CSS Cascading and Inheritance Level 5

W3C Candidate Recommendation Snapshot, 13 January 2022

▼ **More details about this document**

This version:
https://www.w3.org/TR/2022/CR-css-cascade-5-20220113/

Latest published version:
https://www.w3.org/TR/css-cascade-5/

Editor's Draft:
https://drafts.csswg.org/css-cascade-5/

Previous Versions:
https://www.w3.org/TR/2021/WD-css-cascade-5-20211015/
https://www.w3.org/TR/2021/WD-css-cascade-5-20210829/
https://www.w3.org/TR/2021/WD-css-cascade-5-20210608/
https://www.w3.org/TR/2021/WD-css-cascade-5-20210319/
https://www.w3.org/TR/2021/WD-css-cascade-5-20210119/

History:
https://www.w3.org/standards/history/css-cascade-5

Implementation Report:
https://wpt.fyi/results/css/css-cascade

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 cascade layers.

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/.

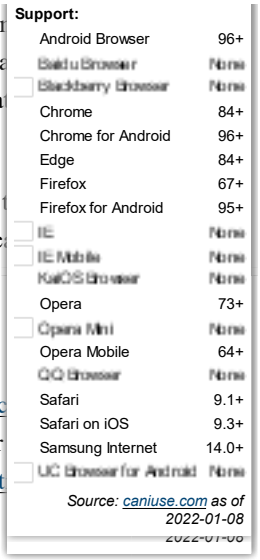
This document was published by the CSS Working Group as a **Candidate Recommendation Snapshot** using the Recommendation track. Publication as a Candidate Recommendation does not imply endorsement by W3C and its

Members. A Candidate Recommendation Snapshot has received wide review, is intended to gather implem
experience, and has commitments from Working Group members to royalty-free licensing for implementa
document is intended to become a W3C Recommendation; it will remain a Candidate Recommendation a
March 2022 to gather additional feedback.

Please send feedback by filing issues in GitHub (preferred), including the spec code "css-cascade" in the t
"[css-cascade] *...summary of comment...*". All issues and comments are archived. Alternately, feedback c
(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
disclosures made in connection with the deliverables of the group; that page also includes instructions for
patent. An individual who has actual knowledge of a patent which the individual believes contains Essent
disclose the information in accordance with section 6 of the W3C Patent Policy.



The following features are at-risk, and may be dropped during the CR period:

- the 'revert-layer' keyword

"At-risk" is a W3C Process term-of-art, and does not necessarily imply that the feature is in danger of being dropped or delayed. It means that the WG believes the feature may have difficulty being interoperably implemented in a timely manner, and marking it as such allows the WG to drop the feature if necessary when transitioning to the Proposed Rec stage, without having to publish a new Candidate Rec without the feature first.

Table of Contents

1. Introduction

CSS defines a finite set of parameters, called ***properties***, that direct the rendering of a document. Each property has a name (e.g., 'color', 'font-size', or 'border-style'), a value space (e.g., <color>, <length-percentage>, '[solid | dashed | dotted | …]'), and a defined behavior on the rendering of the document. Properties values are assigned to various parts of the document via property declarations, which assign the property a value (e.g. 'red', '12pt', 'dotted') for the associated element or box.

One of the fundamental design principles of CSS is cascading, which allows several style sheets to influence the presentation of a document. When different declarations try to set a value for the same element/property combination, the conflicts must somehow be resolved.

The opposite problem arises when no declarations try to set a value for an element/property combination. In this case, a value is be found by way of inheritance or by looking at the property's initial value.

The cascading and defaulting process takes a set of declarations as input, and outputs a specified value for each property on each element.

The rules for finding the specified value for all properties on all elements in the document are described in this specification. The rules for finding the specified values in the page context and its margin boxes are described in [css-page-3].

1.1. Module Interactions

This section is normative.

This module replaces and extends the rules for assigning property values, cascading, and inheritance defined in [CSS2] chapter 6.

Other CSS modules may expand the definitions of some of the syntax and features defined here. For example, the Media Queries Level 4 specification, when combined with this module, expands the definition of the <media-query> value type as used in this specification.

For the purpose of this specification, text nodes are treated as element children of their associated element, and possess the full set of properties; since they cannot be targeted by selectors all of their computed values are assigned by defaulting.

2. Importing Style Sheets: the '@import' rule

The ***'@import'*** rule allows users to import style rules from other style sheets. If an '@import' rule refers to a valid stylesheet, user agents must treat the contents of the stylesheet as if they were written in place of the '@import' rule, with two exceptions:

- If a feature (such as the '@namespace' rule) *explicitly* defines that it only applies to a particular stylesheet, and not any imported ones, then it doesn't apply to the imported stylesheet.
- If a feature relies on the relative ordering of two or more constructs in a stylesheet (such as the requirement that

'@namespace' rules must not have any other rules other than '@import' preceding it), it only applies between constructs in the same stylesheet.

> EXAMPLE 1
> For example, declarations in style rules from imported stylesheets interact with the cascade as if they were written literally into the stylesheet at the point of the '@import'.

Any '@import' rules must precede all other valid at-rules and style rules in a style sheet (ignoring '@charset' and empty '@layer' definitions) and must not have any other valid at-rules or style rules between it and previous '@import' rules, or else the '@import' rule is invalid. The syntax of '@import' is:

```
@import [ <url> | <string> ]
        [ layer | layer(<layer-name>) ]?
        [ supports( [ <supports-condition> | <declaration> ] ) ]?
        <media-query-list>? ;
```

where:

- the <url> or <string> gives the URL of the style sheet to be imported.
- the optional 'layer' keyword or 'layer()' function assigns the contents of the style sheet into its own anonymous cascade layer or into the named cascade layer.

  The layer is added to the layer order even if the import fails to load the stylesheet, but is subject to any import conditions (just as if declared by an '@layer' rule wrapped in the appropriate conditional group rules).
- the optional [<supports-condition>|<declaration>] and <media-query-list> (collectively, the ***import conditions***) state the conditions under which it applies.

> EXAMPLE 2
> The following conditional '@import' rule only loads the style sheet when the UA supports 'display: flex', and only applies the style sheet on a handheld device with a maximum viewport width of 400px.
>
> ```
> @import url("narrow.css") supports(display: flex) handheld and (max-width: 400px);
> ```

> EXAMPLE 3
> The following layer imports load the style sheets into the 'framework.component' layer, and an un-named layer, respectively:
>
> ```
> @import url("tabs.css") layer(framework.component);
> @import url("override.css") layer();
> ```

If a <string> is provided, it must be interpreted as a <url> with the same value.

> EXAMPLE 4
> The following lines are equivalent in meaning and illustrate both '@import' syntaxes (one with 'url()' and one with a bare string):
>
> ```
> @import "mystyle.css";
> @import url("mystyle.css");
> ```

§ 2.1. Conditional '@import' Rules

The import conditions allow the import to be media– or feature-support–dependent. In the absence of any import conditions, the import is unconditional. (Specifying 'all' for the <media-query-list> has the same effect.) If the import conditions do not match, the rules in the imported stylesheet do not apply, exactly as if the imported stylesheet were wrapped in '@media' and/or '@supports' blocks with the given conditions.

