Cascading Style Sheets, level 2
CSS2 Specification

W3C Recommendation *12-May-1998*

This version:
    http://www.w3.org/TR/1998/REC-CSS2-19980512
Latest version:
    http://www.w3.org/TR/REC-CSS2
Previous version:
    http://www.w3.org/TR/1998/PR-CSS2-19980324

Editors:
    Bert Bos <bbos@w3.org>
    Håkon Wium Lie <howcome@w3.org>
    Chris Lilley <chris@w3.org>
    Ian Jacobs <ij@w3.org>

Abstract

This specification defines Cascading Style Sheets, level 2 (CSS2). CSS2 is a style sheet language that allows authors and users to attach style (e.g., fonts, spacing, and aural cues) to structured documents (e.g., HTML documents and XML applications). By separating the presentation style of documents from the content of documents, CSS2 simplifies Web authoring and site maintenance.

CSS2 builds on CSS1 (see [CSS1]) and, with very few exceptions, all valid CSS1 style sheets are valid CSS2 style sheets. CSS2 supports media-specific style sheets so that authors may tailor the presentation of their documents to visual browsers, aural devices, printers, braille devices, handheld devices, etc. This specification also supports content positioning, downloadable fonts, table layout, features for internationalization, automatic counters and numbering, and some properties related to user interface.

Status of this document

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Public discussion of CSS features takes place on www-style@w3.org.

Available formats

The CSS2 specification is available in the following formats:

HTML:
    http://www.w3.org/TR/1998/REC-CSS2-19980512
a plain text file:
    http://www.w3.org/TR/1998/REC-CSS2-19980512/css2.txt,
HTML as a gzip'ed tar file:
    http://www.w3.org/TR/1998/REC-CSS2-19980512/css2.tgz,
HTML as a zip file (this is a '.zip' file not an '.exe'):
    http://www.w3.org/TR/1998/REC-CSS2-19980512/css2.zip,
as well as a gzip'ed PostScript file:
    http://www.w3.org/TR/1998/REC-CSS2-19980512/css2.ps.gz,
and a PDF file:
    http://www.w3.org/TR/1998/REC-CSS2-19980512/css2.pdf.

In case of a discrepancy between the various forms of the specification, http://www.w3.org/TR/1998/REC-CSS2-19980512 is considered the definitive version.

Available languages

The English version of this specification is the only normative version. However, for translations in other languages see http://www.w3.org/Style/css2-updates/translations.html.

Errata

The list of known errors in this specification is available at http://www.w3.org/Style/css2-updates/REC-CSS2-19980512-errata.html. Please report errors in this document to css2-editors@w3.org.

Quick Table of Contents

Full Table of Contents

1 About the CSS2 Specification

Contents

1.1 Reading the specification

This specification has been written with two types of readers in mind: CSS authors and CSS implementors. We hope the specification will provide authors with the tools they need to write efficient, attractive, and accessible documents, without overexposing them to CSS's implementation details. Implementors, however, should find all they need to build conforming user agents [p. 32] . The specification begins with a general presentation of CSS and becomes more and more technical and specific towards the end. For quick access to information, a general table of contents, specific tables of contents at the beginning of each section, and an index provide easy navigation, in both the electronic and printed versions.

The specification has been written with two modes of presentation in mind: electronic and printed. Although the two presentations will no doubt be similar, readers will find some differences. For example, links will not work in the printed version (obviously), and page numbers will not appear in the electronic version. In case of a discrepancy, the electronic version is considered the authoritative version of the document.

1.2 How the specification is organized

The specification is organized into the following sections:

Section 2: An introduction to CSS2
: The introduction includes a brief tutorial on CSS2 and a discussion of design principles behind CSS2.

Sections 3 - 20: CSS2 reference manual.
: The bulk of the reference manual consists of the CSS2 language reference. This reference defines what may go into a CSS2 style sheet (syntax, properties, property values) and how user agents must interpret these style sheets in order to claim conformance [p. 32] .

Appendixes:
: Appendixes contain information about a sample style sheet for HTML 4.0 [p. 291] , changes from CSS1 [p. 293] , implementation and performance notes [p. 295] , the grammar of CSS2 [p. 309] , a list of normative and informative references [p. 313] , and three indexes: one for properties [p. 317] , one for descriptors [p. 323] , and one general index [p. 325] .

1.3 Conventions

1.3.1 Document language elements and attributes

- CSS property, descriptor, and pseudo-class names are delimited by single quotes.
- CSS values are delimited by single quotes.
- Document language element names are in uppercase letters.
- Document language attribute names are in lowercase letters and delimited by double quotes.

1.3.2 CSS property definitions

Each CSS property definition begins with a summary of key information that resembles the following:

’property-name’

Value:	legal values & syntax
Initial:	initial value
Applies to:	elements this property applies to
Inherited:	whether the property is inherited
Percentages:	how percentage values are interpreted
Media:	which media groups the property applies to

Value

This part specifies the set of valid values for the property. Value types may be designated in several ways:

1. keyword values (e.g., auto, disc, etc.)
2. basic data types, which appear between "<" and ">" (e.g., <length>, <percentage>, etc.). In the electronic version of the document, each instance of a basic data type links to its definition.
3. types that have the same range of values as a property bearing the same

name (e.g., <’border-width’> <’background-attachment’>, etc.). In this case, the type name is the property name (complete with quotes) between "<" and ">" (e.g., <’border-width’>). In the electronic version of the document, each instance of this type of non-terminal links to the corresponding property definition.

4. non-terminals that do not share the same name as a property. In this case, the non-terminal name appears between "<" and ">", as in <border-width>. Notice the distinction between <border-width> and <’border-width’>; the latter is defined in terms of the former. The definition of a non-terminal is located near its first appearance in the specification. In the electronic version of the document, each instance of this type of value links to the corresponding value definition.

Other words in these definitions are keywords that must appear literally, without quotes (e.g., red). The slash (/) and the comma (,) must also appear literally.

Values may be arranged as follows:

- Several juxtaposed words mean that all of them must occur, in the given order.
- A bar (|) separates two or more alternatives: exactly one of them must occur.
- A double bar (||) separates two or more options: one or more of them must occur, in any order.
- Brackets ([ ]) are for grouping.

Juxtaposition is stronger than the double bar, and the double bar is stronger than the bar. Thus, the following lines are equivalent:

```
  a b   |   c || d e
[ a b ] | [ c || [ d e ]]
```

Every type, keyword, or bracketed group may be followed by one of the following modifiers:

- An asterisk (*) indicates that the preceding type, word, or group occurs zero or more times.
- A plus (+) indicates that the preceding type, word, or group occurs one or more times.
- A question mark (?) indicates that the preceding type, word, or group is optional.
- A pair of numbers in curly braces ({A,B}) indicates that the preceding type, word, or group occurs at least A and at most B times.

The following examples illustrate different value types:

Value: N | NW | NE
Value: [<length> | thick | thin]{1,4}
Value: [<family-name> ,]* <family-name>
Value: <uri>? <color> [/ <color>]?
Value: <uri> || <color>

Initial

This part specifies the property’s initial value. If the property is inherited, this is the value that is given to the root element of the document tree [p. 30] . Please consult the section on the cascade [p. 69] for information about the interaction between style sheet-specified, inherited, and initial values.

Applies to

This part lists the elements to which the property applies. All elements are considered to have all properties, but some properties have no rendering effect on some types of elements. For example, ’white-space’ only affects block-level elements.

Inherited

This part indicates whether the value of the property is inherited from an ancestor element. Please consult the section on the cascade [p. 69] for information about the interaction between style sheet-specified, inherited, and initial values.

Percentage values

This part indicates how percentages should be interpreted, if they occur in the value of the property. If "N/A" appears here, it means that the property does not accept percentages as values.

Media groups

This part indicates the media groups [p. 79] to which the property applies. The conformance [p. 29] conditions state that user agents must support this property if they support rendering to the media types [p. 78] included in these media groups [p. 79] .

1.3.3 Shorthand properties

Some properties are *shorthand properties*, meaning they allow authors to specify the values of several properties with a single property.

For instance, the ’font’ property is a shorthand property for setting ’font-style’, ’font-variant’, ’font-weight’, ’font-size’, ’line-height’, and ’font-family’ all at once.

When values are omitted from a shorthand form, each "missing" property is assigned its initial value (see the section on the cascade [p. 69]).

Example(s):

The multiple style rules of this example:

```
H1 {
  font-weight: bold;
  font-size: 12pt;
  line-height: 14pt;
  font-family: Helvetica;
  font-variant: normal;
  font-style: normal;
  font-stretch: normal;
  font-size-adjust: none
}
```

may be rewritten with a single shorthand property:

```
H1 { font: bold 12pt/14pt Helvetica }
```

In this example, ’font-variant’, ’font-stretch’, ’font-size-adjust’, and ’font-style’ take their initial values.

1.3.4 Notes and examples

All examples that illustrate illegal usage are clearly marked as "ILLEGAL EXAMPLE".

All HTML examples conform to the HTML 4.0 strict DTD (defined in [HTML40]) unless otherwise indicated by a document type declaration.

All notes are informative only.

Examples and notes are marked within the source HTML for the specification and CSS1 user agents will render them specially.

1.3.5 Images and long descriptions

Most images in the electronic version of this specification are accompanied by "long descriptions" of what they represent. A link to the long description is denoted by a "[D]" to the right of the image.

Images and long descriptions are informative only.

1.4 Acknowledgments

This specification is the product of the W3C Working Group on Cascading Style Sheets and Formatting Properties. In addition to the editors of this specification, the members of the Working Group are: Brad Chase (Bitstream), Chris Wilson (Microsoft), Daniel Glazman (Electricité de France), Dave Raggett (W3C/HP), Ed Tecot (Microsoft), Jared Sorensen (Novell), Lauren Wood (SoftQuad), Laurie Anna Kaplan (Microsoft), Mike Wexler (Adobe), Murray Maloney (Grif), Powell Smith (IBM), Robert Stevahn (HP), Steve Byrne (JavaSoft), Steven Pemberton (CWI), Thom Phillabaum (Netscape), Douglas Rand (Silicon Graphics), Robert Pernett (Lotus), Dwayne Dicks (SoftQuad), and Sho Kuwamoto (Macromedia). We thank them for their continued efforts.

A number of invited experts to the Working Group have contributed: George Kersher, Glenn Rippel (Bitstream), Jeff Veen (HotWired), Markku T. Hakkinen (The Productivity Works), Martin Dürst (W3C, formerly Universität Zürich), Roy Platon (RAL), Todd Fahrner (Verso), Tim Boland (NIST), Eric Meyer (Case Western Reserve University), and Vincent Quint (W3C).

The section on Web Fonts was strongly shaped by Brad Chase (Bitstream) David Meltzer (Microsoft Typography) and Steve Zilles (Adobe). The following people have also contributed in various ways to the section pertaining to fonts: Alex Beamon (Apple), Ashok Saxena (Adobe), Ben Bauermeister (HP), Dave Raggett (W3C/HP), David Opstad (Apple), David Goldsmith (Apple), Ed Tecot (Microsoft), Erik van Blokland (LettError), François Yergeau (Alis), Gavin Nicol (Inso), Herbert van Zijl (Elsevier), Liam Quin, Misha Wolf (Reuters), Paul Haeberli (SGI), and the late Phil Karlton (Netscape).

The section on Paged Media was in large parts authored by Robert Stevahn (HP) and Stephen Waters (Microsoft).

Robert Stevahn (HP), Scott Furman (Netscape), and Scott Isaacs (Microsoft) were key contributors to CSS Positioning.

Mike Wexler (Adobe) was the editor of the interim working draft, which described many of the new features of CSS2.

T.V. Raman (Adobe) made pivotal contributions towards Aural Cascading Style Sheets (ACSS) and the concepts of aural presentation based on his work on AsTeR (Audio System For Technical Readings). He contributed an initial draft of the ACSS specification that shaped the current specification. Values for aural properties in the HTML 4.0 sample style sheet [p. 291] are of his devising; he currently uses them on a daily basis on his audio desktop in conjunction with Emacspeak and the Emacs W3 browser (authored by William Perry, who also implemented the aural extensions on the W3 side of the fence).

Todd Fahrner (Verso) researched contemporary and historical browsers to develop the sample style sheet in the appendix.

Thanks to Jan Kärrman, author of html2ps for helping so much in creating the PostScript version of the specification.

Through electronic and physical encounters, the following people have contributed to the development of CSS2: Alan Borning, Robert Cailliau, Liz Castro, James Clark, Dan Connolly, Donna Converse, Daniel Dardailler, Al Gilman, Daniel Greene, Scott Isaacs, Geir Ivarsøy, Vincent Mallet, Kim Marriott, Brian Michalowski, Lou Montulli, Henrik Frystyk Nielsen, Jacob Nielsen, Eva von Pepel, William Perry, David Siegel, Peter Stuckey, and Jason White.

The discussions on www-style@w3.org have been influential in many key issues for CSS. Especially, we would like to thank Bjorn Backlund, Todd Fahrner, Lars Marius Garshol, Sue Jordan, Ian Hickson, Susan Lesch, Andrew Marshall, MegaZone, Eric Meyer, Russell O’Connor, David Perrell, Liam Quinn, Jon Seymour, Neil St. Laurent, Taylor, Brian Wilson, and Chris Wilson for their participation.

Many thanks to the Web Accessibility Initiative Protocols and Formats Technical Review Working Group (WAI PF) for helping to improve the accessibility of CSS2.

Many thanks to Philippe Le Hégaret, whose CSS validator helped ensure correct examples and a sensible grammar.

Special thanks to Arnaud Le Hors, whose engineering contributions made this document work.

Adam Costello improved this specification by performing a detailed review.

Lastly, thanks to Tim Berners-Lee without whom none of this would have been possible.

1.5 Copyright Notice

---

2 Introduction to CSS2

Contents

2.1 A brief CSS2 tutorial for HTML

In this tutorial, we show how easy it can be to design simple style sheets. For this tutorial, you will need to know a little HTML (see [HTML40]) and some basic desktop publishing terminology.

We begin with a small HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach’s home page</TITLE>
  </HEAD>
  <BODY>
    <H1>Bach’s home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

To set the text color of the H1 elements to blue, you can write the following CSS rule:

```
  H1 { color: blue }
```

A CSS rule consists of two main parts: selector [p. 53] (’H1’) and declaration (’color: blue’). The declaration has two parts: property (’color’) and value (’blue’). While the example above tries to influence only one of the properties needed for rendering an HTML document, it qualifies as a style sheet on its own. Combined with other style sheets (one fundamental feature of CSS is that style sheets are combined) it will determine the final presentation of the document.

The HTML 4.0 specification defines how style sheet rules may be specified for HTML documents: either within the HTML document, or via an external style sheet. To put the style sheet into the document, use the STYLE element:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach’s home page</TITLE>
  <STYLE type="text/css">
    H1 { color: blue }
  </STYLE>
  </HEAD>
```

```
    <BODY>
      <H1>Bach’s home page</H1>
      <P>Johann Sebastian Bach was a prolific composer.
    </BODY>
  </HTML>
```

For maximum flexibility, we recommend that authors specify external style sheets; they may be changed without modifying the source HTML document, and they may be shared among several documents. To link to an external style sheet, you can use the LINK element:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach’s home page</TITLE>
  <LINK rel="stylesheet" href="bach.css" type="text/css">
  </HEAD>
  <BODY>
    <H1>Bach’s home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

The LINK element specifies:

- the type of link: to a "stylesheet".
- the location of the style sheet via the "ref" attribute.
- the type of style sheet being linked: "text/css".

To show the close relationship between a style sheet and the structured markup, we continue to use the STYLE element in this tutorial. Let’s add more colors:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach’s home page</TITLE>
  <STYLE type="text/css">
    BODY { color: red }
    H1 { color: blue }
  </STYLE>
  </HEAD>
  <BODY>
    <H1>Bach’s home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

The style sheet now contains two rules: the first one sets the color of the BODY element to ’red’, while the second one sets the color of the H1 element to ’blue’. Since no color value has been specified for the P element, it will inherit the color from its parent element, namely BODY. The H1 element is also a child element of BODY but the second rule overrides the inherited value. In CSS there are often such conflicts between different values, and this specification describes how to resolve them.

CSS2 has more than 100 different properties, including ’color’. Let’s look at some of the others:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach’s home page</TITLE>
  <STYLE type="text/css">
    BODY {
      font-family: "Gill Sans", sans-serif;
      font-size: 12pt;
      margin: 3em;
    }
  </STYLE>
  </HEAD>
  <BODY>
    <H1>Bach’s home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

The first thing to notice is that several declarations are grouped within a block enclosed by curly braces ({...}), and separated by semicolons, though the last declaration may also be followed by a semicolon.

The first declaration on the BODY element sets the font family to "Gill Sans". If that font isn’t available, the user agent (often referred to as a "browser") will use the ’sans-serif’ font family which is one of five generic font families which all users agents know. Child elements of BODY will inherit the value of the ’font-family’ property.

The second declaration sets the font size of the BODY element to 12 points. The "point" unit is commonly used in print-based typography to indicate font sizes and other length values. It’s an example of an absolute unit which does not scale relative to the environment.

The third declaration uses a relative unit which scales with regard to its surroundings. The "em" unit refers to the font size of the element. In this case the result is that the margins around the BODY element are three times wider than the font size.

2.2 A brief CSS2 tutorial for XML

CSS can be used with any structured document format, for example with applications of the eXtensible Markup Language [XML10]. In fact, XML depends more on style sheets than HTML, since authors can make up their own elements that user agents don’t know how to display.

Here is a simple XML fragment:

```
<ARTICLE>
  <HEADLINE>Fredrick the Great meets Bach</HEADLINE>
  <AUTHOR>Johann Nikolaus Forkel</AUTHOR>
  <PARA>
    One evening, just as he was getting his
    <INSTRUMENT>flute</INSTRUMENT> ready and his
    musicians were assembled, an officer brought him a list of
    the strangers who had arrived.
  </PARA>
</ARTICLE>
```

To display this fragment in a document-like fashion, we must first declare which elements are inline-level (i.e., do not cause line breaks) and which are block-level (i.e., cause line breaks).

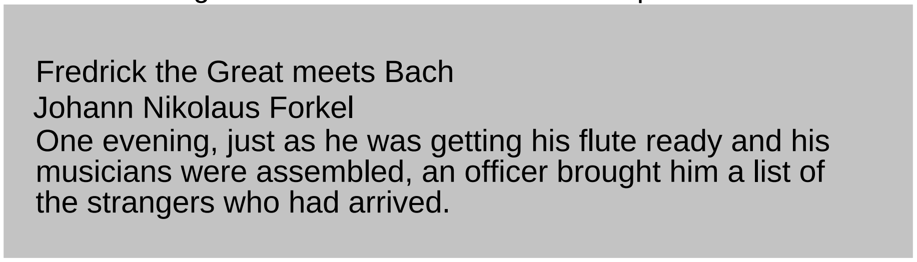
```
INSTRUMENT { display: inline }
ARTICLE, HEADLINE, AUTHOR, PARA { display: block }
```

The first rule declares INSTRUMENT to be inline and the second rule, with its comma-separated list of selectors, declares all the other elements to be block-level.

One proposal for linking a style sheet to an XML document is to use a processing instruction:

```
<?XML:stylesheet type="text/css" href="bach.css"?>
<ARTICLE>
  <HEADLINE>Fredrick the Great meets Bach</HEADLINE>
  <AUTHOR>Johann Nikolaus Forkel</AUTHOR>
  <PARA>
    One evening, just as he was getting his
    <INSTRUMENT>flute</INSTRUMENT> ready and his
    musicians were assembled, an officer brought him a list of
    the strangers who had arrived.
  </PARA>
</ARTICLE>
```

A visual user agent could format the above example as:



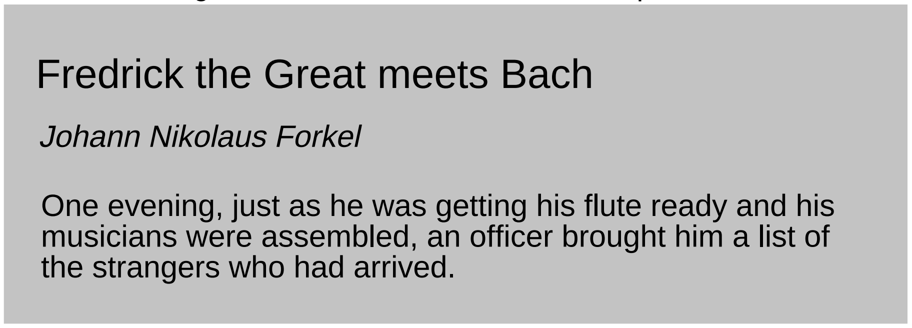
Fredrick the Great meets Bach
Johann Nikolaus Forkel
One evening, just as he was getting his flute ready and his
musicians were assembled, an officer brought him a list of
the strangers who had arrived.

Notice that the word "flute" remains within the paragraph since it is the content of the inline element INSTRUMENT.

Still, the text isn't formatted the way you would expect. For example, the headline font size should be larger than then rest of the text, and you may want to display the author's name in italic:

```
INSTRUMENT { display: inline }
ARTICLE, HEADLINE, AUTHOR, PARA { display: block }
HEADLINE { font-size: 1.3em }
AUTHOR { font-style: italic }
ARTICLE, HEADLINE, AUTHOR, PARA { margin: 0.5em }
```

A visual user agent could format the above example as:



Fredrick the Great meets Bach

Johann Nikolaus Forkel

One evening, just as he was getting his flute ready and his
musicians were assembled, an officer brought him a list of
the strangers who had arrived.

Adding more rules to the style sheet will allow you to further improve the presentation of the document.

2.3 The CSS2 processing model

This section presents one possible model of how user agents that support CSS work. This is only a conceptual model; real implementations may vary.

In this model, a user agent processes a source by going through the following steps:

1. Parse the source document and create a document tree [p. 30] .
2. Identify the target media type [p. 77] .
3. Retrieve all style sheets associated with the document that are specified for the target media type [p. 77] .
4. Annotate every element of the document tree by assigning a single value to every property [p. 41] that is applicable to the target media type [p. 77] . Properties are assigned values according to the mechanisms described in the section on cascading and inheritance [p. 69] .

   Part of the calculation of values depends on the formatting algorithm appropriate for the target media type [p. 77] . For example, if the target medium is the screen, user agents apply the visual formatting model [p. 95] . If the destination medium is the printed page, user agents apply the page model [p. 175] . If the destination medium is an aural rendering device (e.g., speech synthesizer), user agents apply the aural rendering model [p. 277] .
5. From the annotated document tree, generate a *formatting structure*. Often, the formatting structure closely resembles the document tree, but it may also differ significantly, notably when authors make use of pseudo-elements and generated content. First, the formatting structure need not be "tree-shaped" at all -- the nature of the structure depends on the implementation. Second, the formatting structure may contain more or less information than the document tree. For instance, if an element in the document tree has a value of ’none’ for the ’display’ property, that element will generate nothing in the formatting structure. A list element, on the other hand, may generate more information in the formatting structure: the list element’s content and list style information (e.g., a bullet image).

   Note that the CSS user agent does not alter the document tree during this phase. In particular, content generated due to style sheets is not fed back to the document language processor (e.g., for reparsing).
6. Transfer the formatting structure to the target medium (e.g., print the results, display them on the screen, render them as speech, etc.).

Step 1 lies outside the scope of this specification (see, for example, [DOM]).
Steps 2-5 are addressed by the bulk of this specification.
Step 6 lies outside the scope of this specification.

2.3.1 The canvas

For all media, the term *canvas* describes "the space where the formatting structure is rendered." The canvas is infinite for each dimension of the space, but rendering generally occurs within a finite region of the canvas, established by the user agent according to the target medium. For instance, user agents rendering to a screen generally impose a minimum width and choose an initial width based on the dimensions of the viewport [p. 96] . User agents rendering to a page generally impose width and height constraints. Aural user agents may impose limits in audio space, but not in time.

2.3.2 CSS2 addressing model

CSS2 selectors [p. 53] and properties allow style sheets to refer to the following parts of a document or user agent:

- Elements in the document tree and certain relationships between them (see the section on selectors [p. 53]).
- Attributes of elements in the document tree, and values of those attributes (see the section on attribute selectors [p. 57]).
- Some parts of element content (see the :first-line [p. 66] and :first-letter [p. 66] pseudo-elements.
- Elements of the document tree when they are in a certain state (see the section on pseudo-classes [p. 61]).
- Some aspects of the canvas [p. 26] where the document will be rendered.
- Some system information (see the section on user interface [p. 271]).

2.4 CSS design principles

CSS2, as CSS1 before it, is based on a set of design principles:

- **Forward and backward compatibility**. CSS2 user agents will be able to understand CSS1 style sheets. CSS1 user agents will be able to read CSS2 style sheets and discard parts they don’t understand. Also, user agents with no CSS support will be able to display style-enhanced documents. Of course, the stylistic enhancements made possible by CSS will not be rendered, but all content will be presented.
- **Complementary to structured documents**. Style sheets complement structured documents (e.g., HTML and XML applications), providing stylistic information for the marked-up text. It should be easy to change the style sheet with little or no impact on the markup.
- **Vendor, platform, and device independence**. Style sheets enable documents to remain vendor, platform, and device independent. Style sheets themselves are also vendor and platform independent, but CSS2 allows you to target a style sheet for a group of devices (e.g., printers).
- **Maintainability**. By pointing to style sheets from documents, webmasters can simplify site maintenance and retain consistent look and feel throughout the site. For example, if the organization’s background color changes, only one file needs to be changed.

- **Simplicity**. CSS2 is more complex than CSS1, but it remains a simple style language which is human readable and writable. The CSS properties are kept independent of each other to the largest extent possible and there is generally only one way to achieve a certain effect.
- **Network performance**. CSS provides for compact encodings of how to present content. Compared to images or audio files, which are often used by authors to achieve certain rendering effects, style sheets most often decrease the content size. Also, fewer network connections have to be opened which further increases network performance.
- **Flexibility**. CSS can be applied to content in several ways. The key feature is the ability to cascade style information specified in the default (user agent) style sheet, user style sheets, linked style sheets, the document head, and in attributes for the elements forming the document body.
- **Richness**. Providing authors with a rich set of rendering effects increases the richness of the Web as a medium of expression. Designers have been longing for functionality commonly found in desktop publishing and slide-show applications. Some of the requested rendering effects conflict with device independence, but CSS2 goes a long way toward granting designers their requests.
- **Alternative language bindings**. The set of CSS properties described in this specification form a consistent formatting model for visual and aural presentations. This formatting model can be accessed through the CSS language, but bindings to other languages are also possible. For example, a JavaScript program may dynamically change the value of a certain element's 'color' property.
- **Accessibility**. Several CSS features will make the Web more accessible to users with disabilities:
    - Properties to control font appearance allow authors to eliminate inaccessible bit-mapped text images.
    - Positioning properties allow authors to eliminate mark-up tricks (e.g., invisible images) to force layout.
    - The semantics of `!important` rules mean that users with particular presentation requirements can override the author's style sheets.
    - The new 'inherit' value for all properties improves cascading generality and allows for easier and more consistent style tuning.
    - Improved media support, including media groups and the braille, embossed, and tty media types, will allow users and authors to tailor pages to those devices.
    - Aural properties give control over voice and audio output.
    - The attribute selectors, 'attr()' function, and 'content' property give access to alternate content.
    - Counters and section/paragraph numbering can improve document navigability and save on indenting spacing (important for braille devices). The 'word-spacing' and 'text-indent' properties also eliminate the need for extra whitespace in the document.

  Note. *For more information about designing accessible documents using CSS and HTML, please consult [WAI-PAGEAUTH].*

3 Conformance: Requirements and Recommendations

Contents

3.1 Definitions

In this section, we begin the formal specification of CSS2, starting with the contract between authors, users, and implementers.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (see [RFC2119]). However, for readability, these words do not appear in all uppercase letters in this specification.

At times, this specification recommends good practice for authors and user agents. These recommendations are not normative and conformance with this specification does not depend on their realization. These recommendations contain the expression "We recommend ...", "This specification recommends ...", or some similar wording.

Style sheet

A set of statements that specify presentation of a document.

Style sheets may have three different origins: author [p. 31] , user [p. 31] , and user agent [p. 31] . The interaction of these sources is described in the section on cascading and inheritance [p. 69] .

Valid style sheet

The validity of a style sheet depends on the level of CSS used for the style sheet. All valid CSS1 style sheets are valid CSS2 style sheets. However, some changes from CSS1 [p. 294] mean that a few CSS1 style sheets will have slightly different semantics in CSS2.

A valid CSS2 style sheet must be written according to the grammar of CSS2 [p. 309] . Furthermore, it must contain only at-rules, property names, and property values defined in this specification. An **illegal** (invalid) at-rule, property name, or property value is one that is not valid.

Source document

The document to which one or more style sheets refer. This is encoded in some language that represents the document as a tree of elements [p. 30] . Each element consists of a name that identifies the type of element, optionally a number of attributes [p. 30] , and a (possibly empty) content [p. 30] .

Document language
: The encoding language of the source document (e.g., HTML or an XML application).

Element
: (An SGML term, see [ISO8879].) The primary syntactic constructs of the document language. Most CSS style sheet rules use the names of these elements (such as "P", "TABLE", and "OL" for HTML) to specify rendering information for them.

Replaced element
: An element for which the CSS formatter knows only the intrinsic dimensions. In HTML, IMG, INPUT, TEXTAREA, SELECT, and OBJECT elements can be examples of replaced elements. For example, the content of the IMG element is often replaced by the image that the "src" attribute designates. CSS does not define how the intrinsic dimensions are found.

Intrinsic dimensions
: The width and height as defined by the element itself, not imposed by the surroundings. In CSS2 it is assumed that all replaced elements -- and only replaced elements -- come with intrinsic dimensions.

Attribute
: A value associated with an element, consisting of a name, and an associated (textual) value.

Content
: The content associated with an element in the source document; not all elements have content in which case they are called **empty**. The content of an element may include text, and it may include a number of sub-elements, in which case the element is called the **parent** of those sub-elements.

Rendered content
: The content of an element after the rendering that applies to it according to the relevant style sheets has been applied. The rendered content of a replaced element [p. 30] comes from outside the source document. Rendered content may also be alternate text for an element (e.g., the value of the HTML "alt" attribute), and may include items inserted implicitly or explicitly by the style sheet, such as bullets, numbering, etc.

Document tree
: The tree of elements encoded in the source document. Each element in this tree has exactly one parent, with the exception of the **root** element, which has none.

Child
: An element A is called the child of element B if an only if B is the parent of A.

Descendant
: An element A is called a descendant of an element B, if either (1) A is a child of B, or (2) A is the child of some element C that is a descendant of B.

Ancestor
: An element A is called an ancestor of an element B, if and only if B is a descendant of A.

Sibling
: An element A is called a sibling of an element B, if and only if B and A share the same parent element. Element A is a preceding sibling if it comes before B in the document tree. Element B is a following sibling if it comes after B in the document tree.

Preceding element

An element A is called a preceding element of an element B, if and only if (1) A is an ancestor of B or (2) A is a preceding sibling of B.

Following element

An element A is called a following element of an element B, if and only if B is a preceding element of A.

Author

An author is a person who writes documents and associated style sheets. An **authoring tool** generates documents and associated style sheets.

User

A user is a person who interacts with a user agent to view, hear, or otherwise use a document and its associated style sheet. The user may provide a personal style sheet that encodes personal preferences.

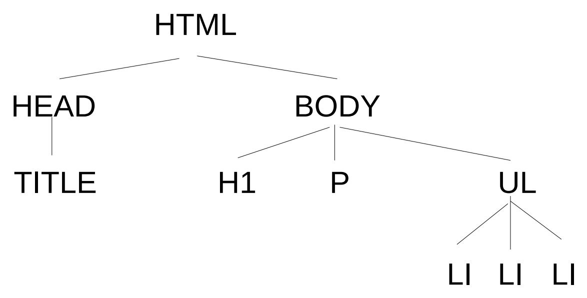
User agent (UA)

A user agent is any program that interprets a document written in the document language and applies associated style sheets according to the terms of this specification. A user agent may display a document, read it aloud, cause it to be printed, convert it to another format, etc.

Here is an example of a source document encoded in HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <TITLE>My home page</TITLE>
  <BODY>
    <H1>My home page</H1>
    <P>Welcome to my home page! Let me tell you about my favorite
                composers:
    <UL>
      <LI> Elvis Costello
      <LI> Johannes Brahms
      <LI> Georges Brassens
    </UL>
  </BODY>
</HTML>
```

This results in the following tree:



According to the definition of HTML, HEAD elements will be inferred during parsing and become part of the document tree even if the HEAD tags are not in the document source. Similarly, the parser knows where the P and LIs end, even though there are no </P> and </LI> tags in the source.

3.2 Conformance

This section defines conformance with the CSS2 specification only. There may be other levels of CSS in the future that may require a user agent to implement a different set of features in order to conform.

In general, the following points must be observed by a user agent claiming conformance to this specification:

1. It must support one or more of the CSS2 media types [p. 77] .
2. For each source document, it must attempt to retrieve all associated style sheets that are appropriate for the supported media types. If it cannot retrieve all associated style sheets (for instance, because of network errors), it must display the document using those it can retrieve.
3. It must parse the style sheets according to this specification. In particular, it must recognize all at-rules, blocks, declarations, and selectors (see the grammar of CSS2 [p. 309]). If a user agent encounters a property that applies for a supported media type, the user agent must parse the value according to the property definition. This means that the user agent must accept all valid values and must ignore declarations with invalid values. User agents must ignore rules that apply to unsupported media types [p. 77] .
4. For each element in a document tree [p. 30] , it must assign a value for every applicable property according to the property's definition and the rules of cascading and inheritance [p. 69] .
5. If the source document comes with alternate style sheets (such as with the "alternate" keyword in HTML 4.0 [HTML40]), the UA must allow the user to select one from among these style sheets and apply the selected one.

