CSS Cascading and Inheritance Level 6

W3C Working Draft, 21 March 2023

▼ **More details about this document**

This version:
https://www.w3.org/TR/2023/WD-css-cascade-6-20230321/

Latest published version:
https://www.w3.org/TR/css-cascade-6/

Editor's Draft:
https://drafts.csswg.org/css-cascade-6/

Previous Versions:
https://www.w3.org/TR/2021/WD-css-cascade-6-20211221/

History:
https://www.w3.org/standards/history/css-cascade-6

Feedback:
CSSWG Issues Repository
Inline In Spec

Editors:
Elika J. Etemad / fantasai (Invited Expert)
Miriam E. Suzanne (Invited Expert)
Tab Atkins Jr. (Google)

Suggest an Edit for this Spec:
GitHub Editor

Abstract

This CSS module describes how to collate style rules and assign values to all properties on all elements. By way of cascading and inheritance, values are propagated for all properties on all elements.

New in this level is § 2.5 Scoping Styles: the @scope rule.

CSS is a language for describing the rendering of structured documents (such as HTML and XML)

on screen, on paper, etc.

Status of this document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the CSS Working Group as a **Working Draft** using the Recommendation track. Publication as a Working Draft does not imply endorsement by W3C and its Members.

This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Please send feedback by filing issues in GitHub (preferred), including the spec code "css-cascade" in the title, like this: "[css-cascade] *...summary of comment...*". All issues and comments are archived. Alternately, feedback can be sent to the (archived) public mailing list www-style@w3.org.

This document is governed by the 2 November 2021 W3C Process Document.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

§ 1. Introduction and Missing Sections

ISSUE 1 This is a diff spec over CSS Cascading and Inheritance Level 5. It is currently an Exploratory Working Draft: if you are implementing anything, please use Level 5 as a reference. We will merge the Level 5 text into this draft once it reaches CR. ¶

§ 2. Cascading

The ***cascade*** takes an unordered list of declared values for a given property on a given element, sorts them by their declaration's precedence as determined below, and outputs a single cascaded value.

§ 2.1. Cascade Sorting Order

The cascade sorts declarations according to the following criteria, in descending order of precedence:

¶ **Origin and Importance**

The origin of a declaration is based on where it comes from and its importance is whether or not it is declared with '!important' (see below). The precedence of the various origins is, in descending order:

1. Transition declarations [css-transitions-1]
2. Important user agent declarations
3. Important user declarations
4. Important author declarations
5. Animation declarations [css-animations-1]
6. Normal author declarations
7. Normal user declarations
8. Normal user agent declarations

Declarations from origins earlier in this list win over declarations from later origins.

¶ **Context**

A document language can provide for blending declarations sourced from different ***encapsulation contexts***, such as the nested tree contexts of shadow trees in the [DOM].

When comparing two declarations that are sourced from different encapsulation contexts, then for normal rules the declaration from the outer context wins, and for important rules the declaration from the inner context wins. For this purpose, [DOM] tree contexts are considered to be nested in shadow-including tree order.

> NOTE: This effectively means that normal declarations belonging to an encapsulation context can set defaults that are easily overridden by the outer context, while important declarations belonging to an encapsulation context can enforce requirements that cannot be overridden by the outer context.

The Style Attribute

Separately for normal and important declarations, declarations that are attached directly to an element (such as the contents of a style attribute) rather than indirectly mapped by means of a style rule selector take precedence over declarations the same importance that are mapped via style rule.

Layers

Declarations within each origin and context can be explicitly assigned to a cascade layer. For the purpose of this step, any declaration not assigned to an explicit layer is added to an implicit final layer.

Cascade layers (like declarations) are sorted by order of appearance, see § 2.4.1 Layer Ordering. When comparing declarations that belong to different layers, then for normal rules the declaration whose cascade layer is latest in the layer order wins, and for important rules the declaration whose cascade layer is earliest wins.

> NOTE: This follows the same logic used for precedence of normal and important origins, thus the '!important' flag maintains the same "override" purpose in both settings.

Strong Scoping Proximity

If two declarations both have elements selected by scoped descendant relationships applying strong scoping proximity, then the declaration with the fewest generational hops between the ancestor/descendant element pair wins.