EXAMPLE 5
The following rules illustrate how '@import' rules can be made media-dependent:

```
@import url("fineprint.css") print;
@import url("bluish.css") projection, tv;
@import url("narrow.css") handheld and (max-width: 400px);
```

User agents may therefore avoid fetching a conditional import as long as the import conditions do not match. Additionally, if a <supports-condition> blocks the application of the imported style sheet, the UA *must not* fetch the style sheet (unless it is loaded through some other link) and *must* return null for the import rule's CSSImportRule.styleSheet value (even if it is loaded through some other link).

EXAMPLE 6
The following rule illustrates how an author can provide fallback rules for legacy user agents without impacting network performance on newer user agents:

```
@import url("fallback-layout.css") supports(not (display: flex));
@supports (display: flex) {
  ...
}
```

The import conditions are given by <media-query-list>, which is parsed and interpreted as a media query list, and <supports-condition>, is parsed and interpreted as a [[supports query]]. If a <declaration> is given in place of a <supports-condition>, it must be interpreted as a <supports-decl> (i.e. the extra set of parentheses is implied) and treated as a <supports-condition>.

EXAMPLE 7
For example, the following two lines are equivalent:

```
@import "mystyle.css" supports(display: flex);
@import "mystyle.css" supports((display: flex));
```

The evaluation and full syntax of the import conditions are defined by the Media Queries [MEDIAQ] and CSS Conditional Rules [CSS-CONDITIONAL-3] specifications.

§ 2.2. Processing Stylesheet Imports

When the same style sheet is imported or linked to a document in multiple places, user agents must process (or act as though they do) each link as though the link were to an independent style sheet.

Note: This does not place any requirements on resource fetching, only how the style sheet is reflected in the CSSOM and used in specs such as this one. Assuming appropriate caching, it is perfectly appropriate for a UA to fetch a style sheet only once, even though it's linked or imported multiple times.

The cascade origin of an imported style sheet is the cascade origin of the style sheet that imported it.

The environment encoding of an imported style sheet is the encoding of the style sheet that imported it. [css-syntax-3]

§ 2.3. Content-Type of CSS Style Sheets

The processing of imported style sheets depends on the actual type of the linked resource:

- If the resource does not have Content-Type metadata, the type is treated as `text/css`.
- If the host document is in quirks mode, and the host document's origin is same origin with the linked resource response's URL's origin, the type is treated as `text/css`.
- Otherwise, the type is determined from its Content-Type metadata.

If the linked resource's type is `text/css`, it must be interpreted as a CSS style sheet. Otherwise, it must be interpreted as a network error.

§ 3. Shorthand Properties

Some properties are ***shorthand properties***, meaning that they allow authors to specify the values of several properties with a single property. A shorthand property sets all of its ***longhand sub-properties***, exactly as if expanded in place.

When values are omitted from a shorthand form, unless otherwise defined, each "missing" sub-property is assigned its initial value.

This means that a shorthand property declaration always sets *all* of its sub-properties, even those that are not explicitly set. Carelessly used, this might result in inadvertently resetting some sub-properties. Carefully used, a shorthand can guarantee a "blank slate" by resetting sub-properties inadvertently cascaded from other sources.

For example, writing 'background: green' rather than 'background-color: green' ensures that the background color overrides any earlier declarations that might have set the background to an image with 'background-image'.

EXAMPLE 8

For example, the CSS Level 1 'font' property is a shorthand property for setting 'font-style', 'font-variant', 'font-weight', 'font-size', 'line-height', and 'font-family' all at once. The multiple declarations of this example:

```
h1 {
  font-weight: bold;
  font-size: 12pt;
  line-height: 14pt;
  font-family: Helvetica;
  font-variant: normal;
  font-style: normal;
}
```

can therefore be rewritten as

```
h1 { font: bold 12pt/14pt Helvetica }
```

As more 'font' sub-properties are introduced into CSS, the shorthand declaration resets those to their initial values as well.

In some cases, a shorthand might have different syntax or special keywords that don't directly correspond to values of its sub-properties. (In such cases, the shorthand will explicitly define the expansion of its values.)

In other cases, a property might be a ***reset-only sub-property*** of the shorthand: Like other sub-properties, it is reset to its initial value by the shorthand when unspecified, but the shorthand might not include syntax to set the sub-property to any of its other values. For example, the 'border' shorthand resets 'border-image' to its initial value of 'none', but has no syntax to set it to anything else. [css-backgrounds-3]

If a shorthand is specified as one of the CSS-wide keywords [css-values-3], it sets all of its sub-properties to that keyword, including any that are reset-only sub-properties. (Note that these keywords cannot be combined with other values in a single declaration, not even in a shorthand.)

Declaring a shorthand property to be '!important' is equivalent to declaring all of its sub-properties to be '!important'.

§ 3.1. Property Aliasing

Properties sometimes change names after being supported for a while, such as vendor-prefixed properties being standardized. The original name still needs to be supported for compatibility reasons, but the new name is preferred. To accomplish this, CSS defines two different ways of "aliasing" old syntax to new syntax.

legacy name aliases

When the old property's value syntax is identical to that of the new property, the two names are aliased with an operation on par with case-mapping: at parse time, the old property is converted into the new property. This conversion also applies in the CSSOM, both for string arguments and property accessors: requests for the old property name transparently transfer to the new property name instead.

> EXAMPLE 9
> For example, if 'old-name' is a legacy name alias for 'new-name', `getComputedStyle(el).oldName` will return the computed style of the `newName` property, and `el.style.setPropertyValue("old-name", "value")` will set the 'new-name' property to `"value"`.

legacy shorthands

When the old property has a distinct syntax from the new property, the two names are aliased using the shorthand mechanism. These shorthands are defined to be legacy shorthands, and their use is *deprecated*. They otherwise behave exactly as regular shorthands, except that the CSSOM will not use them when serializing declarations. [CSSOM]

> EXAMPLE 10
> For example, the 'page-break-*' properties are legacy shorthands for the 'break-*' properties (see *CSS Fragmentation 3* § 3.4 Page Break Aliases: the page-break-before, page-break-after, and page-break-inside properties).
>
> Setting 'page-break-before: always' expands to 'break-before: page' at parse time, like other shorthands do. Similarly, if 'break-before: page' is set, calling `getComputedStyle(el).pageBreakBefore` will return `"always"`. However, when serializing a style block (see *CSSOM 1* § 6.7.2 Serializing CSS Values), the 'page-break-before' property will never be chosen as the shorthand to serialize to, regardless of whether it or 'break-before' was specified; instead, 'break-before' will always be chosen.

§ 3.2. Resetting All Properties: the 'all' property

Name:	***'all'***
Value:	initial | inherit | unset | revert | revert-layer
Initial:	see individual properties

Applies to:	see individual properties
Inherited:	see individual properties
Percentages:	see individual properties
Computed value:	see individual properties
Animation type:	see individual properties
Canonical order:	per grammar

The 'all' property is a shorthand that resets *all* CSS properties except 'direction' and 'unicode-bidi'. It only accepts the CSS-wide keywords. It does not reset custom properties [css-variables-1].

Note: The excepted CSS properties 'direction' and 'unicode-bidi' are actually markup-level features, and should not be set in the author's style sheet. (They exist as CSS properties only to style document languages not supported by the UA.) Authors should use the appropriate markup, such as HTML's `dir` attribute, instead. [css-writing-modes-3]

EXAMPLE 11

For example, if an author specifies 'all: initial' on an element, it will block all inheritance and reset all properties, as if no rules appeared in the author, user, or user-agent levels of the cascade.

