Table of Contents

[Cascade 2](#_Toc142685209)

[Understanding stylesheet origin 2](#_Toc142685210)

[USER AGENT STYLES 2](#_Toc142685211)

[IMPORTANT DECLARATIONS 2](#_Toc142685212)

[Understanding specificity 3](#_Toc142685213)

[INLINE STYLES 3](#_Toc142685214)

[SELECTOR SPECIFICITY 3](#_Toc142685215)

[A NOTATION FOR SPECIFICITY 3](#_Toc142685216)

[SPECIFICITY CONSIDERATIONS 3](#_Toc142685217)

[Understanding source order 3](#_Toc142685218)

[LINK STYLES AND SOURCE ORDER 4](#_Toc142685219)

[CASCADED VALUES 4](#_Toc142685220)

[Two rules of thumb 4](#_Toc142685221)

[Inheritance 4](#_Toc142685222)

[Special values 4](#_Toc142685223)

[Using the inherit keyword 4](#_Toc142685224)

[Using the initial keyword 4](#_Toc142685225)

[Shorthand properties 5](#_Toc142685226)

[Beware shorthands silently overriding other styles 5](#_Toc142685227)

[Understanding the order of shorthand values 5](#_Toc142685228)

[Summary 5](#_Toc142685229)

# Cascade

Predicting how rules behave requires an understanding of the cascade; When two or more rules target the same element on your page, the rules may provide conflicting declarations.

The ***cascade***is the name for this set of rules. It determines how conflicts are resolved, and it’s a fundamental part of how the language works.

* Ex: All three rulesets attempt to set a different font family to this heading. Which one will win? To determine the answer, the browser follows a set of rules, so the result is predictable. In this case, the rules dictate that the second declaration, which has an ID selector, wins;

When declarations conflict, the cascade considers three things to resolve the difference:

* *Stylesheet origin*—Where the styles come from. Your styles are applied in conjunction with the browser’s default styles.
* *Selector specificity*—Which selectors take precedence over which.
* *Source order*—Order in which styles are declared in the stylesheet.

|  |
| --- |
|  |

## Understanding stylesheet origin

The stylesheets you add to your web page aren’t the only ones the browser applies. There are different types, or origins, of stylesheets. Yours are called *author* styles; there are also user agent styles, which are the browser’s default styles. User agent styles have lower priority, so your styles override them.

### USER AGENT STYLES

* Ex: Let’s look again at the example page (figure 1.4). The title is sans-serif because of thestyles you added. A number of other things are determined by the user agent styles: the list has a left padding and a list-style-type of disc to produce the bullets. Linksare blue and underlined.

NOTE You may notice I used ID selectors in this code. There are reasons to avoid doing this;

### IMPORTANT DECLARATIONS

There’s an exception to the style origin rules: declarations that are marked as ***important***. A declaration can be marked important by adding !important to the end of the declaration, before the semicolon:

color: red !important;

## Understanding specificity

If conflicting declarations can’t be resolved based on their origin, the browser next tries to resolve them by looking at their ***specificity***. Understanding specificity is essential. But if you don’t understand specificity, it will bite you. The browser evaluates specificity in two parts: styles applied inline in the HTML and styles applied using a selector.

### INLINE STYLES

If you use an HTML style attribute to apply styles, the declarations are applied only to that element. These are, in effect, “scoped” declarations, which override any declarations applied from your stylesheet or a <style> tag

To override inline declarations in your stylesheet, you’ll need to add an !important to the declaration, shifting it into a higher-priority origin.

### SELECTOR SPECIFICITY

The second part of specificity is determined by the selectors. Different types of selectors also have different specificities. An ID selector has a higher specificity than a class selector, Similarly, a class selector has a higher specificity than a tag selector; The exact rules of specificity are:

* If a selector has more IDs, it wins (that is, it’s more specific).
* If that results in a tie, the selector with the most classes wins.
* If that results in a tie, the selector with the most tag names wins.