Not every user agent must observe every point, however:

- A user agent that *inputs* style sheets must respect points 1 - 3.
- An authoring tool is only required to output valid style sheets [p. 29]
- A user agent that *renders* a document with associated style sheets must respect points 1 - 5 and render the document according to the media-specific requirements set forth in this specification. Values [p. 70] may be approximated when required by the user agent.

The inability of a user agent to implement part of this specification due to the limitations of a particular device (e.g., a user agent cannot render colors on a monochrome monitor or page) does not imply non-conformance.

This specification recommends that a user agent allow the user to specify user style sheets.

3.3 Error conditions

In general, this document does not specify error handling behavior for user agents (e.g., how they behave when they cannot find a resource designated by a URI).

However, user agents must observe the rules for handling parsing errors [p. 42] .

Since user agents may vary in how they handle error conditions, authors and users must not rely on specific error recovery behavior.

3.4 The text/css content type

CSS style sheets that exist in separate files are sent over the Internet as a sequence of bytes accompanied by encoding information (see [HTML40], chapter 5). The structure of the transmission, termed a **message entity,** is defined by RFC 2045 and RFC 2068 (see [RFC2045] and [RFC2068]). A message entity with a content type of "text/css" represents an independent CSS document. The "text/css" content type has been registered by RFC 2138 ([RFC2318]).

4 CSS2 syntax and basic data types

Contents

4.1 Syntax

This section describes a grammar (and *forward-compatible parsing* rules) common to any version of CSS (including CSS2). Future versions of CSS will adhere to this core syntax, although they may add additional syntactic constraints.

These descriptions are normative. They are also complemented by the normative grammar rules presented in Appendix D [p. 309] .

4.1.1 Tokenization

All levels of CSS -- level 1, level 2, and any future levels -- use the same core syntax. This allows UAs to parse (though not completely understand) style sheets written in levels of CSS that didn’t exist at the time the UAs were created. Designers can use this feature to create style sheets that work with older user agents, while also exercising the possibilities of the latest levels of CSS.

At the lexical level, CSS style sheets consist of a sequence of tokens. The list of tokens for CSS2 is as follows. The definitions use Lex-style regular expressions. Octal codes refer to ISO 10646 ([ISO10646]). As in Lex, in case of multiple matches, the longest match determines the token.

Token	**Definition**
IDENT	*{ident}*
ATKEYWORD	@*{ident}*
STRING	*{string}*
HASH	#*{name}*
NUMBER	*{num}*
PERCENTAGE	*{num}*%
DIMENSION	*{num}{ident}*
URI	url\(*{w}{string}{w}*\) |url\(*{w}*([!#$%&*-~]|*{nonascii}*|*{escape}*)**{w}*\)
UNICODE-RANGE	U\+[0-9A-F?]{1,6}(-[0-9A-F]{1,6})?
CDO	<!--
CDC	-->
;	;
{	\{
}	\}
(	\(
)	\)
[	\[
]	\]
S	[\t\r\n\f]+
COMMENT	\/*[^*]**+([^/][^*]**+)*\/
FUNCTION	*{ident}*\(
INCLUDES	~=
DASHMATCH	|=
DELIM	*any other character not matched by the above rules*

The macros in curly braces ({}) above are defined as follows:

Macro	Definition
ident	*{nmstart}{nmchar}**
name	*{nmchar}+*
nmstart	[a-zA-Z]|*{nonascii}*|*{escape}*
nonascii	[^\0-\177]
unicode	\\[0-9a-f]{1,6}[\n\r\t\f]?
escape	*{unicode}*|\\[-~\200-\4177777]
nmchar	[a-z0-9-]|*{nonascii}*|*{escape}*
num	[0-9]+|[0-9]*\.[0-9]+
string	*{string1}*|*{string2}*
string1	\"([\t !#$%&(-~]|\\{nl}|\’|*{nonascii}*|*{escape}*)*\"
string2	\’([\t !#$%&(-~]|\\{nl}|\"|*{nonascii}*|*{escape}*)*\’
nl	\n|\r\n|\r|\f
w	[\t\r\n\f]\*

Below is the core syntax for CSS. The sections that follow describe how to use it. Appendix D [p. 309] describes a more restrictive grammar that is closer to the CSS level 2 language.

```
stylesheet  : [ CDO | CDC | S | statement ]*;
statement   : ruleset | at-rule;
at-rule     : ATKEYWORD S* any* [ block | ’;’ S* ];
block       : ’{’ S* [ any | block | ATKEYWORD S* | ’;’ ]* ’}’ S*;
ruleset     : selector? ’{’ S* declaration? [ ’;’ S* declaration? ]* ’}’ S*;
selector    : any+;
declaration : property ’:’ S* value;
property    : IDENT S*;
value       : [ any | block | ATKEYWORD S* ]+;
any         : [ IDENT | NUMBER | PERCENTAGE | DIMENSION | STRING
              | DELIM | URI | HASH | UNICODE-RANGE | INCLUDES
              | FUNCTION | DASHMATCH | ’(’ any* ’)’ | ’[’ any* ’]’ ] S*;
```

COMMENT tokens do not occur in the grammar (to keep it readable), but any number of these tokens may appear anywhere between other tokens.

The token S in the grammar above stands for whitespace. Only the characters "space" (Unicode code 32), "tab" (9), "line feed" (10), "carriage return" (13), and "form feed" (12) can occur in whitespace. Other space-like characters, such as "em-space" (8195) and "ideographic space" (12288), are never part of whitespace.

4.1.2 Keywords

Keywords have the form of identifiers. Keywords must not be placed between quotes ("..." or '...'). Thus,

```
red
```

is a keyword, but

```
"red"
```

is not. (It is a string [p. 50] .) Other illegal examples:
Illegal example(s):

```
width: "auto";
border: "none";
font-family: "serif";
background: "red";
```

4.1.3 Characters and case

The following rules always hold:

- All CSS style sheets are case-insensitive, except for parts that are not under the control of CSS. For example, the case-sensitivity of values of the HTML attributes "id" and "class", of font names, and of URIs lies outside the scope of this specification. Note in particular that element names are case-insensitive in HTML, but case-sensitive in XML.
- In CSS2, *identifiers* (including element names, classes, and IDs in selectors [p. 53]) can contain only the characters [A-Za-z0-9] and ISO 10646 characters 161 and higher, plus the hyphen (-); they cannot start with a hyphen or a digit. They can also contain escaped characters and any ISO 10646 character as a numeric code (see next item). For instance, the identifier "B&W?" may be written as "B\&W\?" or "B\26 W\3F".

  Note that Unicode is code-by-code equivalent to ISO 10646 (see [UNICODE] and [ISO10646]).
- In CSS2, a backslash (\) character indicates three types of character escapes.

  First, inside a string [p. 50] , a backslash followed by a newline is ignored (i.e., the string is deemed not to contain either the backslash or the newline).

  Second, it cancels the meaning of special CSS characters. Any character (except a hexadecimal digit) can be escaped with a backslash to remove its special meaning. For example, `"\""` is a string consisting of one double quote. Style sheet preprocessors must not remove these backslashes from a style sheet since that would change the style sheet's meaning.

  Third, backslash escapes allow authors to refer to characters they can't easily put in a document. In this case, the backslash is followed by at most six hexadecimal digits (0..9A..F), which stand for the ISO 10646 ([ISO10646]) character with that number. If a digit or letter follows the hexadecimal number, the end of the number needs to be made clear. There are two ways to do that:
  1. with a space (or other whitespace character): "\26 B" ("&B")
  2. by providing exactly 6 hexadecimal digits: "\000026B" ("&B")

In fact, these two methods may be combined. Only one whitespace character is ignored after a hexadecimal escape. Note that this means that a "real" space after the escape sequence must itself either be escaped or doubled.

- Backslash escapes are always considered to be part of an identifier [p. 38] or a string (i.e., "\7B" is not punctuation, even though "{" is, and "\32" is allowed at the start of a class name, even though "2" is not).

4.1.4 Statements

A CSS style sheet, for any version of CSS, consists of a list of *statements* (see the grammar above). There are two kinds of statements: *at-rules* and *rule sets.* There may be whitespace [p. 37] around the statements.

In this specification, the expressions "immediately before" or "immediately after" mean with no intervening whitespace or comments.

4.1.5 At-rules

At-rules start with an *at-keyword*, an ’@’ character followed immediately by an identifier [p. 38] (for example, ’@import’, ’@page’).

An at-rule consists of everything up to and including the next semicolon (;) or the next block, [p. 40] whichever comes first. A CSS user agent that encounters an unrecognized at-rule must ignore [p. 42] the whole of the at-rule and continue parsing after it.

CSS2 user agents must ignore [p. 42] any ’@import’ [p. 71] rule that occurs inside a block [p. 40] or that doesn’t precede all rule sets.

Illegal example(s):

Assume, for example, that a CSS2 parser encounters this style sheet:

```
@import "subs.css";
H1 { color: blue }
@import "list.css";
```

The second ’@import’ is illegal according to CSS2. The CSS2 parser ignores [p. 42] the whole at-rule, effectively reducing the style sheet to:

```
@import "subs.css";
H1 { color: blue }
```

Illegal example(s):

In the following example, the second ’@import’ rule is invalid, since it occurs inside a ’@media’ block [p. 40] .

```
@import "subs.css";
@media print {
  @import "print-main.css";
  BODY { font-size: 10pt }
}
H1 {color: blue }
```

4.1.6 Blocks

A *block* starts with a left curly brace ({) and ends with the matching right curly brace (}). In between there may be any characters, except that parentheses (( )), brackets ([ ]) and braces ({ }) must always occur in matching pairs and may be nested. Single (’) and double quotes (") must also occur in matching pairs, and characters between them are parsed as a string. See Tokenization [p. 35] above for the definition of a string.

Illegal example(s):

Here is an example of a block. Note that the right brace between the double quotes does not match the opening brace of the block, and that the second single quote is an escaped character [p. 38] , and thus doesn’t match the first single quote:

```
{ causta: "}" + ({7} * ’\’’) }
```

Note that the above rule is not valid CSS2, but it is still a block as defined above.

4.1.7 Rule sets, declaration blocks, and selectors

A rule set (also called "rule") consists of a selector followed by a declaration block.

A *declaration-block* (also called a {}-block in the following text) starts with a left curly brace ({) and ends with the matching right curly brace (}). In between there must be a list of zero or more semicolon-separated (;) declarations.

The *selector* (see also the section on selectors [p. 53]) consists of everything up to (but not including) the first left curly brace ({). A selector always goes together with a {}-block. When a user agent can’t parse the selector (i.e., it is not valid CSS2), it must ignore [p. 42] the {}-block as well.

CSS2 gives a special meaning to the comma (,) in selectors. However, since it is not known if the comma may acquire other meanings in future versions of CSS, the whole statement should be ignored [p. 42] if there is an error anywhere in the selector, even though the rest of the selector may look reasonable in CSS2.

Illegal example(s):

For example, since the "&" is not a valid token in a CSS2 selector, a CSS2 user agent must ignore [p. 42] the whole second line, and not set the color of H3 to red:

```
H1, H2 {color: green }
H3, H4 & H5 {color: red }
H6 {color: black }
```

Example(s):

Here is a more complex example. The first two pairs of curly braces are inside a string, and do not mark the end of the selector. This is a valid CSS2 statement.

```
P[example="public class foo\
{\
    private int x;\
\
    foo(int x) {\
        this.x = x;\
    }\
\
}"] { color: red }
```

4.1.8 Declarations and properties

A *declaration* is either empty or consists of a property, followed by a colon (:), followed by a value. Around each of these there may be whitespace [p. 37] .

Because of the way selectors work, multiple declarations for the same selector may be organized into semicolon (;) separated groups.

Example(s):

Thus, the following rules:

```
H1 { font-weight: bold }
H1 { font-size: 12pt }
H1 { line-height: 14pt }
H1 { font-family: Helvetica }
H1 { font-variant: normal }
H1 { font-style: normal }
```

are equivalent to:

```
H1 {
  font-weight: bold;
  font-size: 12pt;
  line-height: 14pt;
  font-family: Helvetica;
  font-variant: normal;
  font-style: normal
}
```

A property is an identifier [p. 38] . Any character may occur in the value, but parentheses ("( )"), brackets ("[ ]"), braces ("{ }"), single quotes (') and double quotes (") must come in matching pairs, and semicolons not in strings must be escaped [p. 38] . Parentheses, brackets, and braces may be nested. Inside the quotes, characters are parsed as a string.

The syntax of values is specified separately for each property, but in any case, values are built from identifiers, strings, numbers, lengths, percentages, URIs, colors, angles, times, and frequencies.

A user agent must ignore [p. 42] a declaration with an invalid property name or an invalid value. Every CSS2 property has its own syntactic and semantic restrictions on the values it accepts.

Illegal example(s):

For example, assume a CSS2 parser encounters this style sheet:

```
H1 { color: red; font-style: 12pt }  /* Invalid value: 12pt */
P { color: blue;  font-vendor: any;  /* Invalid prop.: font-vendor */
    font-variant: small-caps }
EM EM { font-style: normal }
```

The second declaration on the first line has an invalid value ’12pt’. The second declaration on the second line contains an undefined property ’font-vendor’. The CSS2 parser will ignore [p. 42] these declarations, effectively reducing the style sheet to:

```
H1 { color: red; }
P { color: blue;  font-variant: small-caps }
EM EM { font-style: normal }
```

4.1.9 Comments

Comments begin with the characters "/*" and end with the characters "*/". They may occur anywhere between tokens, and their contents have no influence on the rendering. Comments may not be nested.

CSS also allows the SGML comment delimiters ("<!--" and "-->") in certain places, but they do not delimit CSS comments. They are permitted so that style rules appearing in an HTML source document (in the STYLE element) may be hidden from pre-HTML 3.2 user agents. See the HTML 4.0 specification ([HTML40]) for more information.

4.2 Rules for handling parsing errors

In some cases, user agents must ignore part of an illegal style sheet. This specification defines *ignore* to mean that the user agent parses the illegal part (in order to find its beginning and end), but otherwise acts as if it had not been there.

To ensure that new properties and new values for existing properties can be added in the future, user agents are required to obey the following rules when they encounter the following scenarios:

- **Unknown properties.** User agents must ignore [p. 42] a declaration [p. 41] with an unknown property. For example, if the style sheet is:

  ```
  H1 { color: red; rotation: 70minutes }
  ```

  the user agent will treat this as if the style sheet had been

  ```
  H1 { color: red }
  ```

- **Illegal values.** User agents must ignore a declaration with an illegal value. For example:

  ```
  IMG { float: left }       /* correct CSS2 */
  IMG { float: left here }  /* "here" is not a value of ’float’ */
  IMG { background: "red" } /* keywords cannot be quoted in CSS2 */
  IMG { border-width: 3 }   /* a unit must be specified for length values */
  ```

  A CSS2 parser would honor the first rule and ignore [p. 42] the rest, as if the style sheet had been:

  ```
  IMG { float: left }
  IMG { }
  IMG { }
  IMG { }
  ```

A user agent conforming to a future CSS specification may accept one or more of the other rules as well.

- **Invalid at-keywords.** User agents must ignore [p. 42] an invalid at-keyword together with everything following it, up to and including the next semicolon (;) or block ({...}), whichever comes first. For example, consider the following:

```
@three-dee {
  @background-lighting {
    azimuth: 30deg;
    elevation: 190deg;
  }
  H1 { color: red }
}
H1 { color: blue }
```

  The ’@three-dee’ at-rule is not part of CSS2. Therefore, the whole at-rule (up to, and including, the third right curly brace) is ignored. [p. 42] A CSS2 user agent ignores [p. 42] it, effectively reducing the style sheet to:

```
H1 { color: blue }
```

4.3 Values

4.3.1 Integers and real numbers

Some value types may have integer values (denoted by <integer>) or real number values (denoted by <number>). Real numbers and integers are specified in decimal notation only. An <integer> consists of one or more digits "0" to "9". A <number> can either be an <integer>, or it can be zero or more digits followed by a dot (.) followed by one or more digits. Both integers and real numbers may be preceded by a "-" or "+" to indicate the sign.

Note that many properties that allow an integer or real number as a value actually restrict the value to some range, often to a non-negative value.

4.3.2 Lengths

Lengths refer to horizontal or vertical measurements.

The format of a length value (denoted by <length> in this specification) is an optional sign character (’+’ or ’-’, with ’+’ being the default) immediately followed by a <number> (with or without a decimal point) immediately followed by a unit identifier (e.g., px, deg, etc.). After the ’0’ length, the unit identifier is optional.

Some properties allow negative length values, but this may complicate the formatting model and there may be implementation-specific limits. If a negative length value cannot be supported, it should be converted to the nearest value that can be supported.

There are two types of length units: relative and absolute. *Relative length* units specify a length relative to another length property. Style sheets that use relative units will more easily scale from one medium to another (e.g., from a computer display to a laser printer).

Relative units are:

- **em**: the ’font-size’ of the relevant font
- **ex**: the ’x-height’ of the relevant font
- **px**: pixels, relative to the viewing device

Example(s):

```
H1 { margin: 0.5em }      /* em */
H1 { margin: 1ex }        /* ex */
P  { font-size: 12px }    /* px */
```

The ’em’ unit is equal to the computed value of the ’font-size’ property of the element on which it is used. The exception is when ’em’ occurs in the value of the ’font-size’ property itself, in which case it refers to the font size of the parent element. It may be used for vertical or horizontal measurement. (This unit is also sometimes called the quad-width in typographic texts.)

The ’ex’ unit is defined by the font’s ’x-height’. The x-height is so called because it is often equal to the height of the lowercase "x". However, an ’ex’ is defined even for fonts that don’t contain an "x".

Example(s):

The rule:

```
H1 { line-height: 1.2em }
```

means that the line height of H1 elements will be 20% greater than the font size of the H1 elements. On the other hand:

```
H1 { font-size: 1.2em }
```

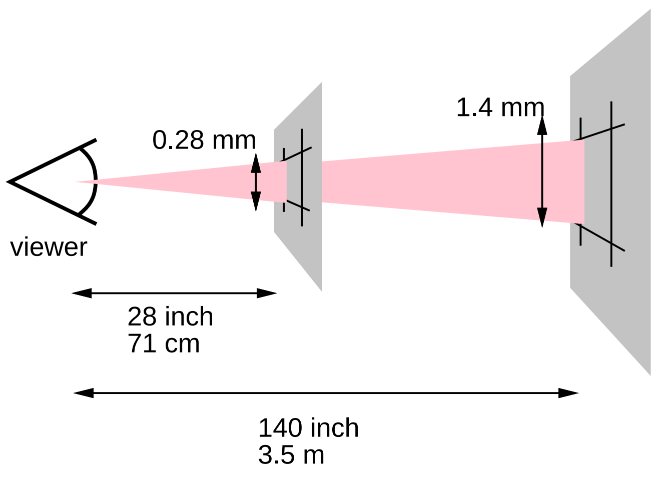
means that the font-size of H1 elements will be 20% greater than the font size inherited by H1 elements.

When specified for the root of the document tree [p. 30] (e.g., "HTML" in HTML), ’em’ and ’ex’ refer to the property’s initial value [p. 69] .

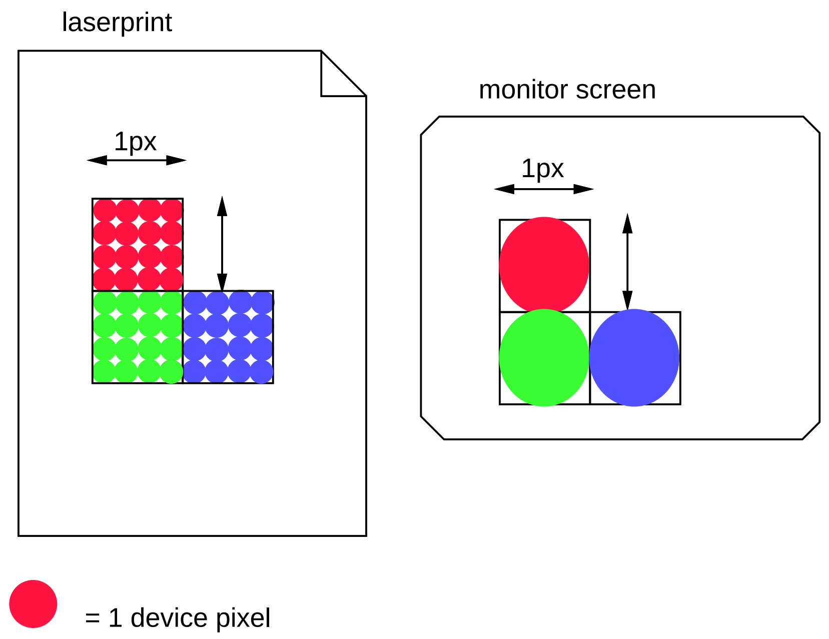
Pixel units are relative to the resolution of the viewing device, i.e., most often a computer display. If the pixel density of the output device is very different from that of a typical computer display, the user agent should rescale pixel values. It is recommended that the *reference pixel* be the visual angle of one pixel on a device with a pixel density of 90dpi and a distance from the reader of an arm’s length. For a nominal arm’s length of 28 inches, the visual angle is therefore about 0.0227 degrees.

For reading at arm’s length, 1px thus corresponds to about 0.28 mm (1/90 inch). When printed on a laser printer, meant for reading at a little less than arm’s length (55 cm, 21 inches), 1px is about 0.21 mm. On a 300 dots-per-inch (dpi) printer, that may be rounded up to 3 dots (0.25 mm); on a 600 dpi printer, it can be rounded to 5 dots.

The two images below illustrate the effect of viewing distance on the size of a pixel and the effect of a device’s resolution. In the first image, a reading distance of 71 cm (28 inch) results in a px of 0.28 mm, while a reading distance of 3.5 m (12 feet) requires a px of 1.4 mm.



In the second image, an area of 1px by 1px is covered by a single dot in a low-resolution device (a computer screen), while the same area is covered by 16 dots in a higher resolution device (such as a 400 dpi laser printer).



Child elements do not inherit the relative values specified for their parent; they (generally) inherit the computed values [p. 70] .

Example(s):

In the following rules, the computed 'text-indent' value of H1 elements will be 36pt, not 45pt, if H1 is a child of the BODY element.

```
BODY {
  font-size: 12pt;
  text-indent: 3em;  /* i.e., 36pt */
}
H1 { font-size: 15pt }
```

Absolute length units are only useful when the physical properties of the output medium are known. The absolute units are:

- **in**: inches -- 1 inch is equal to 2.54 centimeters.
- **cm**: centimeters

- **mm**: millimeters
- **pt**: points -- the points used by CSS2 are equal to 1/72th of an inch.
- **pc**: picas -- 1 pica is equal to 12 points.

Example(s):

```
H1 { margin: 0.5in }      /* inches  */
H2 { line-height: 3cm }   /* centimeters */
H3 { word-spacing: 4mm }  /* millimeters */
H4 { font-size: 12pt }    /* points */
H4 { font-size: 1pc }     /* picas */
```

In cases where the specified length cannot be supported, user agents must approximate it in the actual value.

4.3.3 Percentages

The format of a percentage value (denoted by <percentage> in this specification) is an optional sign character ('+' or '-', with '+' being the default) immediately followed by a <number> immediately followed by '%'.

Percentage values are always relative to another value, for example a length. Each property that allows percentages also defines the value to which the percentage refers. The value may be that of another property for the same element, a property for an ancestor element, or a value of the formatting context (e.g., the width of a containing block [p. 96]). When a percentage value is set for a property of the root [p. 30] element and the percentage is defined as referring to the inherited value of some property, the resultant value is the percentage times the initial value [p. 69] of that property.

Example(s):

Since child elements (generally) inherit the computed values [p. 70] of their parent, in the following example, the children of the P element will inherit a value of 12pt for 'line-height', not the percentage value (120%):

```
P { font-size: 10pt }
P { line-height: 120% }  /* 120% of 'font-size' */
```

4.3.4 URL + URN = URI

URLs (Uniform Resource Locators, see [RFC1738] and [RFC1808]) provide the address of a resource on the Web. An expected new way of identifying resources is called URN (Uniform Resource Name). Together they are called URIs (Uniform Resource Identifiers, see [URI]). This specification uses the term URI.

URI values in this specification are denoted by <uri>. The functional notation used to designate URIs in property values is "url()", as in:

Example(s):

```
BODY { background: url("http://www.bg.com/pinkish.gif") }
```

The format of a URI value is 'url(' followed by optional whitespace [p. 37] followed by an optional single quote (') or double quote (") character followed by the URI itself, followed by an optional single quote (') or double quote (") character followed by optional whitespace followed by ')'. The two quote characters must be the same.

Example(s):
An example without quotes:

```
LI { list-style: url(http://www.redballs.com/redball.png) disc }
```

Parentheses, commas, whitespace characters, single quotes (') and double quotes (") appearing in a URI must be escaped with a backslash: '\(', '\)', '\,'.
Depending on the type of URI, it might also be possible to write the above characters as URI-escapes (where "(" = %28, ")" = %29, etc.) as described in [URI].
In order to create modular style sheets that are not dependent on the absolute location of a resource, authors may use relative URIs. Relative URIs (as defined in [RFC1808]) are resolved to full URIs using a base URI. RFC 1808, section 3, defines the normative algorithm for this process. For CSS style sheets, the base URI is that of the style sheet, not that of the source document.
Example(s):
For example, suppose the following rule:

```
BODY { background: url("yellow") }
```

is located in a style sheet designated by the URI:

```
http://www.myorg.org/style/basic.css
```

The background of the source document's BODY will be tiled with whatever image is described by the resource designated by the URI

```
http://www.myorg.org/style/yellow
```

User agents may vary in how they handle URIs that designate unavailable or inapplicable resources.

4.3.5 Counters

Counters are denoted by identifiers (see the 'counter-increment' and 'counter-reset' properties). To refer to the value of a counter, the notation 'counter(<identifier>)' or 'counter(<identifier>, <list-style-type>)' is used. The default style is 'decimal'.
To refer to a sequence of nested counters of the same name, the notation is 'counters(<identifier>, <string>)' or 'counters(<identifier>, <string>, <list-style-type>)'. See "Nested counters and scope" [p. 162] in the chapter on generated content [p. 153] .
In CSS2, the values of counters can only be referred to from the 'content' property. Note that 'none' is a possible <list-style-type>: 'counter(x, none)' yields an empty string.
Example(s):
Here is a style sheet that numbers paragraphs (P) for each chapter (H1). The paragraphs are numbered with roman numerals, followed by a period and a space:

```
P {counter-increment: par-num}
H1 {counter-reset: par-num}
P:before {content: counter(par-num, upper-roman) ". "}
```

Counters that are not in the scope [p. 162] of any ’counter-reset’, are assumed to have been reset to 0 by a ’counter-reset’ on the root element.

4.3.6 Colors

A <color> is either a keyword or a numerical RGB specification.

The list of keyword color names is: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow. These 16 colors are defined in HTML 4.0 ([HTML40]). In addition to these color keywords, users may specify keywords that correspond to the colors used by certain objects in the user’s environment. Please consult the section on system colors [p. 272] for more information.

Example(s):

```
BODY {color: black; background: white }
H1 { color: maroon }
H2 { color: olive }
```

The RGB color model is used in numerical color specifications. These examples all specify the same color:

Example(s):

```
EM { color: #f00 }              /* #rgb */
EM { color: #ff0000 }           /* #rrggbb */
EM { color: rgb(255,0,0) }      /* integer range 0 - 255 */
EM { color: rgb(100%, 0%, 0%) } /* float range 0.0% - 100.0% */
```

The format of an RGB value in hexadecimal notation is a ’#’ immediately followed by either three or six hexadecimal characters. The three-digit RGB notation (#rgb) is converted into six-digit form (#rrggbb) by replicating digits, not by adding zeros. For example, #fb0 expands to #ffbb00. This ensures that white (#ffffff) can be specified with the short notation (#fff) and removes any dependencies on the color depth of the display.

The format of an RGB value in the functional notation is ’rgb(’ followed by a comma-separated list of three numerical values (either three integer values or three percentage values) followed by ’)’. The integer value 255 corresponds to 100%, and to F or FF in the hexadecimal notation: rgb(255,255,255) = rgb(100%,100%,100%) = #FFF. Whitespace [p. 37] characters are allowed around the numerical values.

All RGB colors are specified in the sRGB color space (see [SRGB]). User agents may vary in the fidelity with which they represent these colors, but using sRGB provides an unambiguous and objectively measurable definition of what the color should be, which can be related to international standards (see [COLORIMETRY]).

Conforming user agents may limit their color-displaying efforts to performing a gamma-correction on them. sRGB specifies a display gamma of 2.2 under specified viewing conditions. User agents should adjust the colors given in CSS such that, in combination with an output device’s "natural" display gamma, an effective display gamma of 2.2 is produced. See the section on gamma correction [p. 193] for further details. Note that only colors specified in CSS are affected; e.g., images are expected to carry their own color information.

Values outside the device gamut should be clipped: the red, green, and blue values must be changed to fall within the range supported by the device. For a typical CRT monitor, whose device gamut is the same as sRGB, the three rules below are equivalent:

Example(s):

```
EM { color: rgb(255,0,0) }       /* integer range 0 - 255 */
EM { color: rgb(300,0,0) }       /* clipped to rgb(255,0,0) */
EM { color: rgb(255,-10,0) }     /* clipped to rgb(255,0,0) */
EM { color: rgb(110%, 0%, 0%) }  /* clipped to rgb(100%,0%,0%) */
```

Other devices, such as printers, have different gamuts to sRGB; some colors outside the 0..255 sRGB range will be representable (inside the device gamut), while other colors inside the 0..255 sRGB range will be outside the device gamut and will thus be clipped.

Note. *Although colors can add significant amounts of information to document and make them more readable, please consider that certain color combinations may cause problems for users with color blindness. If you use a background image or set the background color, please adjust foreground colors accordingly.*

4.3.7 Angles

Angle values (denoted by <angle> in the text) are used with aural style sheets [p. 277] .

Their format is an optional sign character ('+' or '-', with '+' being the default) immediately followed by a <number> immediately followed by an angle unit identifier.

Angle unit identifiers are:

- **deg**: degrees
- **grad**: grads
- **rad**: radians

Angle values may be negative. They should be normalized to the range 0-360deg by the user agent. For example, -10deg and 350deg are equivalent.

For example, a right angle is '90deg' or '100grad' or '1.570796326794897rad'.

4.3.8 Times

Time values (denoted by <time> in the text) are used with aural style sheets [p. 277] .

Their format is a <number> immediately followed by a time unit identifier.

Time unit identifiers are:

- **ms**: milliseconds
- **s**: seconds

Time values may not be negative.

4.3.9 Frequencies

Frequency values (denoted by <frequency> in the text) are used with aural cascading style sheets [p. 277] .

Their format is a <number> immediately followed by a frequency unit identifier.

Frequency unit identifiers are:

- **Hz**: Hertz
- **kHz**: kilo Hertz

Frequency values may not be negative.

For example, 200Hz (or 200hz) is a bass sound, and 6kHz (or 6khz) is a treble sound.

4.3.10 Strings

Strings can either be written with double quotes or with single quotes. Double quotes cannot occur inside double quotes, unless escaped (as ’\"’ or as ’\22’). Analogously for single quotes ("\’" or "\27").

Example(s):

```
"this is a ’string’"
"this is a \"string\""
’this is a "string"’
’this is a \’string\’’
```

A string cannot directly contain a newline. To include a newline in a string, use the escape "\A" (hexadecimal A is the line feed character in Unicode, but represents the generic notion of "newline" in CSS). See the ’content’ property for an example.

It is possible to break strings over several lines, for aesthetic or other reasons, but in such a case the newline itself has to be escaped with a backslash (\). For instance, the following two selectors are exactly the same:

Example(s):

```
A[TITLE="a not s\
o very long title"] {/*...*/}
A[TITLE="a not so very long title"] {/*...*/}
```

4.4 CSS document representation

A CSS style sheet is a sequence of characters from the Universal Character Set (see [ISO10646]). For transmission and storage, these characters must be encoded by a character encoding that supports the set of characters available in US-ASCII (e.g., ISO 8859-x, SHIFT JIS, etc.). For a good introduction to character sets and character encodings, please consult the HTML 4.0 specification ([HTML40], chapter 5), See also the XML 1.0 specification ([XML10], sections 2.2 and 4.3.3, and Appendix F.

When a style sheet is embedded in another document, such as in the STYLE element or "style" attribute of HTML, the style sheet shares the character encoding of the whole document.

When a style sheet resides in a separate file, user agents must observe the following priorities when determining a document’s character encoding (from highest priority to lowest):

1. An HTTP "charset" parameter in a "Content-Type" field.
2. The @charset at-rule.
3. Mechanisms of the language of the referencing document (e.g., in HTML, the "charset" attribute of the LINK element).

At most one @charset rule may appear in an external style sheet -- it must *not* appear in an embedded style sheet -- and it must appear at the very start of the document, not preceded by any characters. After "@charset", authors specify the name of a character encoding. The name must be a charset name as described in the IANA registry (See [IANA]. Also, see [CHARSETS] for a complete list of charsets). For example:

Example(s):

@charset "ISO-8859-1";

This specification does not mandate which character encodings a user agent must support.

