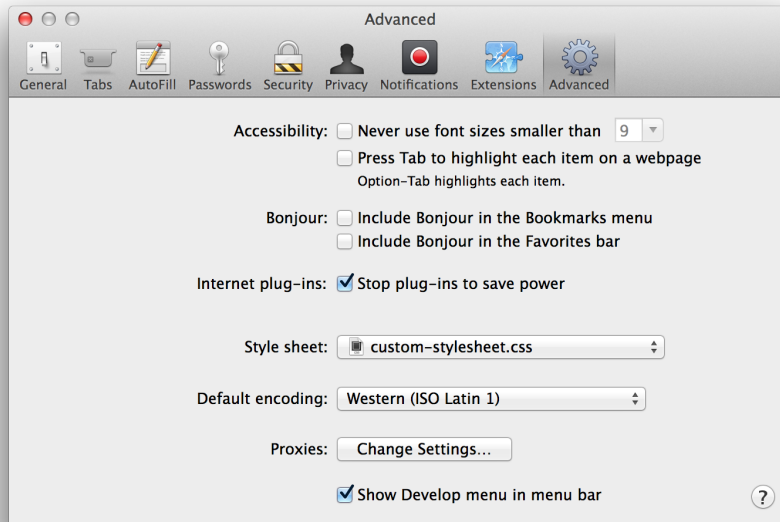
```

/* USER STYLESHEET */
#obnoxious {
  color: green;
  font-style: italic;
  font-size: 24px;
}

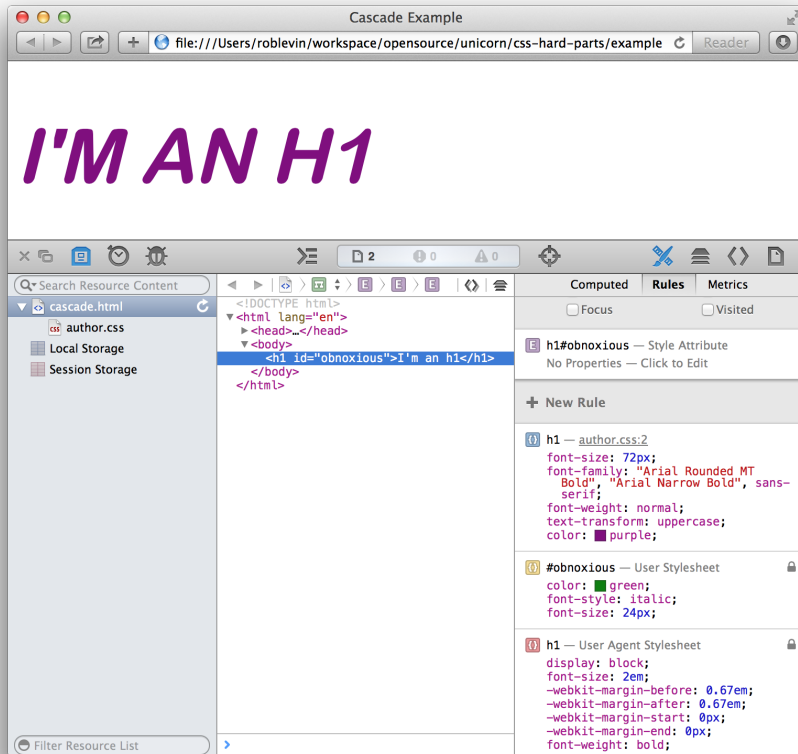
```

We haven't yet discussed the rules for specificity, but an ID selector (like `#obnoxious`), is much more specific than an element selector (like the `h1` defined in the author stylesheet above). Intuitively, you may think that the `#obnoxious` rule would “win the battle” since it's more specific. However, in fact, the less specific `h1` rule from the author stylesheet will get used since the sorting process prefers origin over specificity. Let's take a look at this example a bit more visually.

First we add our custom user stylesheet via Safari's preferences:



When we open our test page in Safari we clearly see that the author style rules override any conflicting user style rules:



We glean the following from this example:

- The **font-style** is not defined in the author stylesheet so the italic defined by the user gets used
- **color** is defined in both but the author's **purple** gets used given the origin principle
- same for **font-size**—the author's **72px** value is used
- **font-weight** is used in two stylesheets—the user agent defines it as **bold** while the author defines it as **normal**. It's a bit hard to see that the weight is *normal* since it's a heavy font (you'll have to trust me). Nevertheless, the author's rule wins that battle as well

This example clearly illustrates that, although it's rarely an issue to page authors since their stylesheet is the most weighty, origin is considered over specificity.