NOTE Pseudo-class selectors (for example, :hover) and attribute selectors (for example, [type="input"]) each have the same specificity as a class selector.

The universal selector (\*) and combinators (>, +, ~) have no effect on specificity.

### A NOTATION FOR SPECIFICITY

A common way to indicate specificity is in a number form, often with commas between each number. For example, “1,2,2” indicates a specificity of one ID, two classes, and two tags. IDs having the highest priority are listed first, followed by classes, then tags.

The selector #page-header #page-title has two IDs, no classes, and no tags. We can say this has a specificity of 2,0,0. A specificity of 1,0,0 takes precedence over a specificity of 0,2,2 and even over 0,10,0 because the first number (IDs) is of the higher priority.

### SPECIFICITY CONSIDERATIONS

When you tried to apply the orange background using the .featured selector, it didn’t work. The selector #main-nav a has an ID that overrides the class selector (specificities 1,0,1 and 0,1,0). To correct this, you have some options to consider. Let’s look at several possible fixes. The quickest fix is to add an !important to the declaration you want to favor.

If you start adding !important to multiple declarations, what happens when you need to trump something already set to important?

What if you raised the specificity of your selector? Update the rulesets in your CSS to match this listing.

It is generally best to keep specificity low when you can, so when you need to override something, your options are open.

## Understanding source order

The third and final step to resolving the cascade is source order. If the origin and the specificity are the same, then the declaration that appears later in the stylesheet—or appears in a stylesheet included later on the page—takes precedence.

Although a featured button inside the nav looks correct, what happens if you want to use the

featured class on another link elsewhere on the page, outside of your nav;

When facing a styling problem, I often tackle it in two phases: First figure out what declarations will get it looking right. Second, think through the possible ways to structure the selectors and choose the one that best fits your needs.

### LINK STYLES AND SOURCE ORDER

you may have learned that your selectors for styling links should go in a certain order. If the user hovers over a visited link, the hover styles take precedence. If the user activates the link (that is, clicks it) while hovering over it, the active styles take precedence. A helpful mnemonic to remember this order is LoVe/HAte—link, visited, hover, active.

### CASCADED VALUES

The browser follows these three steps—origin, specificity, and source order to resolve every property for every element on the page. A declaration that “wins” the cascade is called a *cascaded value*.

### Two rules of thumb

there are two common rules of thumb for working with the cascade.

*Don’t use IDs in your selector.* Even one ID ratchets up the specificity a lot. When you need to override the selector, you often don’t have another meaningful ID you can use, so you wind up having to copy the original selector and add another class to distinguish it from the one you are trying to override.

*Don’t use !important.* This is even more difficult to override than an ID, and once you use it, you’ll need to add it every time you want to override the original declaration— and then you still have to deal with the specificity.

# Inheritance

There’s one last way that an element can receive styles—*inheritance*. The cascade is frequently

conflated with the concept of inheritance. Although the two topics are related, you should understand each individually. If an element has no cascaded value for a given property, it may inherit one from

an ancestor element.

# Special values

There are two special values that you can apply to any property to help manipulate the cascade: inherit and initial.

## Using the inherit keyword

Sometimes, you’ll want inheritance to take place when a cascaded value is preventing it. To do this, you can use the keyword inherit. You can override another value with this, and it will cause the element to inherit that value from its parent. You can also use the inherit keyword to force inheritance of a property not normally inherited, such as border or padding.

## Using the initial keyword

Sometimes you’ll find you have styles applied to an element that you want to undo. You can do this by specifying the keyword initial. Every CSS property has an initial, or default, value. If you assign the value initial to that property, then it effectively resets to its default value.

