Specificity

Specificity is the algorithm used by browsers to determine the CSS declaration that is the most relevant to an element, which in turn, determines the property value to apply to the element. The specificity algorithm calculates the weight of a CSS selector to determine which rule from competing CSS declarations gets applied to an element.

> **Note:** Browsers consider specificity **after** determining cascade origin and importance. In other words, for competing property declarations, specificity is relevant and compared only between selectors from the one cascade origin and layer that has precedence for the property. Order of appearance becomes relevant when the selector specificities of the competing declarations in the cascade layer with precedence are equal.

How is specificity calculated?

Specificity is an algorithm that calculates the weight that is applied to a given CSS declaration. The weight is determined by the number of selectors of each weight category in the selector matching the element (or pseudo-element). If there are two or more declarations providing different property values for the same element, the declaration value in the style block having the matching selector with the greatest algorithmic weight gets applied.

The specificity algorithm is basically a three-column value of three categories or weights - ID, CLASS, and TYPE - corresponding to the three types of selectors. The value represents the count of selector components in each weight category and is written as *ID - CLASS - TYPE*. The three columns are created by counting the number of selector components for each selector weight category in the selectors that match the element.

Selector weight categories

The selector weight categories are listed here in the order of decreasing specificity:

ID column

Includes only ID selectors, such as `#example`. For each ID in a matching selector, add 1-0-0 to the weight value.

CLASS column

Includes class selectors, such as `.myClass`, attribute selectors like `[type="radio"]` and `[lang|="fr"]`, and pseudo-classes, such as `:hover`, `:nth-of-type(3n)`, and `:required`. For each class, attribute selector, or pseudo-class in a matching selector, add 0-1-0 to the weight value.

TYPE column

Includes type selectors, such as `p`, `h1`, and `td`, and pseudo-elements like `::before`, `::placeholder`, and all other selectors with double-colon notation. For each type or pseudo-element in a matching selector, add 0-0-1 to the weight value.

No value

The universal selector (`*`) and the pseudo-class `:where()` and its parameters aren't counted when calculating the weight so their value is 0-0-0, but they do match elements. These selectors do not impact the specificity weight value.

Combinators, such as `+`, `>`, `~`, `" "`, and `||`, may make a selector more specific in what is selected but they don't add any value to the specificity weight.

The negation pseudo-class, `:not()`, itself has no weight. Neither do the `:is()` or the `:has()` pseudo-classes. The parameters in these selectors, however, do. The values of both come from the parameter in the list of parameters that has the highest specificity. The `:not()`, `:is()` and `:has()` exceptions are discussed below.

Matching selector

The specificity weight comes from the matching selector. Take this CSS selector with three comma-separated selectors as an example:

```
[type="password"],
input:focus,
:root #myApp input:required {
  color: blue;
}
```

The `[type="password"]` selector in the above selector list, with a specificity weight of `0-1-0`, applies the `color: blue` declaration to all password input types.

All inputs, no matter the type, when receiving focus, match the second selector in the list, `input:focus`, with a specificity weight of `0-1-1`; this weight is made up of the `:focus` pseudo-class (0-1-0) and the `input` type (0-0-1). If the password input has focus, it will match `input:focus`, and the specificity weight for the `color: blue` style declaration will be `0-1-1`. When that password doesn't have focus, the specificity weight remains at `0-1-0`.

The specificity for a required input nested in an element with attribute `id="myApp"` is `1-2-1`, based on one ID, two pseudo-classes, and one element type.

If the password input type is nested in an element with `id="myApp"` set, the specificity weight will be `1-2-1`, whether or not it has focus. Why is the specificity weight `1-2-1` rather than `0-1-1` or `0-1-0` in this case? Because the specificity weight comes from the matching selector with the greatest specificity weight. The weight is determined by comparing the values in the three columns, from left to right.

```
[type="password"]             /* 0-1-0 */
input:focus                   /* 0-1-1 */
:root #myApp input:required   /* 1-2-1 */
```

Three-column comparison

Once the specificity values of the relevant selectors are determined, the number of selector components in each column are compared, from left to right.

```css
#myElement {
  color: green; /* 1-0-0  - WINS!! */
}
.bodyClass .sectionClass .parentClass [id="myElement"] {
  color: yellow; /* 0-4-0 */
}
```

The first column is the value of the *ID* component, which is the number of IDs in each selector. The numbers in the *ID* columns of competing selectors are compared. The selector with the greater value in the *ID* column wins no matter what the values are in the other columns. In the above example, even though the yellow selector has more components in total, only the value of the first column matters.