If multiple such pairs are represented, their strong scoping proximity weights are compared from innermost scoping relationship to outermost scoping relationship (with any missing pairs weighted as infinity).

Specificity

The Selectors module [SELECT] describes how to compute the specificity of a selector. Each declaration has the same specificity as the style rule it appears in. The declaration with the highest specificity wins.

Weak Scoping Proximity

If two declarations both have elements selected by scoped descendant relationships applying weak scoping proximity, then the declaration with the fewest generational hops between the ancestor/descendant element pair wins.

If multiple such pairs are represented, their weak scoping proximity weights are compared from innermost scoping relationship to outermost scoping relationship (with any missing pairs weighted as infinity).

Order of Appearance

The last declaration in document order wins. For this purpose:

- Style sheets are ordered as in final CSS style sheets.

- Declarations from 'imported style sheets' are ordered as if their style sheets were substituted in place of the '@import' rule.
- Declarations from style sheets independently linked by the originating document are treated as if they were concatenated in linking order, as determined by the host document language.
- Declarations from style attributes are ordered according to the document order of the element the style attribute appears on, and are all placed after any style sheets. [CSSSTYLEATTR]

ISSUE 2 Does ***scope proximity*** belong above or below specificity in the cascade? [Issue #6790]

The ***output of the cascade*** is a (potentially empty) sorted list of declared values for each property on each element.

§ 2.2. Cascading Origins

ISSUE 3 *CSS Cascading 5* § 6.2 Cascading Origins

cascade origin

§ 2.3. Important Declarations: the '!important' annotation

ISSUE 4 *CSS Cascading 5* § 6.3 Important Declarations: the !important annotation

important normal

§ 2.4. Cascade Layers

ISSUE 5 *CSS Cascading 5* § 6.4 Cascade Layers

§ 2.4.1. Layer Ordering

ISSUE 6 *CSS Cascading 5* § 6.4.3 Layer Ordering

§ 2.5. Scoping Styles: the '@scope' rule

A ***scope*** is a subtree or fragment of a document, which can be used by selectors for more targeted matching. A scope is formed by determining:

- The scoping root node, which acts as the upper bound of the scope, and optionally:
- The ***scoping limit*** elements, which act as the lower bounds.

An element is ***in scope*** if:

- It is an inclusive descendant of the scoping root, and
- It is not an inclusive descendant of a scoping limit.

NOTE: In contrast to Shadow Encapsulation, which describes a persistent one-to-one relationship in the DOM between a shadow host and its nested shadow tree, multiple overlapping scopes can be defined in relation to the same elements.

Scoped styles are described in CSS using the ***'@scope'*** block at-rule, which declares a scoping root and optional scoping limits associated with a set of style rules.

EXAMPLE 1 ¶

For example, an author might have wide-reaching color-scheme scopes, which overlap more narrowly-scoped design patterns such as a media object. The selectors in the '@scope' rule establish scoping root and optional scoping limit elements, while the nested selectors only match elements that are in a resulting scope:

```
@scope (.light-scheme) {
  /* Only match links inside a light-scheme */
  a { color: darkmagenta; }
}

@scope (.dark-scheme) {
  /* Only match links inside a dark-scheme */
  a { color: plum; }
}

@scope (.media-object) {
  /* Only match author images inside a media-object */
  .author-image { border-radius: 50%; }
}
```

EXAMPLE 2

By providing scoping limits, an author can limit matching more deeply nested descendants. For example:

```
@scope (.media-object) to (.content > *) {
  img { border-radius: 50%; }
  .content { padding: 1em; }
}
```

The 'img' selector will only match image tags that are in a DOM fragment starting with any '.media-object', and including all descendants up to any intervening children of the '.content' class.

ISSUE 7 Should scoping limits be added to the definition of scoped selectors?

2.5.1. Effects of '@scope'

The '@scope' at-rule has three primary effects on the style rules it contains:

- The style rules in an '@scope' <stylesheet> are scoped style rules.
- The ':scope' selector is defined to match the '@scope' rule's scoping root. The '&' selector is defined to represent the selector representing the scoping root (the <scope-start> selector), or else ':scope' if no selector was specified.
- The cascade prioritizes declarations with a more proximate scoping root, regardless of specificity or source order by applying weak scoping proximity between the scoping root and the subject of each scoped style rule.