This can be useful for the root element of a "widget" included in a page, which does not wish to inherit the styles of the outer page. Note, however, that any "default" style applied to that element (such as, e.g. 'display: block' from the UA style sheet on block elements such as `<div>`) will also be blown away.

§ 4. Value Processing

Once a user agent has parsed a document and constructed a document tree, it must assign, to every element in the tree, and correspondingly to every box in the formatting structure, a value to every property that applies to the target media type.

The final value of a CSS property for a given element or box is the result of a multi-step calculation:

1. First, all the declared values applied to an element are collected, for each property on each element. There may be zero or many declared values applied to the element.
2. Cascading yields the cascaded value. There is at most one cascaded value per property per element.
3. Defaulting yields the specified value. Every element has exactly one specified value per property.
4. Resolving value dependencies yields the computed value. Every element has exactly one computed value per property.
5. Formatting the document yields the used value. An element only has a used value for a given property if that property applies to the element.
6. Finally, the used value is transformed to the actual value based on constraints of the display environment. As with the used value, there may or may not be an actual value for a given property on an element.

Elements that are not connected or are not part of the document's flattened element tree do not participate in CSS value processing, and do not have declared, cascaded, specified, computed, used, or actual values, even if they potentially have style declarations assigned to them (for example, by a `style` attribute).

§ 4.1. Declared Values

Each property declaration applied to an element contributes a ***declared value*** for that property associated with the element. See Filtering Declarations for details.

These values are then processed by the cascade to choose a single "winning value".

§ 4.1.1. Value Aliasing

Some property values have ***legacy value aliases***: at parse time, the legacy syntax is converted into the new syntax, resulting in a declared value different from the parsed input. These aliases are typically used for handling legacy compatibility requirements, such as converting vendor-prefixed values to their standard equivalents.

§ 4.2. Cascaded Values

The ***cascaded value*** represents the result of the cascade: it is the declared value that wins the cascade (is sorted first in the output of the cascade). If the output of the cascade is an empty list, there is no cascaded value.

§ 4.3. Specified Values

The ***specified value*** is the value of a given property that the style sheet authors intended for that element. It is the result of putting the cascaded value through the defaulting processes, guaranteeing that a specified value exists for every property on every element.

In many cases, the specified value is the cascaded value. However, if there is no cascaded value at all, the specified value is defaulted. The CSS-wide keywords are handled specially when they are the cascaded value of a property, setting the specified value as required by that keyword, see § 7.3 Explicit Defaulting.

§ 4.4. Computed Values

The ***computed value*** is the result of resolving the specified value as defined in the "Computed Value" line of the property definition table, generally absolutizing it in preparation for inheritance.

> Note: The computed value is the value that is transferred from parent to child during inheritance. For historical reasons, it is not necessarily the value returned by the `getComputedStyle()` function, which sometimes returns used values. [CSSOM] Furthermore, the computed value is an abstract data representation: their definitions reflect that data representation, not how that data is serialized. For example, serialization rules often allow omitting certain values which are implied during parsing; but those values are nonetheless part of the computed value.

EXAMPLE 12

A specified value can be either absolute (i.e., not relative to another value, as in 'red' or '2mm') or relative (i.e., relative to another value, as in 'auto', '2em'). Computing a relative value generally absolutizes it:

- values with relative units ('em', 'ex', 'vh', 'vw') must be made absolute by multiplying with the appropriate reference size
- certain keywords (e.g., 'smaller', 'bolder') must be replaced according to their definitions
- percentages on some properties must be multiplied by a reference value (defined by the property)
- valid relative URLs must be resolved to become absolute.

See examples (f), (g) and (h) in the table below.

Note: In general, the computed value resolves the specified value as far as possible without laying out the document or performing other expensive or hard-to-parallelize operations, such as resolving network requests or retrieving values other than from the element and its parent.

The computed value exists even when the property does not apply. However, some properties may change how they determine the computed value based on whether the property applies to the element.

§ 4.5. Used Values

The ***used value*** is the result of taking the computed value and completing any remaining calculations to make it the absolute theoretical value used in the formatting of the document.

EXAMPLE 13

For example, a declaration of 'width: auto' can't be resolved into a length without knowing the layout of the element's ancestors, so the computed value is 'auto', while the used value is an absolute length, such as '100px'. [CSS2]

EXAMPLE 14

As another example, a `<div>` might have a computed 'break-before' value of 'auto', but acquire a used 'break-before' value of 'page' by propagation from its first child. [css-break-3]

If a property does not apply to this element or box type then it has no used value for that property.

EXAMPLE 15

For example, the 'flex' property has no used value on elements that aren't flex items.

§ 4.5.1. Applicable Properties

If a property does not ***apply to*** an element or box type—as noted in its "Applies to" line—this means it does not directly take effect on that type of box or element.

Note: A property that does not apply can still have *indirect* formatting effects if its computed value affects the computation of other properties that do apply; and of course its computed value, which always exists, can still inherit to descendants and take effect on them.

EXAMPLE 16

Even though 'writing-mode' and 'text-orientation' do not apply to table rows (they do not affect how the table row or its children are laid out), setting them on such boxes will still affect the calculation of font relative units such as 'ch', and thus possibly any property that takes a <length>.

EXAMPLE 17

Setting 'text-transform' on an HTML <p> element (which is 'display: block' by default) will have an effect, even though 'text-transform' only applies to inline boxes, because the property inherits into the paragraph's anonymous root inline box and applies to the text it contains.

Note: A property defined to apply to "all elements" applies to all elements and display types, but not necessarily to all pseudo-element types, since pseudo-elements often have their own specific rendering models or other restrictions. The '::before' and '::after' pseudo-elements, however, are defined to generate boxes almost exactly like normal elements and are therefore defined accept all properties that apply to "all elements". See [CSS-PSEUDO-4] for more information about pseudo-elements.

§ 4.6. Actual Values

A used value is in principle ready to be used, but a user agent may not be able to make use of the value in a given environment. For example, a user agent may only be able to render borders with integer pixel widths and may therefore have to approximate the used width. Also, the font size of an element may need adjustment based on the availability of fonts or the value of the 'font-size-adjust' property. The ***actual value*** is the used value after any such adjustments have been made.

Note: By probing the actual values of elements, much can be learned about how the document is laid out. However, not all information is recorded in the actual values. For example, the actual value of the 'page-break-after' property does not reflect whether there is a page break or not after the element. Similarly, the actual value of 'orphans' does not reflect how many orphan lines there is in a certain element. See examples (j) and (k) in the table below.

