CSS Easing Functions Level 1

W3C Candidate Recommendation Draft, 13 February 2023

▼ **More details about this document**

This version:
https://www.w3.org/TR/2023/CRD-css-easing-1-20230213/

Latest published version:
https://www.w3.org/TR/css-easing-1/

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

Previous Versions:
https://www.w3.org/TR/2021/CRD-css-easing-1-20210401/
https://www.w3.org/TR/2019/CR-css-easing-1-20190430/

History:
https://www.w3.org/standards/history/css-easing-1

Implementation Report:
https://wpt.fyi/results/css/css-easing?label=master&label=experimental&aligned

Feedback:
CSSWG Issues Repository

Editors:
Brian Birtles (Mozilla)
Dean Jackson (Apple Inc)
Matt Rakow (Microsoft)

Former Editor:
Shane Stephens (Google)

Suggest an Edit for this Spec:
GitHub Editor

Participate:
IRC: #css on W3C's IRC

Tests:
web-platform-tests css/css-easing

Abstract

This CSS module describes a way for authors to define a transformation that controls the rate of change of some value. Applied to animations, such transformations can be used to produce animations that mimic physical phenomena such as momentum or to cause the animation to move in discrete steps producing robot-like movement.

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 Draft** using the Recommendation track. Publication as a Candidate Recommendation does not imply endorsement by W3C and its Members. A Candidate Recommendation Draft integrates changes from the previous Candidate Recommendation that the Working Group intends to include in a subsequent Candidate Recommendation Snapshot.

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-easing” in the title, like this: “[css-easing] *...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

This section is not normative.

It is often desirable to control the rate at which some value changes. For example, gradually increasing the speed at which an element moves can give the element a sense of weight as it appears to gather momentum. This can be used to produce intuitive user interface elements or convincing cartoon props that behave like their physical counterparts. Alternatively, it is sometimes desirable for animation to move forwards in distinct steps such as a segmented wheel that rotates such that the segments always appear in the same position.

Similarly, controlling the rate of change of gradient interpolation can be used to produce different

visual effects such as suggesting a concave or convex surface, or producing a striped effect.

Easing functions provide a means to transform such values by taking an input progress value and producing a corresponding transformed output progress value.

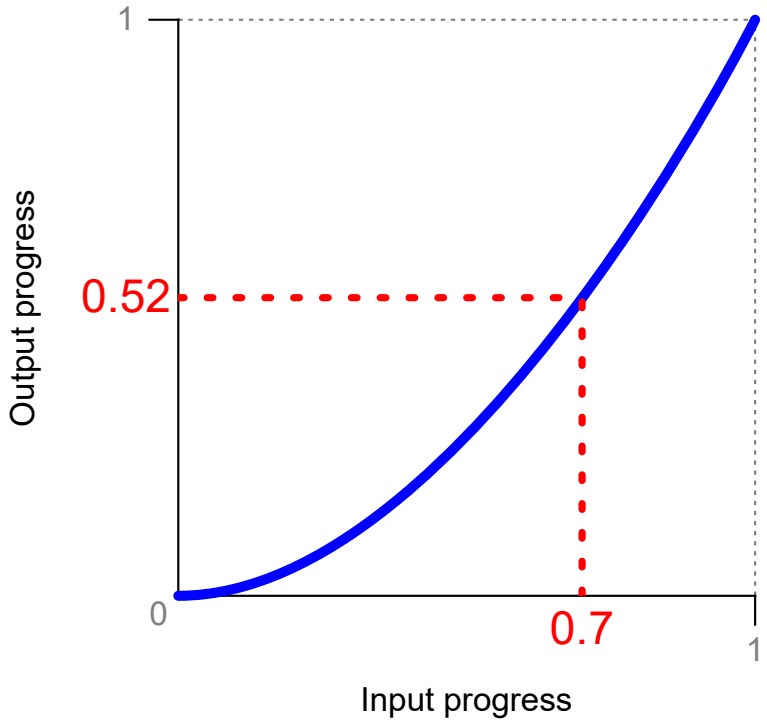

Figure 1 *Example of an easing function that produces an ease-in effect.*
Given an input progress of 0.7, the easing function scales the value to produce an output progress of 0.52.
Applying this easing function to an animation would cause it to progress more slowly at first but then gradually progress more quickly.