ISSUE 8 Should '@scope' use strong or weak scoping proximity? Strong scoping proximity causes declarations to be weighted more strongly by scope proximity than by their selector's specificity. Weak scoping proximity causes declarations of the same specificity to be weighted by proximity to their scoping root before falling back to source ordering, but declarations of higher specificity win over more tightly-scoped declarations. The Working Group currently leans towards weak proximity, and recommends that as a starting point for prototypes. [Issue #6790]

NOTE: Unlike Nesting, selectors within an '@scope' rule do not acquire the specificity of any parent selector(s) in the '@scope' prelude.

EXAMPLE 3 ¶

The following selectors have the same specificity (0,0,1):

```
@scope (#hero) {
  img { border-radius: 50%; }
}

:where(#hero) img { border-radius: 50%; }
```

The additional specificity of the '#hero' selector is not applied to the specificity of the scoped selector. However, since one `<img>` selector is scoped, that selector is weighted more strongly in the cascade with the application of weak scoping proximity.

EXAMPLE 4 ¶

Many existing tools implement "scoped styles" by applying a unique class or attribute to every element in a given scope or "single file component." In this example there are two scopes (`main-component` and `sub-component`) and every element is marked as part of one or both scopes using the `data-scope` attribute:

```
<section data-scope="main-component">
  <p data-scope="main-component">...<p>

  <!-- sub-component root is in both scopes -->
  <section data-scope="main-component sub-component">
    <!-- children are only in the inner scope -->
    <p data-scope="sub-component">...<p>
  </section>
</section>
```

Those custom scope attributes are then appended to every single selector in CSS:

```
p[data-scope~='main-component'] { color: red; }
p[data-scope~='sub-component'] { color: blue; }

/* both sections are part of the outer scope */
section[data-scope~='main-component'] { background: snow; }

/* the inner section is also part of the inner scope */
section[data-scope~='sub-component'] { color: ghostwhite; }
```

Using the ‘@scope’ rule, authors and tools can replicate similar behavior with the unique attribute or class applied only to the scoping roots:

```
<section data-scope="main-component">
  <p>...<p>
  <section data-scope="sub-component">
    <p>...<p>
  </section>
</section>
```

Then the class or attribute can be used for establishing both upper and lower boundaries. Elements matched by a lower boundary selector are excluded from the resulting scope, which allows authors to create non-overlapping scopes by default:

```
@scope ([data-scope='main-component']) to ([data-scope]) {
```

```
  p { color: red; }

  /* only the outer section is part of the outer scope */
  section { background: snow; }
}

@scope ([data-scope='sub-component']) to ([data-scope]) {
  p { color: blue; }

  /* the inner section is only part of the inner scope */
  section { color: ghostwhite; }
}
```

However, authors can use the child combinator and universal selector to create scope boundaries that overlap, such that the inner scope root is part of both scopes:

```
@scope ([data-scope='main-component']) to ([data-scope] > *) {
  p { color: red; }

  /* both sections are part of the outer scope */
  section { background: snow; }
}
```

2.5.2. Syntax of '@scope'

The syntax of the '@scope' rule is:

```
@scope [(<scope-start>)]? [to (<scope-end>)]? {
  <stylesheet>
}
```

where:

- '***<scope-start>***' is a <forgiving-selector-list> selector used to identify the scoping root(s).
- '***<scope-end>***' is a <forgiving-selector-list> selector used to identify any scoping limits.
- the <stylesheet> represents the scoped style rules.

Pseudo-elements cannot be scoping roots or scoping limits; they are invalid both within <scope-start> and <scope-end>.

§ 2.5.3. Scoped Style Rules

Scoped style rules differ from non-scoped rules in the following ways:

- Their selectors can only match elements that are in scope. (This only applies to the subject; the rest of the selector can match unrestricted.)
- They accept a <relative-selector-list> as their prelude (rather than just a <selector-list>). Such relative selectors are relative to ':scope'.
- Any selector in the <relative-selector-list> that does not start with a combinator but does contain the nesting selector or the ':scope' selector, is interpreted as a non-relative selector (but the subject must still be in scope to match).