§ 4.7. Examples

Examples of CSS Value Computation

	Property	Winning declaration	Cascaded value	Specified value	Computed value	Used value	Actual value
(a)	**'text-align'**	`text-align: left`	'left'	'left'	'left'	'left'	'left'
(b)	**'border-top-width'**, **'border-right-width'**, **'border-bottom-width'**, **'border-left-width'**	`border-width: inherit`	'inherit'	'4.2px'	'4.2px'	'4.2px'	'4px'
(c)	**'width'**	(none)	(none)	'auto' (initial value)	'auto'	'120px'	'120px'
(d)	**'list-style-position'**	`list-style-position: inherit`	'inherit'	'inside'	'inside'	'inside'	'inside'
(e)	**'list-style-position'**	`list-style-position: initial`	'initial'	'outside' (initial value)	'outside'	'outside'	'outside'
(f)	**'font-size'**	`font-size: 1.2em`	'1.2em'	'1.2em'	'14.1px'	'14.1px'	'14px'
(g)	**'width'**	`width: 80%`	'80%'	'80%'	'80%'	'354.2px'	'354px'
(h)	**'width'**	`width: auto`	'auto'	'auto'	'auto'	'134px'	'134px'
(i)	**'height'**	`height: auto`	'auto'	'auto'	'auto'	'176px'	'176px'
(j)	**'page-break-after'**	(none)	(none)	'auto' (initial value)	'auto'	'auto'	'auto'
(k)	**'orphans'**	`orphans: 3`	'3'	'3'	'3'	'3'	'3'

§ 4.8. Per-Fragment Value Processing

Certain CSS features can interfere with value processing on a per-fragment basis. See for example *CSS Pseudo-Elements 4* § 2.1.3 Inheritance and the ::first-line Pseudo-element which alters inheritance for fragments within the '::first-line' pseudo-element. In such cases, where individual fragments are given different specified values, any values that resolve based on the computed value of other properties (such as 'currentcolor' or 'em' units) are resolved per box fragment. Subsequent value processing proceeds as normal in each fragment.

APIs that assume a singular value per box (rather than per box fragment) must ignore the effects of non-tree-abiding pseudo-elements. (For example, '::first-line' styles have no effect on the value returned by `getComputedStyle()`.)

EXAMPLE 18

For example, given the following markup:

```
<div><span>First line<br />Second line</span></div>
<div><span>First line</span></div>
<div>First line<br><span>Second line</span></div>
<style>
div { color: blue; }
div::first-line { color: yellow; }
span { border: thin solid currentcolor; }
</style>
```

In each `<div>`, the "First line" text is yellow and the "Second line" text is blue; the border for each fragment of the `<span>`s that wrap each line matches that color.

However, `getComputedStyle()` on all three of the spans will return `"blue"` for 'border-color', because the effects of a '::first-line' pseudo-element are ignored for APIs that aren't fragment-aware.

§ 5. Filtering

In order to find the declared values, implementations must first identify all declarations that apply to each element. A declaration applies to an element if:

- It belongs to a style sheet that currently applies to this document.
- It is not qualified by a conditional rule [CSS-CONDITIONAL-3] with a false condition.
- It belongs to a style rule whose selector matches the element. [SELECT] (Taking scoping into account, if necessary.)
- It is syntactically valid: the declaration's property is a known property name, and the declaration's value matches the syntax for that property.

The values of the declarations that apply form, for each property on each element, a list of declared values. The next section, the cascade, prioritizes these lists.

§ 6. 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.

§ 6.1. Cascade Sorting Order

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

¶ **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.

¶ **Element-Attached Styles**

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.

See [css-style-attr].

> Note: Non-CSS presentational hints (such as presentational markup) are handled separately, see § 6.5 Precedence of Non-CSS Presentational Hints.

¶ **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 ordered by order of appearance. When comparing declarations that belong to different layers, then for normal rules the declaration whose cascade layer is last wins, and for important rules the declaration whose cascade layer is first wins.

> Note: This follows the same logic used for layering normal and important origins, so that the '!important' flag maintains the same "override" purpose in both settings.

¶ **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.

¶ **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.

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

§ 6.2. Cascading Origins

Each style rule has a ***cascade origin***, which determines where it enters the cascade. CSS defines three core origins:

Author Origin
The author specifies style sheets for a source document according to the conventions of the document language. For instance, in HTML, style sheets may be included in the document or linked externally.

User Origin
The user may be able to specify style information for a particular document. For example, the user may specify a file that contains a style sheet or the user agent may provide an interface that generates a user style sheet (or behaves as if it did).

User-Agent Origin
Conforming user agents must apply a default style sheet (or behave as if they did). A user agent's default style sheet should present the elements of the document language in ways that satisfy general presentation expectations for the document language (e.g., for visual browsers, the EM element in HTML is presented using an italic font). See e.g. the HTML user agent style sheet. [HTML]

Extensions to CSS define the following additional origins:

Animation Origin
CSS Animations [css-animations-1] generate "virtual" rules representing their effects when running.

Transition Origin
Like CSS Animations, CSS Transitions [css-transitions-1] generate "virtual" rules representing their effects when running.

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

CSS attempts to create a balance of power between author and user style sheets. By default, rules in an author's style sheet override those in a user's style sheet, which override those in the user-agent's default style sheet. To balance this, a declaration can be marked important, which increases its weight in the cascade and inverts the order of precedence.

A declaration is ***important*** if it has a '!important' annotation as defined by [css-syntax-3], i.e. if the last two (non-whitespace, non-comment) tokens in its value are the delimiter token '!' followed by the identifier token 'important'. All other declarations are ***normal*** (non-important).

EXAMPLE 19

```
[hidden] { display: none !important; }
```

An important declaration takes precedence over a normal declaration. Author and user style sheets may contain important declarations, with user-origin important declarations overriding author-origin important declarations. This CSS feature improves accessibility of documents by giving users with special requirements (large fonts, color combinations, etc.) control over presentation.

Important declarations from all origins take precedence over animations. This allows authors to override animated values in important cases. (Animated values normally override all other rules.) [css-animations-1]

User-agent style sheets may also contain important declarations. These override all author and user declarations.

EXAMPLE 20

The first rule in the user's style sheet in the following example contains an '!important' declaration, which overrides the corresponding declaration in the author's style sheet. The declaration in the second rule will also win due to being marked '!important'. However, the third declaration in the user's style sheet is not '!important' and will therefore lose to the second rule in the author's style sheet (which happens to set style on a shorthand property). Also, the third author rule will lose to the second author rule since the second declaration is '!important'. This shows that '!important' declarations have a function also within author style sheets.

```
/* From the user's style sheet */
p { text-indent: 1em !important }
p { font-style: italic !important }
p { font-size: 18pt }

/* From the author's style sheet */
p { text-indent: 1.5em !important }
p { font: normal 12pt sans-serif !important }
p { font-size: 24pt }
```

Property	Winning value
'text-indent'	'1em'
'font-style'	'italic'
'font-size'	'12pt'
'font-family'	'sans-serif'

§ 6.4. Cascade Layers

In the same way that cascade origins provide a balance of power between user and author styles, ***cascade layers*** provide a structured way to organize and balance concerns within a single origin. Rules within a single cascade layer cascade together, without interleaving with style rules outside the layer.

Authors can create layers to represent element defaults, third-party libraries, themes, components, overrides, and other styling concerns—and are able to re-order the cascade of layers in an explicit way, without altering selectors or specificity within each layer, or relying on source-order to resolve conflicts across layers.

EXAMPLE 21

For example, the following generates an explicit 'reset' layer, with lower cascade weight than any unlayered styles:

```
audio {
  /* specificity of 0,0,1 - implicit (final) layer */
  display: flex;
}

@layer reset {
  audio[controls] {
    /* specificity of 0,1,1 - explicit "reset" layer */
    display: block;
  }
}
```

The unlayered declarations on the `<audio>` element take precedence over the explicitly layered declarations on `audio[controls]` —even though the unlayered styles have a lower specificity, and come first in the source order.

Name-defining at-rules such as '@keyframes' or '@font-face' that are defined inside cascade layers also use the layer order when resolving name collisions.