§ 1.1. Value Definitions

This specification uses the value definition syntax from [CSS-VALUES-3]. Value types not defined in this specification are defined in CSS Values & Units [CSS-VALUES-3]. Combination with other CSS modules may expand the definitions of these value types.

§ 2. Easing functions

An ***easing function*** takes an input progress value and produces an output progress value.

An easing function must be a pure function meaning that for a given set of inputs, it always produces the same output progress value.

The ***input progress value*** is a real number in the range [-∞, ∞]. Typically, the input progress value is in the range [0, 1] but this may not be the case when easing functions are chained together.

> An example of when easing functions are chained together occurs in Web Animations [WEB-ANIMATIONS] where the output of the easing function specified on an animation effect may become the input to an easing function specified on one of the keyframes of a keyframe effect. In this scenario, the input to the easing function on the keyframe effect may be outside the range [0, 1].

The ***output progress value*** is a real number in the range [-∞, ∞].

Some types of easing functions also take an additional boolean before flag input which is defined subsequently.

This specification defines four types of easing functions whose definitions follow.

The syntax for specifying an easing function is as follows:

'<easing-function>' = linear | <cubic-bezier-easing-function> | <step-easing-function>

§ 2.1. The linear easing function: 'linear'

The ***linear easing function*** is an identity function meaning that its output progress value is equal to the input progress value for all inputs.

The syntax for the linear easing function is simply the ***'linear'*** keyword.

§ 2.2. Cubic Bézier easing functions: 'ease', 'ease-in', 'ease-out', 'ease-in-out', 'cubic-bezier()'

A ***cubic Bézier easing function*** is a type of easing function defined by four real numbers that specify the two control points, *P1* and *P2*, of a cubic Bézier curve whose end points *P0* and *P3* are fixed at (0, 0) and (1, 1) respectively. The *x* coordinates of *P1* and *P2* are restricted to the range [0, 1].

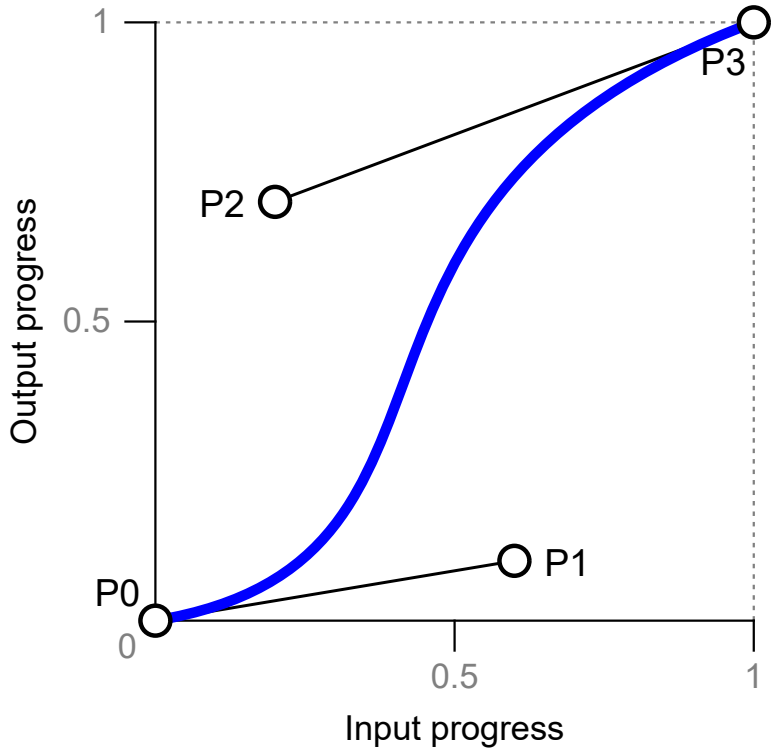

Figure 2 *A cubic Bézier curve used as an easing function.*
The shape of the curve is determined by the location of the control points P1 and P2.
Input progress values serve as x values of the curve, whilst the y values are the output progress values.