EXAMPLE 5

By default, selectors in a scoped style rule are relative selectors, with the scoping root and descendant combinator implied at the start. The following selectors will match the same elements:

```
@scope (#my-component) {
  p { color: green; }
  :scope p { color: green; }
}
```

Authors can adjust the implied relationship by adding an explicit combinator:

```
@scope (#my-component) {
  > p { color: green; }
  :scope > p { color: green; }
}
```

Authors can also target or explicitly position the scoping root in a selector by including either ':scope' or '&' in a given selector:

```
@scope (#my-component) {
  :scope { border: thin solid; }
  & { border: thin solid; }

  main :scope p { color: green; }
  main & p { color: green; }
}
```

While the ':scope' or '&' selectors can both refer to the scoping root, they have otherwise different meanings in this context:

Differences in selector matching
: The ':scope' selector will only match the scoping root itself, while the '&' selector is able to match any element that is matched by the <scope-start> selector list.

Differences in selector specificity
: The ':scope' selector has a specificity equal to other pseudo-classes, while the '&' selector has the specificity equal to the most specific selector in <scope-start>.

§ 2.5.4. Identifying Scoping Roots and Limits

A '@scope' rule produces one or more scopes as follows:

Finding the scoping root(s)

For each element matched by <scope-start>, create a scope using that element as the scoping root. If no <scope-start> is specified, the scoping root is the parent element of the owner node of the stylesheet where the '@scope' rule is defined. (If no such element exists, then the scoping root is the root of the containing node tree.) Any ':scope' or '&' selectors in <scope-start> are interpreted as defined for its outer context.

Finding any scoping limits

For each scope created by a scoping root, its scoping limits are set to all elements that are in scope and that match <scope-end>, interpreting ':scope' and '&' exactly as in scoped style rules.

EXAMPLE 6 ¶

Authors can establish local scoping for `<style>` elements by leaving out the <scope-start> selector. For example:

```
<div>
  <style>
    @scope {
      p { color: red; }
    }
  </style>
  <p>this is red</p>
</div>
<p>not red</p>
```

That would be equivalent to:

```
<div id="foo">
  <style>
    @scope (#foo) {
      p { color: red; }
    }
  </style>
  <p>this is red</p>
</div>
<p>not red</p>
```

EXAMPLE 7 ¶

Scoping limits can use the ':scope' pseudo-class to require a specific relationship to the scoping root:

```
/* .content is only a limit when it is a direct child of the :scope */
@scope (.media-object) to (:scope > .content) { ... }
```

Scoping limits can also reference elements outside their scoping root by using ':scope'. For example:

```
/* .content is only a limit when the :scope is inside .sidebar */
@scope (.media-object) to (.sidebar :scope .content) { ... }
```

§ 2.5.5. Scope Nesting

'@scope' rules can be nested. In this case, just as with the nested style rules, the selectors of the inner '@scope' (including those defining its scope) are scoped by the selectors of the outer one.

EXAMPLE 8 ¶

When nesting '@scope' rules inside other '@scope' rules, or inside other selectors, the <scope-start> selector is relative to the nesting context, while the <scope-end> and any scoped style rules are relative to the scoping root For example, the following code:

```
@scope (.parent-scope) {
  @scope (:scope > .child-scope) to (:scope .limit) {
    :scope .content {
      color: red;
    }
  }
}
```

is equivalent to:

```
@scope (.parent-scope > .child-scope) to (.parent-scope > .child-scope .limi
  .parent-scope > .child-scope .content {
    color: red;
  }
}
```

Global name-defining at-rules such as '@keyframes' or '@font-face' or '@layer' that are defined inside '@scope' are valid, but are not scoped or otherwise affected by the enclosing '@scope' rule. However, any style rules contained by such rules (e.g. within '@layer') are scoped.

§ 2.6. Scoped Descendant Combinator

The ***scoped descendant combinator*** describes a descendant relationship between two elements. A selector of the form 'A >> B' represents an element `B` that is an arbitrary descendant of some ancestor element `A`.

This combinator differs from the descendant combinator in that it applies weak scoping proximity to the relationship between 'A' and 'B'. It does not change the ':scope' element.