EXAMPLE 22

For example, authors could override the animation from a framework, by providing keyframes with the same name in a higher-priority layer:

```
/* establish the layer order, so the "override" layer takes precedence */
@layer framework, override;

@layer override {
  @keyframes slide-left {
    from { translate: 0; }
    to { translate: -100% 0; }
  }
}

@layer framework {
  @keyframes slide-left {
    from { margin-left: 0; }
    to { margin-left: -100%; }
  }
}

.sidebar { animation: slide-left 300ms; }
```

In this case the 'override' layer has a higher cascade priority than the 'framework' layer, so `slide-left` will animate using the `translate` property rather than `margin-left`.

§ 6.4.1. Declaring Cascade Layers

Cascade layers can be declared:

- using an '@import' rule with the 'layer' keyword or 'layer()' function, assigning the contents of the imported file into that layer.

- using a @layer block at-rule, assigning its child style rules into that layer.
- using a @layer statement at-rule, declaring a named layer without assigning any rules.

ISSUE 1 Provide an attribute for assigning link or style elements to cascade layers? [Issue #w3c/csswg-drafts#5853]

§ 6.4.2. Layer Naming and Nesting

A cascade layer has a ***layer name***, which is an ordered list representing each level of layer nesting, each segment of which can be named (as a CSS identifier) or anonymous. (Thus, when a layer is nested inside of another layer, this concatenates their names.) One layer is nested in another when it is declared within the scope of another layer, e.g. an '@layer' rule inside another '@layer', a layered '@import' inside a layered import, or an '@layer' rule inside a layered import.

Layer names represent the same cascade layer if they contain the same segments in the same order; however anonymous segments have unique identities for each occurrence. Note that nesting can cause multiple layers to share the same anonymous segment.

EXAMPLE 23
Explicit layer identifiers provide a way to assign multiple style blocks to a single layer. In the following example, the contents of `headings.css` and `links.css` are cascaded within the same layer as the `audio[controls]` rule:

```
@import url(headings.css) layer(default);
@import url(links.css) layer(default);

@layer default {
  audio[controls] {
    display: block;
  }
}
```

EXAMPLE 24

In this example, the nested ‘framework.base’ layer is distinct from the top-level ‘base’ layer:

```
@layer base {
  p { max-width: 70ch; }
}

@layer framework {
  @layer base {
    p { margin-block: 0.75em; }
  }

  @layer theme {
    p { color: #222; }
  }
}
```

The resulting layers can be represented as a tree:

1. ‘base’
2. ‘framework’
    1. ‘base’
    2. ‘theme’

or as a flat list with nested identifiers:

1. ‘base’
2. ‘framework.base’
3. ‘framework.theme’

Syntactically, an explicit layer name is represented by the <layer-name> in ‘@layer’ and ‘@import’ rules, which is a period-separated list of <ident> tokens with no intervening white space:

<layer-name> = <ident> ['.' <ident>]*

The CSS-wide keywords are reserved for future use, and cause the rule to be invalid at parse time if used as an <ident> in the <layer-name>. When multiple identifiers are concatenated with a period, this is a shorthand representing those layers nested in order.

EXAMPLE 25

```
@layer framework {
  @layer default {
     p { margin-block: 0.75em; }
  }

  @layer theme {
    p { color: #222; }
  }
}

@layer framework.theme {
  /* These styles will be added to the theme layer inside the framework layer */
  blockquote { color: rebeccapurple; }
}
```

Note: A nested layer cannot "escape" its parent layer to reference layers outside itself.

6.4.2.1. Anonymous Layers

When a '@layer' rule omits its <layer-name>, or an '@import' rule uses the 'layer' keyword (which does not provide a <layer-name>), its layer name gains a unique anonymous segment; it therefore cannot be referenced from the outside.

EXAMPLE 26

Each occurrence of an anonymous layer declaration represents a unique cascade layer, thus:

- Multiple unnamed layer rules place their styles into separate layers, as each occurrence is referencing a distinct anonymous layer name.

```
@layer { /* layer 1 */ }
@layer { /* layer 2 */ }
```

- Within a single unnamed layer, child layers with the same name refer to the same cascade layer, because they share the same anonymous parent layer.

```
@layer {
  @layer foo { /* layer 1 */ }
  @layer foo { /* also layer 1 */ }
}
```

- Whereas in separate unnamed layers, child layers with the same name refer to different cascade layers, because they have distinct anonymous parent layers.

```
@layer {
  @layer foo { /* layer 1 */ }
}
@layer {
  @layer foo { /* layer 2 */ }
}
```

EXAMPLE 27

A layer declared without a <layer-name> does not provide any external hook for re-arranging or adding styles.

While this can be a mere convenience for brevity, it can also be used by teams as a way to force an organizing convention (all code in that layer must be defined in the same place), or by libraries wanting to merge & hide a set of internal "private" layers that they don't want exposed to author manipulation:

```
/* bootstrap-base.css */
/* unnamed wrapper layers around each sub-file */
@import url(base-forms.css) layer;
@import url(base-links.css) layer;
@import url(base-headings.css) layer;


/* bootstrap.css */
/* the internal names are hidden from access, subsumed in "base" */
@import url(bootstrap-base.css) layer(base);


/* author.css */
/* author has access to bootstrap.base layer, but not into unnamed layers */
@import url(bootstrap.css) layer(bootstrap);

/* Adds additional styles to the bootstrap layer: */
@layer bootstrap {...}
```

§ 6.4.3. Layer Ordering

Cascade layers are sorted by the order in which they first are declared, with nested layers grouped within their parent layers before any unlayered rules.

EXAMPLE 28

Given the following layer rules:

```
/* unlayered styles come last in the layer order */
h1 { color: darkslateblue; }

@layer reset.type {
  strong { font-weight: bold; }
}

@layer framework {
  .title { font-weight: 100; }

  @layer theme {
    h1, h2 { color: maroon; }
  }
}

@layer reset {
  [hidden] { display: none; }
}
```

The outer layers are sorted first, with any unlayered style rules added to an implicit outer layer which has higher priority than (comes after) the explicit layers:

1. 'reset'
2. 'framework'
3. (implicit outer layer)

Within each layer, nested layers are sorted in appearance order, and style rules without further nesting are similarly added to an implicit sub-layer after the explicitly nested layers:

1. 'reset.type'
2. 'reset' (implicit sub-layer)
3. 'framework.theme'
4. 'framework' (implicit sub-layer)
5. (implicit outer layer)

Layers that are defined inside of a conditional group rule do not contribute to the layer order unless the condition is true or unless the conditional group rule can evaluate differently for different elements in the document.

Note: Since the layer order is global to the document, any layers defined inside an element-sensitive conditional group rule need to be accommodated when establishing the global layer order, regardless of the rule's condition. Conditions that are global to the document, however (such as '@media' and '@supports') can accommodate such '@layer' rules conditionally.

EXAMPLE 29

For example, the following layer order will depend on which media conditions match:

```
@media (min-width: 30em) {
  @layer layout {
    .title { font-size: x-large; }
  }
}

@media (prefers-color-scheme: dark) {
  @layer theme {
    .title { color: white; }
  }
}

@layer theme, layout;
```

If the first media-query matches based on viewport dimensions, then the 'layout' layer will come first in the layer order. If the color-scheme preference query matches, or if neither condition is true, then 'theme' will come first in the layer order.

Authors who want to avoid this behavior can establish an explicit ordering of layers in advance, and avoid defining new layers inside conditional rules.