If the number in the *ID* columns of competing selectors is the same, then the next column, *CLASS*, is compared, as shown below.

```css
#myElement {
  color: yellow; /* 1-0-0 */
}
#myApp [id="myElement"] {
  color: green; /* 1-1-0  - WINS!! */
}
```

The *CLASS* column is the count of class names, attribute selectors, and pseudo-classes in the selector. When the *ID* column value is the same, the selector with the greater value in the *CLASS* column wins, no matter the value in the *TYPE* column. This is shown in the example below.

```css
:root input {
  color: green; /* 0-1-1 - WINS because CLASS column is greater */
}
html body main input {
  color: yellow; /* 0-0-4 */
}
```

If the numbers in the *CLASS* and *ID* columns in competing selectors are the same, the *TYPE* column becomes relevant. The *TYPE* column is the number of element types and pseudo-elements in the selector. When the first two columns have the same value, the selector with the greater number in the *TYPE* column wins.

If the competing selectors have the same values in all the three columns, the proximity rule comes into play, wherein the last declared style gets precedence.

```css
input.myClass {
  color: yellow; /* 0-1-1 */
}
```

```
:root input {

  color: green; /* 0-1-1 WINS because it comes later */
}
```

The `:is()`, `:not()` and `:has()` exceptions

The matches-any pseudo-class `:is()`, the relational pseudo-class `:has()`, and the negation pseudo-class `:not()` are *not* considered as pseudo-classes in the specificity weight calculation. They themselves don't add any weight to the specificity equation. However, the selector parameters passed into the pseudo-class parenthesis are part of the specificity algorithm; the weight of the matches-any and negation pseudo-class in the specificity value calculation is the weight of the parameter's weight.

```
p {
  /* 0-0-1 */
}
:is(p) {
  /* 0-0-1 */
}

h2:nth-last-of-type(n + 2) {
  /* 0-1-1 */
}
h2:has(~ h2) {
  /* 0-0-2 */
}


div.outer p {
  /* 0-1-2 */
}
div:not(.inner) p {
  /* 0-1-2 */
}
```

Note that in the above CSS pairing, the specificity weight provided by the `:is()`, `:has()` and `:not()` pseudo-classes is the value of the selector parameter, not of the pseudo-class.

All three of these pseudo-classes accept complex selector lists, a list of comma-separated selectors, as a parameter. This feature can be used to increase a selector's specificity:

```
:is(p, #fakeId) {
  /* 1-0-0 */
}
h1:has(+ h2, > #fakeId) {
  /* 1-0-1 */
}
p:not(#fakeId) {
  /* 1-0-1 */
  /* 1-0-1 */
}
div:not(.inner, #fakeId) p {
  /* 1-0-2 */
}
```

In the above CSS code block, we have included `#fakeId` in the selectors. This `#fakeId` adds `1-0-0` to the specificity weight of each paragraph.

Generally, you want to keep specificity down to a minimum, but if you need to increase an element's specificity for a particular reason, these three pseudo-classes can help.

```
a:not(#fakeId#fakeId#fakeID) {
  color: blue; /* 3-0-1 */
}
```

In this example, all links will be blue, unless overridden by a link declaration with 3 or more IDs, a color value matching an `a` includes the [`!important` flag](), or if the link has an [inline style]() color declaration. If you use such a technique, add a comment to explain why the hack was needed.

Inline styles

Inline styles added to an element (e.g., `style="font-weight: bold;"`) always overwrite any normal styles in author stylesheets, and therefore, can be thought of as having the highest specificity. Think of inline styles as having a specificity weight of `1-0-0-0`.

The only way to override inline styles is by using `!important`.