ISSUE 9 Should the scoped descendant combinator use strong or weak scoping proximity? Should it even exist? It's defined here to work the way many people expected the regular descendant combinator to work...

EXAMPLE 9

This means that style rules using the scoped descendant combinator are sorted by specificity just like the regular descendant combinator, except that when their specificities are equal the more tightly-scoped declaration wins.

In this example the `<a>` element's color will be determined by the nearest ancestor with either a 'light-scheme' or 'dark-scheme' class. (If the descendant selector had been used, its color would always be 'plum', because it is later in the source order.)

```
.light-scheme >> a { color: darkmagenta; }
.dark-scheme >> a { color: plum; }
```

However if the `<a>` element has a 'light-scheme' ancestor and is focused, its color will be 'teal' even if it has a nearer 'dark-scheme' ancestor, because there is no equivalent 'dark-scheme' rule.

```
.light-scheme >> a:focus { color: teal; }
```

§ 2.7. Precedence of Non-CSS Presentational Hints

ISSUE 10 *CSS Cascading 5* § 6.4 Cascade Layers

3. Changes

This appendix is *informative*.

3.1. Changes since the 21 December 2021 First Public Working Draft

Significant changes since the 21 December 2021 First Public Working Draft include:

- Clarified '@scope' effects on nested ':scope' and '&' selectors. (Issue 8377)
- Removed '@scope' prelude from specificity calculation. (Issue 8500)
- Specified how name-defining at-rules behave in '@scope'. (Issue 6895)
- Added implicit scopes by making "<scope-start>" optional. (Issue 6606)
- Disallowed pseudo-elements in the '@scope' prelude. (Issue 7382)
- Removed selector scoping notation. (Issue 7709)
- Scoping limit elements are excluded from the resulting scope. (Issue 6577)

3.2. Additions Since Level 5

The following features have been added since Level 5:

- The definition of a scope, as described by a combination of <scope-start> and <scope-end> selectors.
- The in-scope (':in()') pseudo-class for selecting with lower-boundaries
- The '@scope' rule for creating scoped stylesheets
- The definition of scope proximity in the cascade

3.3. Additions Since Level 4

The following features have been added since Level 4:

- Added cascade layers to the cascade sort criteria (and defined style attributes as a distinct step of the cascade sort criteria so that they interact appropriately).
- Introduced the '@layer' rule for defining cascade layers.
- Added 'layer'/'layer()' option to '@import' definition.
- Introduced the 'revert-layer' keyword for rolling back values to previous layers.

§ 3.4. Additions Since Level 3

The following features have been added since Level 3:

- Introduced 'revert' keyword, for rolling back the cascade.
- Introduced 'supports()' syntax for supports-conditional '@import' rules.
- Added encapsulation context to the cascade sort criteria to accommodate Shadow DOM. [DOM]
- Defined the property two aliasing mechanisms CSS uses to support legacy syntaxes. See *CSS Cascading 4* § 3.1 Property Aliasing.

§ 3.5. Additions Since Level 2

The following features have been added since Level 2:

- The 'all' shorthand
- The 'initial' keyword
- The 'unset' keyword
- Incorporation of animations and transitions into the cascade.

§ Acknowledgments

David Baron, Tantek Çelik, Keith Grant, Giuseppe Gurgone, Theresa O'Connor, Florian Rivoal, Noam Rosenthal, Simon Sapin, Jen Simmons, Nicole Sullivan, Lea Verou, and Boris Zbarsky contributed to this specification.

§ 4. Privacy Considerations

- User preferences and UA defaults expressed via application of style rules are exposed by the cascade process, and can be inferred from the computed styles they apply to a document.

§ 5. Security Considerations

- The cascade process does not distinguish between same-origin and cross-origin stylesheets, enabling the content of cross-origin stylesheets to be inferred from the computed styles they apply to a document.
- The '@import' rule does not apply the CORS protocol to loading cross-origin stylesheets,

instead allowing them to be freely imported and applied.

- The '@import' rule assumes that resources without `Content-Type` metadata (or any same-origin file if the host document is in quirks mode) are `text/css`, potentially allowing arbitrary files to be imported into the page and interpreted as CSS, potentially allowing sensitive data to be inferred from the computed styles they apply to a document.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words "for example" or are set apart from the normative text with `class="example"`, like this:

> EXAMPLE 10 ¶
>
> This is an example of an informative example.