Note: Cascade layers are scoped to their origin and context, so the ordering of layers in the light DOM has no impact on the order of identically-named layers in the shadow DOM (and vice versa).

ISSUE 2 Allow authors to explicitly place unlayered styles in the layer order [Issue #6323]

6.4.4. Declaring Layers Inline: the '@layer' rule

The ***'@layer'*** rule declares a cascade layer, with the option to assign style rules.

6.4.4.1. Assigning Styles Inline: the '@layer' block at-rule

The '@layer' block at-rule assigns its child style rules to a particular named cascade layer. This block layer-assignment syntax is:

```
@layer <layer-name>? {
  <stylesheet>
}
```

Such '@layer' block rules have the same restrictions and processing as a conditional group rule [CSS-CONDITIONAL-3] with a true condition.

EXAMPLE 30

For example, '@layer' and '@media' can be mixed:

```
@layer framework {
  h1, h2 { color: maroon; background: white;}

  @media (prefers-color-scheme: dark) {
    h1, h2 { color: red; background: black; }
  }
}
```

Note: '@layer' block at-rules cannot be interleaved with '@import' rules.

§ 6.4.4.2. Declaring Without Styles: the *'@layer' statement at-rule*

The '@layer' rule can also be used to define new layers without assigning any style rules, by providing only the layer name:

```
@layer <layer-name>#;
```

Such empty '@layer' rules are allowed before '@import' and '@namespace' rules (after the '@charset' rule, if any) as well as everywhere '@layer' block at-rules are allowed.

Note: No '@layer' rules are allowed between '@import' and '@namespace' rules. Any '@layer' rule that comes after an '@import' or '@namespace' rule will cause any subsequent '@import' or '@namespace' rules to be ignored.

Unlike the block syntax, multiple comma-separated layer names can be provided in this syntax, declaring each of the layers in the order specified.

Note: Since layer ordering is defined by first occurrence of the layer name (see § 6.4.3 Layer Ordering), this rule allows a page to declare the order of its layers up front, so that their order is apparent without having to read the entire style sheet. It also allows inline layers to be interleaved with imported layers, which is not possible with the block syntax.

EXAMPLE 31

The statement syntax allows establishing a layer order in advance, regardless of the order in which style rules are added to each layer. It can be helpful to establish that layer order in advance, before any '@import' rules. In this example, the imported `theme.css` style rules will override any rules added in the later 'default' block since the order of layers has already been established:

```
@layer default, theme, components;
@import url(theme.css) layer(theme);

@layer default {
  audio[controls] {
    display: block;
  }
}
```

It's also possible to have '@import' rules help establish the order, by placing them between '@layer' rules. This example will have the same result:

```
@layer default;
@import url(theme.css) layer(theme);
@layer components;

@layer default {
  audio[controls] {
    display: block;
  }
}
```

However, '@import' and '@namespace' rules must be consecutive, without any intervening rules. The following is invalid:

```
@import url(default.css) layer(default);
@layer theme;
@import url(components.css) layer(components);

@layer theme {
  audio[controls] {
    display: block;
  }
}
```

§ 6.5. Precedence of Non-CSS Presentational Hints

The UA may choose to honor presentational hints in a source document's markup, for example the `bgcolor` attribute or `<s>` element in [HTML]. All document language-based styling must be translated to corresponding CSS rules and enter the cascade as rules in either the UA-origin or a special-purpose ***author presentational hint origin*** between the regular user origin and the author origin. For the purpose of cascading this author presentational hint origin is treated as an independent origin, but for the purpose of the 'revert' keyword it is considered part of the author origin.

A document language may define whether such a presentational hint enters the cascade as UA-origin or author-origin; if so, the UA must behave accordingly. For example, [SVG11] maps its presentation attributes into the author origin.

> Note: Presentational hints entering the cascade as UA-origin rules can be overridden by author-origin or user-origin styles. Presentational hints entering the cascade as author presentational hint origin rules can be overridden by author-origin styles, but not by non-important user-origin styles. Host languages should choose the appropriate origin for presentational hints with these considerations in mind.

7. Defaulting

When the cascade does not result in a value, the specified value must be found some other way. Inherited properties draw their defaults from their parent element through inheritance; all other properties take their initial value. Authors can explicitly request inheritance or initialization via the 'inherit' and 'initial' keywords.

7.1. Initial Values

Each property has an ***initial value***, defined in the property's definition table. If the property is not an inherited property, and the cascade does not result in a value, then the specified value of the property is its initial value.

7.2. Inheritance

Inheritance propagates property values from parent elements to their children. The ***inherited value*** of a property on an element is the computed value of the property on the element's parent element. For the root element, which has no parent element, the inherited value is the initial value of the property.

For a [DOM] tree with shadows, inheritance operates on the flattened element tree. This means that slotted elements inherit from the `<slot>` they're assigned to, rather than directly from their light tree parent. Pseudo-elements inherit according to the fictional tag sequence described for each pseudo-element. [CSS-PSEUDO-4]

Some properties are ***inherited properties***, as defined in their property definition table. This means that, unless the cascade results in a value, the value will be determined by inheritance.

A property can also be explicitly inherited. See the 'inherit' keyword.

> Note: Inheritance follows the document tree and is not intercepted by anonymous boxes, or otherwise affected by manipulations of the box tree.

7.3. Explicit Defaulting

Several CSS-wide property values are defined below; declaring a property to have these values explicitly specifies a particular defaulting behavior. As specified in CSS Values and Units [css-values-3], all CSS properties can accept these values.

7.3.1. Resetting a Property: the 'initial' keyword

If the cascaded value of a property is the ***'initial'*** keyword, the property's specified value is its initial value.

7.3.2. Explicit Inheritance: the 'inherit' keyword

If the cascaded value of a property is the ***'inherit'*** keyword, the property's specified and computed values are the inherited value.

7.3.3. Erasing All Declarations: the 'unset' keyword

If the cascaded value of a property is the ***'unset'*** keyword, then if it is an inherited property, this is treated as 'inherit', and if it is not, this is treated as 'initial'. This keyword effectively erases all declared values occurring earlier in the cascade, correctly inheriting or not as appropriate for the property (or all longhands of a shorthand).

7.3.4. Rolling Back Cascade Origins: the 'revert' keyword

If the cascaded value of a property is the ***'revert'*** keyword, the behavior depends on the cascade origin to which the declaration belongs:

user-agent origin
: Equivalent to 'unset'.

user origin
: Rolls back the cascaded value to the user-agent level, so that the specified value is calculated as if no author-origin or user-origin rules were specified for this property on this element.

author origin
: Rolls back the cascaded value to the user level, so that the specified value is calculated as if no author-origin rules were specified for this property on this element. For the purpose of 'revert', this origin includes the Animation origin.

7.3.5. Rolling Back Cascade Layers: the 'revert-layer' keyword

If the cascaded value of a property is the ***'revert-layer'*** keyword, the cascaded value is rolled back to the layer below, so that the specified value is calculated as if no rules were specified in the current cascade layer—or between its normal and important levels in the cascade—for this property on this element. For 'revert-layer' in important element-attached styles, however, it only reverts the element-attached styles and the intervening animation origin, and not any of the intervening author-origin important rules.

> Note: If there are no lower-priority declarations in the same cascade origin as the 'revert-layer' value, the cascaded value will roll back to the previous origin.

> Note: The animation origin is not collapsed with the author origin for this purpose as it is for 'revert', and thus effectively forms its own cascade layer.