Note that reliance on the @charset construct theoretically poses a problem since there is no *a priori* information on how it is encoded. In practice, however, the encodings in wide use on the Internet are either based on ASCII, UTF-16, UCS-4, or (rarely) on EBCDIC. This means that in general, the initial byte values of a document enable a user agent to detect the encoding family reliably, which provides enough information to decode the @charset rule, which in turn determines the exact character encoding.

4.4.1 Referring to characters not represented in a character encoding

A style sheet may have to refer to characters that cannot be represented in the current character encoding. These characters must be written as escaped [p. 38] references to ISO 10646 characters. These escapes serve the same purpose as numeric character references in HTML or XML documents (see [HTML40], chapters 5 and 25).

The character escape mechanism should be used when only a few characters must be represented this way. If most of a document requires escaping, authors should encode it with a more appropriate encoding (e.g., if the document contains a lot of Greek characters, authors might use "ISO-8859-7" or "UTF-8").

Intermediate processors using a different character encoding may translate these escaped sequences into byte sequences of that encoding. Intermediate processors must not, on the other hand, alter escape sequences that cancel the special meaning of an ASCII character.

Conforming user agents [p. 32] must correctly map to Unicode all characters in any character encodings that they recognize (or they must behave as if they did).

For example, a document transmitted as ISO-8859-1 (Latin-1) cannot contain Greek letters directly: "κουρος" (Greek: "kouros") has to be written as "\3BA\3BF\3C5\3C1\3BF\3C2".

Note. *In HTML 4.0, numeric character references are interpreted in "style" attribute values but not in the content of the STYLE element. Because of this asymmetry, we recommend that authors use the CSS character escape mechanism rather than numeric character references for both the "style" attribute and the STYLE element. For example, we recommend:*

```
<SPAN style="voice-family: D\FC rst">...</SPAN>
```

rather than:

```
<SPAN style="voice-family: D&#252;rst">...</SPAN>
```

---

5 Selectors

Contents

5.1 Pattern matching

In CSS, pattern matching rules determine which style rules apply to elements in the document tree [p. 30] . These patterns, called selectors, may range from simple element names to rich contextual patterns. If all conditions in the pattern are true for a certain element, the selector *matches* the element.

The case-sensitivity of document language element names in selectors depends on the document language. For example, in HTML, element names are case-insensitive, but in XML they are case-sensitive.

The following table summarizes CSS2 selector syntax:

Pattern	Meaning	Described in section
*	Matches any element.	Universal selector [p. 55]

E	Matches any E element (i.e., an element of type E).	Type selectors [p. 56]
E F	Matches any F element that is a descendant of an E element.	Descendant selectors [p. 56]
E > F	Matches any F element that is a child of an element E.	Child selectors [p. 57]
E:first-child	Matches element E when E is the first child of its parent.	The :first-child pseudo-class [p. 62]
E:link E:visited	Matches element E if E is the source anchor of a hyperlink of which the target is not yet visited (:link) or already visited (:visited).	The link pseudo-classes [p. 63]
E:active E:hover E:focus	Matches E during certain user actions.	The dynamic pseudo-classes [p. 63]
E:lang(c)	Matches element of type E if it is in (human) language c (the document language specifies how language is determined).	The :lang() pseudo-class [p. 65]
E + F	Matches any F element immediately preceded by an element E.	Adjacent selectors [p. 57]
E[foo]	Matches any E element with the "foo" attribute set (whatever the value).	Attribute selectors [p. 57]
E[foo="warning"]	Matches any E element whose "foo" attribute value is exactly equal to "warning".	Attribute selectors [p. 57]
E[foo~="warning"]	Matches any E element whose "foo" attribute value is a list of space-separated values, one of which is exactly equal to "warning".	Attribute selectors [p. 57]
E[lang|="en"]	Matches any E element whose "lang" attribute has a hyphen-separated list of values beginning (from the left) with "en".	Attribute selectors [p. 57]
DIV.warning	*HTML only*. The same as DIV[class~="warning"].	Class selectors [p. 59]
E#myid	Matches any E element ID equal to "myid".	ID selectors [p. 60]

5.2 Selector syntax

A *simple selector* is either a type selector [p. 56] or universal selector [p. 55] followed immediately by zero or more attribute selectors [p. 57] , ID selectors [p. 60] , or pseudo-classes [p. 61] , in any order. The simple selector matches if all of its components match.

A *selector* is a chain of one or more simple selectors separated by combinators. *Combinators* are: whitespace, ">", and "+". Whitespace may appear between a combinator and the simple selectors around it.

The elements of the document tree that match a selector are called *subjects* of the selector. A selector consisting of a single simple selector matches any element satisfying its requirements. Prepending a simple selector and combinator to a chain imposes additional matching constraints, so the subjects of a selector are always a subset of the elements matching the rightmost simple selector.

One pseudo-element [p. 61] may be appended to the last simple selector in a chain, in which case the style information applies to a subpart of each subject.

5.2.1 Grouping

When several selectors share the same declarations, they may be grouped into a comma-separated list.

Example(s):

In this example, we condense three rules with identical declarations into one. Thus,

```
H1 { font-family: sans-serif }
H2 { font-family: sans-serif }
H3 { font-family: sans-serif }
```

is equivalent to:

```
H1, H2, H3 { font-family: sans-serif }
```

CSS offers other "shorthand" mechanisms as well, including multiple declarations [p. 41] and shorthand properties [p. 16] .

5.3 Universal selector

The universal selector, written "*", matches the name of any element type. It matches any single element in the document tree. [p. 30]

If the universal selector is not the only component of a simple selector [p. 55] , the "*" may be omitted. For example:

- `*[LANG=fr]` and `[LANG=fr]` are equivalent.
- `*.warning` and `.warning` are equivalent.
- `*#myid` and `#myid` are equivalent.

5.4 Type selectors

A *type selector* matches the name of a document language element type. A type selector matches every instance of the element type in the document tree.

Example(s):

The following rule matches all H1 elements in the document tree:

```
H1 { font-family: sans-serif }
```

5.5 Descendant selectors

At times, authors may want selectors to match an element that is the descendant of another element in the document tree (e.g., "Match those EM elements that are contained by an H1 element"). Descendant selectors express such a relationship in a pattern. A descendant selector is made up of two or more selectors separated by whitespace [p. 37] . A descendant selector of the form "A B" matches when an element B is an arbitrary descendant of some ancestor [p. 30] element A.

Example(s):

For example, consider the following rules:

```
H1 { color: red }
EM { color: red }
```

Although the intention of these rules is to add emphasis to text by changing its color, the effect will be lost in a case such as:

```
<H1>This headline is <EM>very</EM> important</H1>
```

We address this case by supplementing the previous rules with a rule that sets the text color to blue whenever an EM occurs anywhere within an H1:

```
H1 { color: red }
EM { color: red }
H1 EM { color: blue }
```

The third rule will match the EM in the following fragment:

```
<H1>This <SPAN class="myclass">headline
is <EM>very</EM> important</SPAN></H1>
```

Example(s):

The following selector:

```
DIV * P
```

matches a P element that is a grandchild or later descendant of a DIV element. Note the whitespace on either side of the "*".

Example(s):

The selector in the following rule, which combines descendant and attribute selectors [p. 57] , matches any element that (1) has the "href" attribute set and (2) is inside a P that is itself inside a DIV:

```
DIV P *[href]
```

5.6 Child selectors

A *child selector* matches when an element is the child [p. 30] of some element. A child selector is made up of two or more selectors separated by ">".

Example(s):

The following rule sets the style of all P elements that are children of BODY:

```
BODY > P { line-height: 1.3 }
```

Example(s):

The following example combines descendant selectors and child selectors:

```
DIV OL>LI P
```

It matches a P element that is a descendant of an LI; the LI element must be the child of an OL element; the OL element must be a descendant of a DIV. Notice that the optional whitespace around the ">" combinator has been left out.

For information on selecting the first child of an element, please see the section on the :first-child [p. 62] pseudo-class below.

5.7 Adjacent sibling selectors

Adjacent sibling selectors have the following syntax: E1 + E2, where E2 is the subject of the selector. The selector matches if E1 and E2 share the same parent in the document tree and E1 immediately precedes E2.

In some contexts, adjacent elements generate formatting objects whose presentation is handled automatically (e.g., collapsing vertical margins between adjacent boxes). The "+" selector allows authors to specify additional style to adjacent elements.

Example(s):

Thus, the following rule states that when a P element immediately follows a MATH element, it should not be indented:

```
MATH + P { text-indent: 0 }
```

The next example reduces the vertical space separating an H1 and an H2 that immediately follows it:

```
H1 + H2 { margin-top: -5mm }
```

Example(s):

The following rule is similar to the one in the previous example, except that it adds an attribute selector. Thus, special formatting only occurs when H1 has `class="opener"`:

```
H1.opener + H2 { margin-top: -5mm }
```

5.8 Attribute selectors

CSS2 allows authors to specify rules that match attributes defined in the source document.

5.8.1 Matching attributes and attribute values

Attribute selectors may match in four ways:

`[att]`
: Match when the element sets the "att" attribute, whatever the value of the attribute.

`[att=val]`
: Match when the element’s "att" attribute value is exactly "val".

`[att~=val]`
: Match when the element’s "att" attribute value is a space-separated list of "words", one of which is exactly "val". If this selector is used, the words in the value must not contain spaces (since they are separated by spaces).

`[att|=val]`
: Match when the element’s "att" attribute value is a hyphen-separated list of "words", beginning with "val". The match always starts at the beginning of the attribute value. This is primarily intended to allow language subcode matches (e.g., the "lang" attribute in HTML) as described in RFC 1766 ([RFC1766]).

Attribute values must be identifiers or strings. The case-sensitivity of attribute names and values in selectors depends on the document language.

Example(s):

For example, the following attribute selector matches all H1 elements that specify the "title" attribute, whatever its value:

```
H1[title] { color: blue; }
```

Example(s):

In the following example, the selector matches all SPAN elements whose "class" attribute has exactly the value "example":

```
SPAN[class=example] { color: blue; }
```

Multiple attribute selectors can be used to refer to several attributes of an element, or even several times the same attribute.

Example(s):

Here, the selector matches all SPAN elements whose "hello" attribute has exactly the value "Cleveland" and whose "goodbye" attribute has exactly the value "Columbus":

```
SPAN[hello="Cleveland"][goodbye="Columbus"] { color: blue; }
```

Example(s):

The following selectors illustrate the differences between "=" and "~=". The first selector will match, for example, the value "copyright copyleft copyeditor" for the "rel" attribute. The second selector will only match when the "href" attribute has the value "http://www.w3.org/".

```
A[rel~="copyright"]
A[href="http://www.w3.org/"]
```

Example(s):
The following rule hides all elements for which the value of the "lang" attribute is "fr" (i.e., the language is French).

```
*[LANG=fr] { display : none }
```

Example(s):
The following rule will match for values of the "lang" attribute that begin with "en", including "en", "en-US", and "en-cockney":

```
*[LANG|="en"] { color : red }
```

Example(s):
Similarly, the following aural style sheet rules allow a script to be read aloud in different voices for each role:

```
DIALOGUE[character=romeo]
     { voice-family: "Lawrence Olivier", charles, male }

DIALOGUE[character=juliet]
     { voice-family: "Vivien Leigh", victoria, female }
```

5.8.2 Default attribute values in DTDs

Matching takes place on attribute values in the document tree. For document languages other than HTML, default attribute values may be defined in a DTD or elsewhere. Style sheets should be designed so that they work even if the default values are not included in the document tree.
Example(s):
For example, consider an element EXAMPLE with an attribute "notation" that has a default value of "decimal". The DTD fragment might be

```
<!ATTLIST EXAMPLE notation (decimal,octal) "decimal">
```

If the style sheet contains the rules

```
EXAMPLE[notation=decimal] { /*... default property settings ...*/ }
EXAMPLE[notation=octal] { /*... other settings...*/ }
```

then to catch the cases where this attribute is set by default, and not explicitly, the following rule might be added:

```
EXAMPLE { /*... default property settings ...*/ }
```

Because this selector is less specific [p. 74] than an attribute selector, it will only be used for the default case. Care has to be taken that all other attribute values that don’t get the same style as the default are explicitly covered.

5.8.3 Class selectors

For style sheets used with HTML, authors may use the dot (.) notation as an alternative to the "~=" notation when matching on the "class" attribute. Thus, for HTML, "DIV.value" and "DIV[class~=value]" have the same meaning. The attribute value must immediately follow the ".".

Example(s):

For example, we can assign style information to all elements with `class~="pastoral"` as follows:

```
*.pastoral { color: green }  /* all elements with class~=pastoral */
```

or just

```
.pastoral { color: green }  /* all elements with class~=pastoral */
```

The following assigns style only to H1 elements with `class~="pastoral"`:

```
H1.pastoral { color: green }  /* H1 elements with class~=pastoral */
```

Given these rules, the first H1 instance below would not have green text, while the second would:

```
<H1>Not green</H1>
<H1 class="pastoral">Very green</H1>
```

To match a subset of "class" values, each value must be preceded by a ".", in any order.

Example(s):

For example, the following rule matches any P element whose "class" attribute has been assigned a list of space-separated values that includes "pastoral" and "marine":

```
P.pastoral.marine { color: green }
```

This rule matches when `class="pastoral blue aqua marine"` but does not match for `class="pastoral blue"`.

Note. *CSS gives so much power to the "class" attribute, that authors could conceivably design their own "document language" based on elements with almost no associated presentation (such as DIV and SPAN in HTML) and assigning style information through the "class" attribute. Authors should avoid this practice since the structural elements of a document language often have recognized and accepted meanings and author-defined classes may not.*

5.9 ID selectors

Document languages may contain attributes that are declared to be of type ID. What makes attributes of type ID special is that no two such attributes can have the same value; whatever the document language, an ID attribute can be used to uniquely identify its element. In HTML all ID attributes are named "id"; XML applications may name ID attributes differently, but the same restriction applies.

The ID attribute of a document language allows authors to assign an identifier to one element instance in the document tree. CSS ID selectors match an element instance based on its identifier. A CSS ID selector contains a "#" immediately followed by the ID value.

Example(s):

The following ID selector matches the H1 element whose ID attribute has the value "chapter1":

```
H1#chapter1 { text-align: center }
```

In the following example, the style rule matches the element that has the ID value "z98y". The rule will thus match for the P element:

```
<HEAD>
  <TITLE>Match P</TITLE>
  <STYLE type="text/css">
    *#z98y { letter-spacing: 0.3em }
  </STYLE>
</HEAD>
<BODY>
   <P id=z98y>Wide text</P>
</BODY>
```

In the next example, however, the style rule will only match an H1 element that has an ID value of "z98y". The rule will not match the P element in this example:

```
<HEAD>
  <TITLE>Match H1 only</TITLE>
  <STYLE type="text/css">
    H1#z98y { letter-spacing: 0.5em }
  </STYLE>
</HEAD>
<BODY>
   <P id=z98y>Wide text</P>
</BODY>
```

ID selectors have a higher precedence than attribute selectors. For example, in HTML, the selector `#p123` is more specific than `[ID=p123]` in terms of the cascade [p. 69] .

Note. *In XML 1.0 [XML10], the information about which attribute contains an element's IDs is contained in a DTD. When parsing XML, UAs do not always read the DTD, and thus may not know what the ID of an element is. If a style sheet designer knows or suspects that this will be the case, he should use normal attribute selectors instead: `[name=p371]` instead of `#p371`. However, the cascading order of normal attribute selectors is different from ID selectors. It may be necessary to add an "!important" priority to the declarations: `[name=p371] {color: red ! important}`. Of course, elements in XML 1.0 documents without a DTD do not have IDs at all.*

5.10 Pseudo-elements and pseudo-classes

In CSS2, style is normally attached to an element based on its position in the document tree [p. 30] . This simple model is sufficient for many cases, but some common publishing scenarios may not be possible due to the structure of the document tree [p. 30] . For instance, in HTML 4.0 (see [HTML40]), no element refers to the first line of a paragraph, and therefore no simple CSS selector may refer to it.

CSS introduces the concepts of *pseudo-elements* and *pseudo-classes* to permit formatting based on information that lies outside the document tree.

- Pseudo-elements create abstractions about the document tree beyond those specified by the document language. For instance, document languages do not offer mechanisms to access the first letter or first line of an element's content. CSS pseudo-elements allow style sheet designers to refer to this otherwise inaccessible information. Pseudo-elements may also provide style sheet designers a way to assign style to content that does not exist in the source document (e.g., the :before and :after [p. 153] pseudo-elements give access to generated content).

- Pseudo-classes classify elements on characteristics other than their name, attributes or content; in principle characteristics that cannot be deduced from the document tree. Pseudo-classes may be dynamic, in the sense that an element may acquire or lose a pseudo-class while a user interacts with the document. The exception is ':first-child' [p. 62] , which *can* be deduced from the document tree.

Neither pseudo-elements nor pseudo-classes appear in the document source or document tree.

Pseudo-classes are allowed anywhere in selectors while pseudo-elements may only appear after the subject [p. 55] of the selector.

Pseudo-elements and pseudo-class names are case-insensitive.

Some pseudo-classes are mutually exclusive, while others can be applied simultaneously to the same element. In case of conflicting rules, the normal cascading order [p. 73] determines the outcome.

Conforming HTML user agents [p. 32] may ignore [p. 42] all rules with :first-line or :first-letter in the selector, or, alternatively, may only support a subset of the properties on these pseudo-elements.

5.11 Pseudo-classes

5.11.1 :first-child pseudo-class

The :first-child pseudo-class matches an element that is the first child of some other element.

Example(s):

In the following example, the selector matches any P element that is the first child of a DIV element. The rule suppresses indentation for the first paragraph of a DIV:

```
DIV > P:first-child { text-indent: 0 }
```

This selector would match the P inside the DIV of the following fragment:

```
<P> The last P before the note.
<DIV class="note">
   <P> The first P inside the note.
</DIV>
```

but would not match the second P in the following fragment:

```
<P> The last P before the note.
<DIV class="note">
   <H2>Note</H2>
   <P> The first P inside the note.
</DIV>
```

Example(s):
The following rule sets the font weight to ’bold’ for any EM element that is some descendant of a P element that is a first child:

```
P:first-child EM { font-weight : bold }
```

Note that since anonymous [p. 98] boxes are not part of the document tree, they are not counted when calculating the first child.
For example, the EM in:

```
<P>abc <EM>default</EM>
```

is the first child of the P.
The following two selectors are equivalent:

```
* > A:first-child   /* A is first child of any element */
A:first-child       /* Same */
```

5.11.2 The link pseudo-classes: :link and :visited

User agents commonly display unvisited links differently from previously visited ones. CSS provides the pseudo-classes ’:link’ and ’:visited’ to distinguish them:

- The :link pseudo-class applies for links that have not yet been visited.
- The :visited pseudo-class applies once the link has been visited by the user.

Note. *After a certain amount of time, user agents may choose to return a visited link to the (unvisited) ’:link’ state.*
The two states are mutually exclusive.
The document language determines which elements are hyperlink source anchors. For example, in HTML 4.0, the link pseudo-classes apply to A elements with an "href" attribute. Thus, the following two CSS2 declarations have similar effect:

```
A:link { color: red }
:link  { color: red }
```

Example(s):
If the following link:

```
<A class="external" href="http://out.side/">external link</A>
```

has been visited, this rule:

```
A.external:visited { color: blue }
```

will cause it to be blue.

5.11.3 The dynamic pseudo-classes: :hover, :active, and :focus

Interactive user agents sometimes change the rendering in response to user actions. CSS provides three pseudo-classes for common cases:

- The :hover pseudo-class applies while the user designates an element (with some pointing device), but does not activate it. For example, a visual user agent could apply this pseudo-class when the cursor (mouse pointer) hovers over a box generated by the element. User agents not supporting interactive media [p. 79] do not have to support this pseudo-class. Some conforming user agents supporting interactive media [p. 79] may not be able to support this pseudo-class (e.g., a pen device).
- The :active pseudo-class applies while an element is being activated by the user. For example, between the times the user presses the mouse button and releases it.
- The :focus pseudo-class applies while an element has the focus (accepts keyboard events or other forms of text input).

These pseudo-classes are not mutually exclusive. An element may match several of them at the same time.

CSS doesn't define which elements may be in the above states, or how the states are entered and left. Scripting may change whether elements react to user events or not, and different devices and UAs may have different ways of pointing to, or activating elements.

User agents are not required to reflow a currently displayed document due to pseudo-class transitions. For instance, a style sheet may specify that the ’font-size’ of an :active link should be larger than that of an inactive link, but since this may cause letters to change position when the reader selects the link, a UA may ignore the corresponding style rule.

Example(s):

```
A:link    { color: red }    /* unvisited links */
A:visited { color: blue }   /* visited links   */
A:hover   { color: yellow } /* user hovers     */
A:active  { color: lime }   /* active links    */
```

Note that the A:hover must be placed after the A:link and A:visited rules, since otherwise the cascading rules will hide the ’color’ property of the A:hover rule. Similarly, because A:active is placed after A:hover, the active color (lime) will apply when the user both activates and hovers over the A element.

Example(s):

An example of combining dynamic pseudo-classes:

```
A:focus { background: yellow }
A:focus:hover { background: white }
```

The last selector matches A elements that are in pseudo-class :focus and in pseudo-class :hover.

For information about the presentation of focus outlines, please consult the section on dynamic focus outlines [p. 274] .

Note. *In CSS1, the ’:active’ pseudo-class was mutually exclusive with ’:link’ and ’:visited’. That is no longer the case. An element can be both ’:visited’ and ’:active’ (or ’:link’ and ’:active’) and the normal cascading rules determine which properties apply.*

5.11.4 The language pseudo-class: :lang

If the document language specifies how the human language of an element is determined, it is possible to write selectors in CSS that match an element based on its language. For example, in HTML [HTML40], the language is determined by a combination of the "lang" attribute, the META element, and possibly by information from the protocol (such as HTTP headers). XML uses an attribute called XML:LANG, and there may be other document language-specific methods for determining the language.

The pseudo-class ':lang(C)' matches if the element is in language C. Here C is a language code as specified in HTML 4.0 [HTML40] and RFC 1766 [RFC1766]. It is matched the same way as for the '|=' operator [p. 57] .

Example(s):

The following rules set the quotation marks for an HTML document that is either in French or German:

```
HTML:lang(fr) { quotes: '« ' ' »' }
HTML:lang(de) { quotes: '»' '«' '\2039' '\203A' }
:lang(fr) > Q { quotes: '« ' ' »' }
:lang(de) > Q { quotes: '»' '«' '\2039' '\203A' }
```

The second pair of rules actually set the 'quotes' property on Q elements according to the language of its parent. This is done because the choice of quote marks is typically based on the language of the element around the quote, not the quote itself: like this piece of French “à l'improviste” in the middle of an English text uses the English quotation marks.

5.12 Pseudo-elements

5.12.1 The :first-line pseudo-element

The :first-line pseudo-element applies special styles to the first formatted line of a paragraph. For instance:

```
P:first-line { text-transform: uppercase }
```

The above rule means "change the letters of the first line of every paragraph to uppercase". However, the selector "P:first-line" does not match any real HTML element. It does match a pseudo-element that conforming user agents [p. 32] will insert at the beginning of every paragraph.

Note that the length of the first line depends on a number of factors, including the width of the page, the font size, etc. Thus, an ordinary HTML paragraph such as:

```
<P>This is a somewhat long HTML
paragraph that will be broken into several
lines. The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

the lines of which happen to be broken as follows:

```
THIS IS A SOMEWHAT LONG HTML PARAGRAPH THAT
will be broken into several lines. The first
line will be identified by a fictional tag
sequence. The other lines will be treated as
ordinary lines in the paragraph.
```

might be "rewritten" by user agents to include the *fictional tag sequence* for :first-line. This fictional tag sequence helps to show how properties are inherited.

```
<P><P:first-line> This is a somewhat long HTML
paragraph that will </P:first-line> be broken into several
lines. The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

If a pseudo-element breaks up a real element, the desired effect can often be described by a fictional tag sequence that closes and then re-opens the element. Thus, if we mark up the previous paragraph with a SPAN element:

```
<P><SPAN class="test"> This is a somewhat long HTML
paragraph that will be broken into several
lines.</SPAN> The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

the user agent could generate the appropriate start and end tags for SPAN when inserting the fictional tag sequence for :first-line.

```
<P><P:first-line><SPAN class="test"> This is a
somewhat long HTML
paragraph that will </SPAN></P:first-line><SPAN class="test"> be
broken into several
lines.</SPAN> The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

The :first-line pseudo-element can only be attached to a block-level element.

The :first-line pseudo-element is similar to an inline-level element, but with certain restrictions. Only the following properties apply to a :first-line pseudo-element: font properties, [p. 198] color properties, [p. 187] background properties, [p. 188] ’word-spacing’, ’letter-spacing’, ’text-decoration’, ’vertical-align’, ’text-transform’, ’line-height’, ’text-shadow’, and ’clear’.

5.12.2 The :first-letter pseudo-element

The :first-letter pseudo-element may be used for "initial caps" and "drop caps", which are common typographical effects. This type of initial letter is similar to an inline-level element if its ’float’ property is ’none’, otherwise it is similar to a floated element.

These are the properties that apply to :first-letter pseudo-elements: font properties, [p. 198] color properties, [p. 187] background properties, [p. 188] ’text-decoration’, ’vertical-align’ (only if ’float’ is ’none’), ’text-transform’, ’line-height’, margin properties, [p. 85] padding properties, [p. 87] border properties, [p. 88] ’float’, ’text-shadow’, and ’clear’.

The following CSS2 will make a drop cap initial letter span two lines:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
 <HEAD>
  <TITLE>Drop cap initial letter</TITLE>
  <STYLE type="text/css">
   P              { font-size: 12pt; line-height: 12pt }
   P:first-letter { font-size: 200%; font-style: italic;
                    font-weight: bold; float: left }
   SPAN           { text-transform: uppercase }
  </STYLE>
 </HEAD>
 <BODY>
  <P><SPAN>The first</SPAN> few words of an article
    in The Economist.</P>
 </BODY>
</HTML>
```

This example might be formatted as follows:

T HE FIRST few
words of an
article in the
Economist

The fictional tag sequence is:

```
<P>
<SPAN>
<P:first-letter>
T
</P:first-letter>he first
</SPAN>
few words of an article in the Economist.
</P>
```

Note that the :first-letter pseudo-element tags abut the content (i.e., the initial character), while the :first-line pseudo-element start tag is inserted right after the start tag of the element to which it is attached.

In order to achieve traditional drop caps formatting, user agents may approximate font sizes, for example to align baselines. Also, the glyph outline may be taken into account when formatting.

Punctuation (i.e, characters defined in Unicode [UNICODE] in the "open" (Ps), "close" (Pe), and "other" (Po) punctuation classes), that precedes the first letter should be included, as in:

"A bird in
the hand
is worth
two in the bush,"
says an old proverb.

The :first-letter pseudo-element matches parts of block-level [p. 97] elements only.

Some languages may have specific rules about how to treat certain letter combinations. In Dutch, for example, if the letter combination "ij" appears at the beginning of a word, both letters should be considered within the :first-letter pseudo-element.

Example(s):

The following example illustrates how overlapping pseudo-elements may interact. The first letter of each P element will be green with a font size of ’24pt’. The rest of the first formatted line will be ’blue’ while the rest of the paragraph will be ’red’.

```
P { color: red; font-size: 12pt }
P:first-letter { color: green; font-size: 200% }
P:first-line { color: blue }

<P>Some text that ends up on two lines</P>
```

Assuming that a line break will occur before the word "ends", the fictional tag sequence for this fragment might be:

```
<P>
<P:first-line>
<P:first-letter>
S
</P:first-letter>ome text that
</P:first-line>
ends up on two lines
</P>
```

Note that the :first-letter element is inside the :first-line element. Properties set on :first-line are inherited by :first-letter, but are overridden if the same property is set on :first-letter.

5.12.3 The :before and :after pseudo-elements

The ’:before’ and ’:after’ pseudo-elements can be used to insert generated content before or after an element’s content. They are explained in the section on generated text. [p. 153]

Example(s):

```
H1:before {content: counter(chapno, upper-roman) ". "}
```

When the :first-letter and :first-line pseudo-elements are combined with :before and :after, they apply to the first letter or line of the element including the inserted text.

Example(s):

```
P.special:before {content: "Special! "}
P.special:first-letter {color: #ffd800}
```

This will render the "S" of "Special!" in gold.

6 Assigning property values, Cascading, and Inheritance

Contents

6.1 Specified, computed, and actual values

Once a user agent has parsed a document and constructed a document tree [p. 30] , it must assign, for every element in the tree, a value to every property that applies to the target media type [p. 77] .

The final value of a property is the result of a three-step calculation: the value is determined through specification (the "specified value"), then resolved into an absolute value if necessary (the "computed value"), and finally transformed according to the limitations of the local environment (the "actual value").

6.1.1 Specified values

User agents must first assign a specified value to a property based on the following mechanisms (in order of precedence):

1. If the cascade [p. 72] results in a value, use it.
2. Otherwise, if the property is inherited [p. 70] , use the value of the parent element, generally the computed value.
3. Otherwise use the property’s initial value. The initial value of each property is indicated in the property’s definition.

Since it has no parent, the root of the document tree [p. 30] cannot use values from the parent element; in this case, the initial value is used if necessary.

6.1.2 Computed values

Specified values may be absolute (i.e., they are not specified relative to another value, as in ’red’ or ’2mm’) or relative (i.e., they are specified relative to another value, as in ’auto’, ’2em’, and ’12%’). For absolute values, no computation is needed to find the computed value.

Relative values, on the other hand, must be transformed into computed values: percentages must be multiplied by a reference value (each property defines which value that is), values with relative units (em, ex, px) must be made absolute by multiplying with the appropriate font or pixel size, ’auto’ values must be computed by the formulas given with each property, certain keywords (’smaller’, ’bolder’, ’inherit’) must be replaced according to their definitions.

In most cases, elements inherit computed values. However, there are some properties whose specified value may be inherited (e.g., the number value for the ’line-height’ property). In the cases where child elements do not inherit the computed value, this is described in the property definition.

6.1.3 Actual values

A computed 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 computed width. The actual value is the computed value after any approximations have been applied.

6.2 Inheritance

Some values are inherited by the children of an element in the document tree [p. 30] . Each property defines [p. 14] whether it is inherited or not.

Suppose there is an H1 element with an emphasizing element (EM) inside:

```
<H1>The headline <EM>is</EM> important!</H1>
```

If no color has been assigned to the EM element, the emphasized "is" will inherit the color of the parent element, so if H1 has the color blue, the EM element will likewise be in blue.

To set a "default" style property for a document, authors may set the property on the root of the document tree. In HTML, for example, the HTML or BODY elements can serve this function. Note that this will work even if the author omits the BODY tag in the HTML source since the HTML parser will infer the missing tag.

Example(s):

For example, since the ’color’ property is inherited, all descendants of the BODY element will inherit the color ’black’:

```
BODY { color: black; }
```

Specified percentage values are not inherited; computed values are.

Example(s):

For example, given the following style sheet:

```
BODY { font-size: 10pt }
H1 { font-size: 120% }
```

and this document fragment:

```
<BODY>
  <H1>A <EM>large</EM> heading</H1>
</BODY>
```

the ’font-size’ property for the H1 element will have the computed value ’12pt’ (120% times 10pt, the parent’s value). Since the computed value of ’font-size’ is inherited, the EM element will have the computed value ’12pt’ as well. If the user agent does not have the 12pt font available, the actual value of ’font-size’ for both H1 and EM might be, for example, ’11pt’.

6.2.1 The ’inherit’ value

Each property may also have a specified value of ’inherit’, which means that, for a given element, the property takes the same computed value [p. 70] as the property for the element’s parent. The inherited value, which is normally only used as a fallback value, can be strengthened by setting ’inherit’ explicitly.

Example(s):

In the example below, the ’color’ and ’background’ properties are set on the BODY element. On all other elements, the ’color’ value will be inherited and the background will be transparent. If these rules are part of the user’s style sheet, black text on a white background will be enforced throughout the document.

```
BODY {
  color: black !important;
  background: white !important;
}

* {
  color: inherit !important;
  background: transparent;
}
```

6.3 The @import rule

The *’@import’* rule allows users to import style rules from other style sheets. Any @import rules must precede all rule sets in a style sheet. The ’@import’ keyword must be followed by the URI of the style sheet to include. A string is also allowed; it will be interpreted as if it had url(...) around it.

Example(s):

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");
```

So that user agents can avoid retrieving resources for unsupported media types [p. 77] , authors may specify media-dependent @import rules. These conditional imports specify comma-separated media types after the URI.

Example(s):

The following rules have the same effect as if the imported style sheet were wrapped in an @media rule for the same media, but it may save the UA a fruitless download.

```
@import url("fineprint.css") print;
@import url("bluish.css") projection, tv;
```

In the absence of any media types, the import is unconditional. Specifying 'all' for the medium has the same effect.

6.4 The cascade

Style sheets may have three different origins: author, user, and user agent.

- **Author**. 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**: 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 behave as if it did).
- **User agent**: Conforming user agents [p. 32] must apply a *default style sheet* (or behave as if they did) prior to all other style sheets for a document. 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 "A sample style sheet for HTML 4.0" [p. 291] for a recommended default style sheet for HTML 4.0 documents.