Informative notes begin with the word "Note" and are set apart from the normative text with `class="note"`, like this:

> Note, this is an informative note.

Advisements are normative sections styled to evoke special attention and are set apart from other normative text with `<strong class="advisement">`, like this:

> **UAs MUST provide an accessible alternative.**

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
: A CSS style sheet.

renderer
: A UA that interprets the semantics of a style sheet and renders documents that use them.

authoring tool
: A UA that writes a style sheet.

A style sheet is conformant to this specification if all of its statements that use syntax defined in this module are valid according to the generic CSS grammar and the individual grammars of each feature defined in this module.

A renderer is conformant to this specification if, in addition to interpreting the style sheet as defined by the appropriate specifications, it supports all the features defined by this specification by parsing them correctly and rendering the document accordingly. However, the inability of a UA to correctly render a document due to limitations of the device does not make the UA non-conformant. (For example, a UA is not required to render color on a monochrome monitor.)

An authoring tool is conformant to this specification if it writes style sheets that are syntactically correct according to the generic CSS grammar and the individual grammars of each feature in this module, and meet all other conformance requirements of style sheets as described in this module.

§ Partial implementations

So that authors can exploit the forward-compatible parsing rules to assign fallback values, CSS renderers **must** treat as invalid (and ignore as appropriate) any at-rules, properties, property values, keywords, and other syntactic constructs for which they have no usable level of support. In particular, user agents **must not** selectively ignore unsupported component values and honor supported values in a single multi-value property declaration: if any value is considered invalid (as unsupported values must be), CSS requires that the entire declaration be ignored.

§ Implementations of Unstable and Proprietary Features

To avoid clashes with future stable CSS features, the CSSWG recommends following best practices for the implementation of unstable features and proprietary extensions to CSS.

§ Non-experimental implementations

Once a specification reaches the Candidate Recommendation stage, non-experimental implementations are possible, and implementors should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec.

To establish and maintain the interoperability of CSS across implementations, the CSS Working Group requests that non-experimental CSS renderers submit an implementation report (and, if necessary, the testcases used for that implementation report) to the W3C before releasing an unprefixed implementation of any CSS features. Testcases submitted to W3C are subject to review and correction by the CSS Working Group.

Further information on submitting testcases and implementation reports can be found from on the CSS Working Group's website at https://www.w3.org/Style/CSS/Test/. Questions should be directed to the public-css-testsuite@w3.org mailing list.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[CSS-ANIMATIONS-1] defines the following terms:

@keyframes

[CSS-CASCADE-5] defines the following terms:

- @import
- @layer
- all
- author origin
- cascade layers
- cascaded value
- declared value
- initial
- revert
- revert-layer
- unset
- user origin
- user-agent origin

[CSS-COLOR-4] defines the following terms:

- plum
- teal

[CSS-CONDITIONAL-5] defines the following terms:

- supports()

[CSS-FONTS-5] defines the following terms:

- @font-face

[CSS-NESTING-1] defines the following terms:

- &
- contain the nesting selector

[CSS-SCOPING-1] defines the following terms:

- shadow host
- tree context

[CSS-SYNTAX-3] defines the following terms:

- <stylesheet>
- at-rule
- block at-rule
- declaration
- style rule

[CSS-VALUES-3] defines the following terms:

- ?

[CSSOM-1] defines the following terms:

- declarations
- owner node

[DOM] defines the following terms:

- inclusive descendant
- node
- node tree
- parent element
- root
- shadow tree
- shadow-including tree order

[FETCH] defines the following terms:

- cors protocol

[HTML] defines the following terms:

- img
- style

[SELECTORS-4] defines the following terms:

:scope

<forgiving-selector-list>

<relative-selector-list>

<selector-list>

combinator

descendant combinator

pseudo-element

relative selector

scoped selector

scoping root

selector

subject

§ References

§ Normative References

[CSS-ANIMATIONS-1]
David Baron; et al. *CSS Animations Level 1*. 2 March 2023. WD. URL: https://www.w3.org/TR/css-animations-1/