8. Layer APIs

8.1. Extensions to the `CSSImportRule` interface

The `CSSImportRule` interface is extended as follows:

```
partial interface CSSImportRule {
  readonly attribute CSSOMString? layerName;
};
```

Its ***`LayerName`*** attribute represents the layer name declared in the at-rule itself, and is an empty string if the layer is anonymous, or null if the at-rule does not declare a layer.

8.2. The `CSSLayerBlockRule` interface

The `CSSLayerBlockRule` interface represents the '@layer' block rule:

```
[Exposed=Window]
interface CSSLayerBlockRule : CSSGroupingRule {
  readonly attribute CSSOMString name;
};
```

Its ***name*** attribute represents the layer name declared by the at-rule itself, and is an empty string if the layer is anonymous.

EXAMPLE 32

For example, additional nested context is not added from wrapping layer rules.

```
@layer outer {
  @layer foo.bar { }
}
```

in this case the `name` of the inner '@layer' rule is "foo.bar" (and not "outer.foo.bar").

§ 8.3. The `CSSLayerStatementRule` interface

The `CSSLayerStatementRule` interface represents the '@layer' statement:

```
[Exposed=Window]
interface CSSLayerStatementRule : CSSRule {
  readonly attribute FrozenArray<CSSOMString> nameList;
};
```

Its ***nameList*** attribute represents the list of layer names declared by the at-rule, normalized following the same rule as the `CSSLayerBlockRule`'s `name` attribute.

§ 9. Changes

§ 9.1. Changes since the 15 Oct 2021 Working Draft

Non-trivial changes since the 15 October 2021 Working Draft:

- Updated grammar style for @import media queries and supports conditions
- Allowed functional notation parse-time aliases (Issue 6193)
- Made CSSImportRule.layerName nullable (Issue 6576)
- Clarified that revert-layer in style attr does not revert author layers (Issue 6743)
- Clarified revert-layer on style attr and keyframes (Issue 6743)
- Added § 4.1.1 Value Aliasing section. (Issue 6193)
- Added § 8 Layer APIs section. (Issue 6576)
- Clarified the behavior of 'revert-layer' keyword when used in the style attribute or '@keyframes' at-rule. (Issue 6743)
- Clarified the behavior of the 'layer' keyword and 'layer()' function on '@import' rules. (Issue 6776)

§ 9.2. Changes since the 29 August 2021 Working Draft

Changes since the 29 August 2021 Working Draft include:

- Revert the ordering of unlayered styles. (See § 9.3 Changes since the 8 June 2021 Working Draft and Issue 6284)
- Defined presentational hints to use the author presentational hint origin instead of layers, matching update to [CSS-CASCADE-4]. (Issue 6659)

§ 9.3. Changes since the 8 June 2021 Working Draft

Significant changes since the 8 June 2021 Working Draft include:

- Reserve the CSS-wide keywords for future use in layer-names. (Issue 6323)
- Clarify that '@layer' rules respect global conditional rules, but are always applied to the layer order when declared in non-global conditions such as a container query. (Issue 6407)
- Name-defining at-rules follow layer order for collision resolution, similar to specificity resolution. (Issue 6404)
- Disallow interleaving of '@layer' with '@import' or '@namespace' rules. (Issue 6522)

§ 9.4. Changes since the 19 March 2021 Working Draft

Significant changes since the 19 March 2021 Working Draft include:

- Switched the ordering of unlayered styles from highest to lowest priority in the normal origins. (Issue 6284)

§ 9.5. Changes since the 19 January 2021 Working Draft

Significant changes since the 19 January 2021 First Public Working Draft include:

- Switched layer import syntax from using '@layer' to using '@import'. (Issue 5681)
- Added 'revert-layer' keyword. (Issue 5793)

§ 9.6. 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.

§ 9.7. 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 § 3.1 Property Aliasing.

§ 9.8. 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, Florian Rivoal, Simon Sapin, Jen Simmons, and Boris Zbarsky contributed to this specification.

§ Privacy and 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.
- 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.
- 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 33
>
> 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.

§ CR exit criteria

For this specification to be advanced to Proposed Recommendation, there must be at least two independent, interoperable implementations of each feature. Each feature may be implemented by a different set of products, there is no requirement that all features be implemented by a single product. For the purposes of this criterion, we define the following terms:

independent
: each implementation must be developed by a different party and cannot share, reuse, or derive from code used by another qualifying implementation. Sections of code that have no bearing on the implementation of this specification are exempt from this requirement.

interoperable
: passing the respective test case(s) in the official CSS test suite, or, if the implementation is not a Web browser, an equivalent test. Every relevant test in the test suite should have an equivalent test created if such a user agent (UA) is to be used to claim interoperability. In addition if such a UA is to be used to claim interoperability, then there must one or more additional UAs which can also pass those equivalent tests in the same way for the purpose of interoperability. The equivalent tests must be made publicly available for the purposes of peer review.

implementation
: a user agent which:

1. implements the specification.
2. is available to the general public. The implementation may be a shipping product or other publicly available version (i.e., beta version, preview release, or "nightly build"). Non-shipping product releases must have implemented the feature(s) for a period of at least one month in order to demonstrate stability.
3. is not experimental (i.e., a version specifically designed to pass the test suite and is not intended for normal usage going forward).

The specification will remain Candidate Recommendation for at least six months.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[css-2021] defines the following terms:
- vendor-prefixed

[css-align-3] defines the following terms:
- auto

[css-animations-1] defines the following terms:
- @keyframes

[css-backgrounds-3] defines the following terms:
- background
- background-color
- background-image
- border
- border-bottom-width
- border-color
- border-image
- border-left-width
- border-right-width
- border-style
- border-top-width
- dotted

[css-break-3] defines the following terms:
- break-before
- orphans

[css-break-4] defines the following terms:
- box fragment
- page

[css-color-4] defines the following terms:
- <color>
- color
- currentcolor
- red

[CSS-CONDITIONAL-3] defines the following terms:
- <supports-condition>
- <supports-decl>
- @media
- @supports
- conditional group rule

[css-conditional-5] defines the following terms:
- supports()

[css-contain-1] defines the following terms:
- layout

[css-display-3] defines the following terms:
- box
- display
- display type
- element
- inline box
- text node

[css-flexbox-1] defines the following terms:
- flex
- flex item

[css-fonts-4] defines the following terms:
- bolder
- font
- font-family
- font-style
- font-variant
- font-weight
- italic
- sans-serif

[css-fonts-5] defines the following terms:
- @font-face
- font-size
- font-size-adjust

[css-inline-3] defines the following terms:

- root inline box

[css-lists-3] defines the following terms:

- list-style-position

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

- ::after
- ::before
- ::first-line
- tree-abiding

[css-scoping-1] defines the following terms:

- flattened element tree
- tree context

[css-sizing-3] defines the following terms:

- height
- width

[css-syntax-3] defines the following terms:

- <stylesheet>
- @charset
- at-rule
- block at-rule
- environment encoding
- property declarations

[css-text-3] defines the following terms:

- text-align
- text-indent
- text-transform

[css-values-3] defines the following terms:

- ex

[css-values-4] defines the following terms:

- #
- *
- <ident>
- <length-percentage>
- <length>
- <string>
- <url>
- ?
- ch
- css-wide keywords
- em
- url()
- vh
- vw
- |

[css-variables-1] defines the following terms:

- custom property

[css-writing-modes-3] defines the following terms:

- direction
- unicode-bidi

[css-writing-modes-4] defines the following terms:

- text-orientation
- writing-mode

[CSS2] defines the following terms:

- line-height
- page-break-after
- page-break-before

[CSSOM] defines the following terms:

- CSSGroupingRule
- CSSImportRule
- CSSOMString
- CSSRule
- getComputedStyle(elt)

[DOM] defines the following terms:

- connected
- light tree
- quirks mode
- shadow tree
- shadow-including tree order

[FETCH] defines the following terms:

- cors protocol
- response
- url

[HTML] defines the following terms:

- audio
- content-type metadata
- div
- p
- s
- same origin
- slot
- span

[MEDIAQ] defines the following terms:

- <media-query-list>
- <media-query>
- media query list

[selectors-4] defines the following terms:

- pseudo-element

[WEBIDL] defines the following terms:

- Exposed
- FrozenArray

§ References

§ Normative References

[CSS-2021]
Tab Atkins Jr.; Elika Etemad; Florian Rivoal. *CSS Snapshot 2021*. 31 December 2021. NOTE. URL: https://www.w3.org/TR/css-2021/

[CSS-ANIMATIONS-1]

Dean Jackson; et al. *CSS Animations Level 1*. 11 October 2018. WD. URL: https://www.w3.org/TR/css-animations-1/

[CSS-BACKGROUNDS-3]
Bert Bos; Elika Etemad; Brad Kemper. *CSS Backgrounds and Borders Module Level 3*. 26 July 2021. CR. URL: https://www.w3.org/TR/css-backgrounds-3/

[CSS-BREAK-4]
Rossen Atanassov; Elika Etemad. *CSS Fragmentation Module Level 4*. 18 December 2018. WD. URL: https://www.w3.org/TR/css-break-4/

[CSS-COLOR-4]
Tab Atkins Jr.; Chris Lilley; Lea Verou. *CSS Color Module Level 4*. 15 December 2021. WD. URL: https://www.w3.org/TR/css-color-4/

[CSS-CONDITIONAL-3]
David Baron; Elika Etemad; Chris Lilley. *CSS Conditional Rules Module Level 3*. 23 December 2021. CR. URL: https://www.w3.org/TR/css-conditional-3/

[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-DISPLAY-3]
Tab Atkins Jr.; Elika Etemad. *CSS Display Module Level 3*. 3 September 2021. CR. URL: https://www.w3.org/TR/css-display-3/

[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-PSEUDO-4]
Daniel Glazman; Elika Etemad; Alan Stearns. *CSS Pseudo-Elements Module Level 4*. 31 December 2020. WD. URL: https://www.w3.org/TR/css-pseudo-4/

[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*. 6 June 2019. CR. URL: https://www.w3.org/TR/css-values-3/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. 16 December 2021. WD. URL: https://www.w3.org/TR/css-values-4/

[CSS-VARIABLES-1]
Tab Atkins Jr.. *CSS Custom Properties for Cascading Variables Module Level 1*. 11 November 2021. CR. URL: https://www.w3.org/TR/css-variables-1/

[CSS-WRITING-MODES-3]
Elika Etemad; Koji Ishii. *CSS Writing Modes Level 3*. 10 December 2019. REC. URL: https://www.w3.org/TR/css-writing-modes-3/

[CSS2]
Bert Bos; et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. 7 June 2011. REC. URL: https://www.w3.org/TR/CSS21/

[CSSOM]
Daniel Glazman; Emilio Cobos Álvarez. *CSS Object Model (CSSOM)*. 26 August 2021. WD. URL:

https://www.w3.org/TR/cssom-1/

[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/

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

[MEDIAQ]
Florian Rivoal; Tab Atkins Jr.. *Media Queries Level 4*. 25 December 2021. CR. URL: https://www.w3.org/TR/mediaqueries-4/

[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*. 21 November 2018. WD. URL: https://www.w3.org/TR/selectors-4/

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

§ Informative References

[CSS-ALIGN-3]
Elika Etemad; Tab Atkins Jr.. *CSS Box Alignment Module Level 3*. 24 December 2021. WD. URL: https://www.w3.org/TR/css-align-3/

[CSS-BREAK-3]
Rossen Atanassov; Elika Etemad. *CSS Fragmentation Module Level 3*. 4 December 2018. CR. URL: https://www.w3.org/TR/css-break-3/

[CSS-CASCADE-4]
Elika Etemad; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 4*. 3 December 2021. WD. URL: https://www.w3.org/TR/css-cascade-4/

[CSS-CONTAIN-1]
Tab Atkins Jr.; Florian Rivoal. *CSS Containment Module Level 1*. 22 December 2020. REC. URL: https://www.w3.org/TR/css-contain-1/

[CSS-FLEXBOX-1]
Tab Atkins Jr.; et al. *CSS Flexible Box Layout Module Level 1*. 19 November 2018. CR. URL: https://www.w3.org/TR/css-flexbox-1/

[CSS-FONTS-4]
John Daggett; Myles Maxfield; Chris Lilley. *CSS Fonts Module Level 4*. 21 December 2021. WD. URL: https://www.w3.org/TR/css-fonts-4/

[CSS-INLINE-3]
Dave Cramer; Elika Etemad; Steve Zilles. *CSS Inline Layout Module Level 3*. 27 August 2020. WD. URL: https://www.w3.org/TR/css-inline-3/

[CSS-LISTS-3]
Elika Etemad; Tab Atkins Jr.. *CSS Lists and Counters Module Level 3*. 17 November 2020. WD. URL: https://www.w3.org/TR/css-lists-3/

[CSS-PAGE-3]
Elika Etemad; Simon Sapin. *CSS Paged Media Module Level 3*. 18 October 2018. WD. URL: https://www.w3.org/TR/css-page-3/

[CSS-SIZING-3]

Tab Atkins Jr.; Elika Etemad. *CSS Box Sizing Module Level 3*. 17 December 2021. WD. URL: https://www.w3.org/TR/css-sizing-3/

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

[CSS-TEXT-3]
Elika Etemad; Koji Ishii; Florian Rivoal. *CSS Text Module Level 3*. 22 April 2021. CR. URL: https://www.w3.org/TR/css-text-3/

[CSS-WRITING-MODES-4]
Elika Etemad; Koji Ishii. *CSS Writing Modes Level 4*. 30 July 2019. CR. URL: https://www.w3.org/TR/css-writing-modes-4/

[SVG11]
Erik Dahlström; et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. 16 August 2011. REC. URL: https://www.w3.org/TR/SVG11/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Animation type	Canonical order	Computed value
'all'	initial | inherit | unset | revert | revert-layer	see individual properties	see individual properties	see individual properties	see individual properties	see individual properties	per grammar	see individual properties

§ IDL Index

```
partial interface CSSImportRule {
  readonly attribute CSSOMString? layerName;
};

[Exposed=Window]
interface CSSLayerBlockRule : CSSGroupingRule {
  readonly attribute CSSOMString name;
};

[Exposed=Window]
interface CSSLayerStatementRule : CSSRule {
  readonly attribute FrozenArray<CSSOMString> nameList;
};
```

§ Issues Index

ISSUE 1 Provide an attribute for assigning link or style elements to cascade layers? [Issue #w3c/csswg-drafts#5853]

ISSUE 2 Allow authors to explicitly place unlayered styles in the layer order [Issue #6323]