Styles That Stick

There are a few ways that styles rules will get applied. The most obvious is an explicit rule gets defined such directly such as:

```
h1.gaudy {  
  color: hotpink;  
}
```

With this definition, we are directly applying the color **hotpink** to any level one header with the class **gaudy**. This is a cascading style—simple enough. There are a couple of more subtle ways that a style rule may get applied such as **inheritance** and **initial** value. Let's look at those next.

Inheritance

According to the [spec](#):

When inheritance occurs, elements inherit computed values. The computed value from the parent element becomes both the specified value and the computed value on the child.

Inheritance happens when descendent elements take on the styles defined by an ancestor element. In the following example, the paragraph element will inherit the font family defined in its parent element `.outer` since **font-family** is an [inherited property](#):

```
<style>  
  .outer {  
    font-family: sans-serif;  
  }  
</style>  
<div class="outer">  
  <p>I will be a san-serif</p>  
</div>
```

Note that some types of CSS properties are inherited, others are not. [See this table](#), but, for example, **color** is an inherited property, while **display** and **border** are **not**. These rules tend to make intuitive sense—if **borders** were inherited, it would be a big issue; you'd end up with a maze of borders for elements that have children!

In terms of precedence, inheritance comes after cascading values, but before initial values.

Initial Value

[Sitepoint](#) provides an overview of how initial values work:

The default value for a property, when it's not specified explicitly or inherited, is called the initial value. The initial value of each property is defined in the CSS specification.

Browsers have user agent style sheets that define the default rendering of the various HTML element types. In some cases, the rules in those built-in style sheets define values other than the initial values from the CSS specification. For example, links are usually underlined, even though the initial value for the `text-decoration` property is none.

Another example of initial value is `border-color`, which will get a default of the [currentColor](#) (essentially, the current value of the `color` property). Initial value comes after cascading and inherited values in terms of precedence.

The browser has to work out the details of how cascaded, inherited, and initial style values are applied to any given element. Let's examine how that process works next.

Rule Evaluation

Once the browser has constructed a document tree—a tree-based hierarchy of elements representing the source document—it needs to take any **specified values** and resolve them to **computed values** (for example, `em` units are converted to pixels and URIs are made absolute), and then finally resolve those computed values to **used values** and then **actual values**. A used value might be the result of evaluating a rule like: `width: 75%`. Obviously, determining what that percentage width is calculating from has to happen at run-time. The actual value is the value that the browser actually uses when it renders the page. An example from the spec is:

the user agent may be forced to use only black and white shades instead of full color

The details of this process are defined in the [spec on value-stages](#).

As CSS authors, we have the most control over specified values so let's focus our attention on how those get evaluated:

6.1.1 Specified values

User agents must first assign a specified value to each property based on the following mechanisms (in order of precedence):

If the cascade results in a value, use it. Otherwise, if the property is inherited and the element is not the root of the document tree, use the computed value of the parent element. Otherwise use the property’s initial value. The initial value of each property is indicated in the property’s definition.

So, the cascade is still the first and foremost consideration, but now we start to look at cases where no direct style rule has been defined for a particular element. In such cases, inheritance and initial values will be considered. You’re probably asking yourself, “What is this *inheritance* thing, and what’s a property’s *initial value*? Let’s look at those now.

Take a Deep Breath

We’ve delved pretty deep into the details of the Cascade. Let’s try not to get too overwhelmed by these details and recap the most important bits:

- Rules from the Cascade are considered first
- Then inherited rules
- Then initial values

Whew!

Specificity Details

I’ve shown you earlier that origin is considered before specificity, but I have a confession to make... page authors generally only need to worry about specificity since, as stated, the author stylesheet gets the most weight.

Calculating Specificity

In order to determine how two or more conflicting style rules will get resolved (providing they have the same *weight*), the browser uses a sort of “scoring system” called [specificity](#). The main idea is that rules with a higher specificity will take precedence over those with a lower specificity.