[CSS-CASCADE-4]
Elika Etemad; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 4*. 13 January 2022. CR. URL: https://www.w3.org/TR/css-cascade-4/

[CSS-CASCADE-5]
Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 5*. 13 January 2022. CR. URL: https://www.w3.org/TR/css-cascade-5/

[CSS-CONDITIONAL-5]
David Baron; Elika Etemad; Chris Lilley. *CSS Conditional Rules Module Level 5*. 21 December 2021. WD. URL: https://www.w3.org/TR/css-conditional-5/

[CSS-FONTS-5]
Myles Maxfield; Chris Lilley. *CSS Fonts Module Level 5*. 21 December 2021. WD. URL: https://www.w3.org/TR/css-fonts-5/

[CSS-NESTING-1]
Tab Atkins Jr.; Adam Argyle. *CSS Nesting Module*. 14 February 2023. WD. URL: https://www.w3.org/TR/css-nesting-1/

[CSS-SCOPING-1]
Tab Atkins Jr.; Elika Etemad. *CSS Scoping Module Level 1*. 3 April 2014. WD. URL:

https://www.w3.org/TR/css-scoping-1/

[CSS-SYNTAX-3]
Tab Atkins Jr.; Simon Sapin. *CSS Syntax Module Level 3*. 24 December 2021. CR. URL: https://www.w3.org/TR/css-syntax-3/

[CSS-TRANSITIONS-1]
David Baron; et al. *CSS Transitions*. 11 October 2018. WD. URL: https://www.w3.org/TR/css-transitions-1/

[CSS-VALUES-3]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 3*. 1 December 2022. CR. URL: https://www.w3.org/TR/css-values-3/

[CSSOM-1]
Daniel Glazman; Emilio Cobos Álvarez. *CSS Object Model (CSSOM)*. 26 August 2021. WD. URL: https://www.w3.org/TR/cssom-1/

[CSSSTYLEATTR]
Tantek Çelik; Elika Etemad. *CSS Style Attributes*. 7 November 2013. REC. URL: https://www.w3.org/TR/css-style-attr/

[DOM]
Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

[FETCH]
Anne van Kesteren. *Fetch Standard*. Living Standard. URL: https://fetch.spec.whatwg.org/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[SELECT]
Tantek Çelik; et al. *Selectors Level 3*. 6 November 2018. REC. URL: https://www.w3.org/TR/selectors-3/

[SELECTORS-4]
Elika Etemad; Tab Atkins Jr.. *Selectors Level 4*. 11 November 2022. WD. URL: https://www.w3.org/TR/selectors-4/

§ Informative References

[CSS-COLOR-4]
Tab Atkins Jr.; Chris Lilley; Lea Verou. *CSS Color Module Level 4*. 1 November 2022. CR. URL: https://www.w3.org/TR/css-color-4/

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

§ Issues Index

ISSUE 1 This is a diff spec over CSS Cascading and Inheritance Level 5. It is currently an Exploratory Working Draft: if you are implementing anything, please use Level 5 as a reference. We will merge the Level 5 text into this draft once it reaches CR.

ISSUE 2 Does scope proximity belong above or below specificity in the cascade? [Issue #6790]

ISSUE 3 *CSS Cascading 5* § 6.2 Cascading Origins

ISSUE 4 *CSS Cascading 5* § 6.3 Important Declarations: the !important annotation

ISSUE 5 *CSS Cascading 5* § 6.4 Cascade Layers

ISSUE 6 *CSS Cascading 5* § 6.4.3 Layer Ordering

ISSUE 7 Should scoping limits be added to the definition of scoped selectors?

ISSUE 8 Should '@scope' use strong or weak scoping proximity? Strong scoping proximity causes declarations to be weighted more strongly by scope proximity than by their selector's specificity. Weak scoping proximity causes declarations of the same specificity to be weighted by proximity to their scoping root before falling back to source ordering, but declarations of higher specificity win over more tightly-scoped declarations. The Working Group currently leans towards weak proximity, and recommends that as a starting point for prototypes. [Issue #6790]

ISSUE 9 Should the scoped descendant combinator use strong or weak scoping proximity? Should it even exist? It's defined here to work the way many people expected the regular descendant combinator to work...

ISSUE 10 *CSS Cascading 5* § 6.4 Cascade Layers