

Cascade and inheritance

English ▼

[↑ Overview: Building blocks](#)[Next →](#)

The aim of this lesson is to develop your understanding of some of the most fundamental concepts of CSS — the cascade, specificity, and inheritance — which control how CSS is applied to HTML and how conflicts are resolved.

While working through this lesson may seem less immediately relevant and a little more academic than some other parts of the course, an understanding of these things will save you much pain later on! We encourage you to work through this section carefully, and check that you understand the concepts before moving on.

Prerequisites: Basic computer literacy, [basic software installed](#), basic knowledge of [working with files](#), HTML basics (study [Introduction to HTML](#)), and an idea of how CSS works (study [CSS first steps](#).)

Objective: To learn about the cascade and specificity, and how inheritance works in CSS.

Conflicting rules

CSS stands for **Cascading Style Sheets**, and that first word *cascading* is incredibly important to understand — the way that the cascade behaves is key to understanding CSS.

At some point, you will be working on a project and you will find that the CSS you thought should be applied to an element is not working. Usually the problem is that you have created two rules which could potentially apply to the same element. The **cascade**, and the closely-related concept of **specificity**, are mechanisms that control which rule applies when there is such a conflict. Which rule is styling your element may not be the one you expect, so you need to understand how these mechanisms work.

Also significant here is the concept of **inheritance**, which means that some CSS properties by default inherit values set on the current element's parent element, and some don't. This can also cause some behavior that you might not expect.

Let's start by taking a quick look at the key things we are dealing with, then we'll look at each in turn and see how they interact with each other and your CSS. This can seem like a set of tricky concepts to understand. As you get more practice writing CSS, however, the way it works will become more obvious to you.

The cascade

Stylesheets **cascade** — at a very simple level this means that the order of CSS rules matter; when two rules apply that have equal specificity the one that comes last in the CSS is the one that will be used.

In the below example, we have two rules that could apply to the `h1`. The `h1` ends up being colored blue — these rules have an identical selector and therefore carry the same specificity, so the last one in the source order wins.

This is my heading.

```
h1 {  
  color: red;  
}  
h1 {  
  color: blue;  
}
```

```
<h1>This is my heading.</h1>
```

Reset

Specificity

Specificity is how the browser decides which rule applies if multiple rules have different selectors, but could still apply to the same element. It is basically a measure of how specific a selector's selection will be:

- An element selector is less specific — it will select all elements of that type that appear on a page — so will get a lower score.
- A class selector is more specific — it will select only the elements on a page that have a specific `class` attribute value — so will get a higher score.

Example time! Below we again have two rules that could apply to the `h1`. The below `h1` ends up being colored red — the class selector gives its rule a higher specificity, and so it will be applied even though the rule with the element selector appears further down in the source order.

This is my heading.

```
.main-heading {  
  color: red;  
}  
  
h1 {  
  color: blue;  
}
```

```
<h1 class="main-heading">This is my heading.</h1>
```

Reset

We'll explain specificity scoring and other such things later on.

Inheritance

Inheritance also needs to be understood in this context — some CSS property values set on parent elements are inherited by their child elements, and some aren't.

For example, if you set a `color` and `font-family` on an element, every element inside it will also be styled with that color and font, unless you've applied different color and font values directly to them.

As the body has been set to have a color of blue this is inherited through the descendants.

We can change the color by targetting the element with a selector, such as this `span`.

```
body {  
  color: blue;  
}  
  
span {  
  color: black;  
}
```

//

```
<p>As the body has been set to have a color of blue this  
is inherited through the descendants.</p>  
<p>We can change the color by targetting the element with  
a selector, such as this <span>span</span>.</p>
```

//

Reset

Some properties do not inherit — for example if you set a `width` of 50% on an element, all of its descendants do not get a width of 50% of their parent's width. If this was the case, CSS would be very frustrating to use!



Note: On MDN CSS property reference pages you can find a technical information box, usually at the bottom of the specifications section, which lists a number of data points about that property, including whether it is inherited or not. See the [color property Specifications section](#), for example.

Understanding how the concepts work together

These three concepts together control which CSS applies to what element; in the below sections we'll see how they work together. It can sometimes seem a little bit complicated, but you will start to remember them as you get more experienced with CSS, and you can always look up the details if you forget! Even experienced developers don't remember all the details.

