Understanding CSS Casecade

Garen Ikezian

What is CSS?

It's an imperative language that allows rules for the browser to design a web page. It stands for **C**ascading **S**tyle **S**heets.

With CSS, browsers will handle the actual styling of the webpage. The styler only makes styling requests to the browser.

The reasoning behind CSS being and imperative and not a procedural language is so that webpages can render consistently and efficiently. It also simplifies the design process and makes it abstract across all different clients. Instead of telling a particular browser how to behave (procedurally), we instead make styling requests and the browsers does the styling for us.

What is a cascade?

The cascade *defines* the rules for which rule should go first. It helps us to keep track which "style" will run.

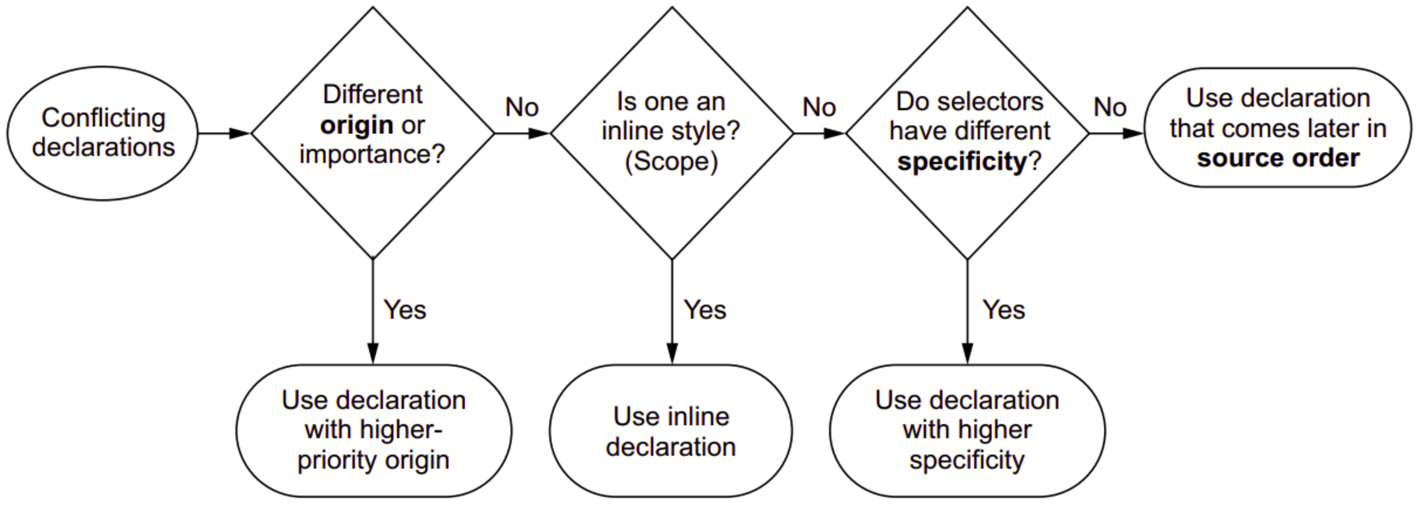
Conflicting declarations
Different **origin** or importance?
No
Is one an inline style? (Scope)
No
Do selectors have different **specificity**?
No
Use declaration that comes later in **source order**
Yes
Use declaration with higher-priority origin
Yes
Use inline declaration
Yes
Use declaration with higher specificity

Figure 1.3 High-level flowchart of the cascade showing declaration precedence

Figure 1: CSS Casecade

Cascading are determined by three factors:

1. Stylesheet Origin:

Where does the style come from? Is it inline, internal, or external css? (90% of our concerns are in the author style origin)

Author styles origin override the browser's default agent styles origin. The agent styles origin is the reason why `<li>` elements have underlined and blue by default

1. Author Important Origin: `!important` is never overriden. It is a bad practice if overused.
2. Author Style Origin: Whichever is the closest is called (starting from the closest).
    1. Inline Inside any HTML tag inside the `<body>` tag in an .html file
    2. Internal Inside the `<style>` in the `<head>` tag in an .html file
    3. External Inside any .css file

3. User Agent Origin: The "default style settings" of the browser so to speak

2. Selector Specificity:

What about specificity?

It helps us to categorize selectors by order of importance. They are (in order) inline styles, ID selectors, class selectors, and type/tag/element selectors.

Different types of selectors have different types of specifities.