  Note that the default style sheet may change if system settings are modified by the user (e.g., system colors). However, due to limitations in a user agent's internal implementation, it may be impossible to change the values in the default style sheet.

Style sheets from these three origins will overlap in scope, and they interact according to the cascade.

The CSS cascade assigns a weight to each style rule. When several rules apply, the one with the greatest weight takes precedence.

By default, rules in author style sheets have more weight than rules in user style sheets. Precedence is reversed, however, for "!important" rules. All rules user and author rules have more weight than rules in the UA's default style sheet.

Imported style sheets also cascade and their weight depends on their import order. Rules specified in a given style sheet override rules imported from other style sheets. Imported style sheets can themselves import and override other style sheets, recursively, and the same precedence rules apply.

6.4.1 Cascading order

To find the value for an element/property combination, user agents must apply the following sorting order:

1. Find all declarations that apply to the element and property in question, for the target media type [p. 77] . Declarations apply if the associated selector matches [p. 53] the element in question.
2. The primary sort of the declarations is by weight and origin: for normal declarations, author style sheets override user style sheets which override the default style sheet. For "!important" declarations, user style sheets override author style sheets which override the default style sheet. "!important" declaration override normal declarations. An imported style sheet has the same origin as the style sheet that imported it.
3. The secondary sort is by specificity [p. 74] of selector: more specific selectors will override more general ones. Pseudo-elements and pseudo-classes are counted as normal elements and classes, respectively.
4. Finally, sort by order specified: if two rules have the same weight, origin and specificity, the latter specified wins. Rules in imported style sheets are considered to be before any rules in the style sheet itself.

Apart from the "!important" setting on individual declarations, this strategy gives author's style sheets higher weight than those of the reader. It is therefore important that the user agent give the user the ability to turn off the influence of a certain style sheet, e.g., through a pull-down menu.

6.4.2 !important rules

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 (see cascade rule 3).

However, for balance, an "!important" declaration (the keywords "!" and "important" follow the declaration) takes precedence over a normal declaration. Both author and user style sheets may contain "!important" declarations, and user "!important" rules override author "!important" rules. This CSS feature improves accessibility of documents by giving users with special requirements (large fonts, color combinations, etc.) control over presentation.

Note. *This is a semantic change since CSS1. In CSS1, author "!important" rules took precedence over user "!important" rules.*

Declaring a shorthand property (e.g., 'background') to be "!important" is equivalent to declaring all of its sub-properties to be "!important".

Example(s):

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 styles sheet. The second declaration will also win due to being marked "!important". However, the third rule 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 rule 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: 12pt sans-serif !important }
P { font-size: 24pt }
```

6.4.3 Calculating a selector’s specificity

A selector’s specificity is calculated as follows:

- count the number of ID attributes in the selector (= a)
- count the number of other attributes and pseudo-classes in the selector (= b)
- count the number of element names in the selector (= c)
- ignore pseudo-elements.

Concatenating the three numbers a-b-c (in a number system with a large base) gives the specificity.

Example(s):

Some examples:

```
*              {}  /* a=0 b=0 c=0 -> specificity =   0 */
LI             {}  /* a=0 b=0 c=1 -> specificity =   1 */
UL LI          {}  /* a=0 b=0 c=2 -> specificity =   2 */
UL OL+LI       {}  /* a=0 b=0 c=3 -> specificity =   3 */
H1 + *[REL=up]{}  /* a=0 b=1 c=1 -> specificity =  11 */
UL OL LI.red   {}  /* a=0 b=1 c=3 -> specificity =  13 */
LI.red.level   {}  /* a=0 b=2 c=1 -> specificity =  21 */
#x34y          {}  /* a=1 b=0 c=0 -> specificity = 100 */
```

In HTML, values of an element’s "style" attribute are style sheet rules. These rules have no selectors, but for the purpose of step 3 of the cascade algorithm, they are considered to have an ID selector (specificity: a=1, b=0, c=0). For the purpose of step 4, they are considered to be after all other rules.

```
<HEAD>
<STYLE type="text/css">
  #x97z { color: blue }
</STYLE>
</HEAD>
<BODY>
<P ID=x97z style="color: red">
</BODY>
```

In the above example, the color of the P element would be red. Although the specificity is the same for both declarations, the declaration in the "style" attribute will override the one in the STYLE element because of cascading rule 4.

6.4.4 Precedence of non-CSS presentational hints

The UA may choose to honor presentational hints from other sources than style sheets, for example the FONT element or the "align" attribute in HTML. If so, the non-CSS presentational hints must be translated to the corresponding CSS rules with specificity equal to zero. The rules are assumed to be at the start of the author style sheet and may be overridden by subsequent style sheet rules.

Note. *In a transition phase, this policy will make it easier for stylistic attributes to coexist with style sheets.*

Note. *In CSS1, the non-CSS presentational hints were given a specificity equal to 1, not 0. The change is due to the introduction of the universal selector, which has a specificity of 0.*

---

7 Media types

Contents

7.1 Introduction to media types

One of the most important features of style sheets is that they specify how a document is to be presented on different media: on the screen, on paper, with a speech synthesizer, with a braille device, etc.

Certain CSS properties are only designed for certain media (e.g., the ’cue-before’ property for aural user agents). On occasion, however, style sheets for different media types may share a property, but require different values for that property. For example, the ’font-size’ property is useful both for screen and print media. However, the two media are different enough to require different values for the common property; a document will typically need a larger font on a computer screen than on paper. Experience also shows that sans-serif fonts are easier to read on screen, while fonts with serifs are easier to read on paper. For these reasons, it is necessary to express that a style sheet -- or a section of a style sheet -- applies to certain media types.

7.2 Specifying media-dependent style sheets

There are currently two ways to specify media dependencies for style sheets:

- Specify the target medium from a style sheet with the @media or @import at-rules.

  Example(s):

  ```
  @import url("loudvoice.css") aural;
  @media print {
    /* style sheet for print goes here */
  }
  ```

- Specify the target medium within the document language. For example, in HTML 4.0 ([HTML40]), the "media" attribute on the LINK element specifies the target media of an external style sheet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
   <HEAD>
      <TITLE>Link to a target medium</TITLE>
      <LINK rel="stylesheet" type="text/css"
         media="print, handheld" href="foo.css">
   </HEAD>
   <BODY>
      <P>The body...
   </BODY>
</HTML>
```

The @import [p. 71] rule is defined in the chapter on the cascade [p. 69] .

7.2.1 The @media rule

An @media rule specifies the target media types [p. 78] (separated by commas) of a set of rules (delimited by curly braces). The @media construct allows style sheet rules for various media in the same style sheet:

```
@media print {
  BODY { font-size: 10pt }
}
@media screen {
  BODY { font-size: 12pt }
}
@media screen, print {
  BODY { line-height: 1.2 }
}
```

7.3 Recognized media types

A CSS *media type* names a set of CSS properties. A user agent that claims to support a media type by name must implement all of the properties that apply to that media type.

The names chosen for CSS media types reflect target devices for which the relevant properties make sense. In the following list of CSS media types, the parenthetical descriptions are not normative. They only give a sense of what device the media type is meant to refer to.

all
: Suitable for all devices.

aural
: Intended for speech synthesizers. See the section on aural style sheets [p. 277] for details.

braille
: Intended for braille tactile feedback devices.

embossed
: Intended for paged braille printers.

handheld
: Intended for handheld devices (typically small screen, monochrome, limited bandwidth).

print
: Intended for paged, opaque material and for documents viewed on screen in print preview mode. Please consult the section on paged media [p. 175] for

information about formatting issues that are specific to paged media.

projection
: Intended for projected presentations, for example projectors or print to transparencies. Please consult the section on paged media [p. 175] for information about formatting issues that are specific to paged media.

screen
: Intended primarily for color computer screens.

tty
: Intended for media using a fixed-pitch character grid, such as teletypes, terminals, or portable devices with limited display capabilities. Authors should not use pixel units [p. 44] with the "tty" media type.

tv
: Intended for television-type devices (low resolution, color, limited-scrollability screens, sound available).

Media type names are case-insensitive.

Due to rapidly changing technologies, CSS2 does not specify a definitive list of media types that may be values for @media.

Note. *Future versions of CSS may extend this list. Authors should not rely on media type names that are not yet defined by a CSS specification.*

7.3.1 Media groups

Each CSS property definition specifies the media types for which the property must be implemented by a conforming user agent [p. 32] . Since properties generally apply to several media, the "Applies to media" section of each property definition lists media groups rather than individual media types. Each property applies to all media types in the media groups listed in its definition.

CSS2 defines the following media groups:

- **continuous** or **paged**. "Both" means that the property in question applies to both media groups.
- **visual**, **aural**, or **tactile**.
- **grid** (for character grid devices), or **bitmap**. "Both" means that the property in question applies to both media groups.
- **interactive** (for devices that allow user interaction), or **static** (for those that don’t). "Both" means that the property in question applies to both media groups.
- **all** (includes all media types)

The following table shows the relationships between media groups and media types:

Relationship between media groups and media types

Media Types	**Media Groups**			
	continuous/paged	**visual/aural/tactile**	**grid/bitmap**	**interactive/static**
aural	continuous	aural	N/A	both
braille	continuous	tactile	grid	both
emboss	paged	tactile	grid	both
handheld	both	visual	both	both
print	paged	visual	bitmap	static
projection	paged	visual	bitmap	static
screen	continuous	visual	bitmap	both
tty	continuous	visual	grid	both
tv	both	visual, aural	bitmap	both

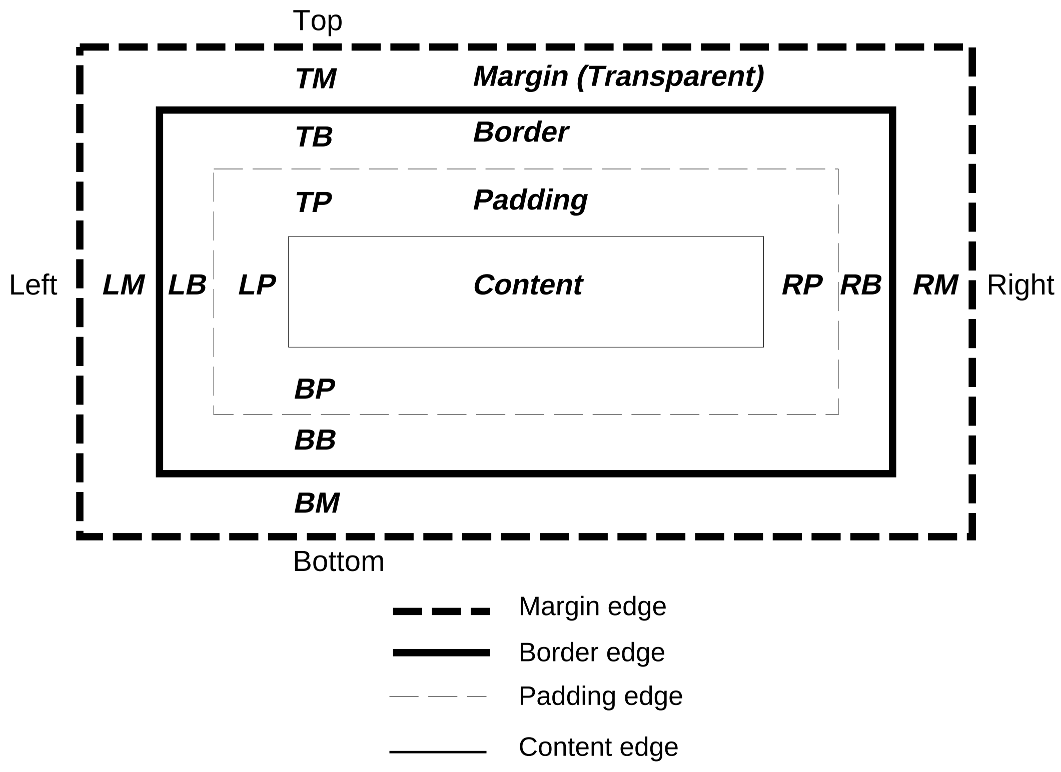
8 Box model

Contents

The CSS box model describes the rectangular boxes that are generated for elements in the document tree [p. 30] and laid out according to the visual formatting model [p. 95] . The page box [p. 175] is a special kind of box that is described in detail in the section on paged media [p. 175] .

8.1 Box dimensions

Each box has a *content area* (e.g., text, an image, etc.) and optional surrounding *padding*, *border*, and *margin* areas; the size of each area is specified by properties defined below. The following diagram shows how these areas relate and the terminology used to refer to pieces of margin, border, and padding:



The margin, border, and padding can be broken down into left, right, top, and bottom segments (e.g., in the diagram, "LM" for left margin, "RP" for right padding, "TB" for top border, etc.).

The perimeter of each of the four areas (content, padding, border, and margin) is called an "edge", so each box has four edges:

content edge or **inner edge**
: The content edge surrounds the element’s rendered content [p. 30] .

padding edge
: The padding edge surrounds the box padding. If the padding has 0 width, the padding edge is the same as the content edge. The padding edge of a box defines the edges of the containing block [p. 96] established by the box.

border edge
: The border edge surrounds the box’s border. If the border has 0 width, the border edge is the same as the padding edge.

margin edge or **outer edge**
: The margin edge surrounds the box margin. If the margin has 0 width, the margin edge is the same as the border edge.

Each edge may be broken down into a left, right, top, and bottom edge.

The dimensions of the content area of a box -- the *content width* and *content height* -- depend on several factors: whether the element generating the box has the ’width’ or ’height’ property set, whether the box contains text or other boxes, whether the box is a table, etc. Box widths and heights are discussed in the chapter on visual formatting model details [p. 131] .

The *box width* is given by the sum of the left and right margins, border, and padding, and the content width. The *height* is given by the sum of the top and bottom margins, border, and padding, and the content height.

The background style of the various areas of a box are determined as follows:

- *Content area*: The ’background’ property of the generating element.
- *Padding area*: The ’background’ property of the generating element.
- *Border area*: The border properties [p. 88] of the generating element.
- *Margin area*: Margins are always transparent.

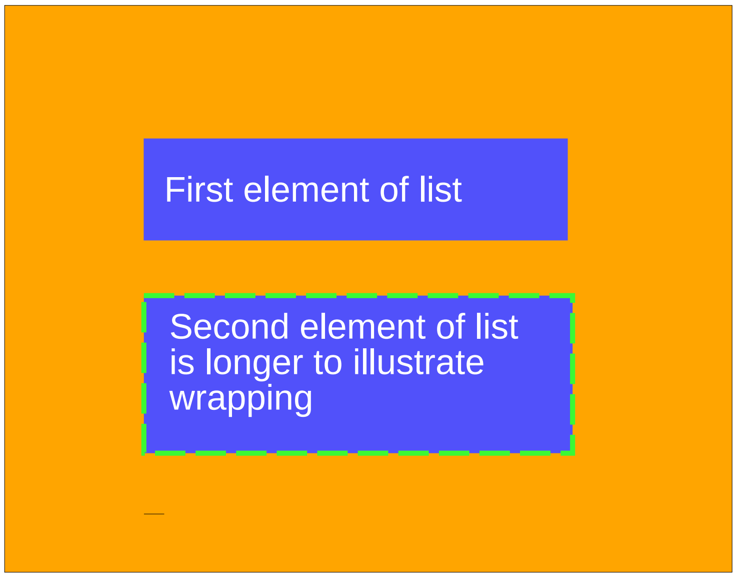
8.2 Example of margins, padding, and borders

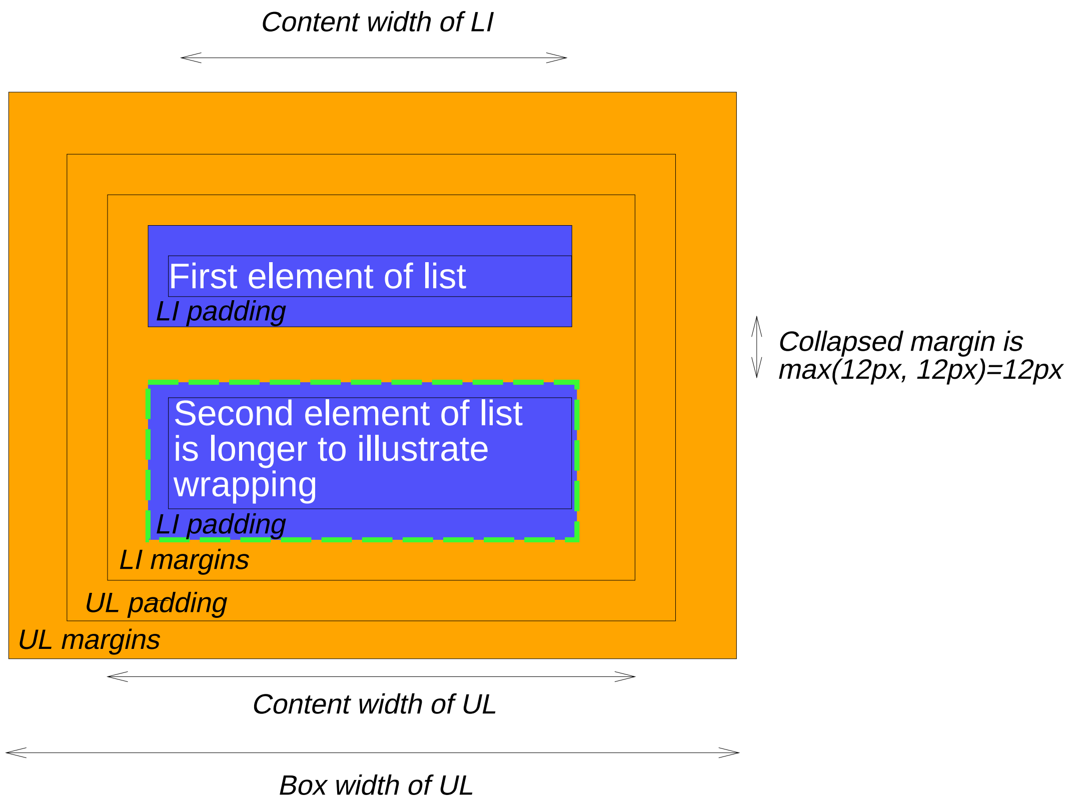
This example illustrates how margins, padding, and borders interact. The example HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Examples of margins, padding, and borders</TITLE>
    <STYLE type="text/css">
      UL {
        background: green;
        margin: 12px 12px 12px 12px;
        padding: 3px 3px 3px 3px;
                                     /* No borders set */
      }
      LI {
        color: black;                /* text color is black */
        background: gray;            /* Content, padding will be gray */
        margin: 12px 12px 12px 12px;
        padding: 12px 0px 12px 12px; /* Note 0px padding right */
        list-style: none             /* no glyphs before a list item */
                                     /* No borders set */
      }
      LI.withborder {
        border-style: dashed;
        border-width: medium;        /* sets border width on all sides */
        border-color: black;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <UL>
      <LI>First element of list
      <LI class="withborder">Second element of list is longer
           to illustrate wrapping.
    </UL>
  </BODY>
</HTML>
```

results in a document tree [p. 30] with (among other relationships) a UL element that has two LI children.

The first of the following diagrams illustrates what this example would produce. The second illustrates the relationship between the margins, padding, and borders of the UL elements and those of its children LI elements.





Note that:

- The content width [p. 82] for each LI box is calculated top-down; the containing block [p. 96] for each LI box is established by the UL element.
- The height of each LI box is given by its content height [p. 82] , plus top and bottom padding, borders, and margins. Note that vertical margins between the LI boxes collapse. [p. 86]
- The right padding of the LI boxes has been set to zero width (the ’padding’

property). The effect is apparent in the second illustration.

- The margins of the LI boxes are transparent -- margins are always transparent -- so the background color (green) of the UL padding and content areas shines through them.
- The second LI element specifies a dashed border (the ’border-style’ property).

8.3 Margin properties: ’margin-top’, ’margin-right’, ’margin-bottom’, ’margin-left’, and ’margin’

Margin properties specify the width of the margin area [p. 81] of a box. The ’margin’ shorthand property sets the margin for all four sides while the other margin properties only set their respective side.

The properties defined in this section refer to the **<margin-width>** value type, which may take one of the following values:

<length>
Specifies a fixed width.
<percentage>
The percentage is calculated with respect to the *width* of the generated box’s containing block [p. 96] . This is true for ’margin-top’ and ’margin-bottom’, except in the page context [p. 176] , where percentages refer to page box height.
auto
See the section on computing widths and margins [p. 134] for behavior.

Negative values for margin properties are allowed, but there may be implementation-specific limits.

’margin-top’, **’margin-right’**, **’margin-bottom’**, **’margin-left’**

Value:	<margin-width> | inherit
Initial:	0
Applies to:	all elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

These properties set the top, right, bottom, and left margin of a box.
Example(s):

```
H1 { margin-top: 2em }
```

’margin’

Value:	<margin-width>{1,4} | inherit
Initial:	not defined for shorthand properties
Applies to:	all elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

The ’margin’ property is a shorthand property for setting ’margin-top’, ’margin-right’, ’margin-bottom’, and ’margin-left’ at the same place in the style sheet.

If there is only one value, it applies to all sides. If there are two values, the top and bottom margins are set to the first value and the right and left margins are set to the second. If there are three values, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third. If there are four values, they apply to the top, right, bottom, and left, respectively.

Example(s):

```
BODY { margin: 2em }         /* all margins set to 2em */
BODY { margin: 1em 2em }     /* top & bottom = 1em, right & left = 2em */
BODY { margin: 1em 2em 3em } /* top=1em, right=2em, bottom=3em, left=2em */
```

The last rule of the example above is equivalent to the example below:

```
BODY {
  margin-top: 1em;
  margin-right: 2em;
  margin-bottom: 3em;
  margin-left: 2em;        /* copied from opposite side (right) */
}
```

8.3.1 Collapsing margins

In this specification, the expression *collapsing margins* means that adjoining margins (no padding or border areas separate them) of two or more boxes (which may be next to one another or nested) combine to form a single margin.

In CSS2, horizontal margins never collapse.

Vertical margins may collapse between certain boxes:

- Two or more adjoining vertical margins of block [p. 97] boxes in the normal flow [p. 105] collapse. The resulting margin width is the maximum of the adjoining margin widths. In the case of negative margins, the absolute maximum of the negative adjoining margins is deducted from the maximum of the positive adjoining margins. If there are no positive margins, the absolute maximum of the negative adjoining margins is deducted from zero.
- Vertical margins between a floated [p. 108] box and any other box do not collapse.
- Margins of absolutely [p. 113] and relatively positioned boxes do not collapse.

Please consult the examples of margin, padding, and borders [p. 83] for an illustration of collapsed margins.

8.4 Padding properties: ’padding-top’, ’padding-right’, ’padding-bottom’, ’padding-left’, and ’padding’

The padding properties specify the width of the padding area [p. 81] of a box. The ’padding’ shorthand property sets the padding for all four sides while the other padding properties only set their respective side.

The properties defined in this section refer to the **<padding-width>** value type, which may take one of the following values:

<length>
: Specifies a fixed width.

<percentage>
: The percentage is calculated with respect to the *width* of the generated box’s containing block [p. 96] , even for ’padding-top’ and ’padding-bottom’.

Unlike margin properties, values for padding values cannot be negative. Like margin properties, percentage values for padding properties refer to the width of the generated box’s containing block.

’padding-top’, **’padding-right’**, **’padding-bottom’**, **’padding-left’**

Value:	<padding-width> | inherit
Initial:	0
Applies to:	all elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

These properties set the top, right, bottom, and left padding of a box.

Example(s):

```
BLOCKQUOTE { padding-top: 0.3em }
```

’padding’

Value:	<padding-width>{1,4} | inherit
Initial:	not defined for shorthand properties
Applies to:	all elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

The ’padding’ property is a shorthand property for setting ’padding-top’, ’padding-right’, ’padding-bottom’, and ’padding-left’ at the same place in the style sheet.

If there is only one value, it applies to all sides. If there are two values, the top and bottom paddings are set to the first value and the right and left paddings are set to the second. If there are three values, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third. If there are four

values, they apply to the top, right, bottom, and left, respectively.
The surface color or image of the padding area is specified via the ’background’ property:
Example(s):

```
H1 {
  background: white;
  padding: 1em 2em;
}
```

The example above specifies a ’1em’ vertical padding (’padding-top’ and ’padding-bottom’) and a ’2em’ horizontal padding (’padding-right’ and ’padding-left’). The ’em’ unit is relative [p. 43] to the element’s font size: ’1em’ is equal to the size of the font in use.

8.5 Border properties

The border properties specify the width, color, and style of the border area [p. 81] of a box. These properties apply to all elements.
Note. *Notably for HTML, user agents may render borders for certain elements (e.g., buttons, menus, etc.) differently than for "ordinary" elements.*

8.5.1 Border width: ’border-top-width’, ’border-right-width’, ’border-bottom-width’, ’border-left-width’, and ’border-width’

The border width properties specify the width of the border area [p. 81] . The properties defined in this section refer to the **<border-width>** value type, which may take one of the following values:

thin
A thin border.
medium
A medium border.
thick
A thick border.
<length>
The border’s thickness has an explicit value. Explicit border widths cannot be negative.

The interpretation of the first three values depends on the user agent. The following relationships must hold, however:
’thin’ <=’medium’ <= ’thick’.
Furthermore, these widths must be constant throughout a document.

’border-top-width’, **’border-right-width’**, **’border-bottom-width’**, **’border-left-width’**

Value:	<border-width> | inherit
Initial:	medium
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

These properties set the width of the top, right, bottom, and left border of a box.

’border-width’

Value:	<border-width>{1,4} | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

This property is a shorthand property for setting ’border-top-width’, ’border-right-width’, ’border-bottom-width’, and ’border-left-width’ at the same place in the style sheet.

If there is only one value, it applies to all sides. If there are two values, the top and bottom borders are set to the first value and the right and left are set to the second. If there are three values, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third. If there are four values, they apply to the top, right, bottom, and left, respectively.

Example(s):

In the examples below, the comments indicate the resulting widths of the top, right, bottom, and left borders:

```
H1 { border-width: thin }                   /* thin thin thin thin */
H1 { border-width: thin thick }             /* thin thick thin thick */
H1 { border-width: thin thick medium }      /* thin thick medium thick */
```

8.5.2 Border color: ’border-top-color’, ’border-right-color’, ’border-bottom-color’, ’border-left-color’, and ’border-color’

The border color properties specify the color of a box’s border.

’border-top-color’, **’border-right-color’**, **’border-bottom-color’**, **’border-left-color’**

Value:	<color> | inherit
Initial:	the value of the ’color’ property
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

’border-color’

Value:	<color>{1,4} | transparent | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

The ’border-color’ property sets the color of the four borders. Values have the following meanings:

<color>
Specifies a color value.

transparent
The border is transparent (though it may have width).

The ’border-color’ property can have from one to four values, and the values are set on the different sides as for ’border-width’.

If an element’s border color is not specified with a border property, user agents must use the value of the element’s ’color’ property as the computed value [p. 70] for the border color.

Example(s):

In this example, the border will be a solid black line.

```
P {
  color: black;
  background: white;
  border: solid;
}
```

8.5.3 Border style: ’border-top-style’, ’border-right-style’, ’border-bottom-style’, ’border-left-style’, and ’border-style’

The border style properties specify the line style of a box’s border (solid, double, dashed, etc.). The properties defined in this section refer to the **<border-style>** value type, which make take one of the following:

none
No border. This value forces the computed value of ’border-width’ to be ’0’.

hidden
Same as ’none’, except in terms of border conflict resolution [p. 264] for table elements [p. 245] .

dotted
The border is a series of dots.

dashed
The border is a series of short line segments.

solid
The border is a single line segment.

double
: The border is two solid lines. The sum of the two lines and the space between them equals the value of ’border-width’.

groove
: The border looks as though it were carved into the canvas.

ridge
: The opposite of ’grove’: the border looks as though it were coming out of the canvas.

inset
: The border makes the entire box look as though it were embedded in the canvas.

outset
: The opposite of ’inset’: the border makes the entire box look as though it were coming out of the canvas.

All borders are drawn on top of the box’s background. The color of borders drawn for values of ’groove’, ’ridge’, ’inset’, and ’outset’ depends on the element’s ’color’ property.

Conforming HTML user agents [p. 32] may interpret ’dotted’, ’dashed’, ’double’, ’groove’, ’ridge’, ’inset’, and ’outset’ to be ’solid’.

’border-top-style’, **’border-right-style’**, **’border-bottom-style’**, **’border-left-style’**

Value:	<border-style> | inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

’border-style’

Value:	<border-style>{1,4} | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

The ’border-style’ property sets the style of the four borders. It can have from one to four values, and the values are set on the different sides as for ’border-width’ above.

Example(s):

```
#xy34 { border-style: solid dotted }
```

In the above example, the horizontal borders will be ’solid’ and the vertical borders will be ’dotted’.

Since the initial value of the border styles is ’none’, no borders will be visible unless the border style is set.

8.5.4 Border shorthand properties: ’border-top’, ’border-bottom’, ’border-right’, ’border-left’, and ’border’

’border-top’, **’border-right’**, **’border-bottom’**, **’border-left’**

Value:	[<’border-top-width’> || <’border-style’> || <color>] | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

This is a shorthand property for setting the width, style, and color of the top, right, bottom, and left border of a box.
Example(s):

```
H1 { border-bottom: thick solid red }
```

The above rule will set the width, style, and color of the border **below** the H1 element. Omitted values are set to their initial values [p. 69] . Since the following rule does not specify a border color, the border will have the color specified by the ’color’ property:

```
H1 { border-bottom: thick solid }
```

’border’

Value:	[<’border-width’> || <’border-style’> || <color>] | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

The ’border’ property is a shorthand property for setting the same width, color, and style for all four borders of a box. Unlike the shorthand ’margin’ and ’padding’ properties, the ’border’ property cannot set different values on the four borders. To do so, one or more of the other border properties must be used.
Example(s):
For example, the first rule below is equivalent to the set of four rules shown after it:

```
P { border: solid red }
P {
  border-top: solid red;
  border-right: solid red;
  border-bottom: solid red;
  border-left: solid red
}
```

Since, to some extent, the properties have overlapping functionality, the order in which the rules are specified is important.

Example(s):

Consider this example:

```
BLOCKQUOTE {
  border-color: red;
  border-left: double;
  color: black
}
```

In the above example, the color of the left border is black, while the other borders are red. This is due to 'border-left' setting the width, style, and color. Since the color value is not given by the 'border-left' property, it will be taken from the 'color' property. The fact that the 'color' property is set after the 'border-left' property is not relevant.

---

9 Visual formatting model

Contents

9.1 Introduction to the visual formatting model

This chapter and the next describe the visual formatting model: how user agents process the document tree [p. 30] for visual media [p. 77] .

In the visual formatting model, each element in the document tree generates zero or more boxes according to the box model [p. 81] . The layout of these boxes is governed by:

- box dimensions [p. 81] and type [p. 97] .
- positioning scheme [p. 102] (normal flow, float, and absolute).
- relationships between elements in the document tree. [p. 30]
- external information (e.g., viewport size, intrinsic [p. 30] dimensions of images, etc.).

The properties defined in this chapter and the next apply to both continuous media [p. 79] and paged media [p. 79] . However, the meanings of the margin properties [p. 85] vary when applied to paged media (see the page model [p. 175] for details).

The visual formatting model does not specify all aspects of formatting (e.g., it does not specify a letter-spacing algorithm). Conforming user agents [p. 32] may behave differently for those formatting issues not covered by this specification.

9.1.1 The viewport

User agents for continuous media [p. 79] generally offer users a *viewport* (a window or other viewing area on the screen) through which users consult a document. User agents may change the document's layout when the viewport is resized (see the initial containing block [p. 96]). When the viewport is smaller than the document's initial containing block, the user agent should offer a scrolling mechanism. There is at most one viewport per canvas [p. 26] , but user agents may render to more than one canvas (i.e., provide different views of the same document).

9.1.2 Containing blocks

In CSS2, many box positions and sizes are calculated with respect to the edges of a rectangular box called a *containing block*. In general, generated boxes act as containing blocks for descendant boxes; we say that a box "establishes" the containing block for its descendants. The phrase "a box's containing block" means "the containing block in which the box lives," not the one it generates.

Each box is given a position with respect to its containing block, but it is not confined by this containing block; it may overflow [p. 145] .

The root of the document tree [p. 30] generates a box that serves as the *initial containing block* for subsequent layout.

The width of the initial containing block may be specified with the 'width' property for the root element. If this property has the value 'auto', the user agent supplies the initial width (e.g., the user agent uses the current width of the viewport [p. 96]).

The height of the initial containing block may be specified with the 'height' property for the root element. If this property has the value 'auto', the containing block height will grow to accommodate the document's content.

The initial containing block cannot be positioned or floated (i.e., user agents ignore [p. 42] the 'position' and 'float' properties for the root element).

The details [p. 131] of how a containing block's dimensions are calculated are described in the next chapter [p. 131] .

9.2 Controlling box generation

The following sections describe the types of boxes that may be generated in CSS2. A box’s type affects, in part, its behavior in the visual formatting model. The ’display’ property, described below, specifies a box’s type.

9.2.1 Block-level elements and block boxes

Block-level elements are those elements of the source document that are formatted visually as blocks (e.g., paragraphs). Several values of the ’display’ property make an element block-level: ’block’, ’list-item’, ’compact’ and ’run-in’ (part of the time; see compact [p. 99] and run-in boxes [p. 100]), and ’table’.

Block-level elements generate a *principal block box* that only contains *block boxes*. The principal block box establishes the containing block [p. 96] for descendant boxes and generated content and is also the box involved in any positioning scheme. Principal block boxes participate in a block formatting context [p. 105] .

Some block-level elements generate additional boxes outside of the principal box: ’list-item’ elements and those with markers [p. 164] . These additional boxes are placed with respect to the principal box.

Anonymous block boxes

In a document like this:

```
<DIV>
  Some text
  <P>More text
</DIV>
```

(and assuming the DIV and the P both have ’display: block’), the DIV appears to have both inline content and block content. To make it easier to define the formatting, we assume that there is an *anonymous block box* around "Some text".

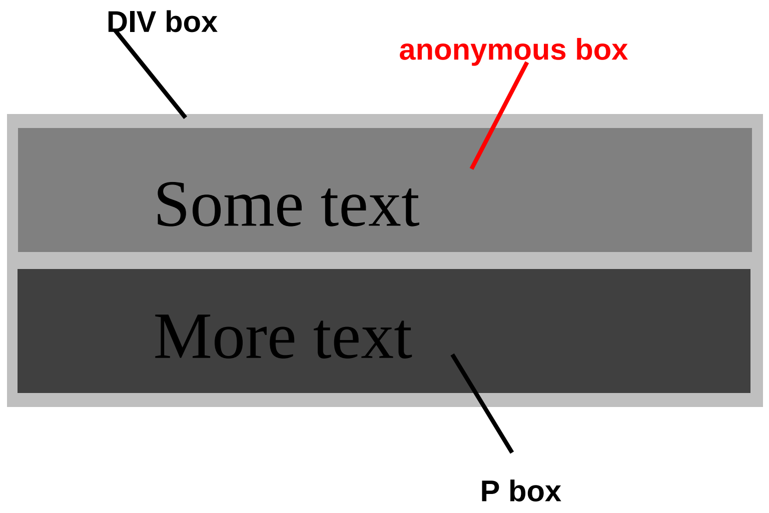


Diagram showing the three boxes, of which one is anonymous, for the example above.

In other words: if a block box (such as that generated for the DIV above) has another block box inside it (such as the P above), then we force it to have *only* block boxes inside it, by wrapping any inline boxes in an anonymous block box.

Example(s):

This model would apply in the following example if the following rules:

```
/* Note: HTML UAs may not respect these rules */
BODY { display: inline }
P    { display: block }
```

were used with this HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HEAD>
<TITLE>Anonymous text interrupted by a block</TITLE>
</HEAD>
<BODY>
This is anonymous text before the P.
<P>This is the content of P.</>
This is anonymous text after the P.
</BODY>
```

The BODY element contains a chunk (C1) of anonymous text followed by a block-level element followed by another chunk (C2) of anonymous text. The resulting boxes would be an anonymous block box for BODY, containing an anonymous block box around C1, the P block box, and another anonymous block box around C2.