Understanding inheritance

We'll start with inheritance. In the example below we have a ``, with two levels of unordered lists nested inside it. We have given the outer `` a border, padding, and a font color.

The color has applied to the direct children, but also the indirect children — the immediate child ``s, and those inside the first nested list. We have then added a class of `special` to the second nested list and applied a different color to it. This then inherits down through its children.

- Item One
- Item Two
 - 2.1
 - 2.2
- Item Three
 - **3.1**
 - **3.1.1**
 - **3.1.2**
 - **3.2**

```
.main {  
  color: rebeccapurple;  
  border: 2px solid #ccc;  
  padding: 1em;  
}
```

```
.special {  
  color: black;  
  font-weight: bold;  
}
```

```
<ul class="main">  
  <li>Item One</li>  
  <li>Item Two  
    <ul>  
      <li>2.1</li>  
      <li>2.2</li>  
    </ul>  
  </li>  
</ul>
```

Things like widths (as mentioned above), margins, padding, and borders do not inherit. If a border were to be inherited by the children of our list, every single list and list item would gain a border — probably not an effect we would ever want!

Which properties are inherited by default and which aren't is largely down to common sense.

Controlling inheritance

CSS provides four special universal property values for controlling inheritance. Every CSS property accepts these values.

`inherit`


Sets the property value applied to a selected element to be the same as that of its parent element. Effectively, this "turns on inheritance".


`initial`

Sets the property value applied to a selected element to be the same as the value set for that property on that element in the browser's default style sheet. If no value is set by the browser's default style sheet and the property is naturally inherited, then the property value is set to `inherit` instead.

`unset`

Resets the property to its natural value, which means that if the property is naturally inherited it acts like `inherit`, otherwise it acts like `initial`.

 **Note:** There is also a newer value, `revert`, which has limited browser support.

 **Note:** See [Origin of CSS declarations](#) in [Introducing the CSS Cascade](#) for more information on each of these and how they work.

We can look at a list of links and explore how the universal values work. The live example below allows you to play with the CSS and see what happens when you make changes. Playing with code really is the best way to get to grips with HTML and CSS.

For example:

1. The second list item has the class `my-class-1` applied. This sets the color of the `<a>` element nested inside to inherit. If you remove the rule how does it change the color of the link?
2. Do you understand why the third and fourth links are the color that they are? Check the description of the values above if not.
3. Which of the links will change color if you define a new color for the `<a>` element — for example `a { color: red; }`?

- Default [link](#) color
- Inherit the [link](#) color
- Reset the [link](#) color
- Unset the [link](#) color

```
body {  
  color: green;  
}  
  
.my-class-1 a {  
  color: inherit;  
}  
  
.my-class-2 a {  
  color: initial;  
}  
  
.my-class-3 a {  
  color: unset;  
}
```

```
<ul>  
  <li>Default <a href="#">link</a> color</li>  
  <li class="my-class-1">Inherit the <a href="#">link</a> color</li>  
  <li class="my-class-2">Reset the <a href="#">link</a> color</li>  
  <li class="my-class-3">Unset the <a href="#">link</a> color</li>  
</ul>
```

Reset

Resetting all property values

The CSS shorthand property `all` can be used to apply one of these inheritance values to (almost) all properties at once. Its value can be any one of the inheritance values (`inherit`, `initial`, `unset`, or `revert`). It's a convenient way to undo changes made to styles so that you can get back to a known starting point before beginning new changes.

In the below example we have two blockquotes. The first has styling applied to the blockquote element itself, the second has a class applied to the blockquote which sets the value of `all` to `unset`.

This blockquote is styled

This blockquote is not styled

```
blockquote {  
  background-color: red;  
  border: 2px solid green;  
}  
  
.fix-this {  
  all: unset;  
}
```

```
<blockquote>  
  <p>This blockquote is styled</p>  
</blockquote>  
  
<blockquote class="fix-this">  
  <p>This blockquote is not styled</p>  
</blockquote>
```

Reset

Try setting the value of `all` to some of the other available values and observe what the difference is.