A cubic Bézier easing function has the following syntax (using notation from [CSS-VALUES-3]):

'<cubic-bezier-easing-function>' = ease | ease-in | ease-out | ease-in-out | cubic-bezier(<number [0,1]>, <number>, <number [0,1]>, <number>)

The meaning of each value is as follows:

'ease'
Equivalent to 'cubic-bezier(0.25, 0.1, 0.25, 1)'.

'ease-in'
Equivalent to 'cubic-bezier(0.42, 0, 1, 1)'.

'ease-out'
Equivalent to 'cubic-bezier(0, 0, 0.58, 1)'.

'ease-in-out'
Equivalent to 'cubic-bezier(0.42, 0, 0.58, 1)'.

'cubic-bezier(<number [0,1]>, <number>, <number [0,1]>, <number>)'
Specifies a cubic Bézier easing function. The four numbers specify points *P1* and *P2* of the curve as (*x1*, *y1*, *x2*, *y2*). Both *x* values must be in the range [0, 1] or the definition is invalid.

The keyword values listed above are illustrated below.

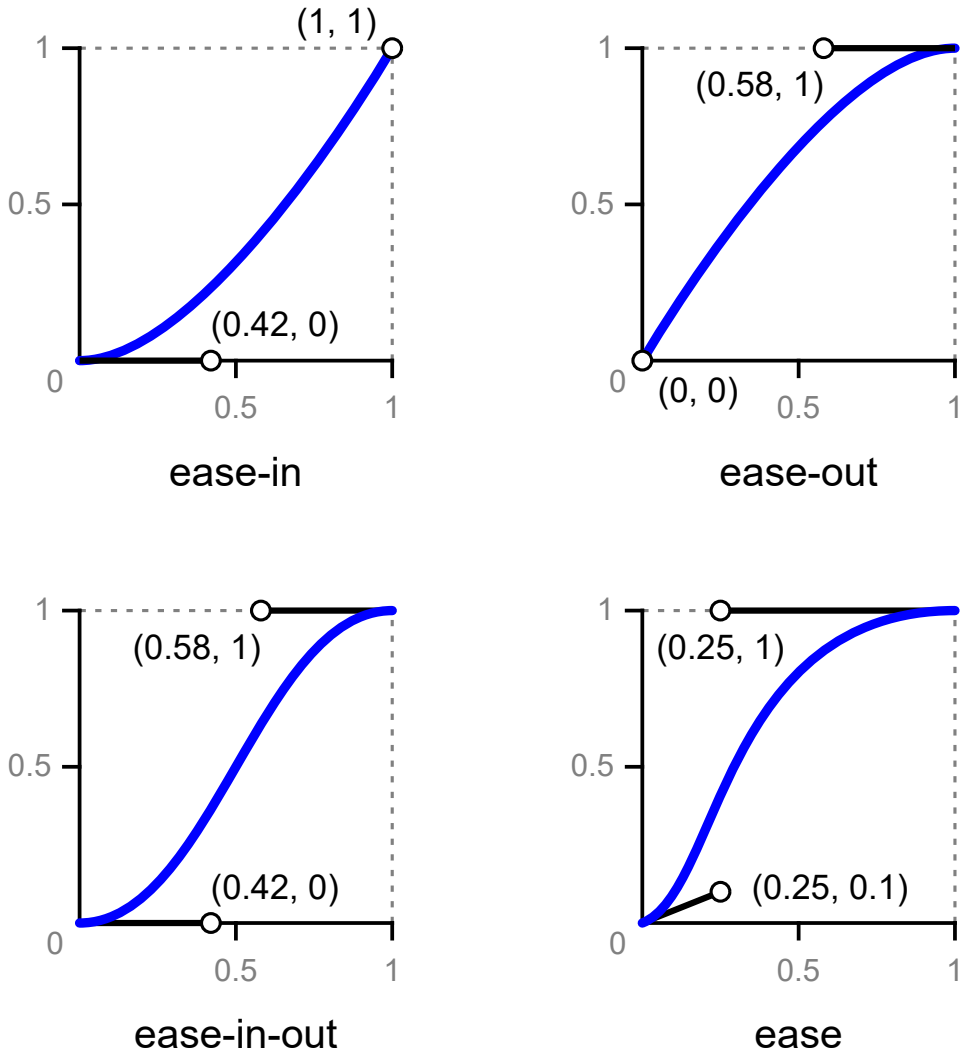

Figure 3 *The easing functions produced by each of the cubic Bézier easing function keyword values.*

§ 2.2.1. Output of a cubic bézier easing function

The mapping from input progress to output progress is performed by determining the corresponding *y* value (output progress value) for a given *x* value (input progress value). The evaluation of this curve is covered in many sources such as [FUND-COMP-GRAPHICS].