The properties of anonymous boxes are inherited from the enclosing non-anonymous box (in the example: the one for DIV). Non-inherited properties have their initial value. For example, the font of the anonymous box is inherited from the DIV, but the margins will be 0.

9.2.2 Inline-level elements and inline boxes

Inline-level elements are those elements of the source document that do not form new blocks of content; the content is distributed in lines (e.g., emphasized pieces of text within a paragraph, inline images, etc.). Several values of the 'display' property make an element inline: 'inline', 'inline-table', 'compact' and 'run-in' (part of the time; see compact [p. 99] and run-in boxes [p. 100]). Inline-level elements generate inline boxes.

Inline boxes may participate in several formatting contexts:

- Within a block box, an inline boxes participate in an inline formatting context [p. 105] .
- A compact [p. 99] inline box is given a position in the margin of a block box.
- Marker [p. 164] boxes are also given positions outside of a block box.

Anonymous inline boxes

In a document like this:

```
<P>Some <EM>emphasized</em> text</P>
```

The P generates a block box, with several inline boxes inside it. The box for "emphasized" is an inline box generated by an inline element (EM), but the other boxes ("Some" and "text") are inline boxes generated by a block-level element (P). The latter are called anonymous inline boxes, because they don’t have an associated inline-level element.

Such anonymous inline boxes inherit inheritable properties from their block parent box. Non-inherited properties have their initial value. In the example, the color of the anonymous initial boxes is inherited from the P, but the background is transparent.

If it is clear from the context which type of anonymous box is meant, both anonymous inline boxes and anonymous block boxes are simply called anonymous boxes in this specification.

There are more types of anonymous boxes that arise when formatting tables [p. 248] .

9.2.3 Compact boxes

A *compact box* behaves as follows:

- If a block [p. 97] box (that does not float and is not absolutely positioned [p. 113]) follows the compact box, the compact box is formatted like a one-line inline box. The resulting box width is compared to one of the side margins of the block box. The choice of left or right margin is determined by the ’direction’ specified for the element producing the containing block [p. 96] for the compact box and following box. If the inline box width is less than or equal to the margin, the inline box is given a position in the margin as described immediately below.
- Otherwise, the compact box becomes a block box.

The compact box is given a position in the margin as follows: it is outside (to the left or right) of the first line box [p. 105] of the block, but it affects the calculation of that line box’s height [p. 141] . The ’vertical-align’ property of the compact box determines the vertical position of the compact box relative to that line box. The horizontal position of the compact box is always in the margin of the block box.

An element that cannot be formatted on one line cannot be placed in the margin of the following block. For example, a ’compact’ element in HTML that contains a BR element will always be formatted as a block box (assuming the default style for BR, which inserts a newline). For placing multi-line texts in the margin, the ’float’ property is often more appropriate.

The following example illustrates a compact box.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>A compact box example</TITLE>
    <STYLE type="text/css">
      DT { display: compact }
      DD { margin-left: 4em }
    </STYLE>
  </HEAD>
  <BODY>
    <DL>
```

```
      <DT>Short
        <DD><P>Description goes here.
      <DT>too long for the margin
        <DD><P>Description goes here.
    </DL>
  </BODY>
</HTML>
```

This example might be formatted as:

```
short    Description goes here

too long for the margin
         Description goes here
```

The 'text-align' property can be used to align the compact element inside the margin: against the left edge of the margin ('left'), against the right edge ('right'), or centered in the margin ('center'). The value 'justify' doesn't apply, and is handled as either 'left' or 'right', depending on the 'direction' of the block-level element in whose margin the compact element is formatted. ('left' if the direction is 'ltr', 'right' if it is 'rtl'.)

Please consult the section on generated content [p. 156] for information about how compact boxes interact with generated content.

9.2.4 Run-in boxes

A *run-in box* behaves as follows:

- If a block [p. 97] box (that does not float and is not absolutely positioned [p. 113]) follows the run-in box, the run-in box becomes the first inline box of the block box.
- Otherwise, the run-in box becomes a block box.

A 'run-in' box is useful for run-in headers, as in this example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>A run-in box example</TITLE>
    <STYLE type="text/css">
      H3 { display: run-in }
    </STYLE>
  </HEAD>
  <BODY>
    <H3>A run-in heading.</H3>
    <P>And a paragraph of text that
       follows it.
  </BODY>
</HTML>
```

This example might be formatted as:

```
  A run-in heading. And a
  paragraph of text that
  follows it.
```

The properties of the run-in element are inherited from its parent in the source tree, not from the block box it visually becomes part of.

Please consult the section on generated content [p. 156] for information about how run-in boxes interact with generated content.

9.2.5 The ’display’ property

’display’

Value:	inline | block | list-item | run-in | compact | marker | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none | inherit
Initial:	inline
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	all

The values of this property have the following meanings:

block
This value causes an element to generate a principal block box.

inline
This value causes an element to generate one or more inline boxes.

list-item
This value causes an element (e.g., LI in HTML) to generate a principal block box and a list-item inline box. For information about lists and examples of list formatting, please consult the section on lists [p. 168] .

marker
This value declares generated content [p. 153] before or after a box to be a marker. This value should only be used with :before and :after pseudo-elements [p. 153] attached to block-level elements. In other cases, this value is interpreted as ’inline’. Please consult the section on markers [p. 164] for more information.

none
This value causes an element to generate **no** boxes in the formatting structure [p. 25] (i.e., the element has no effect on layout). Descendant elements do not generate any boxes either; this behavior **cannot** be overridden by setting the ’display’ property on the descendants.

Please note that a display of ’none’ does not create an invisible box; it creates no box at all. CSS includes mechanisms that enable an element to generate boxes in the formatting structure that affect formatting but are not visible themselves. Please consult the section on visibility [p. 149] for details.

run-in and **compact**
These values create either block or inline boxes, depending on context. Properties apply to run-in and compact boxes based on their final status (inline-level or block-level). For example, the ’white-space’ property only applies if the box becomes a block box.

table, **inline-table**, **table-row-group**, **table-column**, **table-column-group**, **table-header-group**, **table-footer-group**, **table-row**, **table-cell**, and **table-caption**

These values cause an element to behave like a table element (subject to restrictions described in the chapter on tables [p. 245]).

Note that although the initial value [p. 69] of ’display’ is ’inline’, rules in the user agent’s default style sheet [p. 72] may override [p. 69] this value. See the sample style sheet [p. 291] for HTML 4.0 in the appendix.

Example(s):

Here are some examples of the ’display’ property:

```
P   { display: block }
EM  { display: inline }
LI  { display: list-item }
IMG { display: none }      /* Don’t display images */
```

Conforming [p. 32] HTML user agents may ignore [p. 42] the ’display’ property.

9.3 Positioning schemes

In CSS2, a box may be laid out according to three *positioning schemes:*

1. Normal flow [p. 105] . In CSS2, normal flow includes block formatting [p. 105] of block [p. 97] boxes, inline formatting [p. 105] of inline [p. 98] boxes, relative positioning [p. 107] of block or inline boxes, and positioning of compact [p. 99] and run-in [p. 100] boxes.
2. Floats [p. 108] . In the float model, a box is first laid out according to the normal flow, then taken out of the flow and shifted to the left or right as far as possible. Content may flow along the side of a float.
3. Absolute positioning [p. 113] . In the absolute positioning model, a box is removed from the normal flow entirely (it has no impact on later siblings) and assigned a position with respect to a containing block.

Note. *CSS2’s positioning schemes help authors make their documents more accessible by allowing them to avoid mark-up tricks (e.g., invisible images) used for layout effects.*

9.3.1 Choosing a positioning scheme: ’position’ property

The ’position’ and ’float’ properties determine which of the CSS2 positioning algorithms is used to calculate the position of a box.

’position’

Value:	static | relative | absolute | fixed | inherit
Initial:	static
Applies to:	all elements, but not to generated content
Inherited:	no
Percentages:	N/A
Media:	visual

The values of this property have the following meanings:

static
: The box is a normal box, laid out according to the normal flow [p. 105] . The ’left’ and ’top’ properties do not apply.

relative
: The box’s position is calculated according to the normal flow [p. 105] (this is called the position in normal flow). Then the box is offset relative [p. 107] to its normal position. When a box B is relatively positioned, the position of the following box is calculated as though B were not offset.

absolute
: The box’s position (and possibly size) is specified with the ’left’, ’right’, ’top’, and ’bottom’ properties. These properties specify offsets with respect to the box’s containing block [p. 96] . Absolutely positioned boxes are taken out of the normal flow. This means they have no impact on the layout of later siblings. Also, though absolutely positioned [p. 113] boxes have margins, they do not collapse [p. 86] with any other margins.

fixed
: The box’s position is calculated according to the ’absolute’ model, but in addition, the box is fixed [p. 113] with respect to some reference. In the case of continuous media [p. 79] , the box is fixed with respect to the viewport [p. 96] (and doesn’t move when scrolled). In the case of paged media [p. 79] , the box is fixed with respect to the page, even if that page is seen through a viewport [p. 96] (in the case of a print-preview, for example). Authors may wish to specify ’fixed’ in a media-dependent way. For instance, an author may want a box to remain at the top of the viewport [p. 96] on the screen, but not at the top of each printed page. The two specifications may be separated by using an @media rule [p. 78] , as in:

Example(s):

```
@media screen {
  H1#first { position: fixed }
}
@media print {
  H1#first { position: static }
}
```

9.3.2 Box offsets: ’top’, ’right’, ’bottom’, ’left’

An element is said to be *positioned* if its ’position’ property has a value other than ’static’. Positioned elements generate positioned boxes, laid out according to four properties:

’top’

Value:	<length> | <percentage> | auto | inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to height of containing block
Media:	visual

This property specifies how far a box’s top content edge is offset below the top edge of the box’s containing block [p. 96] .

’right’

Value:	<length> | <percentage> | auto | inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

This property specifies how far a box’s right content edge is offset to the left of the right edge of the box’s containing block [p. 96] .

’bottom’

Value:	<length> | <percentage> | auto | inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to height of containing block
Media:	visual

This property specifies how far a box’s bottom content edge is offset above the bottom of the box’s containing block [p. 96] .

’left’

Value:	<length> | <percentage> | auto | inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

This property specifies how far a box’s left content edge is offset to the right of the left edge of the box’s containing block [p. 96] .

The values for the four properties have the following meanings:

<length>

The offset is a fixed distance from the reference edge.

<percentage>

The offset is a percentage of the containing block's width (for 'left' or 'right') or height (for 'top' and 'bottom'). For 'top' and 'bottom', if the height of the containing block is not specified explicitly (i.e., it depends on content height), the percentage value is interpreted like 'auto'.

auto

The effect of this value depends on which of related properties have the value 'auto' as well. See the sections on the width [p. 135] and height [p. 139] of absolutely positioned [p. 113] , non-replaced elements for details.

For absolutely positioned [p. 113] boxes, the offsets are with respect to the box's containing block [p. 96] . For relatively positioned boxes, the offsets are with respect to the outer edges of the box itself (i.e., the box is given a position in the normal flow, then offset from that position according to these properties).

9.4 Normal flow

Boxes in the normal flow belong to a formatting context, which may be block or inline, but not both simultaneously. Block [p. 97] boxes participate in a block formatting [p. 105] context. Inline boxes [p. 98] participate in an inline formatting [p. 105] context.

9.4.1 Block formatting context

In a block formatting context, boxes are laid out one after the other, vertically, beginning at the top of a containing block. The vertical distance between two sibling boxes is determined by the 'margin' properties. Vertical margins between adjacent block boxes in a block formatting context collapse [p. 86] .

In a block formatting context, each box's left outer edge touches the left edge of the containing block (for right-to-left formatting, right edges touch). This is true even in the presence of floats (although a box's *content* area may shrink due to the floats).

For information about page breaks in paged media, please consult the section on allowed page breaks [p. 183] .

9.4.2 Inline formatting context

In an inline formatting context, boxes are laid out horizontally, one after the other, beginning at the top of a containing block. Horizontal margins, borders, and padding are respected between these boxes. The boxes may be aligned vertically in different ways: their bottoms or tops may be aligned, or the baselines of text within them may be aligned. The rectangular area that contains the boxes that form a line is called a *line box*.

The width of a line box is determined by a containing block [p. 96] . The height of a line box is determined by the rules given in the section on line height calculations [p. 141] . A line box is always tall enough for all of the boxes it contains. However, it may be taller than the tallest box it contains (if, for example, boxes are aligned so that baselines line up). When the height of a box B is less than the height of the line box containing it, the vertical alignment of B within the line box is determined by the 'vertical-align' property.

When several inline boxes cannot fit horizontally within a single line box, they are distributed among two or more vertically-stacked line boxes. Thus, a paragraph is a vertical stack of line boxes. Line boxes are stacked with no vertical separation and they never overlap.

In general, the left edge of a line box touches the left edge of its containing block and the right edge touches the right edge of its containing block. However, floating boxes may come between the containing block edge and the line box edge. Thus, although line boxes in the same inline formatting context generally have the same width (that of the containing block), they may vary in width if available horizontal space is reduced due to floats [p. 108] . Line boxes in the same inline formatting context generally vary in height (e.g., one line might contain a tall image while the others contain only text).

When the total width of the inline boxes on a line is less than the width of the line box containing them, their horizontal distribution within the line box is determined by the 'text-align' property. If that property has the value 'justify', the user agent may stretch the inline boxes as well.

Since an inline box may not exceed the width of a line box, long inline boxes are split into several boxes and these boxes distributed across several line boxes. When an inline box is split, margins, borders, and padding have no visual effect where the split occurs. Formatting of margins, borders, and padding may not be fully defined if the split occurs within a bidirectional embedding.

Inline boxes may also be split into several boxes *within the same line box* due to bidirectional text processing [p. 127] .

Here is an example of inline box construction. The following paragraph (created by the HTML block-level element P) contains anonymous text interspersed with the elements EM and STRONG:

```
<P>Several <EM>emphasized words</EM> appear
<STRONG>in this</STRONG> sentence, dear.</P>
```

The P element generates a block box that contains five inline boxes, three of which are anonymous:

- Anonymous: "Several"
- EM: "emphasized words"
- Anonymous: "appear"
- STRONG: "in this"
- Anonymous: "sentence, dear."

To format the paragraph, the user agent flows the five boxes into line boxes. In this example, the box generated for the P element establishes the containing block for the line boxes. If the containing block is sufficiently wide, all the inline boxes will fit into a single line box:

Several *emphasized words* appear **in this** sentence, dear.

If not, the inline boxes will be split up and distributed across several line boxes. The previous paragraph might be split as follows:

Several *emphasized words* appear
in this sentence, dear.

or like this:

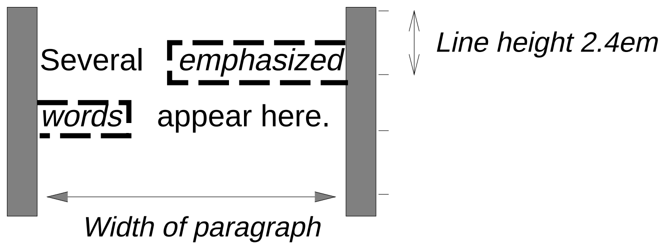
```
Several emphasized
words appear in this
sentence, dear.
```

In the previous example, the EM box was split into two EM boxes (call them "split1" and "split2"). Margins, borders, padding, or text decorations have no visible effect after split1 or before split2.

Consider the following example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Example of inline flow on several lines</TITLE>
    <STYLE type="text/css">
      EM {
        padding: 2px;
        margin: 1em;
        border-width: medium;
        border-style: dashed;
        line-height: 2.4em;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <P>Several <EM>emphasized words</EM> appear here.</P>
  </BODY>
</HTML>
```

Depending on the width of the P, the boxes may be distributed as follows:



- The margin is inserted before "emphasized" and after "words".
- The padding is inserted before, above, and below "emphasized" and after, above, and below "words". A dashed border is rendered on three sides in each case.

9.4.3 Relative positioning

Once a box has been laid out according to the normal flow [p. 105] , it may be shifted relative to this position. This is called *relative positioning*. Offsetting a box (B1) in this way has no effect on the box (B2) that follows: B2 is given a position as if B1 were not offset and B2 is not re-positioned after B1's offset is applied. This implies that relative positioning may cause boxes to overlap.

Relatively positioned boxes keep their normal flow size, including line breaks and the space originally reserved for them. A relatively positioned box establishes a new a new containing block [p. 96] for normal flow children and positioned descendants.

A relatively positioned box is generated when the ’position’ property for an element has the value ’relative’. The offset is specified by the ’top’, ’bottom’, ’left’, and ’right’ properties.

Dynamic movement of relatively positioned boxes can produce animation effects in scripting environments (see also the ’visibility’ property). Relative positioning may also be used as a general form of superscripting and subscripting except that line height is not automatically adjusted to take the positioning into consideration. See the description of line height calculations [p. 141] for more information.

Examples of relative positioning are provided in the section comparing normal flow, floats, and absolute positioning [p. 115] .

9.5 Floats

A float is a box that is shifted to the left or right on the current line. The most interesting characteristic of a float (or "floated" or "floating" box) is that content may flow along its side (or be prohibited from doing so by the ’clear’ property). Content flows down the right side of a left-floated box and down the left side of a right-floated box. The following is an introduction to float positioning and content flow; the exact rules [p. 112] governing float behavior are given in the description of the ’float’ property.

A floated box must have an explicit width (assigned via the ’width’ property, or its intrinsic [p. 30] width in the case of replaced elements [p. 30]). Any floated box becomes a block box [p. 97] that is shifted to the left or right until its outer edge touches the containing block edge or the outer edge of another float. The top of the floated box is aligned with the top of the current line box (or bottom of the preceding block box if no line box exists). If there isn’t enough horizontal room on the current line for the float, it is shifted downward, line by line, until a line has room for it.

Since a float is not in the flow, non-positioned block boxes created before and after the float box flow vertically as if the float didn’t exist. However, line boxes created next to the float are shortened to make room for the floated box. Any content in the current line before a floated box is reflowed in the first available line on the other side of the float.

Several floats may be adjacent, and this model also applies to adjacent floats in the same line.

Example(s):

The following rule floats all IMG boxes with `class="icon"` to the left (and sets the left margin to ’0’):

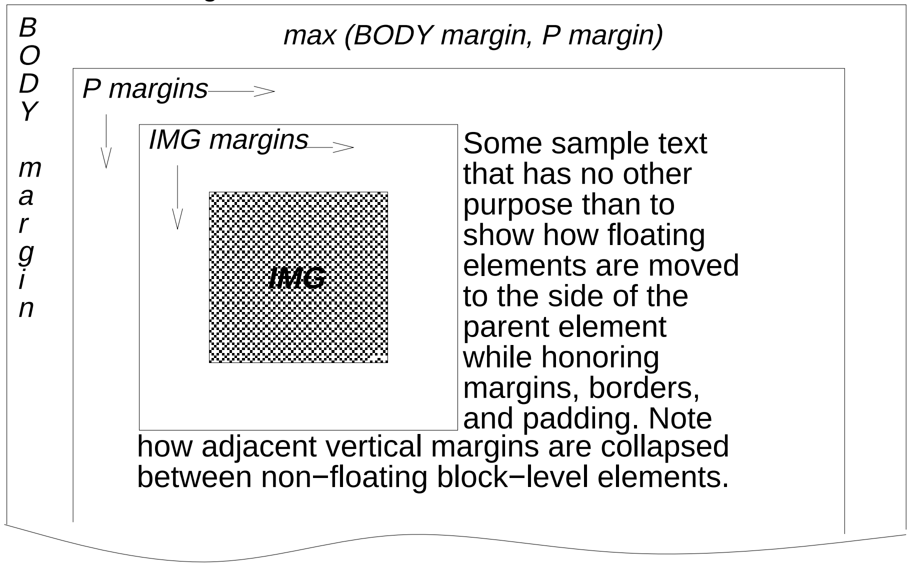
```
IMG.icon {
  float: left;
  margin-left: 0;
}
```

Consider the following HTML source and style sheet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Float example</TITLE>
    <STYLE type="text/css">
```

```
        IMG { float: left }
        BODY, P, IMG { margin: 2em }
      </STYLE>
    </HEAD>
    <BODY>
      <P><IMG src=img.gif alt="This image will illustrate floats">
         Some sample text that has no other...
    </BODY>
  </HTML>
```

The IMG box is floated to the left. The content that follows is formatted to the right of the float, starting on the same line as the float. The line boxes to the right of the float are shortened due to the float's presence, but resume their "normal" width (that of the containing block established by the P element) after the float. This document might be formatted as:


B O D Y margin
max (BODY margin, P margin)
P margins
IMG margins
IMG
Some sample text that has no other purpose than to show how floating elements are moved to the side of the parent element while honoring margins, borders, and padding. Note how adjacent vertical margins are collapsed between non−floating block−level elements.

Formatting would have been exactly the same if the document had been:

```
  <BODY>
    <P>Some sample text
    <IMG src=img.gif alt="This image will illustrate floats">
             that has no other...
  </BODY>
```

because the content to the left of the float is displaced by the float and reflowed down its right side.

The margins of floating boxes never collapse [p. 86] with margins of adjacent boxes. Thus, in the previous example, vertical margins do not collapse [p. 86] between the P box and the floated IMG box.

A float can overlap other boxes in the normal flow (e.g., when a normal flow box next to a float has negative margins). When an inline box overlaps with a float, the content, background, and borders of the inline box are rendered in front of the float. When a block box overlaps, the background and borders of the block box are rendered behind the float and are only be visible where the box is transparent. The content of the block box is rendered in front of the float.

Example(s):

Here is another illustration, showing what happens when a float overlaps borders of elements in the normal flow.

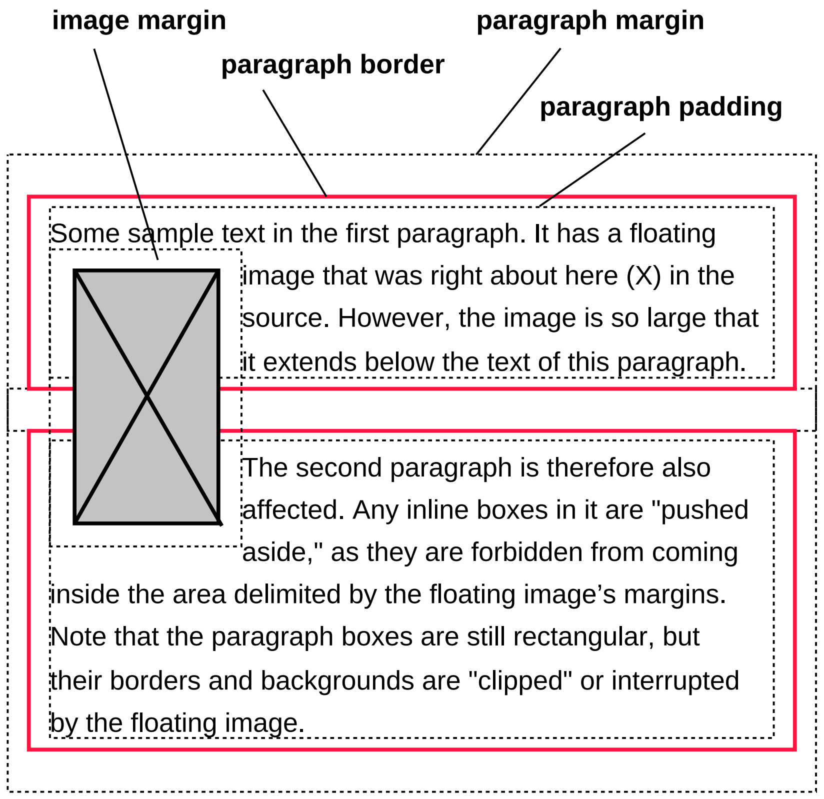


image margin
paragraph border
paragraph margin
paragraph padding
Some sample text in the first paragraph. It has a floating image that was right about here (X) in the source. However, the image is so large that it extends below the text of this paragraph.
The second paragraph is therefore also affected. Any inline boxes in it are "pushed aside," as they are forbidden from coming inside the area delimited by the floating image's margins. Note that the paragraph boxes are still rectangular, but their borders and backgrounds are "clipped" or interrupted by the floating image.

A floating image obscures borders of block boxes it overlaps.

The following example illustrates the use of the ’clear’ property to prevent content from flowing next to a float.

Example(s):

Assuming a rule such as this:

```
P { clear: left }
```

formatting might look like this:

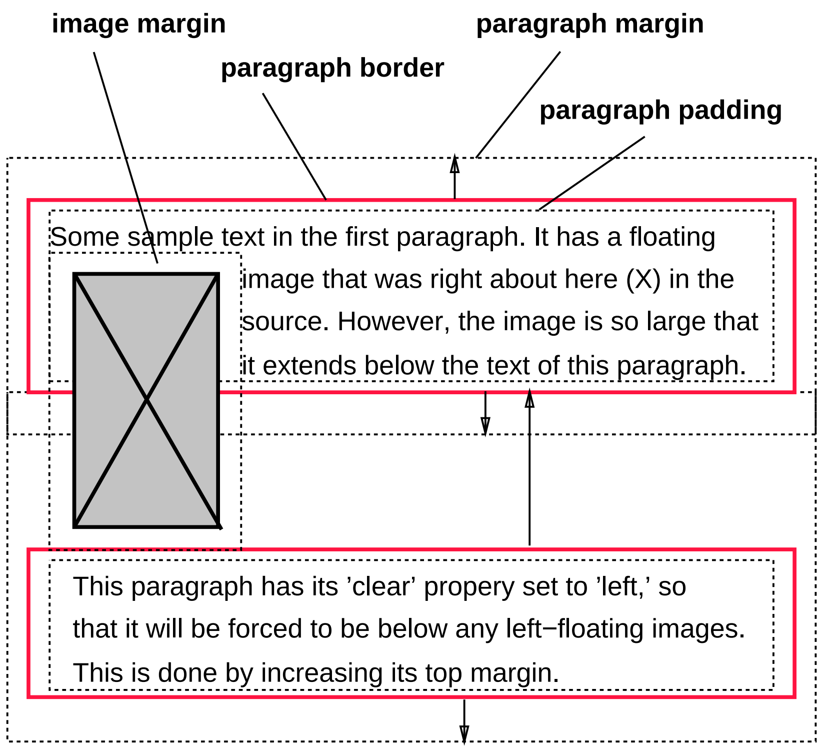


image margin
paragraph border
paragraph margin
paragraph padding
Some sample text in the first paragraph. It has a floating image that was right about here (X) in the source. However, the image is so large that it extends below the text of this paragraph.
This paragraph has its 'clear' propery set to 'left,' so that it will be forced to be below any left−floating images. This is done by increasing its top margin.

Both paragraphs have set 'clear: left', which causes the second paragraph to be "pushed down" to a position below the float -- its top margin expands to accomplish this (see the 'clear' property).

9.5.1 Positioning the float: the 'float' property

'float'

Value:	left | right | none | inherit
Initial:	none
Applies to:	all but positioned elements and generated content
Inherited:	no
Percentages:	N/A
Media:	visual

This property specifies whether a box should float to the left, right, or not at all. It may be set for elements that generate boxes that are not absolutely positioned [p. 113] . The values of this property have the following meanings:

left
The element generates a block [p. 97] box that is floated to the left. Content flows on the right side of the box, starting at the top (subject to the 'clear' property). The 'display' is ignored, unless it has the value 'none'.

right
Same as 'left', but content flows on the left side of the box, starting at the top.

none
The box is not floated.

Here are the precise rules that govern the behavior of floats:

1. The left outer edge [p. 82] of a left-floating box may not be to the left of the left edge of its containing block [p. 96] . An analogous rule holds for right-floating elements.
2. If the current box is left-floating, and there are any left floating boxes generated by elements earlier in the source document, then for each such earlier box, either the left outer edge [p. 82] of the current box must be to the right of the right outer edge [p. 82] of the earlier box, or its top must be lower than the bottom of the earlier box. Analogous rules hold for right-floating boxes.
3. The right outer edge [p. 82] of a left-floating box may not be to the right of the left outer edge [p. 82] of any right-floating box that is to the right of it. Analogous rules hold for right-floating elements.
4. A floating box's outer top [p. 82] may not be higher than the top of its containing block [p. 96] .
5. The outer top [p. 82] of a floating box may not be higher than the outer top of any block [p. 97] or floated [p. 108] box generated by an element earlier in the source document.
6. The outer top [p. 82] of an element's floating box may not be higher than the top of any line-box [p. 105] containing a box generated by an element earlier in the source document.
7. A left-floating box that has another left-floating box to its left may not have its right outer edge to the right of its containing block's right edge. (Loosely: a left float may not stick out at the right edge, unless it is already as far to the left as possible.) An analogous rule holds for right-floating elements.
8. A floating box must be placed as high as possible.
9. A left-floating box must be put as far to the left as possible, a right-floating box as far to the right as possible. A higher position is preferred over one that is further to the left/right.

9.5.2 Controlling flow next to floats: the ’clear’ property

’clear’

Value:	none | left | right | both | inherit
Initial:	none
Applies to:	block-level elements
Inherited:	no
Percentages:	N/A
Media:	visual

This property indicates which sides of an element's box(es) may *not* be adjacent to an earlier floating box. (It may be that the element itself has floating descendants; the ’clear’ property has no effect on those.)

This property may only be specified for block-level [p. 97] elements (including floats). For compact [p. 99] and run-in boxes [p. 100] , this property applies to the final block box to which the compact or run-in box belongs.

Values have the following meanings when applied to non-floating block boxes:

left
: The top margin of the generated box is increased enough that the top border edge is below the bottom outer edge of any left-floating boxes that resulted from elements earlier in the source document.

right
: The top margin of the generated box is increased enough that the top border edge is below the bottom outer edge of any right-floating boxes that resulted from elements earlier in the source document.

both
: The generated box is moved below all floating boxes of earlier elements in the source document..

none
: No constraint on the box's position with respect to floats.

When the property is set on floating elements, it results in a modification of the rules [p. 112] for positioning the float. An extra constraint (#10) is added:

- The top outer edge [p. 82] of the float must be below the bottom outer edge of all earlier left-floating boxes (in the case of 'clear: left'), or all earlier right-floating boxes (in the case of 'clear: right'), or both ('clear: both').