Understanding the cascade

We now understand why a paragraph nested deep in the structure of your HTML is the same color as the CSS applied to the body, and from the introductory lessons we have an understanding of how to change the CSS applied to something at any point in the document — whether by assigning CSS to an element or creating a class. We will now take a proper look at how the cascade defines which CSS rules apply when more than one thing could style an element.

There are three factors to consider, listed here in decreasing order of importance. Earlier ones overrule later ones:

1. **Importance**
2. **Specificity**
3. **Source order**

We will look at these from the bottom up, to see how browsers figure out exactly what CSS should be applied.

Source order

We have already seen how source order matters to the cascade. If you have more than one rule, which has exactly the same weight, then the one that comes last in the CSS will win. You can think of this as rules which are nearer the element itself overwriting early ones until the last one wins and gets to style the element.

Specificity

Once you understand the fact that source order matters, at some point you will run into a situation where you know that a rule comes later in the stylesheet, but an earlier, conflicting, rule is applied. This is because that earlier rule has a **higher specificity** — it is more specific, and therefore is being chosen by the browser as the one that should style the element.

As we saw earlier in this lesson, a class selector has more weight than an element selector, so the properties defined on the class will override those applied directly to the element.

Something to note here is that although we are thinking about selectors, and the rules that are applied to the thing they select, it isn't the entire rule which is overwritten, only the properties which are the same.

This behavior helps avoid repetition in your CSS. A common practice is to define generic styles for the basic elements, and then create classes for those which are different. For example, in the stylesheet below we have defined generic styles for level 2 headings, and then created some classes which change only some of the properties and values. The values defined initially are applied to all headings, then the more specific values are applied to the headings with the classes.

Heading with no class

Heading with class of small

Heading with class of bright

```
h2 {  
  font-size: 2em;  
  color: #000;  
  font-family: Georgia, 'Times New Roman', Times, serif;  
}  
  
.small {  
  font-size: 1em;  
}  
  
.bright {  
  color: rebeccapurple;  
}
```


//

```
<h2>Heading with no class</h2>  
<h2 class="small">Heading with class of small</h2>  
<h2 class="bright">Heading with class of bright</h2>
```

Let's now have a look at how the browser will calculate specificity. We already know that an element selector has low specificity and can be overwritten by a class. Essentially a value is awarded to different types of selectors, and adding these up gives you the weight of that particular selector, which can then be assessed against other potential matches.

The amount of specificity a selector has is measured using four different values (or components), which can be thought of as thousands, hundreds, tens and ones — four single digits in four columns:

1. **Thousands:** Score one in this column if the declaration is inside a `style` attribute, aka inline styles. Such declarations don't have selectors, so their specificity is always simply 1000.
2. **Hundreds:** Score one in this column for each ID selector contained inside the overall selector.
3. **Tens:** Score one in this column for each class selector, attribute selector, or pseudo-class contained inside the overall selector.
4. **Ones:** Score one in this column for each element selector or pseudo-element contained inside the overall selector.

 **Note:** The universal selector (`*`), combinators (`+`, `>`, `~`, `'`), and negation pseudo-class (`:not`) have no effect on specificity.

The following table shows a few isolated examples to get you in the mood. Try going through these, and making sure you understand why they have the specificity that we have given them. We've not covered selectors in detail yet, but you can find details of each selector on the [MDN selectors reference](#).

Selector	Thousands	Hundreds	Tens	Ones	Total specificity
<code>h1</code>	0	0	0	1	0001
<code>h1 + p::first-letter</code>	0	0	0	3	0003
<code>li > a[href*="en-US"] > .inline-warning</code>	0	0	2	2	0022
<code>#identifier</code>	0	1	0	0	0100
No selector, with a rule inside an element's <code>style</code> attribute	1	0	0	0	1000

Before we move on, let's look at an example in action.

One

Two

```
/* specificity: 0101 */
#outer a {
  background-color: red;
}

/* specificity: 0201 */
#outer #inner a {
  background-color: blue;
}

/* specificity: 0104 */
#outer div ul li a {
  color: yellow;
}

/* specificity: 0112 */


<div id="outer" class="container">
  <div id="inner" class="container">
    <ul>
      <li class="nav"><a href="#">One</a></li>
      <li class="nav"><a href="#">Two</a></li>
    </ul>
  </div>
</div>
```

Reset

So what's going on here? First of all, we are only interested in the first seven rules of this example, and as you'll notice, we have included their specificity values in a comment before each one.