Many JavaScript frameworks and libraries add inline styles. Using `!important` with a very targeted selector, such as an attribute selector using the inline style, is one way to override these inline styles.

```
<p style="color: purple">…</p>
```

```
p[style*="purple"] {
  color: rebeccapurple !important;
}
```

Make sure to include a comment with every inclusion of the important flag so code maintainers understand why a CSS anti-pattern was used.

The `!important` exception

CSS declarations marked as important override any other declarations within the same cascade layer and origin. Although technically, [`!important`]() has nothing to do with specificity, it interacts directly with specificity and the cascade. It reverses the [cascade]() order of stylesheets.

If declarations from the same origin and cascade layer conflict and one property value has the `!important` flag set, the important declaration
[illegible]

is applied no matter the specificity. When conflicting declarations from the same origin and cascade layer with the `!important` flag are applied to the same element, the declaration with a greater specificity is applied.

Using `!important` to override specificity is considered a **bad practice** and should be avoided for this purpose. Understanding and effectively using specificity and the cascade can remove any need for the `!important` flag.

Instead of using `!important` to override foreign CSS (from external libraries, like Bootstrap or normalize.css), import the third-party scripts directly into cascade layers. If you must use `!important` in your CSS, comment your usage so future code maintainers know why the declaration was marked important and know not to override it. But definitely, don't use `!important` when writing plugins or frameworks that other developers will need to incorporate without being able to control.

The `:where()` exception

The specificity-adjustment pseudo-class `:where()` always has its specificity replaced with zero, `0-0-0`. It enables making CSS selectors very specific in what element is targeted without any increase to specificity.

In creating third-party CSS to be used by developers who don't have access to edit your CSS, it's considered a good practice to create CSS with the lowest possible specificity. For example, if your theme includes the following CSS:

```
:where(#defaultTheme) a {
  /* 0-0-1 */
  color: red;
}
```

Then the developer implementing the widget can easily override the link color using only type selectors.

```
footer a {
  /* 0-0-2 */
  color: blue;
}
```

Tips for handling specificity headaches

Instead of using `!important`, consider using cascade layers and using low weight specificity throughout your CSS so that styles are easily overridden with slightly more specific rules. Using semantic HTML helps provide anchors from which to apply styling.

Making selectors specific with and without adding specificity

By indicating the section of the document you're styling before the element you're selecting, the rule becomes more specific. Depending on how you add it, you can add some, a lot, or no specificity, as shown below:

```
<main id="myContent">
  <h1>Text</h1>
</main>
```

```
#myContent h1 {
  color: green; /* 1-0-1 */
}
[id="myContent"] h1 {
  color: yellow; /* 0-1-1 */
}
:where(#myContent) h1 {
  color: blue; /* 0-0-1 */
}
```

No matter the order, the heading will be green because that rule is the most specific.

Reducing ID specificity

Specificity is based on the form of a selector. Including the `id` of an element as an attribute selector rather than an id selector is a good way to make an element more specific without adding an overabundance of specificity. In the previous example, the selector `[id="myContent"]` counts as an attribute selector for the purpose of determining the selector's specificity, even though it selects an ID.

You can also include the `id` or any part of a selector as a parameter in the `:where()` specificity-adjustment pseudo class if you need to make a selector more specific but don't want to add any specificity at all.

Increasing specificity by duplicating selector

As a special case for increasing specificity, you can duplicate weights from the *CLASS* or *ID* columns. Duplicating id, class, pseudo-class or attribute selectors within a compound selector will increase specificity when overriding very specific selectors over which you have no control.

```
#myId#myId#myId span {
  /* 3-0-1 */
}
.myClass.myClass.myClass span {
  /* 0-3-1 */
}
```

Use this sparingly, if at all. If using selector duplication, always comment your CSS.

By using `:is()` and `:not()` (and also `:has()`), you can increase specificity even if you can't add an `id` to a parent element:

```
:not(#fakeID#fakeId#fakeID) span {
  /* 3-0-1 */
}
:is(#fakeID#fakeId#fakeID, span) {
  /* 3-0-0 */
}
```

Precedence over third-party CSS

Leveraging cascade layers is the standard way of enabling one set of styles to take precedence over another set of styles; cascade layers enable this without using specificity! Normal (not important) author styles imported into cascade layers have lower precedence than unlayered author styles.