9.6 Absolute positioning

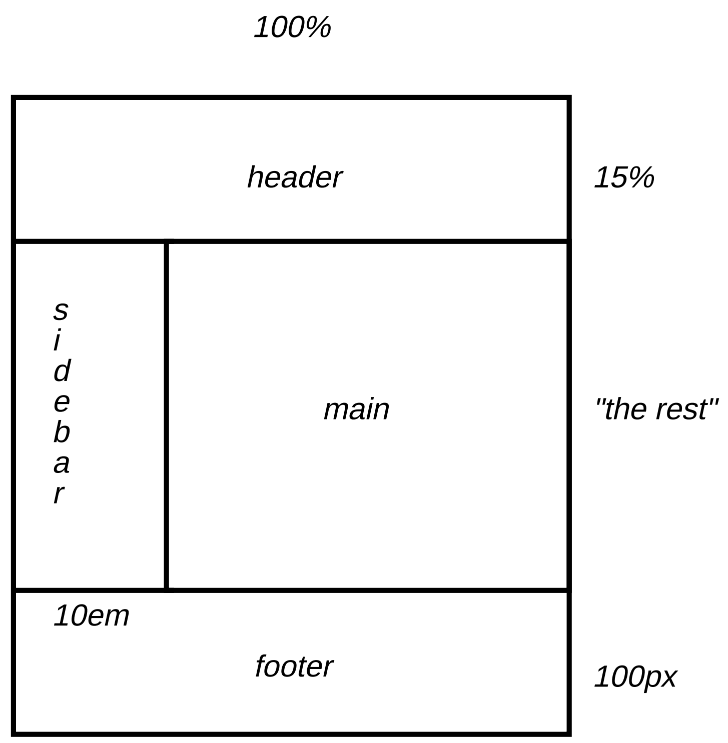
In the absolute positioning model, a box is explicitly offset with respect to its containing block. It is removed from the normal flow entirely (it has no impact on later siblings). An absolutely positioned box establishes a new containing block for normal flow children and positioned descendants. However, the contents of an absolutely positioned element do not flow around any other boxes. They may or may not obscure the contents of another box, depending on the stack levels [p. 125] of the overlapping boxes.

References in this specification to an *absolutely positioned element* (or its box) imply that the element's 'position' property has the value 'absolute' or 'fixed'.

9.6.1 Fixed positioning

Fixed positioning is a subcategory of absolute positioning. The only difference is that for a fixed positioned box, the containing block is established by the viewport [p. 96] . For continuous media [p. 79] , fixed boxes do not move when the document is scrolled. In this respect, they are similar to fixed background images [p. 188] . For paged media [p. 175] , boxes with fixed positions are repeated on every page. This is useful for placing, for instance, a signature at the bottom of each page.

Authors may use fixed positioning to create frame-like presentations. Consider the following frame layout:



This might be achieved with the following HTML document and style rules:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>A frame document with CSS2</TITLE>
    <STYLE type="text/css">
      BODY { height: 8.5in } /* Required for percentage heights below */
      #header {
        position: fixed;
        width: 100%;
        height: 15%;
        top: 0;
        right: 0;
        bottom: auto;
        left: 0;
      }
      #sidebar {
        position: fixed;
        width: 10em;
        height: auto;
        top: 15%;
        right: auto;
        bottom: 100px;
        left: 0;
      }
      #main {
        position: fixed;
        width: auto;
        height: auto;
        top: 15%;
        right: 0;
        bottom: 100px;
        left: 10em;
      }
      #footer {
        position: fixed;
        width: 100%;
        height: 100px;
        top: auto;
        right: 0;
        bottom: 0;
        left: 0;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <DIV id="header"> ...  </DIV>
    <DIV id="sidebar"> ...  </DIV>
    <DIV id="main"> ...  </DIV>
    <DIV id="footer"> ...  </DIV>
  </BODY>
</HTML>
```

9.7 Relationships between ’display’, ’position’, and ’float’

The three properties that affect box generation and layout -- ’display’, ’position’, and ’float’ -- interact as follows:

1. If ’display’ has the value ’none’, user agents must ignore [p. 42] ’position’ and ’float’. In this case, the element generates no box.
2. Otherwise, ’position’ has the value ’absolute’ or ’fixed’, ’display’ is set to ’block’ and ’float’ is set to ’none’. The position of the box will be determined by the ’top’, ’right’, ’bottom’ and ’left’ properties and the box’s containing block.
3. Otherwise, if ’float’ has a value other than ’none’, ’display’ is set to ’block’ and the box is floated.
4. Otherwise, the remaining ’display’ properties apply as specified.

Note. *CSS2 does not specify layout behavior when values for these properties are changed by scripts. For example, what happens when an element having ’width: auto’ is repositioned? Do the contents reflow, or do they maintain their original formatting? The answer is outside the scope of this document, and such behavior is likely to differ in initial implementations of CSS2.*

9.8 Comparison of normal flow, floats, and absolute positioning

To illustrate the differences between normal flow, relative positioning, floats, and absolute positioning, we provide a series of examples based on the following HTML fragment:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Comparison of positioning schemes</TITLE>
  </HEAD>
  <BODY>
    <P>Beginning of body contents.
```

```
      <SPAN id="outer"> Start of outer contents.
      <SPAN id="inner"> Inner contents.</SPAN>
      End of outer contents.</SPAN>
      End of body contents.
    </P>
  </BODY>
</HTML>
```

In this document, we assume the following rules:

```
BODY { display: block; line-height: 200%;
       width: 400px; height: 400px }
P    { display: block }
SPAN { display: inline }
```

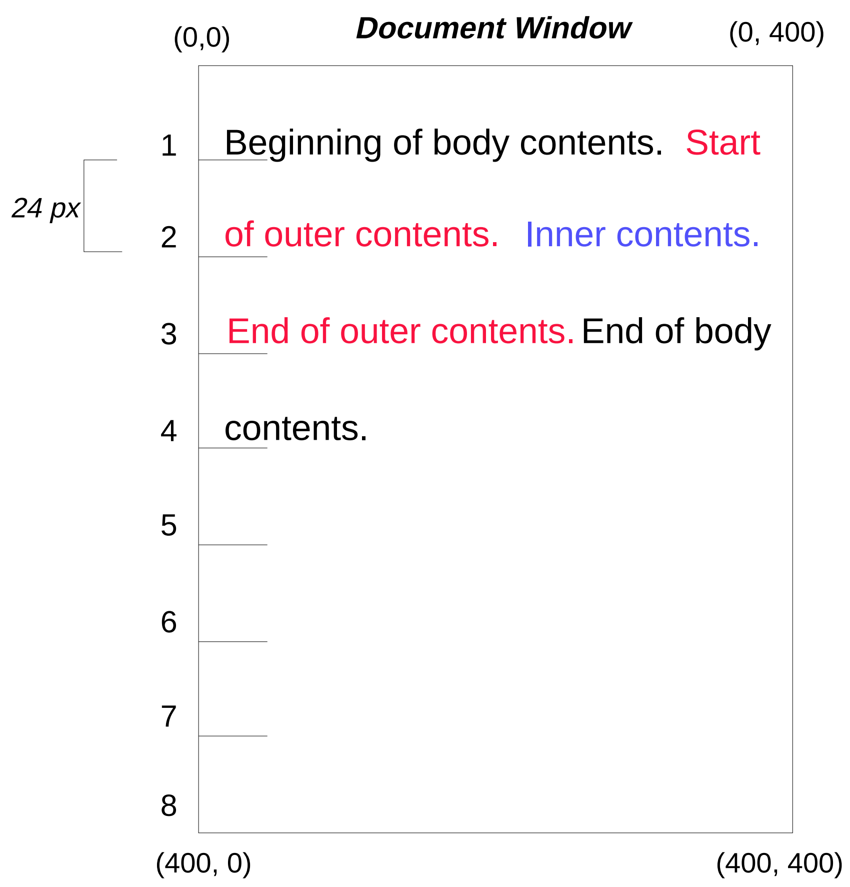
The final positions of boxes generated by the *outer* and *inner* elements vary in each example. In each illustration, the numbers to the left of the illustration indicate the normal flow [p. 105] position of the double-spaced (for clarity) lines. (Note: the illustrations use different horizontal and vertical scales.)

9.8.1 Normal flow

Consider the following CSS declarations for *outer* and *inner* that don't alter the normal flow [p. 105] of boxes:

```
#outer { color: red }
#inner { color: blue }
```

The P element contains all inline content: anonymous inline text [p. 98] and two SPAN element. Therefore, all of the content will be laid out in an inline formatting context, within a containing block established by the P element, producing something like:


Document Window
(0,0)
(0, 400)
24 px
1
2
3
4
5
6
7
8
Beginning of body contents. Start
of outer contents. Inner contents.
End of outer contents. End of body
contents.
(400, 0)
(400, 400)

9.8.2 Relative positioning

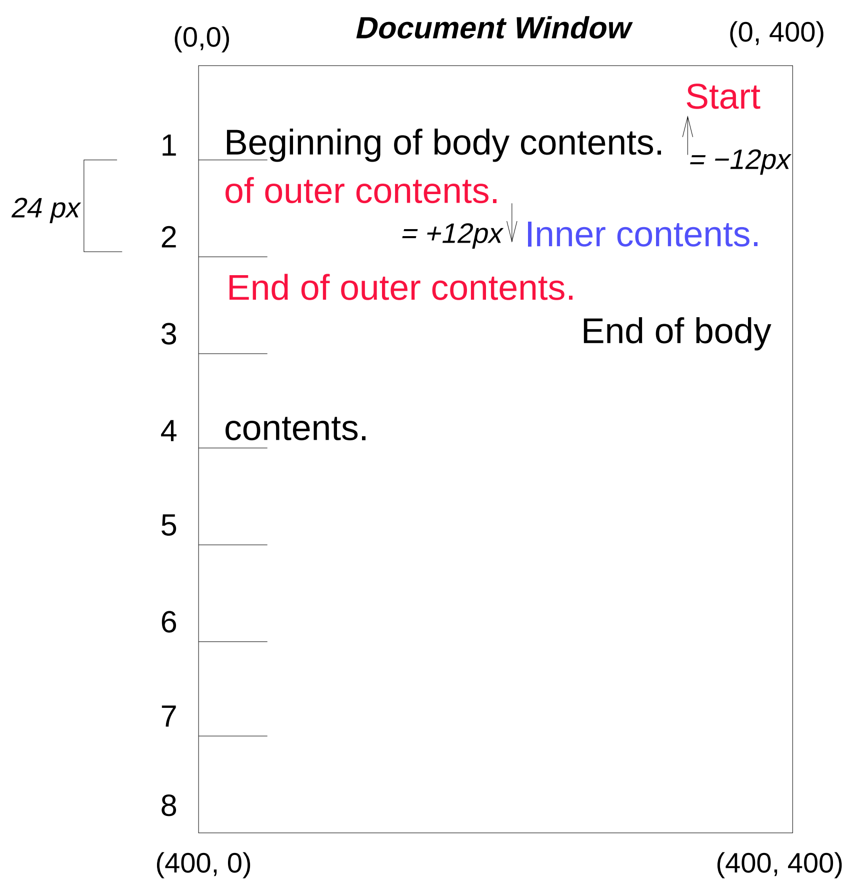
To see the effect of relative positioning [p. 107] , we specify:

```
#outer { position: relative; top: -12px; color: red }
#inner { position: relative; top: 12px; color: blue }
```

Text flows normally up to the *outer* element. The *outer* text is then flowed into its normal flow position and dimensions at the end of line 1. Then, the inline boxes containing the text (distributed over three lines) are shifted as a unit by ’-12px’ (upwards).

The contents of *inner*, as a child of *outer*, would normally flow immediately after the words "of outer contents" (on line 1.5). However, the *inner* contents are themselves offset relative to the *outer* contents by ’12px’ (downwards), back to their original position on line 2.

Note that the content following *outer* is not affected by the relative positioning of *outer*.



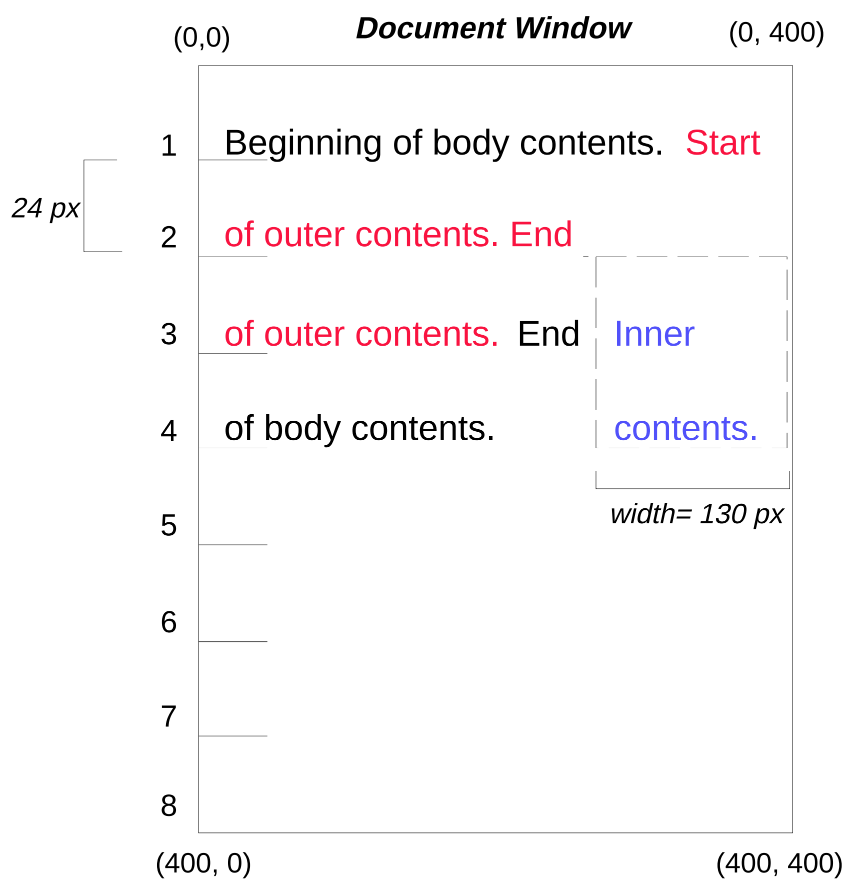
Note also that had the offset of *outer* been ’-24px’, the text of *outer* and the body text would have overlapped.

9.8.3 Floating a box

Now consider the effect of floating [p. 108] the *inner* element’s text to the right by means of the following rules:

```
#outer { color: red }
#inner { float: right; width: 130px; color: blue }
```

Text flows normally up to the *inner* box, which is pulled out of the flow and floated to the right margin (its ’width’ has been assigned explicitly). Line boxes to the left of the float are shortened, and the document’s remaining text flows into them.



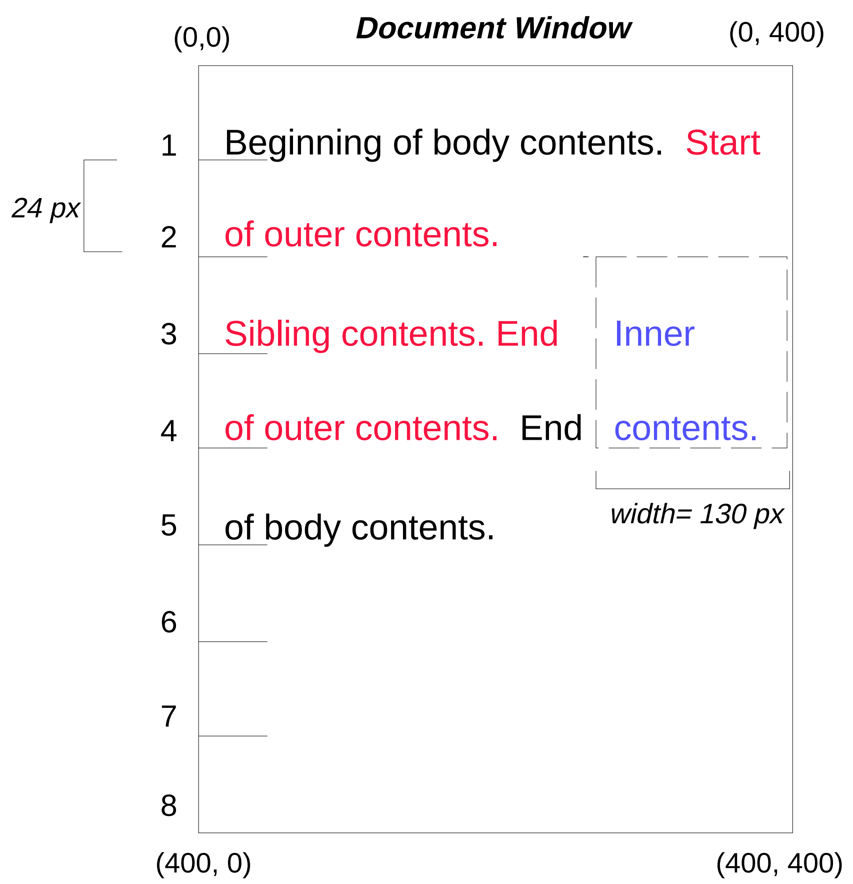
To show the effect of the ’clear’ property, we add a *sibling* element to the example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Comparison of positioning schemes II</TITLE>
  </HEAD>
  <BODY>
    <P>Beginning of body contents.
      <SPAN id=outer> Start of outer contents.
      <SPAN id=inner> Inner contents.</SPAN>
      <SPAN id=sibling> Sibling contents.</SPAN>
      End of outer contents.</SPAN>
      End of body contents.
    </P>
  </BODY>
</HTML>
```

The following rules:

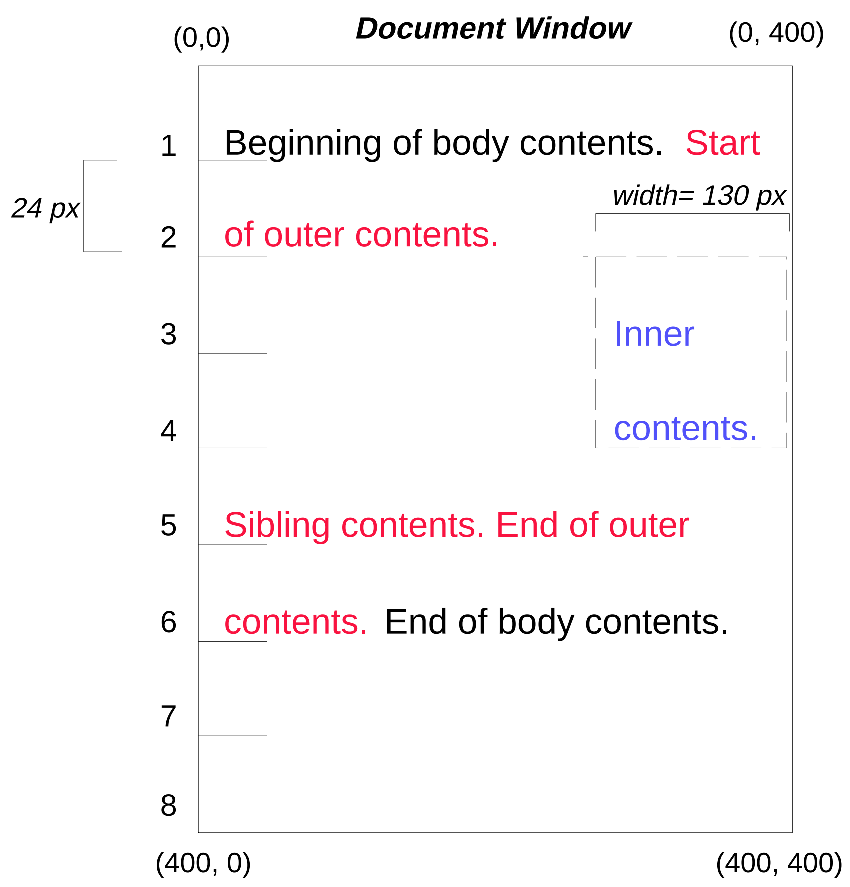
```
#inner { float: right; width: 130px; color: blue }
#sibling { color: red }
```

cause the *inner* box to float to the right as before and the document’s remaining text to flow into the vacated space:



However, if the ’clear’ property on the *sibling* element is set to ’right’ (i.e., the generated *sibling* box will not accept a position next to floating boxes to its right), the *sibling* content begins to flow below the float:

```
#inner { float: right; width: 130px; color: blue }
#sibling { clear: right; color: red }
```

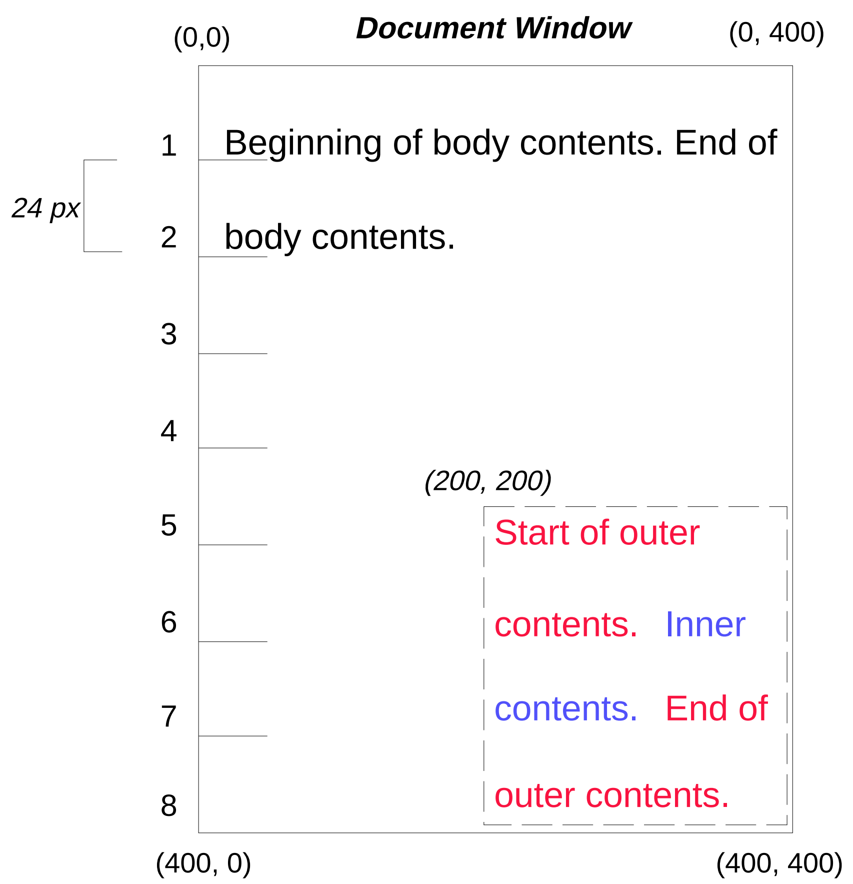


9.8.4 Absolute positioning

Finally, we consider the effect of absolute positioning [p. 113] . Consider the following CSS declarations for *outer* and *inner*:

```
#outer {
    position: absolute;
    top: 200px; left: 200px;
    width: 200px;
    color: red;
}
#inner { color: blue }
```

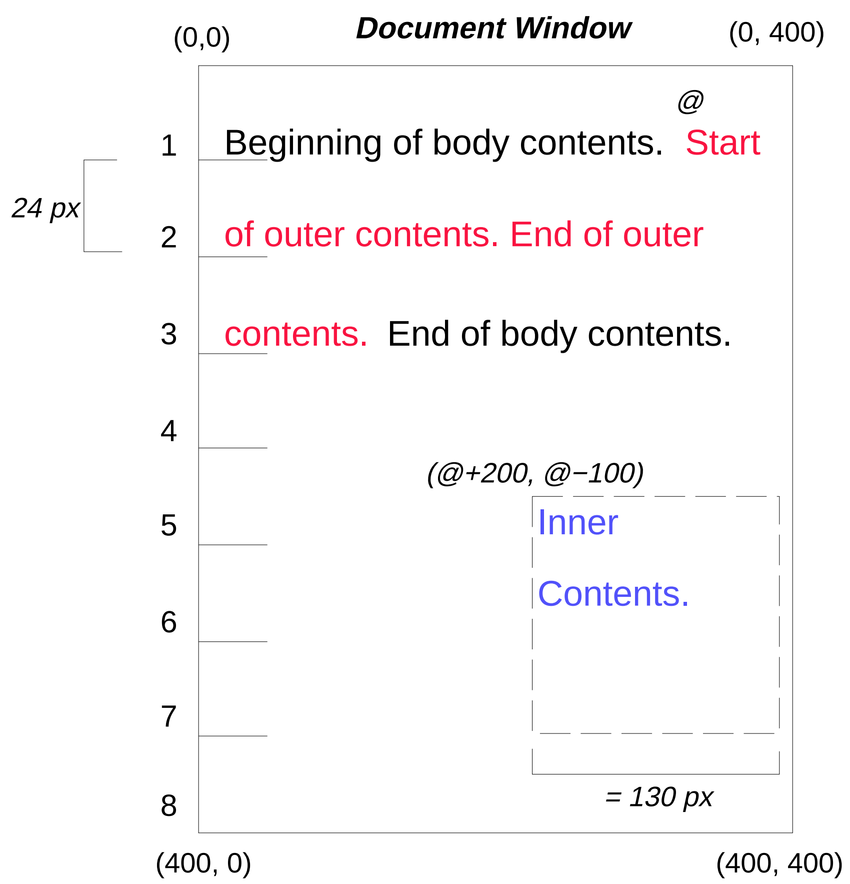
which cause the top of the *outer* box to be positioned with respect to its containing block. The containing block for a positioned box is established by the nearest positioned ancestor (or, if none exists, the initial containing block [p. 96] , as in our example). The top side of the *outer* box is ’200px’ below the top of the containing block and the left side is ’200px’ from the left side. The child box of *outer* is flowed normally with respect to its parent.



The following example shows an absolutely positioned box that is a child of a relatively positioned box. Although the parent *outer* box is not actually offset, setting its ’position’ property to ’relative’ means that its box may serve as the containing block for positioned descendants. Since the *outer* box is an inline box that is split across several lines, the first inline box’s top and left edges (depicted by thick dashed lines in the illustration below) serve as references for ’top’ and ’left’ offsets.

```
#outer {
  position: relative;
  color: red
}
#inner {
  position: absolute;
  top: 200px; left: -100px;
  height: 130px; width: 130px;
  color: blue;
}
```

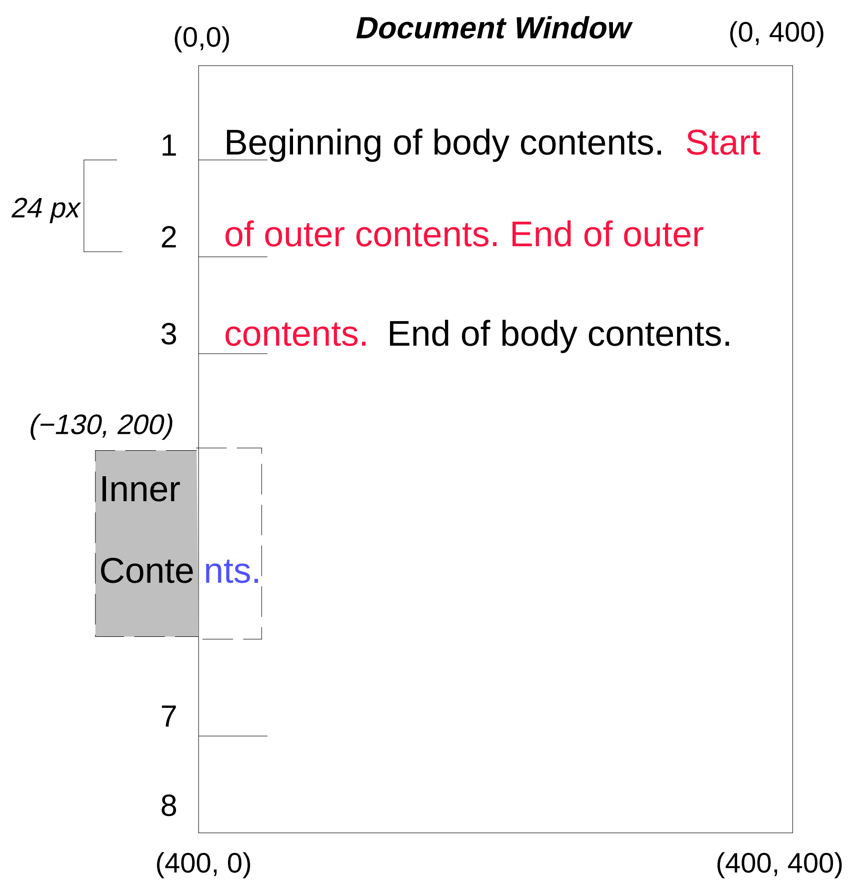
This results in something like the following:



If we do not position the *outer* box:

```
#outer { color: red }
#inner {
  position: absolute;
  top: 200px; left: -100px;
  height: 130px; width: 130px;
  color: blue;
}
```

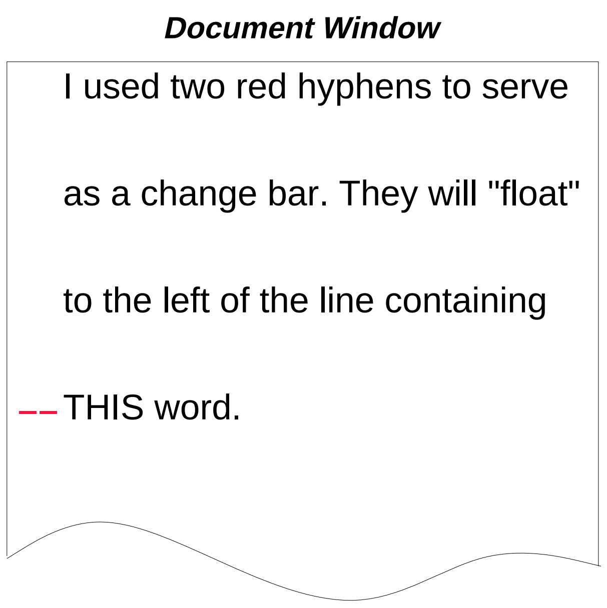
the containing block for *inner* becomes the initial containing block [p. 96] (in our example). The following illustration shows where the *inner* box would end up in this case.


(0,0)
Document Window
(0, 400)
1
Beginning of body contents. Start
24 px
2
of outer contents. End of outer
3
contents. End of body contents.
(−130, 200)
Inner
Conte nts.
7
8
(400, 0)
(400, 400)

Relative and absolute positioning may be used to implement change bars, as shown in the following example. The following document:

```
<P style="position: relative; margin-right: 10px; left: 10px;">
I used two red hyphens to serve as a change bar. They
will "float" to the left of the line containing THIS
<SPAN style="position: absolute; top: auto; left: -1em; color: red;">--</SPAN>
word.</P>
```

might result in something like:


Document Window
I used two red hyphens to serve
as a change bar. They will "float"
to the left of the line containing
-- THIS word.

First, the paragraph (whose containing block sides are shown in the illustration) is flowed normally. Then it is offset '10px' from the left edge of the containing block (thus, a right margin of '10px' has been reserved in anticipation of the offset). The two hyphens acting as change bars are taken out of the flow and positioned at the current line (due to 'top: auto'), '-1em' from the left edge of its containing block (established by the P in its final position). The result is that the change bars seem to "float" to the left of the current line.

9.9 Layered presentation

In the following sections, the expression "in front of" means closer to the user as the user faces the screen.

In CSS2, each box has a position in three dimensions. In addition to their horizontal and vertical positions, boxes lie along a "z-axis" and are formatted one on top of the other. Z-axis positions are particularly relevant when boxes overlap visually. This section discusses how boxes may be positioned along the z-axis.

Each box belongs to one *stacking context*. Each box in a given stacking context has an integer *stack level*, which is its position on the z-axis relative to other boxes in the same stacking context. Boxes with greater stack levels are always formatted in front of boxes with lower stack levels. Boxes may have negative stack levels. Boxes with the same stack level in a stacking context are stacked bottom-to-top according to document tree order.

The root [p. 30] element creates a *root stacking context*, but other elements may establish *local stacking contexts*. Stacking contexts are inherited. A local stacking context is atomic; boxes in other stacking contexts may not come between any of its boxes.

An element that establishes a local stacking context generates a box that has two stack levels: one for the stacking context it creates (always '0') and one for the stacking context to which it belongs (given by the 'z-index' property).

An element's box has the same stack level as its parent's box unless given a different stack level with the 'z-index' property.

9.9.1 Specifying the stack level: the ’z-index’ property

’z-index’

Value:	auto | <integer> | inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	N/A
Media:	visual

For a positioned box, the ’z-index’ property specifies:

1. The stack level of the box in the current stacking context.
2. Whether the box establishes a local stacking context.

Values have the following meanings:

<integer>
This integer is the stack level of the generated box in the current stacking context. The box also establishes a local stacking context in which its stack level is ’0’.

auto
The stack level of the generated box in the current stacking context is the same as its parent’s box. The box does not establish a new local stacking context.

In the following example, the stack levels of the boxes (named with their "id" attributes) are: "text2"=0, "image"=1, "text3"=2, and "text1"=3. The "text2" stack level is inherited from the root box. The others are specified with the ’z-index’ property.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Z-order positioning</TITLE>
    <STYLE type="text/css">
      .pile {
        position: absolute;
        left: 2in;
        top: 2in;
        width: 3in;
        height: 3in;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <P>
      <IMG id="image" class="pile"
           src="butterfly.gif" alt="A butterfly image"
           style="z-index: 1">

    <DIV id="text1" class="pile"
         style="z-index: 3">
      This text will overlay the butterfly image.
      </DIV>

      <DIV id="text2">
        This text will be beneath everything.
      </DIV>

      <DIV id="text3" class="pile"
           style="z-index: 2">
        This text will underlay text1, but overlay the butterfly image
      </DIV>
    </BODY>
  </HTML>
```

This example demonstrates the notion of *transparency*. The default behavior of a box is to allow boxes behind it to be visible through transparent areas in its content. In the example, each box transparently overlays the boxes below it. This behavior can be overridden by using one of the existing background properties [p. 188] .