- The first two selectors are competing over the styling of the link's background color — the second one wins and makes the background color blue because it has an extra ID selector in the chain: its specificity is 201 vs. 101.

- The third and fourth selectors are competing over the styling of the link's text color — the second one wins and makes the text white because although it has one less element selector, the missing selector is swapped out for a class selector, which is worth ten rather than one. So the winning specificity is 113 vs. 104.
- Selectors 5–7 are competing over the styling of the link's border when hovered. Selector six clearly loses to five with a specificity of 23 vs. 24 — it has one fewer element selectors in the chain. Selector seven, however, beats both five and six — it has the same number of sub-selectors in the chain as five, but an element has been swapped out for a class selector. So the winning specificity is 33 vs. 23 and 24.

 **Note:** This has only been an approximate example for ease of understanding. In actuality, each selector type has its own level of specificity that cannot be overwritten by selectors with a lower specificity level. For example, a *million* **class** selectors combined would not be able to overwrite the rules of *one* **id** selector.

A more accurate way to evaluate specificity would be to score the specificity levels individually starting from highest and moving on to lowest when necessary. Only when there is a tie between selector scores within a specificity level do you need to evaluate the next level down; otherwise, you can disregard the lower specificity level selectors since they can never overwrite the higher specificity levels.

!important

There is a special piece of CSS that you can use to overrule all of the above calculations, however you should be very careful with using it — `!important`. This is used to make a particular property and value the most specific thing, thus overriding the normal rules of the cascade.

Take a look at this example where we have two paragraphs, one of which has an ID.

This is a paragraph.

One selector to rule them all!

```
#winning {  
  background-color: red;  
  border: 1px solid black;  
}  
  
.better {  
  background-color: gray;  
  border: none !important;  
}  
  
p {  
  background-color: blue;  
  color: white;  
  padding: 5px;  
}
```


```
<p class="better">This is a paragraph.</p>  
<p class="better" id="winning">One selector to rule them  
all!</p>
```

Reset

Let's walk through this to see what's happening — try removing some of the properties to see what happens if you are finding it hard to understand:

1. You'll see that the third rule's `color` and `padding` values have been applied, but the `background-color` hasn't. Why? Really all three should surely apply, because rules later in the source order generally override earlier rules.
2. However, The rules above it win, because class selectors have higher specificity than element selectors.

- Both elements have a `class` of `better`, but the 2nd one has an `id` of `winning` too. Since IDs have an *even higher* specificity than classes (you can only have one element with each unique ID on a page, but many elements with the same class — ID selectors are *very specific* in what they target), the red background color and the 1 pixel black border should both be applied to the 2nd element, with the first element getting the gray background color, and no border, as specified by the class.
- The 2nd element *does* get the red background color, but no border. Why? Because of the `!important` declaration in the second rule — including this after `border: none` means that this declaration will win over the border value in the previous rule, even though the ID has higher specificity.

 **Note:** The only way to override this `!important` declaration would be to include another `!important` declaration on a declaration with the *same specificity* later in the source order, or one with a higher specificity.

It is useful to know that `!important` exists so that you know what it is when you come across it in other people's code. **However, we strongly recommend that you never use it unless you absolutely have to.** `!important` changes the way the cascade normally works, so it can make debugging CSS problems really hard to work out, especially in a large stylesheet.

One situation in which you may have to use it is when you are working on a CMS where you can't edit the core CSS modules, and you really want to override a style that can't be overridden in any other way. But really, don't use it if you can avoid it.

The effect of CSS location

Finally, it is also useful to note that the importance of a CSS declaration depends on what stylesheet it is specified in — it is possible for users to set custom stylesheets to override the developer's styles, for example the user might be visually impaired, and want to set the font size on all web pages they visit to be double the normal size to allow for easier reading.