Specificity is calculated as a four digit comma delimited number where each number can be labeled from left to right as: ‘a’, ‘b’, ‘c’, ‘d’.

- Add 1 to ‘a’ if there’s an inline ‘style’ attribute is used

- Add 1 to ‘b’ for each ID in the selector
- Add 1 to ‘c’ for each class, attribute and pseudo-classes in the selector
- Add 1 to ‘d’ for each element names and pseudo-elements in the selector

To gather the total specificity we simply concatenate those four numbers. Here are some examples straight from the [spec](#):

```
*          {} /* a=0 b=0 c=0 d=0 -> specificity = 0,0,0,0 */
li         {} /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1 */
li:first-line {} /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
ul li      {} /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
ul ol+li   {} /* a=0 b=0 c=0 d=3 -> specificity = 0,0,0,3 */
h1 + *[rel=up] {} /* a=0 b=0 c=1 d=1 -> specificity = 0,0,1,1 */
ul ol li.red {} /* a=0 b=0 c=1 d=3 -> specificity = 0,0,1,3 */
li.red.level {} /* a=0 b=0 c=2 d=1 -> specificity = 0,0,2,1 */
#x34y      {} /* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0 */
style=""   {} /* a=1 b=0 c=0 d=0 -> specificity = 1,0,0,0 */
```

Let’s take a look at a few more examples from within a simple html page:

```
<head>
<style type="text/css">
body { color: blue; }
#abc { color: red; }
</style>
</head>
<body>
  <p id="abc">I have and ID so I will render red</p>
  <h1 style="color: green">I have an inline style and render green</h1>
</body>
```

In this example, we have two defined style rules for `body` and `#abc`. The `body` rule has a specificity of 0001 and the `#abc` has a specificity of 0100. The ID clearly “wins”. The `h1` has an inline style and will thus calculate to 1000.

Let’s look at one more fairly complex selector:

```
#foo > #bar ul.klass li p:first-line {}
/* a=0 b=2 c=1 d=4 -> specificity = 0,2,1,4 */
```

- No inline style so a=0
- Two IDs, `#foo` and `#bar` so b=2
- One class `.klass` so c=1

- Three elements `ul`, `li`, and `p`, as well as one pseudo-element `:first-line` so `d=4`

Selector syntax is out of the scope for this book, but you can find the general rules for [selectors here](#).

Media

As mentioned earlier, the first step in evaluating the cascade, is to gather all relevant rules that apply for the current medium. This can be a factor when you're web site or application targets more than one medium. We'll take on print as that's a common use case.

Print Media

When a user elects to print one of your web pages, the current medium will be **print**. When this happens, it's preferable to only have to deal with one print stylesheet. However, if you've set your most general stylesheets to apply to **all** media (e.g. `media="all"`) your print stylesheet will have to "fight" against the more general stylesheets to win the specificity battle. You might instead want to target the media type of **screen** for your general stylesheets to avoid this. Generally, if you see a ton of `!important` values in the print stylesheet, it's a code smell that indicates the misuse of `media="all"`.

Debugging Print Styles

Eric Meyer provides a wonderful trick for debugging stylesheets in his book [Smashing CSS: Professional Techniques for Modern Layout](#). Essentially, you turn off the general stylesheets, by purposely setting the media type to something other than **all** or **screen**. Then, you set your print stylesheet to **screen**. Once that's done, you should be able to load your site in a normal browser—which is the **screen** medium—and have access to all your favorite developer tools as well as frequent page reloading. Here's an example of setting up such an approach:

```
<link rel="stylesheet" type="text/css" href="styles.css" media="tv">
<link rel="stylesheet" type="text/css" href="print.css" media="screen">
```

Once you've got your print styles looking the way you want, you'd simply switch `styles.css` to point to `media="screen"`, and `print.css` to point to `media="print"`. Neat trick, huh!

Resources

If you'd like a visual presentation of how the cascade rules are applied you may want to check out Russ Weakley's excellent [CSS Cascade Slideshare](#) chock full of extremely useful examples.

[CSS 2.1 Spec](#)

[CSS: The Missing Manual](#)

[CSS: The Definitive Guide, 3rd Edition](#)

[Smashing CSS: Professional Techniques for Modern Layout](#)