`!important` has the highest specificity of them all. (We're not going to include it here since it is rarely used)

It is necessary to understand certain terminology through code before proceeding (pseudoClass and pseudoELements will be discussed later):

Example 1: CSS Important Terminology

```
.post:hover::before { color: red; }

.selector:pseudoClass::pseudoElement { property: value; }

/*
They all have the same specifity value
.post is a class selector
:hover is a pseudoClass
::before is a pseudoElement
.post:hover::before are a group of selectors
*/

/*
color: red; is a declaration
color is a property
red is a value
*/
```

Together, they make up what is called a *ruleset*.

Ok, but how do we determine simplicity?

We determine simplicity with a notation made of zeroes as such:

$$[0, 0, 0, 0]$$

1. 1st zero is inline specifity
2. 2nd zero is ID specificity
3. 3rd zero is class specifity
4. 4th zero is element specificity

Let's take these rulesets inside this html code as an example:

Example 2: What is the colour of h1?

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            /*  [0,0,0,3]  */
            html body h1 { color : red; }

            /*  [0,1,0,0]  */
            #h1 { color : green; }
        </style>
    </head>

    <body>
```

```
        <h1>Test</h1>
    </body>
</html>
```

What color do you think <h1> 's colour will be? It is red

But wait, doesn't the ID selector have a larger simplicity than an element selector?

Yes. But the reason why it is not green is because `<h1>` needs an HTML attribute called `id` for the id selector to be recognized as such.

Example 3: Fixing h1's colour

```
<!DOCTYPE html>
<html>
     <head>
         <style>
             /*  [0,0,0,3]  */
             html body h1 { color : red; }

             /*  [0,1,0,0]  */
             #h1 { color : green; }
         </style>
     </head>

     <body>
         /* We now have the attribute id added*/
         <h1 id="h1">Test</h1>

     </body>
</html>
```

Now it is green.

Again, the id attribute is looking for an id **selector** in the css file. If there is one, it will be used to perform styling stuff (In this case, changing the text colour to green).

Notice that the element h1 and the id "h1" selectors share the same name. This is to demonstrate that as long as there is the pound symbol (#), the selector will be regarded as an id.

The same thing applies for classes with the period at the beginning (.classname). In other words, there are html class attributes that are behave the same way as html `id` attributes but they only look for class **selectors**.

Now let's take a more complex example:

Example 4: Complex Specificity

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            /*  [0,0,0,3]  */
            html body h1 { color : red; }

            /*[0,0,1,1]*/
            body .colorClass{ color: green; }

            /*  [0,0,1,0]  */
            .colorClass{ color : blue; }

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

        </style>
    </head>

    <body>
```

```
        <h1 id="h1" class="colorClass">
            Geography Question
        </h1>

        <p class="body .colorClass">
          What is the capital of Burkina Faso?
        </p>

        <ul >
            <li> Pretoria </li>
            <li> Kinshasa </li>
            <li> Ouagadougou </li>
        </ul>
    </body>
</html>
```

Tip!: Hovering over rulesets with Visual Studio Code is a fantastic way to figure out specificity.

3. Source Order

It is related to selector specificty of which that if there are two selectors of equal specifity, the one at the bottom will win.

For example:

Example 5: Source Order

```
h1 { color: red; }

h1 { color: blue; }

/*h1 will be blue because the same specifier is below
the one above.*/
```

The same thing applies to classes and id specifiers.

To Conclude. . .

CSS can get really ugly if used unwisely. The general idea is that it is best to start general then write more and more specific rules. That way, the design of the webpage can be consistent and modifying the file can be less painful.

It will not change the fact that it will require overrides and too much of it will defeat the purpose of a consitent design.

There are now two methodologies, one has to either accept the casecade (more modular, less repetitive), resist (more modular, less reptitive) it, or do a little bit of both.

More Reading/Exercises:

- All the well-known CSS methodologies (like OOCSS, SMACSS, BEM etc.) are explained in this article
  - A look at Some CSS Methodologies by *William Craig*: https://www.webfx.com/blog/web-design/css-methodologies/
- Exploring SMACSS: Scalable and Modular Architecture for CSS by *Slobodan Gajic*: https://www.toptal.com/css/smacss-scalable-modular-architecture-css
- These exercises by flukeout are an excellent start to get to know with CSS selectors (`:nth-of-type`, `last-child`, `:first-child` etc.): https://flukeout.github.io/.