**Inheritance and Cascade**

Inheritance is the mechanism by which some property values are passed on from an element to its descendants.

When determining which values should apply to an element, a user agent must consider not only inheritance but also the specificity of the declarations, as well as the origin of the declarations themselves. This process of consideration is what’s known as the cascade

The answer is found in the specificity of each selector. For every rule, the user agent evaluates the specificity of the selector and attaches it to each declaration in the rule. When an element has two or more conflicting property declarations, the one with the highest specificity will win out.

A selector’s specificity is determined by the components of the selector itself. A specificity value can be expressed in four parts, like this: 0,0,0,0. The actual specificity of a selector is determined as follows:

* For every ID attribute value given in the selector, add 0,1,0,0.
* For every class attribute value, attribute selection, or pseudo-class given in the selector, add 0,0,1,0.
* For every element and pseudo-element given in the selector, add 0,0,0,1. CSS2 contradicted itself as to whether pseudo-elements had any specificity at all, but CSS2.1 made it clear that they do, and this is where they belong.
* Combinators and the universal selector do not contribute anything to the specificity.

**In order to determine which style rules apply, specificity values of selectors are determined from left to right.**

**What happens after specificity is decided?**

Once the specificity of a selector has been determined, the specificity value will be conferred on all of its associated declarations. Consider this rule:

h1 {color: silver; background: black;}

For specificity purposes, the user agent must treat the rule as if it were “ungrouped” into separate rules. Thus, the previous example would become:

h1 {color: silver;}

h1 {background: black;}

Consider another example:

h1 **+** p {color: black; font-style: *italic*;}              */\* 0,0,0,2 \*/*

p {color: gray; background: white; font-style: *normal*;} */\* 0,0,0,1 \*/*

\*.aside {color: black; background: silver;}             */\* 0,0,1,0 \*/*

When applied to the following markup, the content will be rendered as shown

<h1>Greetings!</h1>

<p class="aside">

    It's a fine way to start a day, don't you think?

</p>

<p>

    There are many ways to greet a person, but the words are not as important as

    the act of greeting itself.

</p>

<h1>Salutations!</h1>

<p>

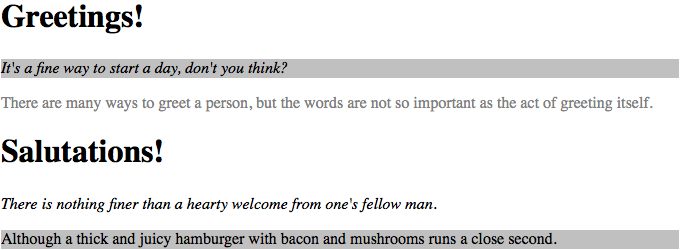
    There is nothing finer than a hearty welcome from one's fellow man.

</p>

<p class="aside">

    Although a thick and juicy hamburger with bacon and mushrooms runs a close second.

</p>



In every case, the user agent determines which rules match a given element, calculates all of the associated declarations and their specificities, determines which rules win out, and then applies the winners to the element to get the styled result. These machinations must be performed on every element, selector, and declaration. Fortunately, the user agent does it all automatically.

**Universal Selector Specificity**

The universal selector does not contribute to specificity. In other words, it has a specificity of 0,0,0,0, which is different than having no specificity

***Combinators, by comparison, have no specificity at all—not even zero specificity. Thus, they have no impact on a selector’s overall specificity.***

**Inline Style Specificity**

So far, we’ve only seen specificities that begin with a zero, so you may be wondering why it’s there at all. As it happens, that first zero is reserved for inline style declarations, which trump any other declaration’s specificity.

For Ex:

<h1 id="meadow" style="color: green;">The Meadow Party</h1>

h1#meadow {color: red;}

Font color for the above text will be green instead of red because inline-styling (1,0,0,0) has higher specificity than that of id’s (0,1,0,0).