To summarize

Conflicting declarations will be applied in the following order, with later ones overriding earlier ones:

1. Declarations in user agent style sheets (e.g. the browser's default styles, used when no other styling is set).
2. Normal declarations in user style sheets (custom styles set by a user).
3. Normal declarations in author style sheets (these are the styles set by us, the web developers).
4. Important declarations in author style sheets
5. Important declarations in user style sheets

It makes sense for web developers' stylesheets to override user stylesheets, so the design can be kept as intended, but sometimes users have good reasons to override web developer styles, as mentioned above — this can be achieved by using `!important` in their rules.

Active learning: playing with the cascade

In this active learning, we'd like you to experiment with writing a single new rule that will override the color and background color that we've applied to the links by default. Can you use one of the special values we looked at in the [Controlling inheritance](#) section to write a declaration in a new rule that will reset the background color back to white, without using an actual color value?

If you make a mistake, you can always reset it using the *Reset* button. If you get really stuck, [take a look at the solution here](#).

- [One](#)
- [Two](#)

```
#outer div ul .nav a {  
  background-color: blue;  
  padding: 5px;  
  display: inline-block;  
  margin-bottom: 10px;  
}  
  
div div li a {  
  color: yellow;  
}
```

```
<div id="outer" class="container">  
  <div id="inner" class="container">  
    <ul>  
      <li class="nav"><a href="#">One</a></li>  
      <li class="nav"><a href="#">Two</a></li>  
    </ul>  
  </div>  
</div>
```

Reset

What's next

If you understood most of this article, then well done — you've started getting familiar with the fundamental mechanics of CSS. Next up, we'll look at selectors in detail.

If you didn't fully understand the cascade, specificity, and inheritance, then don't worry! This is definitely the most complicated thing we've covered so far in the course, and is something that even professional web developers sometimes find tricky. We'd advise that you return to this article a few times as you continue through the course, and keep thinking about it.

Refer back here if you start to come across strange issues with styles not applying as expected. It could be a specificity issue.

[↑ Overview: Building blocks](#)

[Next →](#)

In this module

1. [Cascade and inheritance](#)
2. [CSS selectors](#)
 - [Type, class, and ID selectors](#)
 - [Attribute selectors](#)
 - [Pseudo-classes and pseudo-elements](#)
 - [Combinators](#)
3. [The box model](#)
4. [Backgrounds and borders](#)
5. [Handling different text directions](#)
6. [Overflowing content](#)
7. [Values and units](#)
8. [Sizing items in CSS](#)
9. [Images, media, and form elements](#)
10. [Styling tables](#)
11. [Debugging CSS](#)
12. [Organizing your CSS](#)

Conflicting rules

Understanding how the concepts work together

Understanding inheritance

Understanding the cascade

The effect of CSS location

To summarize

Active learning: playing with the cascade

What's next

In this module

Related Topics

Complete beginners start here!

- ▶ [Getting started with the Web](#)

HTML — Structuring the Web

- ▶ [Introduction to HTML](#)
- ▶ [Multimedia and embedding](#)
- ▶ [HTML tables](#)
- ▶ [HTML forms](#)

CSS — Styling the Web

- ▶ [CSS first steps](#)
- ▼ [CSS building blocks](#)
 - [CSS building blocks overview](#)
 - [Cascade and inheritance](#)
 - [CSS selectors](#)
 - [The box model](#)
 - [Backgrounds and borders](#)

Handling different text directions

Overflowing content

Values and units

Sizing items in CSS

Images, media, and form elements

Styling tables

Debugging CSS

Organizing your CSS

- ▶ Styling text

- ▶ CSS layout

JavaScript — Dynamic client-side scripting

- ▶ JavaScript first steps

- ▶ JavaScript building blocks

- ▶ Introducing JavaScript objects

- ▶ Asynchronous JavaScript

- ▶ Client-side web APIs

Accessibility — Make the web usable by everyone

- ▶ Accessibility guides

- ▶ Accessibility assessment

Tools and testing

- ▶ Cross browser testing

Server-side website programming

- ▶ First steps

- ▶ [Django web framework \(Python\)](#)
- ▶ [Express Web Framework \(node.js/JavaScript\)](#)

Further resources

- ▶ [Common questions](#)

[How to contribute](#)



Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

Sign up now