9.10 Text direction: the ’direction’ and ’unicode-bidi’ properties

The characters in certain scripts are written from right to left. In some documents, in particular those written with the Arabic or Hebrew script, and in some mixed-language contexts, text in a single (visually displayed) block may appear with mixed directionality. This phenomenon is called *bidirectionality*, or "bidi" for short.

The Unicode standard ([UNICODE], section 3.11) defines a complex algorithm for determining the proper directionality of text. The algorithm consists of an implicit part based on character properties, as well as explicit controls for embeddings and overrides. CSS2 relies on this algorithm to achieve proper bidirectional rendering. The ’direction’ and ’unicode-bidi’ properties allow authors to specify how the elements and attributes of a document language map to this algorithm.

If a document contains right-to-left characters, and if the user agent displays these characters (with appropriate glyphs, not arbitrary substitutes such as a question mark, a hex code, a black box, etc.), the user agent must apply the bidirectional algorithm. This seemingly one-sided requirement reflects the fact that, although not every Hebrew or Arabic document contains mixed-directionality text, such documents are much more likely to contain left-to-right text (e.g., numbers, text from other languages) than are documents written in left-to-right languages.

Because the directionality of a text depends on the structure and semantics of the document language, these properties should in most cases be used only by designers of document type descriptions (DTDs), or authors of special documents. If a default style sheet specifies these properties, authors and users should not specify rules to override them. A typical exception would be to override bidi behavior in a user agent if that user agent transliterates Yiddish (usually written with Hebrew letters) to Latin letters at the user’s request.

The HTML 4.0 specification ([HTML40], section 8.2) defines bidirectionality behavior for HTML elements. Conforming [p. 32] HTML user agents may therefore ignore the ’direction’ and ’unicode-bidi’ properties in author and user style sheets. The style sheet rules that would achieve the bidi behavior specified in [HTML40] are given in the sample style sheet [p. 292] . The HTML 4.0 specification also contains more information on bidirectionality issues.

’direction’

Value:	ltr | rtl | inherit
Initial:	ltr
Applies to:	all elements, but see prose
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies the base writing direction of blocks and the direction of embeddings and overrides (see ’unicode-bidi’) for the Unicode bidirectional algorithm. In addition, it specifies the direction of table [p. 245] column layout, the direction of horizontal overflow [p. 145] , and the position of an incomplete last line in a block in case of ’text-align: justify’.

Values for this property have the following meanings:

ltr
Left-to-right direction.

rtl
Right-to-left direction.

For the ’direction’ property to have any effect on inline-level elements, the ’unicode-bidi’ property’s value must be ’embed’ or ’override’.

Note. *The ’direction’ property, when specified for table column elements, is not inherited by cells in the column since columns don’t exist in the document tree. Thus, CSS cannot easily capture the "dir" attribute inheritance rules described in [HTML40], section 11.3.2.1.*

’unicode-bidi’

Value:	normal | embed | bidi-override | inherit
Initial:	normal
Applies to:	all elements, but see prose
Inherited:	no
Percentages:	N/A
Media:	visual

Values for this property have the following meanings:

normal
The element does not open an additional level of embedding with respect to the bidirectional algorithm. For inline-level elements, implicit reordering works across element boundaries.

embed
If the element is inline-level, this value opens an additional level of embedding with respect to the bidirectional algorithm. The direction of this embedding level is given by the ’direction’ property. Inside the element, reordering is done implicitly. This corresponds to adding a LRE (U+202A; for ’direction: ltr’) or RLE (U+202B; for ’direction: rtl’) at the start of the element and a PDF

(U+202C) at the end of the element.

bidi-override

If the element is inline-level or a block-level element that contains only inline-level elements, this creates an override. This means that inside the element, reordering is strictly in sequence according to the ’direction’ property; the implicit part of the bidirectional algorithm is ignored. This corresponds to adding a LRO (U+202D; for ’direction: ltr’) or RLO (U+202E; for ’direction: rtl’) at the start of the element and a PDF (U+202C) at the end of the element.

The final order of characters in each block-level element is the same as if the bidi control codes had been added as described above, markup had been stripped, and the resulting character sequence had been passed to an implementation of the Unicode bidirectional algorithm for plain text that produced the same line-breaks as the styled text. In this process, non-textual entities such as images are treated as neutral characters, unless their ’unicode-bidi’ property has a value other than ’normal’, in which case they are treated as strong characters in the ’direction’ specified for the element.

Please note that in order to be able to flow inline boxes in a uniform direction (either entirely left-to-right or entirely right-to-left), more inline boxes (including anonymous inline boxes) may have to be created, and some inline boxes may have to be split up and reordered before flowing.

Because the Unicode algorithm has a limit of 15 levels of embedding, care should be taken not to use ’unicode-bidi’ with a value other than ’normal’ unless appropriate. In particular, a value of ’inherit’ should be used with extreme caution. However, for elements that are, in general, intended to be displayed as blocks, a setting of ’unicode-bidi: embed’ is preferred to keep the element together in case display is changed to inline (see example below).

The following example shows an XML document with bidirectional text. It illustrates an important design principle: DTD designers should take bidi into account both in the language proper (elements and attributes) and in any accompanying style sheets. The style sheets should be designed so that bidi rules are separate from other style rules. The bidi rules should not be overridden by other style sheets so that the document language’s or DTD’s bidi behavior is preserved.

Example(s):

In this example, lowercase letters stand for inherently left-to-right characters and uppercase letters represent inherently right-to-left characters:

```
<HEBREW>
  <PAR>HEBREW1 HEBREW2 english3 HEBREW4 HEBREW5</PAR>
  <PAR>HEBREW6 <EMPH>HEBREW7</EMPH> HEBREW8</PAR>
</HEBREW>
<ENGLISH>
  <PAR>english9 english10 english11 HEBREW12 HEBREW13</PAR>
  <PAR>english14 english15 english16</PAR>
  <PAR>english17 <HE-QUO>HEBREW18 english19 HEBREW20</HE-QUO></PAR>
</ENGLISH>
```

Since this is XML, the style sheet is responsible for setting the writing direction. This is the style sheet:

```
/* Rules for bidi */
HEBREW, HE-QUO  {direction: rtl; unicode-bidi: embed}
ENGLISH         {direction: ltr; unicode-bidi: embed}

/* Rules for presentation */
HEBREW, ENGLISH, PAR  {display: block}
EMPH                  {font-weight: bold}
```

The HEBREW element is a block with a right-to-left base direction, the ENGLISH element is a block with a left-to-right base direction. The PARs are blocks that inherit the base direction from their parents. Thus, the first two PARs are read starting at the top right, the final three are read starting at the top left. Please note that HEBREW and ENGLISH are chosen as element names for explicitness only; in general, element names should convey structure without reference to language.

The EMPH element is inline-level, and since its value for ’unicode-bidi’ is ’normal’ (the initial value), it has no effect on the ordering of the text. The HE-QUO element, on the other hand, creates an embedding.

The formatting of this text might look like this if the line length is long:

```
               5WERBEH 4WERBEH english3 2WERBEH 1WERBEH

                                8WERBEH 7WERBEH 6WERBEH

english9 english10 english11 13WERBEH 12WERBEH

english14 english15 english16

english17 20WERBEH english19 18WERBEH
```

Note that the HE-QUO embedding causes HEBREW18 to be to the right of english19.

If lines have to be broken, it might be more like this:

```
       2WERBEH 1WERBEH
  -EH 4WERBEH english3
                 5WERB

   -EH 7WERBEH 6WERBEH
                 8WERB

english9 english10 en-
glish11 12WERBEH
13WERBEH

english14 english15
english16

english17 18WERBEH
20WERBEH english19
```

Because HEBREW18 must be read before english19, it is on the line above english19. Just breaking the long line from the earlier formatting would not have worked. Note also that the first syllable from english19 might have fit on the previous line, but hyphenation of left-to-right words in a right-to-left context, and vice versa, is usually suppressed to avoid having to display a hyphen in the middle of a line.

10 Visual formatting model details

Contents

10.1 Definition of "containing block"

The position and size of an element's box(es) are sometimes computed relative to a certain rectangle, called the *containing block* of the element. The containing block of an element is defined as follows:

1. The containing block (called the initial containing block) in which the root element [p. 30] lives is chosen by the user agent.
2. For other elements, unless the element is absolutely positioned [p. 113] , the containing block is formed by the content edge of the nearest block-level ancestor box.
3. If the element has 'position: fixed', the containing block is established by the viewport [p. 96] .
4. If the element has 'position: absolute', the containing block is established by the nearest ancestor with a 'position' other than 'static', in the following way:
   1. In the case that the ancestor is block-level, the containing block is

formed by the padding edge [p. 82] of the ancestor.
2. In the case that the ancestor is inline-level, the containing block depends on the ’direction’ property of the ancestor:
    1. If the ’direction’ is ’ltr’, the top and left of the containing block are the top and left content edges of the first box generated by the ancestor, and the bottom and right are the bottom and right content edges of the last box of the ancestor.
    2. If the ’direction’ is ’rtl’, the top and right are the top and right edges of the first box generated by the ancestor, and the bottom and left are the bottom and left content edges of the last box of the ancestor.

If there is no such ancestor, the content edge of the root element’s box establishes the containing block.

Example(s):
With no positioning, the containing blocks (C.B.) in the following document:

```
<HTML>
   <HEAD>
      <TITLE>Illustration of containing blocks</TITLE>
   </HEAD>
   <BODY id="body">
      <DIV id="div1">
      <P id="p1">This is text in the first paragraph...</P>
      <P id="p2">This is text <EM id="em1"> in the
      <STRONG id="strong1">second</STRONG> paragraph.</EM></P>
      </DIV>
   </BODY>
</HTML>
```

are established as follows:

For box generated by	**C.B. is established by**
body	initial C.B. (UA-dependent)
div1	body
p1	div1
p2	div1
em1	p2
strong1	p2

If we position "div1":

```
   #div1 { position: absolute; left: 50px; top: 50px }
```

its containing block is no longer "body"; it becomes the initial containing block (since there are no other positioned ancestor boxes).
If we position "em1" as well:

```
#div1 { position: absolute; left: 50px; top: 50px }
#em1  { position: absolute; left: 100px; top: 100px }
```

the table of containing blocks becomes:

For box generated by	C.B. is established by
body	initial C.B.
div1	initial C.B.
p1	div1
p2	div1
em1	div1
strong1	em1

By positioning "em1", its containing block becomes the nearest positioned ancestor box (i.e., that generated by "div1").

10.2 Content width: the ’width’ property

’width’

Value:	<length> | <percentage> | auto | inherit
Initial:	auto
Applies to:	all elements but non-replaced inline elements, table rows, and row groups
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

This property specifies the content width [p. 82] of boxes generated by block-level and replaced [p. 30] elements.

This property does not apply to non-replaced inline-level [p. 98] elements. The width of a non-replaced inline element’s boxes is that of the rendered content within them (*before* any relative offset of children). Recall that inline boxes flow into line boxes [p. 105] . The width of line boxes is given by the their containing block [p. 96] , but may be shorted by the presence of floats [p. 108] .

The width of a replaced element’s box is intrinsic [p. 30] and may be scaled by the user agent if the value of this property is different than ’auto’.

Values have the following meanings:

<length>
Specifies a fixed width.

<percentage>
Specifies a percentage width. The percentage is calculated with respect to the width of the generated box’s containing block [p. 96] .

auto

The width depends on the values of other properties. See the sections below.

Negative values for ’width’ are illegal.

Example(s):

For example, the following rule fixes the content width of paragraphs at 100 pixels:

```
P { width: 100px }
```

10.3 Computing widths and margins

The computed values of an element’s ’width’, ’margin-left’, ’margin-right’, ’left’ and ’right’ properties depend on the type of box generated and on each other. In principle, the computed values are the same as the specified values, with ’auto’ replaced by some suitable value, but there are exceptions. The following situations need to be distinguished:

1. inline, non-replaced elements
2. inline, replaced elements
3. block-level, non-replaced elements in normal flow
4. block-level, replaced elements in normal flow
5. floating, non-replaced elements
6. floating, replaced elements
7. absolutely positioned, non-replaced elements
8. absolutely positioned, replaced elements

Points 1-6 include relative positioning.

10.3.1 Inline, non-replaced elements

The ’width’ property does not apply. A specified value of ’auto’ for ’left’, ’right’, ’margin-left’ or ’margin-right’ becomes a computed value of ’0’.

10.3.2 Inline, replaced elements

A specified value of ’auto’ for ’left’, ’right’, ’margin-left’ or ’margin-right’ becomes a computed value of ’0’. A specified value of ’auto’ for ’width’ gives the element’s intrinsic [p. 30] width as the computed value.

10.3.3 Block-level, non-replaced elements in normal flow

If ’left’ or ’right’ are given as ’auto’, their computed value is 0. The following constraints must hold between the other properties:

’margin-left’ + ’border-left-width’ + ’padding-left’ + ’width’ + ’padding-right’ + ’border-right-width’ + ’margin-right’ = width of containing block [p. 131]

(If the border style is ’none’, use ’0’ as the border width.) If all of the above have a specified value other than ’auto’, the values are said to be "over-constrained" and one of the computed values will have to be different from its specified value. If the ’direction’ property has the value ’ltr’, the specified value of ’margin-right’ is ignored and the value is computed so as to make the equality true. If the value of ’direction’ is ’ltr’, this happens to ’margin-left’ instead.

If there is exactly one value specified as ’auto’, its computed value follows from the equality.

If ’width’ is set to ’auto’, any other ’auto’ values become ’0’ and ’width’ follows from the resulting equality.

If both ’margin-left’ and ’margin-right’ are ’auto’, their computed values are equal.

10.3.4 Block-level, replaced elements in normal flow

If ’left’ or ’right’ are ’auto’, their computed value is 0. If ’width’ is specified as ’auto’, its value is the element’s intrinsic [p. 30] width. If one of the margins is ’auto’, its computed value is given by the constraints [p. 134] above. Furthermore, if both margins are ’auto’, their computed values are equal.

10.3.5 Floating, non-replaced elements

If ’left’, ’right’, ’width’, ’margin-left’, or ’margin-right’ are specified as ’auto’, their computed value is ’0’.

10.3.6 Floating, replaced elements

If ’left’, ’right’, ’margin-left’ or ’margin-right’ are specified as ’auto’, their computed value is ’0’. If ’width’ is ’auto’, its value is the element’s intrinsic [p. 30] width.

10.3.7 Absolutely positioned, non-replaced elements

The constraint that determines the computed values for these elements is:

> ’left’ + ’margin-left’ + ’border-left-width’ + ’padding-left’ + ’width’ + ’padding-right’ + ’border-right-width’ + ’margin-right’ + ’right’ = width of containing block [p. 131]

(If the border style is ’none’, use ’0’ as the border width.) The solution to this constraint is reached through a number of substitutions in the following order:

1. If ’left’ has the value ’auto’ while ’direction’ is ’ltr’, replace ’auto’ with the distance from the left edge of the containing block to the left margin edge of a hypothetical box that would have been the first box of the element if its ’position’ property had been ’static’. (But rather than actually computing that box, user agents are free to make a guess at its probable position.) The value is negative if the hypothetical box is to the left of the containing block.
2. If ’right’ has the value ’auto’ while ’direction’ is ’rtl’, replace ’auto’ with the distance from the right edge of the containing block to the right margin edge of the same hypothetical box as above. The value is positive if the hypothetical box is to the left of the containing block’s edge.

3. If ’width’ is ’auto’, replace any remaining ’auto’ for ’left’ or ’right’ with ’0’.
4. If ’left’, ’right’ or ’width’ are (still) ’auto’, replace any ’auto’ on ’margin-left’ or ’margin-right’ with ’0’.
5. If at this point both ’margin-left’ and ’margin-right’ are still ’auto’, solve the equation under the extra constraint that the two margins must get equal values.
6. If at this point there is only one ’auto’ left, solve the equation for that value.
7. If at this point the values are over-constrained, ignore the value for either ’left’ (in case ’direction’ is ’rtl’) or ’right’ (in case ’direction’ is ’ltr’) and solve for that value.

10.3.8 Absolutely positioned, replaced elements

This situation is similar to the previous one, except that the element has an intrinsic [p. 30] width. The sequence of substitutions is now:

1. If ’width’ is ’auto’, substitute the element’s intrinsic [p. 30] width.
2. If ’left’ has the value ’auto’ while ’direction’ is ’ltr’, replace ’auto’ with the distance from the left edge of the containing block to the left margin edge of a hypothetical box that would have been the first box of the element if its ’position’ property had been ’static’. (But rather than actually computing that box, user agents are free to make a guess at its probable position.) The value is negative if the hypothetical box is to the left of the containing block.
3. If ’right’ has the value ’auto’ while ’direction’ is ’rtl’, replace ’auto’ with the distance from the right edge of the containing block to the right margin edge of the same hypothetical box as above. The value is positive if the hypothetical box is to the left of the containing block’s edge.
4. If ’left’ or ’right’ are ’auto’, replace any ’auto’ on ’margin-left’ or ’margin-right’ with ’0’.
5. If at this point both ’margin-left’ and ’margin-right’ are still ’auto’, solve the equation under the extra constraint that the two margins must get equal values.
6. If at this point there is only one ’auto’ left, solve the equation for that value.
7. If at this point the values are over-constrained, ignore the value for either ’left’ (in case ’direction’ is ’rtl’) or ’right’ (in case ’direction’ is ’ltr’) and solve for that value.

10.4 Minimum and maximum widths: ’min-width’ and ’max-width’

’min-width’

Value:	<length> | <percentage> | inherit
Initial:	UA dependent
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

’max-width’

Value:	<length> | <percentage> | none | inherit
Initial:	none
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual

These two properties allow authors to constrain box widths to a certain range. Values have the following meanings:

<length>
Specifies a fixed minimum or maximum computed width.

<percentage>
Specifies a percentage for determining the computed value. The percentage is calculated with respect to the width of the generated box’s containing block [p. 96] .

none
(Only on ’max-width’) No limit on the width of the box.

The following algorithm describes how the two properties influence the computed value [p. 70] of the ’width’ property:

1. The width is computed (without ’min-width’ and ’max-width’) following the rules under "Computing widths and margins" [p. 134] above.
2. If the computed value of ’min-width’ is greater than the value of ’max-width’, ’max-width’ is set to the value of ’min-width’.
3. If the computed width is greater than ’max-width’, the rules above [p. 134] are applied again, but this time using the value of ’max-width’ as the specified value for ’width’.
4. If the computed width is smaller than ’min-width’, the rules above [p. 134] are applied again, but this time using the value of ’min-width’ as the specified value for ’width’.

The user agent may define a non-negative minimum value for the ’min-width’ property, which may vary from element to element and even depend on other properties. If ’min-width’ goes below this limit, either because it was set explicitly, or because it was ’auto’ and the rules below would make it too small, the user agent may use the minimum value as the computed value.

10.5 Content height: the ’height’ property

’height’

Value:	<length> | <percentage> | auto | inherit
Initial:	auto
Applies to:	all elements but non-replaced inline elements, table columns, and column groups
Inherited:	no
Percentages:	see prose
Media:	visual

This property specifies the content height [p. 82] of boxes generated by block-level and replaced [p. 30] elements.

This property does not apply to non-replaced inline-level [p. 98] elements. The height of a non-replaced inline element's boxes is given by the element's (possibly inherited) 'line-height' value.

Values have the following meanings:

<length>
Specifies a fixed height.

<percentage>
Specifies a percentage height. The percentage is calculated with respect to the height of the generated box's containing block [p. 96] . If the height of the containing block is not specified explicitly (i.e., it depends on content height), the value is interpreted like 'auto'.

auto
The height depends on the values of other properties. See the prose below.

Negative values for 'height' are illegal.

Example(s):

For example, the following rule fixes the height of paragraphs to 100 pixels:

```
P { height: 100px }
```

Paragraphs that require more than 100 pixels of height will overflow [p. 145] according to the 'overflow' property.

10.6 Computing heights and margins

For computing the values of 'top', 'margin-top', 'height', 'margin-bottom', and 'bottom' a distinction must be made between various kinds of boxes:

1. inline, non-replaced elements
2. inline, replaced elements
3. block-level, non-replaced elements in normal flow
4. block-level, replaced elements in normal flow
5. floating, non-replaced elements
6. floating, replaced elements
7. absolutely positioned, non-replaced elements
8. absolutely positioned, replaced elements

Points 1-6 include relative positioning.

10.6.1 Inline, non-replaced elements

If ’top’, ’bottom’, ’margin-top’, or ’margin-bottom’ are ’auto’, their computed value is 0. The ’height’ property doesn’t apply, but the height of the box is given by the ’line-height’ property.

10.6.2 Inline, replaced elements block-level, replaced elements in normal flow, and floating, replaced elements

If ’top’, ’bottom’, ’margin-top’, or ’margin-bottom’ are ’auto’, their computed value is 0. If ’height’ is ’auto’, the computed value is the intrinsic [p. 30] height.

10.6.3 Block-level, non-replaced elements in normal flow, and floating, non-replaced elements

If ’top’, ’bottom’, ’margin-top’, or ’margin-bottom’ are ’auto’, their computed value is 0. If ’height’ is ’auto’, the height depends on whether the element has any block-level children. If it only has inline-level children, the height is from the top of the topmost line box to the bottom of the bottommost line box. If it has block-level children, it is the distance from the top border-edge of the topmost block-level child box, to the bottom border-edge of the bottommost block-level child box. Only children in the normal flow are taken into account (i.e., floating boxes and absolutely positioned boxes are ignored, and relatively positioned boxes are considered without their offset). Note that the child box may be an anonymous box. [p. 97]

10.6.4 Absolutely positioned, non-replaced elements

For absolutely positioned elements, the vertical dimensions must satisfy this constraint:

’top’ + ’margin-top’ + ’border-top-width’ + ’padding-top’ + ’height’ + ’padding-bottom’ + ’border-bottom-width’ + ’margin-bottom’ + ’bottom’ = height of containing block

(If the border style is ’none’, use ’0’ as the border width.) The solution to this constraint is reached through a number of substitutions in the following order:

1. If ’top’ has the value ’auto’ replace it with the distance from the top edge of the containing block to the top margin edge of a hypothetical box that would have been the first box of the element if its ’position’ property had been ’static’. (But rather than actually computing that box, user agents are free to make a guess at its probable position.) The value is negative if the hypothetical box is above the containing block.
2. If both ’height’ and ’bottom’ are ’auto’, replace ’bottom’ with 0.
3. If ’bottom’ or ’height’ are (still) ’auto’, replace any ’auto’ on ’margin-top’ or ’margin-bottom’ with ’0’.
4. If at this point both ’margin-top’ and ’margin-bottom’ are still ’auto’, solve the equation under the extra constraint that the two margins must get equal

values.
5. If at this point there is only one ’auto’ left, solve the equation for that value.
6. If at this point the values are over-constrained, ignore the value for ’bottom’ and solve for that value.

10.6.5 Absolutely positioned, replaced elements

This situation is similar to the previous one, except that the element has an intrinsic [p. 30] height. The sequence of substitutions is now:

1. If ’height’ is ’auto’, substitute the element’s intrinsic [p. 30] height.
2. If ’top’ has the value ’auto’, replace it with the distance from the top edge of the containing block to the top margin edge of a hypothetical box that would have been the first box of the element if its ’position’ property had been ’static’. (But rather than actually computing that box, user agents are free to make a guess at its probable position.) The value is negative if the hypothetical box is above the containing block.
3. If ’bottom’ is ’auto’, replace any ’auto’ on ’margin-top’ or ’margin-bottom’ with ’0’.
4. If at this point both ’margin-top’ and ’margin-bottom’ are still ’auto’, solve the equation under the extra constraint that the two margins must get equal values.
5. If at this point there is only one ’auto’ left, solve the equation for that value.
6. If at this point the values are over-constrained, ignore the value for ’bottom’ and solve for that value.

10.7 Minimum and maximum heights: ’min-height’ and ’max-height’

It is sometimes useful to constrain the height of elements to a certain range. Two properties offer this functionality:

’min-height’

Value:	<length> | <percentage> | inherit
Initial:	0
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	refer to height of containing block
Media:	visual

’max-height’

Value:	<length> | <percentage> | none | inherit
Initial:	none
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	refer to height of containing block
Media:	visual

These two properties allow authors to constrain box heights to a certain range. Values have the following meanings:

<length>
Specifies a fixed minimum or maximum computed height.

<percentage>
Specifies a percentage for determining the computed value. The percentage is calculated with respect to the height of the generated box’s containing block [p. 96] . If the height of the containing block is not specified explicitly (i.e., it depends on content height), the percentage value is interpreted like ’auto’.

none
(Only on ’max-height’) No limit on the height of the box.

The following algorithm describes how the two properties influence the computed value [p. 70] of the ’height’ property:

1. The height is computed (without ’min-height’ and ’max-height’) following the rules under "Computing heights and margins" [p. 138] above.
2. If the computed value of ’min-height’ is greater than the value of ’max-height’, ’max-height’ is set to the value of ’min-height’.
3. If the computed height is greater than ’max-height’, the rules above [p. 138] are applied again, but this time using the value of ’max-height’ as the specified value for ’height’.
4. If the computed height is smaller than ’min-height’, the rules above [p. 138] are applied again, but this time using the value of ’min-height’ as the specified value for ’height’.

10.8 Line height calculations: the ’line-height’ and ’vertical-align’ properties

As described in the section on inline formatting contexts [p. 105] , user agents flow inline boxes into a vertical stack of line boxes [p. 105] . The height of a line box is determined as follows:

1. The height of each inline box in the line box is calculated (see "Computing heights and margins" [p. 138] and the ’line-height’ property).
2. The inline boxes are aligned vertically according to their ’vertical-align’ property.
3. The line box height is the distance between the uppermost box top and the lowermost box bottom.

Empty inline elements generate empty inline boxes, but these boxes still have margins, padding, borders and a line height, and thus influence these calculations just like elements with content.

Note that if all the boxes in the line box are aligned along their bottoms, the line box will be exactly the height of the tallest box. If, however, the boxes are aligned along a common baseline, the line box top and bottom may not touch the top and bottom of the tallest box.

10.8.1 Leading and half-leading

Since the height of an inline box may be different from the font size of text in the box (e.g., ’line-height’ > 1em), there may be space above and below rendered glyphs. The difference between the font size and the computed value of ’line-height’ is called the *leading*. Half the leading is called the *half-leading*.

User agents center glyphs vertically in an inline box, adding half-leading on the top and bottom. For example, if a piece of text is ’12pt’ high and the ’line-height’ value is ’14pt’, 2pts of extra space should be added: 1pt above and 1pt below the letters. (This applies to empty boxes as well, as if the empty box contained an infinitely narrow letter.)

When the ’line-height’ value is less than the font size, the final inline box height will be less than the font size and the rendered glyphs will "bleed" outside the box. If such a box touches the edge of a line box, the rendered glyphs will also "bleed" into the adjacent line box.

Although margins, borders, and padding of non-replaced elements do not enter into inline box height calculation (and thus the line box calculation), they are still rendered around inline boxes. This means that if the height of a line box is shorter than the outer edges [p. 82] of the boxes it contains, backgrounds and colors of padding and borders may "bleed" into adjacent line boxes. However, in this case, some user agents may use the line box to "clip" the border and padding areas (i.e., not render them).

’line-height’

Value:	normal | <number> | <length> | <percentage> | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	refer to the font size of the element itself
Media:	visual

If the property is set on a block-level [p. 97] element whose content is composed of inline-level [p. 98] elements, it specifies the *minimal* height of each generated inline box.

If the property is set on an inline-level [p. 98] element, it specifies the *exact* height of each box generated by the element. (Except for inline replaced [p. 30] elements, where the height of the box is given by the ’height’ property.)

Values for this property have the following meanings:

normal
Tells user agents to set the computed value [p. 70] to a "reasonable" value based on the font size of the element. The value has the same meaning as

<number>. We recommend a computed value for ’normal’ between 1.0 to 1.2.

<length>
The box height is set to this length. Negative values are illegal.

<number>
The computed value [p. 70] of the property is this number multiplied by the element’s font size. Negative values are illegal. However, the number, not the computed value [p. 70] , is inherited.

<percentage>
The computed value [p. 70] of the property is this percentage multiplied by the element’s computed font size. Negative values are illegal.

Example(s):
The three rules in the example below have the same resultant line height:

```
DIV { line-height: 1.2; font-size: 10pt }     /* number */
DIV { line-height: 1.2em; font-size: 10pt }   /* length */
DIV { line-height: 120%; font-size: 10pt }    /* percentage */
```

When an element contains text that is rendered in more than one font, user agents should determine the ’line-height’ value according to the largest font size.
Generally, when there is only one value of ’line-height’ for all inline boxes in a paragraph (and no tall images), the above will ensure that baselines of successive lines are exactly ’line-height’ apart. This is important when columns of text in different fonts have to be aligned, for example in a table.
Note that replaced elements have a ’font-size’ and a ’line-height’ property, even if they are not used directly to determine the height of the box. The ’font-size’ is, however, used to define the ’em’ and ’ex’ units, and the ’line-height’ has a role in the ’vertical-align’ property.

’vertical-align’

Value:	baseline | sub | super | top | text-top | middle | bottom | text-bottom | <percentage> | <length> | inherit
Initial:	baseline
Applies to:	inline-level and ’table-cell’ elements
Inherited:	no
Percentages:	refer to the ’line-height’ of the element itself
Media:	visual

This property affects the vertical positioning inside a line box of the boxes generated by an inline-level element. The following values only have meaning with respect to a parent inline-level element, or to a parent block-level element, if that element generates anonymous inline boxes [p. 98] ; they have no effect if no such parent exists.
Note. *Values of this property have slightly different meanings in the context of tables. Please consult the section on table height algorithms [p. 258] for details.*

baseline
Align the baseline of the box with the baseline of the parent box. If the box doesn’t have a baseline, align the bottom of the box with the parent’s baseline.

middle
Align the vertical midpoint of the box with the baseline of the parent box plus half the x-height of the parent.

sub
Lower the baseline of the box to the proper position for subscripts of the parent’s box. (This value has no effect on the font size of the element’s text.)

super
Raise the baseline of the box to the proper position for superscripts of the parent’s box. (This value has no effect on the font size of the element’s text.)

text-top
Align the top of the box with the top of the parent element’s font.

text-bottom
Align the bottom of the box with the bottom of the parent element’s font.

<percentage>
Raise (positive value) or lower (negative value) the box by this distance (a percentage of the ’line-height’ value). The value ’0%’ means the same as ’baseline’.

<length>
Raise (positive value) or lower (negative value) the box by this distance. The value ’0cm’ means the same as ’baseline’.

The remaining values refer to the line box in which the generated box appears:

top
Align the top of the box with the top of the line box.

bottom
Align the bottom of the box with the bottom of the line box.

---

11 Visual effects

Contents

11.1 Overflow and clipping

Generally, the content of a block box is confined to the content edges of the box. In certain cases, a box may *overflow*, meaning its content lies partly or entirely outside of the box, e.g.:

- A line cannot be broken, causing the line box to be wider than the block box.
- A block-level box is too wide for the containing block. This may happen when an element’s ’width’ property has a value that causes the generated block box to spill over sides of the containing block.
- An element’s height exceeds an explicit height assigned to the containing block (i.e., the containing block’s height is determined by the ’height’ property, not by content height).
- A box is positioned absolutely [p. 113] .
- It has negative margins [p. 85] .

Whenever overflow occurs, the ’overflow’ property specifies how (and whether) a box is clipped. The ’clip’ property specifies the size and shape of the clipping region. Specifying a small clipping region may cause clipping of otherwise visible contents.

11.1.1 Overflow: the ’overflow’ property

’overflow’

Value:	visible | hidden | scroll | auto | inherit
Initial:	visible
Applies to:	block-level and replaced elements
Inherited:	no
Percentages:	N/A
Media:	visual

This property specifies whether the content of a block-level element is clipped when it overflows the element’s box (which is acting as a containing block for the content). Values have the following meanings:

visible
: This value indicates that content is not clipped, i.e., it may be rendered outside the block box.

hidden
: This value indicates that the content is clipped and that no scrolling mechanism should be provided to view the content outside the clipping region; users will not have access to clipped content. The size and shape of the clipping region is specified by the ’clip’ property.

scroll
: This value indicates that the content is clipped and that if the user agent uses scrolling mechanism that is visible on the screen (such as a scroll bar or a panner), that mechanism should be displayed for a box whether or not any of its content is clipped. This avoids any problem with scrollbars appearing and disappearing in a dynamic environment. When this value is specified and the target medium is ’print’ or ’projection’, overflowing content should be printed.

auto
: The behavior of the ’auto’ value is user agent-dependent, but should cause a scrolling mechanism to be provided for overflowing boxes.

Even if ’overflow’ is set to ’visible’, content may be clipped to a UA’s document window by the native operating environment.