For input progress values outside the range [0, 1], the curve is extended infinitely using tangent of the curve at the closest endpoint as follows:

- For input progress values less than zero,
    1. If the *x* value of P1 is greater than zero, use a straight line that passes through P1 and P0 as the tangent.
    2. Otherwise, if the *x* value of P2 is greater than zero, use a straight line that passes through P2 and P0 as the tangent.
    3. Otherwise, let the output progress value be zero for all input progress values in the range [-∞, 0).

- For input progress values greater than one,

  1. If the *x* value of P2 is less than one, use a straight line that passes through P2 and P3 as the tangent.
  2. Otherwise, if the *x* value of P1 is less than one, use a straight line that passes through P1 and P3 as the tangent.
  3. Otherwise, let the output progress value be one for all input progress values in the range (1, ∞].

§ 2.3. Step easing functions: 'step-start', 'step-end', 'steps()'

A ***step easing function*** is a type of easing function that divides the input time into a specified number of intervals that are equal in length. It is defined by a number of ***steps***, and a ***step position***. It has following syntax:

'<step-easing-function>' = step-start | step-end | steps(<integer>[, <step-position>]?)

'<step-position>' = jump-start | jump-end | jump-none | jump-both | start | end

The meaning of each value is as follows:

'step-start'
: Computes to 'steps(1, start)'

'step-end'
: Computes to 'steps(1, end)'

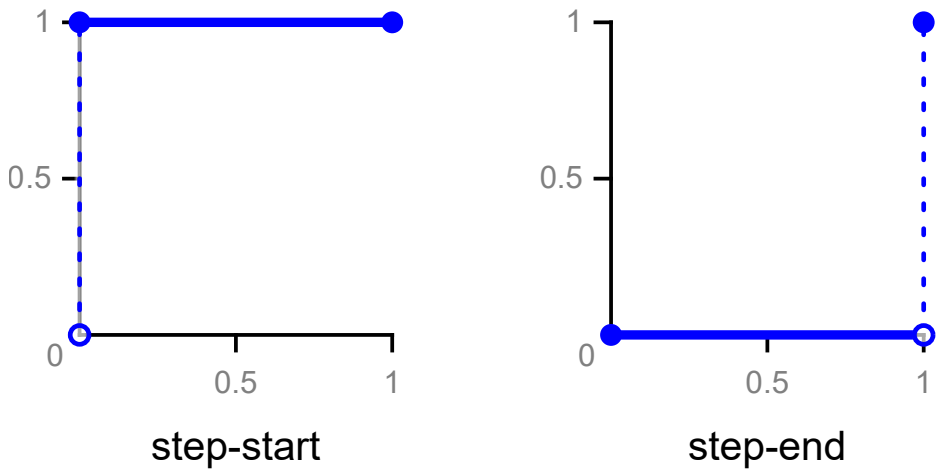

Figure 4 *Example step easing function keyword values.*

'steps(<integer>[, <step-position>]?)'
: The first parameter specifies the number of intervals in the function. It must be a positive integer greater than 0 unless the second parameter is 'jump-none' in which case it must be a

positive integer greater than 1.

The second parameter, which is optional, specifies the step position using one of the following values:

'jump-start'
: The first rise occurs at input progress value of 0.

'jump-end'
: The last rise occurs at input progress value of 1.

'jump-none'
: All rises occur within the range (0, 1).

'jump-both'
: The first rise occurs at input progress value of 0 and the last rise occurs at input progress value of 1.

'start'
: Behaves as 'jump-start'.

'end'
: Behaves as 'jump-end'.

If the second parameter is omitted, the value 'end' is assumed.