If styles are coming from a stylesheet you can't edit or don't understand and you need to override styles, a strategy is to import the styles you don't control into a cascade layer. Styles in subsequently declared layers take precedence, with unlayered styles having precedence over all layered styles from the same origin.

When two selectors from different layers match the same element, origin and importance take precedence; the specificity of the selector in the losing stylesheet is irrelevant.

```
<style>
  @import TW.css layer();
  p,
  p * {
    font-size: 1rem;
  }
</style>
```

In the above example, all paragraph text, including the nested content, will be `1rem` no matter how many class names the paragraphs have that match the TW stylesheet.

Avoiding and overriding `!important`

The best approach is to not use `!important`. The above explanations on specificity should be helpful in avoiding using the flag and removing it altogether when encountered.

To remove the perceived need for `!important`, you can do one of the following:

- Increase the specificity of the selector of the formerly `!important` declaration so that it is greater than other declarations
- Give it the same specificity and put it after the declaration it is meant to override
- Reduce the specificity of the selector you are trying to override.

All these methods are covered in preceding sections.

If you're unable to remove `!important` flags from an authors style sheet, the only solution to overriding the important styles is by using `!important`. Creating a cascade layer of important declaration overrides is an excellent solution. Two ways of doing this include:

Method #1

1. Create a separate, short style sheet containing only important declarations specifically overriding any important declarations you were unable to remove.
2. Import this stylesheet as the first import in your CSS using `layer()`, including the `@import` statement, before linking to other stylesheets. This is to ensure that the important overrides is imported as the first layer.

```
<style>
```

```
  @import importantOverrides.css layer();

</style>
```

Method #2

1. At the beginning of your stylesheet declarations, create a named cascade layer, like so:

   ```
   @layer importantOverrides;
   ```

2. Each time you need to override an important declaration, declare it within the named layer. Only declare important rules within the layer.

   ```
   [id="myElement"] p {
     /* normal styles here */
   }
   @layer importantOverrides {
     [id="myElement"] p {
       /* important style here */
     }
   }
   ```

The specificity of the selector of the important style within the layer can be low, as long as it matches the element you are trying to override. Normal layers should be declared outside the layer because layered styles have lower precedence than unlayered styles.

Tree proximity ignorance

The proximity of an element to other elements that are referenced in a given selector has no impact on specificity.

```
body h1 {
  color: green;
}

html h1 {
  color: purple;
}
```

The `<h1>` elements will be purple because when declarations have the same specificity, the last declared selector has precedence.

Directly targeted elements vs. inherited styles

Styles for a directly targeted element will always take precedence over inherited styles, regardless of the specificity of the inherited rule. Given the following CSS and HTML:

```
#parent {
  color: green;
}

h1 {
```

```
  color: purple;
}
```

```html
<html lang="en">
  <body id="parent">
    <h1>Here is a title!</h1>
  </body>
</html>
```

The `h1` will be purple because the `h1` selector targets the element specifically, while the green is inherited from the `#parent` declarations.

Examples

In the following CSS, we have three selectors targeting [`<input>`] elements to set a color. For a given input, the specificity weight of the color declaration having precedence is the matching selector with the greatest weight:

```css
#myElement input.myClass {
  color: red;
} /* 1-1-1 */
input[type="password"]:required {
  color: blue;
} /* 0-2-1 */
html body main input {
  color: green;
} /* 0-0-4 */
```

If the above selectors all target the same input, the input will be red, as the first declaration has the highest value in the *ID* column.

The last selector has four *TYPE* components. While it has the highest integer value, no matter how many elements and pseudo-elements are included, even if there were 150, TYPE components never have precedence over *CLASS* components. The column values are compared starting from left to right when column values are equal.

Had we converted the id selector in the example code above to an attribute selector, the first two selectors would have the same specificity, as shown below:

```css
[id="myElement"] input.myClass {
  color: red;
} /* 0-2-1 */
input[type="password"]:required {
  color: blue;
} /* 0-2-1 */
```

When multiple declarations have equal specificity, the last declaration found in the CSS is applied to the element. If both selectors match the same [`<input>`], the color will be blue.