## Shorthand properties

*Shorthand properties* are properties that let you set the values of several other properties at one time. For example, font is a shorthand property that lets you set several font properties like background, border;

### Beware shorthands silently overriding other styles

This can silently override styles you specify elsewhere. If, for example, you were to use the shorthand font property for the page title without specifying a font-weight, a font weight of normal would still be set;

### Understanding the order of shorthand values

**TOP, RIGHT, BOTTOM, LEFT**

Remembering this order can keep you out of trouble. In fact, the word ***TRouBLe***is an mnemonic you can use to remember the order: top, right, bottom, left. Specify three values, and the left and right side will both use the second one. Specify two values, and the top and bottom will use the first one. If you specify only one value, it will apply to all four sides.

**HORIZONTAL, VERTICAL**

Whereas **padding: 1em 2em** specifies the vertical top/bottom values first, followed by the horizontal right/left values, **background-position: 25% 75%** specifies the horizontal right/left values first, followed by the vertical top/bottom values;

## Summary

* Keep selector specificity under control;
* Don’t confuse cascade with inheritance;
* Certain properties are inherited, including those for text, lists, and table borders.
* Don’t confuse initial and auto values;
* Stay out of TRouBLe with shorthand properties;

# **Working with relative units**

One of the most familiar, and probably easiest to work with, is pixels. These are known as ***absolute* units;** that is, 5 px always means the same thing; Other units, such as **em and rem**, are not absolute, but *relative*. The value of relative unit’s changes, based on external factors; Naturally, this makes relative units more difficult to work with.

## **The power of relative values**

### The struggle for pixel-perfect design

In the web environment, the user can have their browser window set to any number of sizes, and the CSS has to apply to it. Furthermore, users can resize the page after it’s opened, and the CSS needs to adjust to new constraints. This means that styles can’t be applied when you create your page; the browser must calculate those when the page is rendered onscreen. This adds a layer of abstraction to CSS. We can’t style an element according to an ideal context;

## **Ems and rems**

Ems are defined by the current element’s font size. the most common relative length unit, are a measure used in typography, referring to a specified font size. In CSS, 1 em means the font size of the current element; its exact value varies depending on the element you’re applying it to;

|  |
| --- |
| .padded {  font-size: 16px;  padding: 1em;  } |

The padding has a specified value of 1em. This is multiplied by the font size, producing a rendered padding of 16 px. This is important: Values declared using relative units are evaluated by the browser to an absolute value, called the *computed value*. In this example, editing the padding to **2 em** would produce a computed value of 32 px.

Using **ems** can be convenient when setting properties like padding, height, width, or border-radius because these will scale evenly with the element if it inherits different font sizes, or if the user changes the font settings.

It’s helpful to know that, for most browsers, the default font size is 16 px.

If you know the **pixel-based font size** you’d like, but want to specify the declaration in ems, here’s a simple formula: divide the desired pixel size by the parent (inherited) pixel size. For example, if you want a 10 px font and your element is inheriting a 12 px font, 10 / 12 = 0.8333 em.

### **EMS FOR FONT SIZE TOGETHER WITH EMS FOR OTHER PROPERTIES**

You’ve now defined ems for font-size (based on an inherited font size). And, you’ve defined ems for other properties like padding and border-radius (based on the current element’s font size). What makes ems tricky is when you use them for both font size and any other properties on the same element. When you do this, the browser must calculate the font size first, and then it uses that value to calculate the other values. Both properties can have the same declared value, but they’ll have different computed values.

What’s happening here is the paragraph inherits a font size of 16 px from the body, producing a calculated font size of 19.2 px. This means that 19.2 px is now the local value for an em, and that value is used to calculate the padding;

### **THE SHRINKING FONT PROBLEM**

Ems can produce unexpected results when you use them to specify the font sizes of multiple nested elements. To know the exact value for each element, you’ll need to know its **inherited font size,** which, if defined on the parent element in ems, requires you to know the parent element’s inherited size, and so on up the tree.

Shrinking text occurs when you **nest** **lists** **several** levels deep and apply an **em-based font size** to each level.

They’re nice for padding, margins, and element sizing, but when it comes to font size, they can get complicated. Thankfully, there is a better option—rems.

## Using rems for font-size