These values are illustrated below:

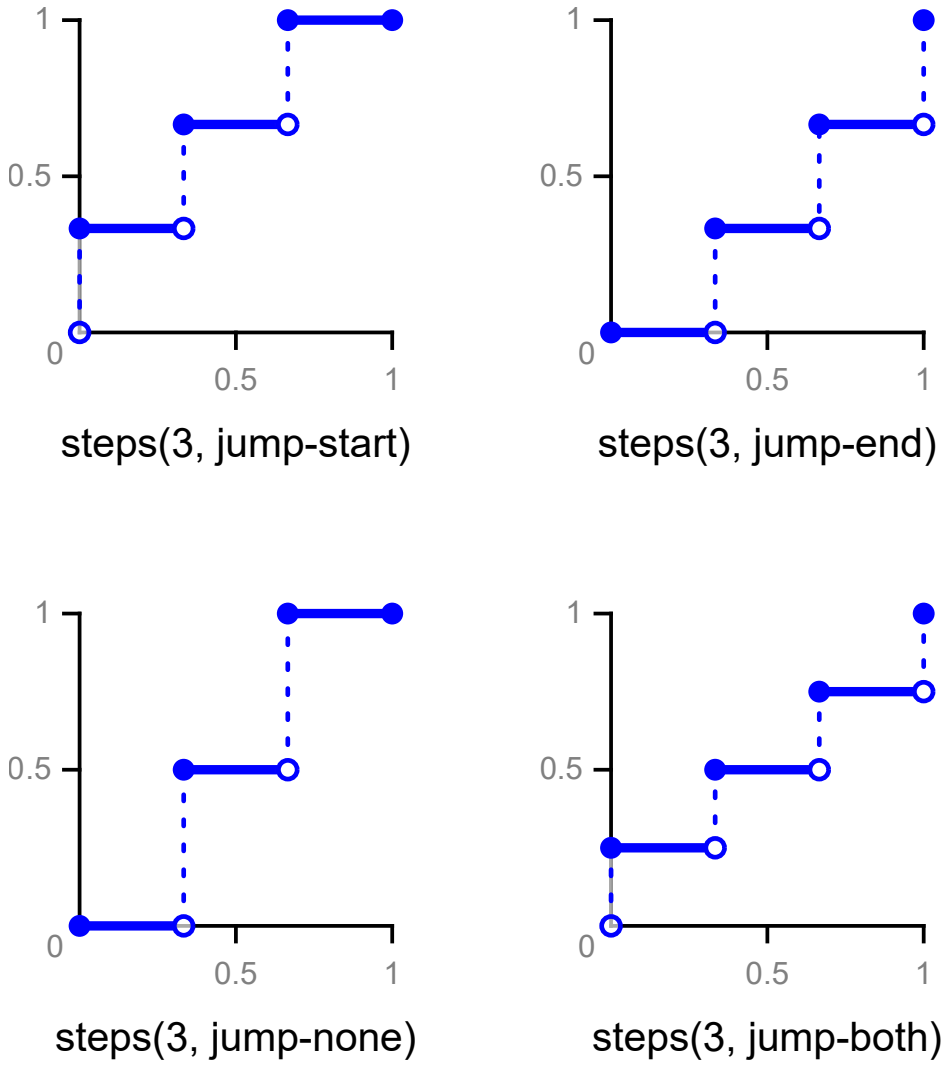

Figure 5 *Example step easing functions.*

§ 2.3.1. Output of a step easing function

At the exact point where a step occurs, the result of the function is conceptually the top of the step. However, an additional ***before flag*** passed as input to the step easing function, if true, will cause the result of the function to correspond to the bottom of the step at the step point.

EXAMPLE 1 ¶

As an example of how the before flag affects the behavior of this function, consider an animation with a step easing function whose step position is 'start' and which has a positive delay and backwards fill.

For example, using CSS animation:

```
animation: moveRight 5s 1s steps(5, start);
```

During the delay phase, the input progress value will be zero but if the before flag is set to indicate that the animation has yet to reach its animation interval, the easing function will produce zero as its output progress value, i.e. the bottom of the first step.

At the exact moment when the animation interval begins, the input progress value will still be zero, but the before flag will not be set and hence the result of the easing function will correspond to the top of the first step.