**Importance**

Sometimes, a declaration is so important that it outweighs all other considerations. CSS calls these important declarations (for hopefully obvious reasons) and lets you mark them by inserting !important just before the terminating semicolon in a declaration:

p.dark {color: #333 **!important**; background: white;}

**If needed, each declaration has to be marked “!important”. “!important” keyword should always appear at the end of the declaration but before the semilcolon. Also, the ‘!’ (bang) preceding the word ‘important’ does not indicate as the declaration being not important.**

Declarations that are marked !important do not have a special specificity value, but are instead considered separately from non-important declarations. In effect, all !important declarations are grouped together, and specificity conflicts are resolved relatively within that group. Similarly, all non-important declarations are considered together, with any conflicts within the non-important group are resolved using specificity. Thus, in any case where an important and a non-important declaration conflict, the important declaration always wins.

**Inheritance**

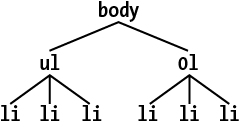
As important as specificity may be to understanding how declarations are applied to a document, another key concept is inheritance. Inheritance is the mechanism by which some styles are applied not only to a specified element, but also to its descendants. If a color is applied to an h1 element, for example, then that color is applied to all text inside the h1, even the text enclosed within child elements of that h1:

h1 {color: gray;}

<h1>Meerkat <em>Central</em></h1>

Both the ordinary h1 text and the em text are colored gray because the em element inherits the value of color from the h1. If property values could not be inherited by descendant elements, the em text would be black, not gray, and we’d have to color the elements separately.

For Ex:



Considering the above figure, with the following style:

ul {color: gray;}

When the declaration color: gray; is applied to the ul element, that element takes on that declaration. The value is then propagated down the tree to the descendant elements and continues on until there are no more descendants to inherit the value. Values are never propagated upward; that is, an element never passes values up to its ancestors.

**Note: There is an exception to the upward propagation rule in HTML: background styles applied to the body element can be passed to the html element, which is the document’s root element and therefore defines its canvas. This only happens if the body element has a defined background and the html element does not.**

**Things need to be considered about Inheritance:**

First, note that many properties are not inherited—generally in order to avoid undesirable outcomes. For example, the property border (which is used to set borders on elements) does not inherit. A quick glance at [Figure 3-5](https://learning.oreilly.com/library/view/css-the-definitive/9781449325053/ch03.html#why_borders_arenat_inherited) reveals why this is the case. If borders were inherited, documents would become much more cluttered—unless the author took the extra effort to turn off the inherited borders.

Second, inherited values have no specificity at all, not even zero specificity. This seems like an academic distinction until you work through the consequences of the lack of inherited specificity. Consider the following rules and markup fragment and compare them to the result shown in the following figure:

\* {color: gray;}

h1#page-title {color: black;}

<h1 id="page-title">Meerkat <em>Central</em></h1>

<p>

Welcome to the best place on the web for meerkat information!

</p>



Since the universal selector applies to all elements and has zero specificity, its color declaration’s value of gray wins out over the inherited value of black, which has no specificity at all. Therefore, the em element is rendered gray instead of black.

This example vividly illustrates one of the potential problems of using the universal selector indiscriminately. Because it can match any element, the universal selector often has the effect of short-circuiting inheritance. This can be worked around, but it’s usually more sensible to avoid the problem in the first place by not using the universal selector indiscriminately.

The complete lack of specificity for inherited values is not a trivial point. For example, assume that a style sheet has been written such that all text in a “toolbar” is to be white on black:

#toolbar {color: white; background: black;}

This will work so long as the element with an id of toolbar contains nothing but plain text. If, however, the text within this element is all hyperlinks (a elements), then the user agent’s styles for hyperlinks will take over. In a web browser, this means they’ll likely be colored blue, since the browser’s internal style sheet probably contains an entry like this:

a:link {color: blue;}

To overcome this problem, you must declare something like this:

#toolbar {color: white; background: black;}

#toolbar a:link {color: white;}

**The Cascade**

Throughout this chapter, we’ve skirted one rather important issue: what happens when two rules of equal specificity apply to the same element? How does the browser resolve the conflict? For example, consider the following rules:

h1 {color: red;}

h1 {color: blue;}

The cascade rules for CSS are:

1. Find all rules that contain a selector that matches a given element.
2. Sort all declarations applying to the given element by *explicit weight*. Those rules marked !important have a higher weight than those that are not.
3. Sort all declarations applying to the given element by *origin*. There are three basic origins: author, reader, and user agent. Under normal circumstances, the author’s styles win out over the reader’s styles. !important reader styles are stronger than any other styles, including !important author styles. Both author and reader styles override the user agent’s default styles.
4. Sort all declarations applying to the given element by *specificity*. Those elements with a higher specificity have more weight than those with lower specificity.
5. Sort all declarations applying to the given element by *order*. The later a declaration appears in the style sheet or document, the more weight it is given. Declarations that appear in an imported style sheet are considered to come before all declarations within the style sheet that imports them.

Consider following examples to better understand the above 5 points

p {color: gray **!important**;}

<p style="color: black;">Well, <em>hello</em> there!</p>

Despite the fact that there is a color assigned in the style attribute of the paragraph, the !important rule wins out, and the paragraph is gray. This gray is inherited by the em element as well.

## Sorting by Weight and Origin

In situations where the explicit weight is the same, the origin of a rule is considered. If an element is matched by normal-weight styles in both the author’s style sheet and the reader’s style sheet, then the author’s styles are used. For example, assume that the following styles come from the indicated origins:

p em {color: black;}    */\* author's style sheet \*/*

p em {color: yellow;}   */\* reader's style sheet \*/*

In this case, emphasized text within paragraphs is colored black, not yellow, because normal-weight author styles win out over normal-weight reader styles. However, if both rules are marked !important, the situation changes:

p em {color: black **!important**;}    */\* author's style sheet \*/*

p em {color: yellow **!important**;}   */\* reader's style sheet \*/*

To sum up, there are five basic levels to consider in terms of declaration weight. In order of most to least weight, these are:

1. Reader important declarations
2. Author important declarations
3. Author normal declarations
4. Reader normal declarations
5. User agent declarations

Authors typically need to worry about only the first four weight levels, since anything declared by an author will win out over the user agent’s styles.

**Sorting by Specificity**

If conflicting declarations apply to an element and they all have the same explicit weight and origin, they should be sorted by specificity, with the most specific declaration winning out, like this:

## Sorting by Order

Finally, if two rules have exactly the same explicit weight, origin, and specificity, then the one that occurs later in the style sheet wins out. Let’s return to our earlier example, where we find the following two rules in the document’s style sheet:

So what happens if rules from completely separate style sheets conflict? For example, suppose the following:

**@import** url(basic.css);

h1 {color: blue;}

What if h1 {color: red;} appears in basic.css? The entire contents of basic.css are treated as if they were pasted into the style sheet at the point where the @import occurs. Thus, any rule contained in the document’s style sheet occurs later than those from the @import. If they tie in terms of explicit weight and specificity, the document’s style sheet contains the winner.

Order sorting is the reason behind the often-recommended ordering of link styles. The recommendation is that you array your link styles in the order link-visited-focus-hover-active, or LVFHA, like this:

a:link {color: blue;}

a:visited {color: purple;}

a:focus {color: green;}

a:hover {color: red;}

a:active {color: orange;}

Thanks to the information in this chapter, you now know that the specificity of all of these selectors is the same: 0,0,1,1. Because they all have the same explicit weight, origin, and specificity, the last one that matches an element will win out. An unvisited link that is being “clicked” or otherwise activated, such as via the keyboard, is matched by four of the rules—:link, Lfocus, :hover, and :active—so the last one of those four will win out. Given the LVFHA ordering, :active will win, which is likely what the author intended.

The ability to chain pseudo-classes together eliminates all these worries.

a:link {color: blue;}

a:visited {color: purple;}

a:link:hover {color: red;}

a:visited:hover {color: gray;}

## Non-CSS Presentational Hints

It is possible that a document will contain presentational hints that are not CSS—for example, the font element. Such presentational hints are treated as if they have a specificity of 0 and appear at the beginning of the author’s stylesheet. Such presentation hints will be overridden by any author or reader styles, but not by the user agent’s styles. In CSS3, presentational hints from outside CSS are treated as if they belong to the user agent’s stylesheet, presumably at the end (although as of this writing, the specification doesn’t say).