Cascading: inheritance and precedence

Cascading: inheritance and precedence

Inheritance

Now that we've covered several ways of defining CSS selectors, we need to understand what happens when multiple selectors resolve to the same element, and how an element can get inherit rules from its parent.

Remember when we said *"For now don't worry about the 'Cascading' part..."* at the beginning of this module? Well, that was then, this is now. From this moment on, you will need to worry about cascading.

Most CSS rules once applied to an element are *also* applied to all the children of that element, and to their children, and theirs *ad infinitum*. There are exceptions, notably the layout properties (margin, padding, position, width, etc.) and the decorative properties (border, background, etc.) **do not** cascade. This cascading of a CSS property from parent to child is also called "inheritance".

Generally, inheritance is a *good* thing. Do you want the whole page to use your corporate approved Web-font? `body { font-family: "Soulless", serif; }` is all you need. There is no need to apply the same `font-family` property to each and every tag used within the page. Thank you, Cascading!

However, sometimes inheritance can be a bad thing. An element may suddenly display in a way that you weren't expecting and you can't find any relevant CSS rule for that element. In this case, one likely culprit is a CSS rule that has been inherited from a parent. Thanks, Cascading!

Inheritance can be explicitly leveraged. Many CSS properties accept the value of `inherit`, which means to inherit the value from the parent. By smartly leveraging `inherit`, you can reduce repetition in your CSS rules and make your project easier to maintain.

In the sample below, we see a paragraph with children and grand-children. A CSS rule is applied to the paragraph that sets the font-family to be monospace, and the padding is set to 40 pixels. Note that in the result, the font-family is applied to all the children, while the padding is only applied to the paragraph itself, none of its children inherit the padding.

HTML	**CSS**	**Result**

`<p>This paragraph has children <span>spans</span> and <span>q</span>, which, in turn, have their own child <span>spans</span>.`
 	
`<q>With this structure, we can see how some CSS <span>rules</span> are <span>applied across a <q>variety</q></span> of scopes.</q>`
`</p>`
	`p {`
	`  /* inherited by children of p */`
	`  font-family: monospace;`
	
	`  /* not inherited */`
	`  padding: 40px;`
	`}`
		This paragraph has children spans and q, which, in turn, have their own child spans. With this structure, we can see how some CSS rules are applied across a variety of scopes.

Discussion	**CSS**	**Result**

To the right we add another CSS rule, this one instructing that the padding for spans and q elements should be inherited from their parent. Look at the result on the right, the padding is very evident.	`span, q { padding: inherit; }`	This paragraph has children spans and q , which, in turn, have their own child spans . With this structure, we can see how some CSS rules are applied across a variety of scopes.

Which rules are inheritable?

There is no reliable rule for which CSS properties are inheritable by default and which are not. However, generally, the properties associated with positioning and layout are *not* inherited. Likewise, the decorative properties (borders, background images, etc.) do not inherit. Most properties that begin with `text-` or `font-` inherit.

Precedence

It is possible, and easy, to have several different CSS rules all applying to the same element. This is often advantageous because most CSS properties are *orthogonal* to one another, meaning they do not interfere with each other. This gives us freedom to organize the CSS properties in rules in ways that make sense to us as developers, knowing that they can compose nicely. For example, a bit of text can be made italic by one rule, bold by another, and underlined by a third. We do not have to put all those properties into one place if that is not convenient for us.

However, what happens when there are different rules competing to set different values for the same property? This is where CSS *precedence* comes into play. When rendering CSS, the browser has some guidelines it follows for resolving conflicting rules. Here is rough summary, in order:

1 - Most specific rule

A more *specific* rule takes precedence over a less specific rule. A rule that more tightly matches a particular element than a general rule will be applied.

```
span { color: blue; }
ul li span { color: red; }
```

In the example above, both rules are attempting to set a span color for a span inside a list item. However, the second rule will "win" when there is a conflict (like color in this case).

2 - `#id` selector is the most specific

Rules with an id selector (e.g. `#someid`) are considered more specific than rules without.

3- `.class` selector is more specific than a tag selector

Rules employing a class selector (e.g. `.someclass`) are considered more specific than rules without (but not as specific as an #id selector, which trumps everything).

4- Rules that come later override those that come earlier

This guideline is for two CSS rulesets with the same selector. Where there are conflicts, the rules from the later one apply.

```
.hortense { color: red; text-decoration: underline; }
.hortense { color: blue; }
```

In the example above, an element with the `.hortense` class will be underlined and its color will be **blue**, because that rule came later than when it was set red.

No fear

These guidelines seem fairly straightforward, but situations can quickly get rather knotty. For example, what color should we expect in this situation?

HTML	CSS

arbol	p.forest span { color: green; } p span.tree { color: blue; }

What if the order of the CSS rules were reversed? Would it make a difference?

If this problem seems difficult to figure out, don't worry about it. In the next section, we will be looking at Chrome Developer Tools. You'll see how you can use the tools in the browser itself to inspect your elements and see exactly what CSS rules and properties are being inherited, applied and what is their precedence.

!important

```
p { color: orange !important; }
```

Because multiple CSS selectors can resolve to the same element, and because the rules that govern precedence are complex, you may from time to time encounter a situation where you need to apply a particular CSS property and you want it to take precedence over all others, no matter what. `!important` will do that.

The exclamation point is required, and the whole symbol (`!important`) goes after the value and before the semi-colon (`;`).

This may seem like an attractive option, but using it is not recommended. Once you start to use it, then you'll eventually run into a conflict with the various rules that are using `!important`, and from that conflict there is no escape. If you are having problems with precedence the best advice is to fix them directly, rather than using `!important`.