For the purposes of calculating the output progress value, the step position 'start' is considered equivalent to 'jump-start'. Likewise 'end' is considered equivalent to 'jump-end'. As a result, the following algorithm does not make explicit reference to start or end.

NOTE: User agents must still differentiate between 'jump-start' and 'start' for the purpose of serialization (see § 2.4 Serialization).

The output progress value is calculated from the input progress value and before flag as follows:

1. Calculate the *current step* as `floor(input progress value × steps)`.
2. If the step position property is one of:
   - 'jump-start',
   - 'jump-both',

   increment *current step* by one.
3. If *both* of the following conditions are true:
   - the before flag is set, *and*
   - input progress value × steps mod 1 equals zero (that is, if input progress value × steps is integral), then

   decrement *current step* by one.

4. If input progress value ≥ 0 and *current step* < 0, let *current step* be zero.
5. Calculate *jumps* based on the step position as follows:

   'jump-start' or 'jump-end'
   : steps

   'jump-none'
   : steps - 1

   'jump-both'
   : steps + 1

6. If input progress value ≤ 1 and *current step* > *jumps*, let *current step* be *jumps*.

   > Steps 4 and 6 in this procedure ensure that given an input progress value in the range [0, 1], a step easing function does not produce an output progress value outside that range.
   >
   > For example, although mathematically we might expect that a step easing function with a step position of 'jump-start' would step up (i.e. beyond 1) when the input progress value is 1, intuitively, when we apply such an easing function to a forwards-filling animation, we expect it to produce an output progress value of 1 as the animation fills forwards.
   >
   > A similar situation arises for a step easing function with a step position of 'jump-end' when applied to an animation during its delay phase.

7. The output progress value is `current step / jumps`.

§ 2.4. Serialization

Easing functions are serialized using the common serialization patterns defined in [CSSOM] with the following additional requirements:

- The keyword values 'ease', 'linear', 'ease-in', 'ease-out', and 'ease-in-out' are serialized as-is, that is, they are *not* converted to the equivalent 'cubic-bezier()' function before serializing.
- Step easing functions, whether they are specified using the 'steps()' function or either of the 'step-start' or 'step-end' keywords, are serialized as follows:

  1. If the step position is 'jump-end' or 'end', serialize as 'steps(<integer>)'.
  2. Otherwise, serialize as 'steps(<integer>, <step-position>)'.

§ 3. Privacy and Security Considerations

This specification does not directly introduce any new capabilities to the Web platform but rather

provides common definitions that may be referenced by other specifications. As a result, it does not introduce any new privacy and security concerns.

Specifications referencing the features defined in this specification should consider that while easing functions most commonly take an input progress value in the range [0,1] and produce an output progress value in the range [0, 1], this is not always the case. Applications of easing functions should define the behavior for inputs and outputs outside this range to ensure they do not introduce new security considerations.

§ 4. Changes

The following changes have been made since the 30 April 2019 Candidate Recommendation:

- Updated 'cubic-bezier()' syntax definition to annotate range restrictions using CSS bracketed range notation. (Editorial)
- Added an example of chaining easing functions. (Editorial)

§ 5. Acknowledgements

This specification is based on the CSS Transitions specification edited by L. David Baron, Dean Jackson, David Hyatt, and Chris Marrin. The editors would also like to thank Douglas Stockwell, Steve Block, Tab Atkins, Rachel Nabors, Martin Pitt, and the Animation at Work slack community for their feedback and contributions.

§ 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 2 ¶

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

steps, in § 2.3

steps(), in § 2.3

step-start, in § 2.3

timing function, in § 2

§ Terms defined by reference

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

,

<integer>

<number>

?

css bracketed range notation

|

§ References

§ Normative References

[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/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. 19 October 2022. WD. URL: https://www.w3.org/TR/css-values-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

§ Informative References

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

[FUND-COMP-GRAPHICS]
Peter Shirley; Michael Ashikhmin; Steve Marschner. *Fundamentals of Computer Graphics*. 2009.

[WEB-ANIMATIONS]
Brian Birtles; et al. *Web Animations*. 8 September 2022. WD. URL: https://www.w3.org

/TR/web-animations-1/