Example(s):

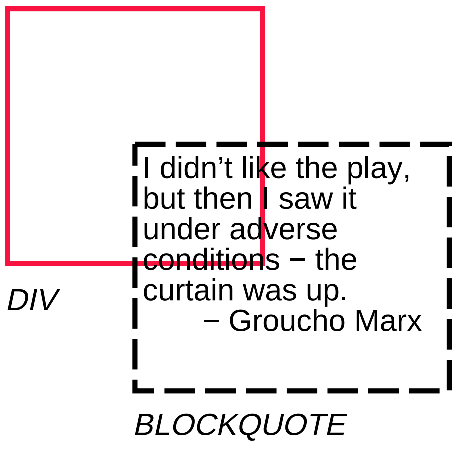
Consider the following example of a block quotation (BLOCKQUOTE) that is too big for its containing block (established by a DIV). Here is the source document:

```
<DIV>
<BLOCKQUOTE>
<P>I didn’t like the play, but then I saw
it under adverse conditions - the curtain was up.
<DIV class="attributed-to">- Groucho Marx</DIV>
</BLOCKQUOTE>
</DIV>
```

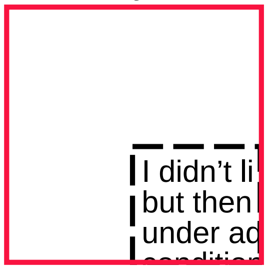
Here is the style sheet controlling the sizes and style of the generated boxes:

```
DIV { width : 100px; height: 100px;
      border: thin solid red;
      }

BLOCKQUOTE   { width : 125px; height : 100px;
      margin-top: 50px; margin-left: 50px;
      border: thin dashed black
      }

DIV.attributed-to { text-align : right; }
```

The initial value of ’overflow’ is ’visible’, so the BLOCKQUOTE would be formatted without clipping, something like this:



Setting 'overflow' to 'hidden' for the DIV element, on the other hand, causes the BLOCKQUOTE to be clipped by the containing block:



A value of 'scroll' would tell UAs that support a visible scrolling mechanism to display one so that users could access the clipped content.

11.1.2 Clipping: the 'clip' property

A *clipping region* defines what portion of an element's rendered content [p. 30] is visible. By default, the clipping region has the same size and shape as the element's box(es). However, the clipping region may be modified by the 'clip' property.

'clip'

Value:	<shape> | auto | inherit
Initial:	auto
Applies to:	block-level and replaced elements
Inherited:	no
Percentages:	N/A
Media:	visual

The 'clip' property applies to elements that have a 'overflow' property with a value other than 'visible'. Values have the following meanings:

auto
The clipping region has the same size and location as the element's box(es).

<shape>
In CSS2, the only valid <shape> value is: rect (<top> <right> <bottom> <left>) where <top>, <bottom> <right>, and <left> specify offsets from the respective sides of the box.

    <top>, <right>, <bottom>, and <left> may either have a <length> value or 'auto'. Negative lengths are permitted. The value 'auto' means that a given edge of the clipping region will be the same as the edge of the element's generated box (i.e., 'auto' means the same as '0'.)

    When coordinates are rounded to pixel coordinates, care should be taken that no pixels remain visible when <left> + <right> is equal to the element's width (or <top> + <bottom> equals the element's height), and conversely that no pixels remain hidden when these values are 0.

The element's ancestors may also have clipping regions (in case their 'overflow' property is not 'visible'); what is rendered is the intersection of the various clipping regions.

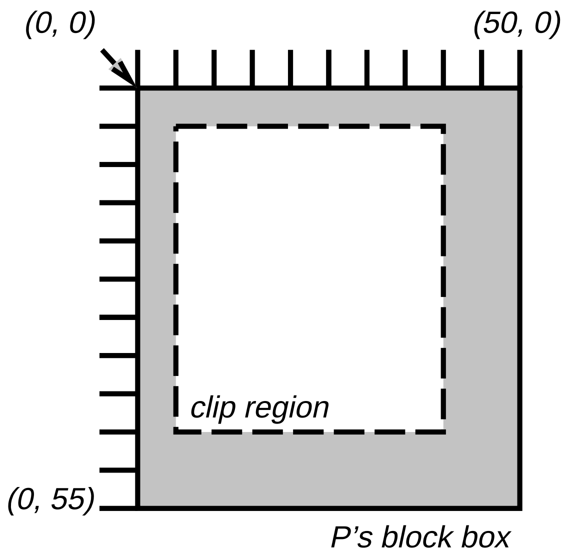
If the clipping region exceeds the bounds of the UA's document window, content may be clipped to that window by the native operating environment.

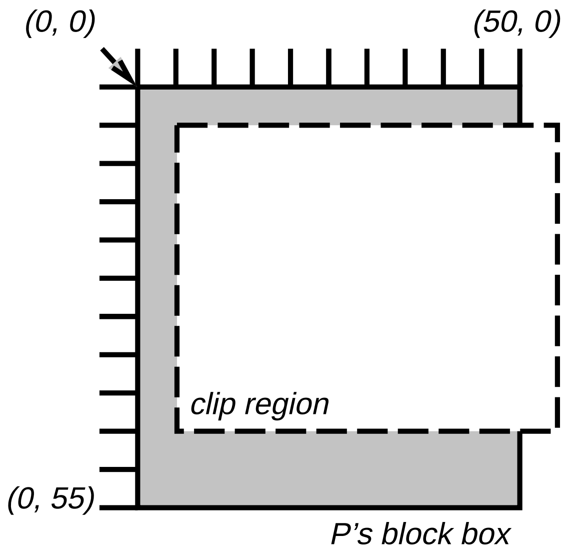
Example(s):

The following two rules:

```
P { clip: rect(5px, 10px, 10px, 5px); }
P { clip: rect(5px, -5px, 10px, 5px); }
```

will create the rectangular clipping regions delimited by the dashed lines in the following illustrations:





Note. *In CSS2, all clipping regions are rectangular. We anticipate future extensions to permit non-rectangular clipping.*

11.2 Visibility: the ’visibility’ property

’visibility’

Value:	visible | hidden | collapse | inherit
Initial:	inherit
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

The ’visibility’ property specifies whether the boxes generated by an element are rendered. Invisible boxes still affect layout (set the ’display’ property to ’none’ to suppress box generation altogether). Values have the following meanings:

visible
  The generated box is visible.
hidden
  The generated box is invisible (fully transparent), but still affects layout.
collapse
  Please consult the section on dynamic row and column effects [p. 261] in tables. If used on elements other than rows or columns, ’collapse’ has the same meaning as ’hidden’.

This property may be used in conjunction with scripts to create dynamic effects.

In the following example, pressing either form button invokes a user-defined script function that causes the corresponding box to become visible and the other to be hidden. Since these boxes have the same size and position, the effect is that one replaces the other. (The script code is in a hypothetical script language. It may or may not have any effect in a CSS-capable UA.)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>
<STYLE type="text/css">
<!--
   #container1 { position: absolute;
                 top: 2in; left: 2in; width: 2in }
   #container2 { position: absolute;
                 top: 2in; left: 2in; width: 2in;
                 visibility: hidden; }
-->
</STYLE>
</HEAD>
<BODY>
<P>Choose a suspect:</P>
<DIV id="container1">
   <IMG alt="Al Capone"
        width="100" height="100"
        src="suspect1.jpg">
   <P>Name: Al Capone</P>
   <P>Residence: Chicago</P>
</DIV>

<DIV id="container2">
   <IMG alt="Lucky Luciano"
        width="100" height="100"
        src="suspect2.jpg">
   <P>Name: Lucky Luciano</P>
   <P>Residence: New York</P>
</DIV>

<FORM method="post"
      action="http://www.suspect.org/process-bums">
   <P>
   <INPUT name="Capone" type="button"
          value="Capone"
          onclick=’show("container1");hide("container2")’>
   <INPUT name="Luciano" type="button"
          value="Luciano"
```

```
            onclick='show("container2");hide("container1")'>
</FORM>
</BODY>
</HTML>
```

12 Generated content, automatic numbering, and lists

Contents

In some cases, authors may want user agents to render content that does not come from the document tree [p. 30] . One familiar example of this is a numbered list; the author does not want to list the numbers explicitly, he or she wants the user agent to generate them automatically. Similarly, authors may want the user agent to insert the word "Figure" before the caption of a figure, or "Chapter 7" before the seventh chapter title. For audio or braille in particular, user agents should be able to insert these strings.

In CSS2, content may be generated by several mechanisms:

- The ’content’ property, in conjunction with the :before and :after pseudo-elements.
- The ’cue-before’, ’cue-after’ aural properties (see the chapter on aural style sheets [p. 277]). When the ’content’ property is combined with the aural properties they are rendered in the following order: :before, ’cue-before’, (’pause-before’), the element’s content, (’pause-after’), ’cue-after’, and :after.
- Elements with a value of ’list-item’ for the ’display’ property.

Below we describe the mechanisms associated with the ’content’ property.

12.1 The :before and :after pseudo-elements

Authors specify the style and location of generated content with the :before and :after pseudo-elements. As their names indicate, the :before and :after pseudo-elements specify the location of content before and after an element’s document tree [p. 30] content. The ’content’ property, in conjunction with these pseudo-elements, specifies what is inserted.

Example(s):

For example, the following rule inserts the string "Note: " before the content of every P element whose "class" attribute has the value "note":

```
P.note:before { content: "Note: " }
```

The formatting objects (e.g., boxes) generated by an element include generated content. So, for example, changing the above style sheet to:

```
P.note:before { content: "Note: " }
P.note        { border: solid green }
```

would cause a solid green border to be rendered around the entire paragraph, including the initial string.

The :before and :after pseudo-elements inherit [p. 70] any inheritable properties from the element in the document tree to which they are attached.

Example(s):

For example, the following rules insert an open quote mark before every Q element. The color of the quote mark will be red, but the font will be the same as the font of the rest of the Q element:

```
Q:before {
  content: open-quote;
  color: red
}
```

In a :before or :after pseudo-element declaration, non-inherited properties take their initial values [p. 69] .

Example(s):

So, for example, because the initial value of the ’display’ property is ’inline’, the quote in the previous example is inserted as an inline box (i.e., on the same line as the element’s initial text content). The next example explicitly sets the ’display’ property to ’block’, so that the inserted text becomes a block:

```
BODY:after {
    content: "The End";
    display: block;
    margin-top: 2em;
    text-align: center;
}
```

Note that an audio user agent would speak the words "The End" after rendering the rest of the BODY content.

User agents must ignore the following properties with :before and :after pseudo-elements: ’position’, ’float’, list [p. 168] properties, and table [p. 245] properties.

The :before and :after pseudo-elements elements allow values of the ’display’ property as follows:

- If the subject [p. 55] of the selector is a block-level [p. 97] element, allowed values are ’none’, ’inline’, ’block’, and ’marker’. If the value of the ’display’ has any other value, the pseudo-element will behave as if the value were ’block’.
- If the subject [p. 55] of the selector is an inline-level [p. 98] element, allowed values are ’none’ and ’inline’. If the value of the ’display’ has any other value,

the pseudo-element will behave as if the value were ’inline’.

Note. *Other values may be permitted in future levels of CSS.*

12.2 The ’content’ property

’content’

Value:	[<string> | <uri> | <counter> | attr(X) | open-quote | close-quote | no-open-quote | no-close-quote]+ | inherit
Initial:	empty string
Applies to:	:before and :after pseudo-elements
Inherited:	no
Percentages:	N/A
Media:	all

This property is used with the :before and :after pseudo-elements to generate content in a document. Values have the following meanings:

<string>
: Text content (see the section on strings [p. 50]).

<uri>
: The value is a URI that designates an external resource. If a user agent cannot support the resource because of the media types [p. 77] it supports, it must ignore the resource. **Note.** CSS2 offers no mechanism to change the size of the embedded object, or to provide a textual description, like the "alt" or "longdesc" attributes do for images in HTML. This may change in future levels of CSS.

<counter>
: Counters [p. 47] may be specified with two different functions: ’counter()’ or ’counters()’. The former has two forms: ’counter(*name*)’ or ’counter(*name*, *style*)’. The generated text is the value of the named counter at this point in the formatting structure; it is formatted in the indicated style [p. 163] (’decimal’ by default). The latter function also has two forms: ’counter(*name*, *string*)’ or ’counter(*name*, *string*, *style*)’. The generated text is the value of all counters with the given name at this point in the formatting structure, separated by the specified string. The counters are rendered in the indicated style [p. 163] (’decimal’ by default). See the section on automatic counters and numbering [p. 160] for more information.

open-quote and **close-quote**
: These values are replaced by the appropriate string from the **’quotes’** property.

no-open-quote and **no-close-quote**
: Inserts nothing (the empty string), but increments (decrements) the level of nesting for quotes.

attr(X)
: This function returns as a string the value of attribute X for the subject of the selector. The string is not parsed by the CSS processor. If the subject of the selector doesn’t have an attribute X, an empty string is returned. The case-sensitivity of attribute names depends on the document language.

Note. In CSS2, it is not possible to refer to attribute values for other elements of the selector.

The ’display’ property controls whether the content is placed in a block, inline, or marker box.

Authors should put ’content’ declarations in @media [p. 78] rules when the content is media-sensitive. For instance, literal text may be used for any media group, but images only apply to the visual + bitmap media groups, and sound files only apply to the aural media group.

Example(s):

The following rule causes a sound file to be played at the end of a quotation (see the section on aural style sheets [p. 277] for additional mechanisms):

```
@media aural {
   BLOCKQUOTE:after { content: url("beautiful-music.wav") }
   }
```

Example(s):

The next rule inserts the text of the HTML "alt" attribute before the image. If the image is not displayed, the reader will still see the "alt" text.

```
IMG:before { content: attr(alt) }
```

Authors may include newlines in the generated content by writing the "\A" escape sequence in one of the strings after the ’content’ property. This inserts a *forced line break*, similar to the BR element in HTML. See "Strings" [p. 50] and "Characters and case" [p. 38] for more information on the "\A" escape sequence.

Example(s):

```
H1:before {
    display: block;
    text-align: center;
    content: "chapter\A hoofdstuk\A chapitre"
}
```

Generated content does not alter the document tree. In particular, it is not fed back to the document language processor (e.g., for reparsing).

Note. *In future levels of CSS, the ’content’ property may accept additional values, allowing it to vary the style of pieces of the generated content, but in CSS2, all the content of the :before or :after pseudo-element has the same style.*

12.3 Interaction of :before and :after with ’compact’ and ’run-in’ elements

The following cases can occur:

1. **A ’run-in’ or ’compact’ element has a :before pseudo-element of type ’inline’:** the pseudo-element is taken into account when the size of the element’s box is computed (for ’compact’) and is rendered inside the same block box as the element.
2. **A ’run-in’ or ’compact’ element has an :after pseudo-element of type ’inline’:** The rules of the previous point apply.
3. **A ’run-in’ or ’compact’ element has a :before pseudo-element of type**

’block’: the pseudo-element is formatted as a block above the element, and does not take part in the computation of the element’s size (for ’compact’).

4. **A ’run-in’ or ’compact’ element has an :after pseudo-element of type ’block’:** both the element and its :after pseudo-element are formatted as block boxes. The element is *not* formatted as an inline box in its own :after pseudo-element.
5. **The element following a ’run-in’ or ’compact’ element has a :before of type ’block’:** the decision how to format the ’run-in’/’compact’ element is made with respect to the block box resulting from the :before pseudo-element.
6. **The element following a ’run-in’ or ’compact’ element has an :before of type ’inline’:** the decision how to format the ’run-in’/’compact’ element depends on the ’display’ value of the element to which the :before is attached.

Example(s):

Here is an example of a ’run-in’ header with an :after pseudo-element, followed by a paragraph with a :before pseudo-element. All pseudo-elements are inline (the default) in this example. When the style sheet:

```
H3 { display: run-in }
H3:after { content: ": " }
P:before { content: "... " }
```

is applied to this source document:

```
<H3>Centaurs</H3>
<P>have hoofs
<P>have a tail
```

The visual formatting will resemble:

```
Centaurs: ... have hoofs
... have a tail
```

12.4 Quotation marks

In CSS2, authors may specify, in a style-sensitive and context-dependent manner, how user agents should render quotation marks. The ’quotes’ property specifies pairs of quotation marks for each level of embedded quotation. The ’content’ property gives access to those quotation marks and causes them to be inserted before and after a quotation.

12.4.1 Specifying quotes with the ’quotes’ property

’quotes’

Value:	[<string> <string>]+ | none | inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies quotation marks for any number of embedded quotations. Values have the following meanings:

none

The ’open-quote’ and ’close-quote’ values of the ’content’ property produce no quotations marks.

[**<string> <string>**]+

Values for the ’open-quote’ and ’close-quote’ values of the ’content’ property are taken from this list of pairs of quotation marks (opening and closing). The first (leftmost) pair represents the outermost level of quotation, the second pair the first level of embedding, etc. The user agent must apply the appropriate pair of quotation marks according to the level of embedding.

Example(s):
For example, applying the following style sheet:

```
/* Specify pairs of quotes for two levels in two languages */
Q:lang(en) { quotes: ’"’ ’"’ "’" "’" }
Q:lang(no) { quotes: "«" "»" "<" ">" }

/* Insert quotes before and after Q element content */
Q:before { content: open-quote }
Q:after  { content: close-quote }
```

to the following HTML fragment:

```
<HTML lang="en">
  <HEAD>
  <TITLE>Quotes</TITLE>
  </HEAD>
  <BODY>
    <P><Q>Quote me!</Q>
  </BODY>
</HTML>
```

would allow a user agent to produce:

```
"Quote me!"
```

while this HTML fragment:

```
<HTML lang="no">
  <HEAD>
  <TITLE>Quotes</TITLE>
  </HEAD>
  <BODY>
    <P><Q>Trøndere gråter når <Q>Vinsjan på kaia</Q> blir deklamert.</Q>
  </BODY>
</HTML>
```

would produce:

```
«Trøndere gråter når <Vinsjan på kaia> blir deklamert.»
```

Note. *While the quotation marks specified by ’quotes’ in the previous examples are conveniently located on computer keyboards, high quality typesetting would require different ISO 10646 characters. The following informative table lists some of the ISO 10646 quotation mark characters:*

Approximate rendering	ISO 10646 code (hex)	Description
"	0022	QUOTATION MARK [the ASCII double quotation mark]
'	0027	APOSTROPHE [the ASCII single quotation mark]
<	2039	SINGLE LEFT-POINTING ANGLE QUOTATION MARK
>	203A	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK
«	00AB	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
»	00BB	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
‘	2018	LEFT SINGLE QUOTATION MARK [single high-6]
’	2019	RIGHT SINGLE QUOTATION MARK [single high-9]
‘‘	201C	LEFT DOUBLE QUOTATION MARK [double high-6]
’’	201D	RIGHT DOUBLE QUOTATION MARK [double high-9]
,,	201E	DOUBLE LOW-9 QUOTATION MARK [double low-9]

12.4.2 Inserting quotes with the ’content’ property

Quotation marks are inserted in appropriate places in a document with the ’open-quote’ and ’close-quote’ values of the ’content’ property. Each occurrence of ’open-quote’ or ’close-quote’ is replaced by one of the strings from the value of ’quotes’, based on the depth of nesting.

’Open-quote’ refers to the first of a pair of quotes, ’close-quote’ refers to the second. Which pair of quotes is used depends on the nesting level of quotes: the number of occurrences of ’open-quote’ in all generated text before the current occurrence, minus the number of occurrences of ’close-quote’. If the depth is 0, the first pair is used, if the depth is 1, the second pair is used, etc. If the depth is greater than the number of pairs, the last pair is repeated.

Note that this quoting depth is independent of the nesting of the source document or the formatting structure.

Some typographic styles require open quotation marks to be repeated before every paragraph of a quote spanning several paragraphs, but only the last paragraph ends with a closing quotation mark. In CSS, this can be achieved by inserting "phantom" closing quotes. The keyword ’no-close-quote’ decrements the quoting level, but does not insert a quotation mark.

Example(s):

The following style sheet puts opening quotation marks on every paragraph in a BLOCKQUOTE, and inserts a single closing quote at the end:

```
BLOCKQUOTE P:before     { content: open-quote }
BLOCKQUOTE P:after      { content: no-close-quote }
BLOCKQUOTE P.last:after { content: close-quote }
```

This relies on the last paragraph being marked with a class "last", since there are no selectors that can match the last child of an element.

For symmetry, there is also a ’no-open-quote’ keyword, which inserts nothing, but increments the quotation depth by one.

Note. *If a quotation is in a different language than the surrounding text, it is customary to quote the text with the quote marks of the language of the surrounding text, not the language of the quotation itself.*

Example(s):

For example, French inside English:

The device of the order of the garter is “Honi soit qui mal y pense.”

English inside French:

Il disait: « Il faut mettre l’action en ‹ fast forward ›.»

A style sheet like the following will set the ’quotes’ property so that ’open-quote’ and ’close-quote’ will work correctly on all elements. These rules are for documents that contain only English, French, or both. One rule is needed for every additional language. Note the use of the child combinator (">") to set quotes on elements based on the language of the surrounding text:

```
[LANG|=fr] > *  { quotes: "«" "»" "\2039" "\203A" }
[LANG|=en] > *  { quotes: "\201C" "\201D" "\2018" "\2019" }
```

The quotation marks for English are shown here in a form that most people will be able to type. If you can type them directly, they will look like this:

```
[LANG|=fr] > * { quotes: "«" "»" "‹" "›" }
[LANG|=en] > * { quotes: "“" "”" "‘" "’" }
```

12.5 Automatic counters and numbering

Automatic numbering in CSS2 is controlled with two properties, ’counter-increment’ and ’counter-reset’. The counters defined by these properties are used with the counter() and counters() functions of the the ’content’ property.

’counter-reset’

Value:	[<identifier> <integer>?]+ | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	all

’counter-increment’

Value:	[<identifier> <integer>?]+ | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	all

The ’counter-increment’ property accepts one or more names of counters (identifiers), each one optionally followed by an integer. The integer indicates by how much the counter is incremented for every occurrence of the element. The default increment is 1. Zero and negative integers are allowed.

The ’counter-reset’ property also contains a list of one or more names of counters, each one optionally followed by an integer. The integer gives the value that the counter is set to on each occurrence of the element. The default is 0.

If ’counter-increment’ refers to a counter that is not in the scope (see below [p. 162]) of any ’counter-reset’, the counter is assumed to have been reset to 0 by the root element.

Example(s):

This example shows a way to number chapters and sections with "Chapter 1", "1.1", "1.2", etc.

```
H1:before {
    content: "Chapter " counter(chapter) ". ";
    counter-increment: chapter;  /* Add 1 to chapter */
    counter-reset: section;      /* Set section to 0 */
}
H2:before {
    content: counter(chapter) "." counter(section) " ";
    counter-increment: section;
}
```

If an element increments/resets a counter and also uses it (in the ’content’ property of its :before or :after pseudo-element), the counter is used *after* being incremented/reset.

If an element both resets and increments a counter, the counter is reset first and then incremented.

The ’counter-reset’ property follows the cascading rules. Thus, due to cascading, the following style sheet:

```
H1 { counter-reset: section -1 }
H1 { counter-reset: imagenum 99 }
```

will only reset ’imagenum’. To reset both counters, they have to be specified together:

```
H1 { counter-reset: section -1 imagenum 99 }
```

12.5.1 Nested counters and scope

Counters are "self-nesting", in the sense that re-using a counter in a child element automatically creates a new instance of the counter. This is important for situations like lists in HTML, where elements can be nested inside themselves to arbitrary depth. It would be impossible to define uniquely named counters for each level.

Example(s):

Thus, the following suffices to number nested list items. The result is very similar to that of setting ’display:list-item’ and ’list-style: inside’ on the LI element:

```
OL { counter-reset: item }
LI { display: block }
LI:before { content: counter(item) ". "; counter-increment: item }
```

The self-nesting is based on the principle that every element that has a ’counter-reset’ for a counter X, creates a fresh counter X, the *scope* of which is the element, its preceding siblings, and all the descendants of the element and its preceding siblings.

In the example above, an OL will create a counter, and all children of the OL will refer to that counter.

If we denote by item[n] the nth instance of the "item" counter, and by "(" and ")" the beginning and end of a scope, then the following HTML fragment will use the indicated counters. (We assume the style sheet as given in the example above).

```
<OL>               <!-- (set item[0] to 0          -->
  <LI>item         <!--  increment item[0] (= 1)   -->
  <LI>item         <!--  increment item[0] (= 2)   -->
    <OL>           <!--  (set item[1] to 0         -->
      <LI>item     <!--   increment item[1] (= 1)  -->
      <LI>item     <!--   increment item[1] (= 2)  -->
      <LI>item     <!--   increment item[1] (= 3)  -->
        <OL>       <!--   (set item[2] to 0        -->
          <LI>item <!--    increment item[2] (= 1) -->
        </OL>      <!--   )                        -->
        <OL>       <!--   (set item[3] to 0        -->
          <LI>     <!--    increment item[3] (= 1) -->
        </OL>      <!--   )                        -->
      <LI>item     <!--   increment item[1] (= 4)  -->
    </OL>          <!--  )                         -->
  <LI>item         <!--  increment item[0] (= 3)   -->
  <LI>item         <!--  increment item[0] (= 4)   -->
</OL>              <!-- )                          -->
<OL>               <!-- (reset item[4] to 0        -->
  <LI>item         <!--  increment item[4] (= 1)   -->
  <LI>item         <!--  increment item[4] (= 2)   -->
</OL>              <!-- )                          -->
```

The ’counters()’ function generates a string composed of the values of all counters with the same name, separated by a given string.

Example(s):

The following style sheet numbers nested list items as "1", "1.1", "1.1.1", etc.

```
OL { counter-reset: item }
LI { display: block }
LI:before { content: counters(item, "."); counter-increment: item }
```

12.5.2 Counter styles

By default, counters are formatted with decimal numbers, but all the styles available for the ’list-style-type’ property are also available for counters. The notation is:

```
counter(name)
```

for the default style, or:

```
counter(name, ’list-style-type’)
```

All the styles are allowed, including ’disc’, ’circle’, ’square’, and ’none’.

Example(s):

```
H1:before        { content: counter(chno, upper-latin) ". " }
H2:before        { content: counter(section, upper-roman) " - " }
BLOCKQUOTE:after { content: " [" counter(bq, hebrew) "]" }
DIV.note:before  { content: counter(notecntr, disc) " " }
P:before         { content: counter(p, none) }
```

12.5.3 Counters in elements with ’display: none’

An element that is not displayed (’display’ set to ’none’) cannot increment or reset a counter.

Example(s):

For example, with the following style sheet, H2s with class "secret" do not increment ’count2’.

```
H2.secret {counter-increment: count2; display: none}
```

Elements with ’visibility’ set to ’hidden’, on the other hand, *do* increment counters.

12.6 Markers and lists

Most block-level elements in CSS generate one principal block box. In this section, we discuss two CSS mechanisms that cause an element to generate two boxes: one principal [p. 97] block box (for the element’s content) and one separate marker box (for decoration such as a bullet, image, or number). The marker box may be positioned inside or outside the principal box. Unlike :before and :after content, the marker box does not affect the position of the principal box, whatever the positioning scheme.

The more general of the two mechanisms is new in CSS2 and is called *markers*. The more limited mechanism involves the list [p. 168] properties of CSS1. The list properties give authors quick results for many common ordered and unordered list scenarios. However, markers give authors precise control over marker content and position. Markers may be used with counters [p. 160] to

create new list styles, to number margin notes, and much more.
For instance, the following example illustrates how markers may be used to add periods after each numbered list item. This HTML program and style sheet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
   <HEAD>
     <TITLE>Creating a list with markers</TITLE>
     <STYLE type="text/css">
          LI:before {
              display: marker;
              content: counter(mycounter, lower-roman) ".";
              counter-increment: mycounter;
          }
     </STYLE>
  </HEAD>
  <BODY>
    <OL>
      <LI> This is the first item.
      <LI> This is the second item.
      <LI> This is the third item.
    </OL>
  </BODY>
</HTML>
```

should produce something like this:

```
   i. This is the first item.
  ii. This is the second item.
 iii. This is the third item.
```

With descendant selectors [p. 56] and child selectors [p. 57] , it’s possible to specify different marker types depending on the depth of embedded lists.

12.6.1 Markers: the ’marker-offset’ property

Markers are created by setting the ’display’ property to ’marker’ inside a :before or :after pseudo-element. While ’block’ and ’inline’ :before and :after content is part of the principal box [p. 97] generated by the element, ’marker’ content is formatted in an independent marker box, outside the principal box. Marker boxes are formatted as a single line (i.e., one line box [p. 105]), so they are not as flexible as floats [p. 95] . The marker box is only created if the ’content’ property for the pseudo-element actually generates content.
Marker boxes have padding and borders, but no margins.
For the :before pseudo-element, the baseline of text in the marker box will be vertically aligned with the baseline of text in the first line of content in the principal box. If the principal box contains no text, the top outer edge of the marker box will be aligned with the top outer edge of the principal box. For the :after pseudo-element, the baseline of text in the marker box will be vertically aligned with the baseline of text in the last line of content in the principal box. If the principal box contains no text, the bottom outer edge of the marker box will be aligned with the bottom outer edge of the principal box.
The height of a marker box is given by the ’line-height’ property. The :before (:after) marker box participates in the height calculation of the principal box’s first (last) line box. Thus, markers are aligned with the first and last line of an element’s content, even though the marker boxes live in distinct line boxes. If no

first or last line box exists in a principal box, the marker box establishes its line box alone.

The vertical alignment of a marker box within its line box is specified with the ’vertical-align’ property.

If the value of the ’width’ property is ’auto’, the marker box content width [p. 82] is that of the content, otherwise it is the value of ’width’. For values of ’width’ less than the content width, the ’overflow’ property specifies overflow behavior. Marker boxes may overlap principal boxes. For values of ’width’ greater than the content width, the ’text-align’ property determines the horizontal alignment of the content in the marker box.

The ’marker-offset’ property specifies the horizontal offset between a marker box and the associated principal box [p. 97] . The distance is measured between their nearest border edges [p. 82] . **Note.** If a marker flows to the right of a float in a left-to-right formatting context, the principal box will flow down the float’s right side, but the marker boxes will appear to the left of the float. Since the principal box left border edge lies to the left of the float (see the description of floats [p. 108]), and marker boxes lie outside the border edge of the principal box, the marker will also lie to the left of the float. Analogous behavior applies for right-to-left formatting when a marker flows to the left of a float.

When the ’display’ property has the value ’marker’ for content generated by an element with ’display: list-item’, a marker box generated for ’:before’ replaces the normal list item marker.

In the following example, the content is centered within a marker box of a fixed width. This document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
 <HTML>
    <HEAD>
      <TITLE>Content alignment in the marker box</TITLE>
      <STYLE type="text/css">
           LI:before {
               display: marker;
               content: "(" counter(counter) ")";
               counter-increment: counter;
               width: 6em;
               text-align: center;
           }
      </STYLE>
   </HEAD>
   <BODY>
     <OL>
       <LI> This is the first item.
       <LI> This is the second item.
       <LI> This is the third item.
     </OL>
   </BODY>
 </HTML>
```

should produce something like this:

```
(1)    This is the
       first item.
(2)    This is the
       second item.
(3)    This is the
       third item.
```

The next example creates markers before and after list items.
This document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Markers before and after list items</TITLE>
    <STYLE type="text/css">
      @media screen, print {
         LI:before {
              display: marker;
              content: url("smiley.gif");
         LI:after {
              display: marker;
              content: url("sad.gif");
         }
      }
    </STYLE>
  </HEAD>
  <BODY>
    <UL>
      <LI>first list item comes first
      <LI>second list item comes second
    </UL>
  </BODY>
</HTML>
```

should produce something like this (ascii art is used instead of smiley gif images here):

```
:-) first list item
    comes first      :-(
:-) second list item
    comes second     :-(
```

The next example uses markers to number notes (paragraphs).
The following document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Markers to create numbered notes4>/TITLE>
    <STYLE type="text/css">
      P { margin-left: 12 em; }
      @media screen, print {
         P.Note:before         {
              display: marker;
              content: url("note.gif")
                       "Note " counter(note-counter) ":";
              counter-increment: note-counter;
              text-align: left;
              width: 10em;
         }
     }
    </STYLE>
  </HEAD>
  <BODY>
    <P>This is the first paragraph in this document.</P>
    <P CLASS="Note">This is a very short document.</P>
    <P>This is the end.</P>
  </BODY>
</HTML>
```

should produce something like:

```
          This is the first paragraph
          in this document.

Note 1:   This is a very short
          document.

          This is the end.
```

’marker-offset’

Value:	<length> | auto | inherit
Initial:	auto
Applies to:	elements with ’display: marker’
Inherited:	no
Percentages:	N/A
Media:	visual

This property specifies the distance between the nearest border edges [p. 82] of a marker box and its associated principal box [p. 97] . The offset may either be a user-specified (<length>) or chosen by the UA (’auto’). Lengths may be negative, but there may be implementation-specific limits.

The following example illustrates how markers may be used to add periods after each numbered list item. This HTML program and style sheet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
 <HTML>
    <HEAD>
      <TITLE>Marker example 5</TITLE>
      <STYLE type="text/css">
           P { margin-left: 8em } /* Make space for counters */
           LI:before {
               display: marker;
               marker-offset: 3em;
               content: counter(mycounter, lower-roman) ".";
               counter-increment: mycounter;
           }
      </STYLE>
   </HEAD>
   <BODY>
     <P> This is a long preceding paragraph ...
     <OL>
       <LI> This is the first item.
       <LI> This is the second item.
       <LI> This is the third item.
     </OL>
     <P> This is a long following paragraph ...
   </BODY>
 </HTML>
```

should produce something like this:

```
       This is a long preceding
       paragraph ...

  i.   This is the first item.
 ii.   This is the second item.
iii.   This is the third item.

       This is a long following
       paragraph ...
```

12.6.2 Lists: the ’list-style-type’, ’list-style-image’, ’list-style-position’, and ’list-style’ properties

The *list properties* allow basic visual formatting of lists. As with more general markers, a element with ’display: list-item’ generates a principal box [p. 97] for the element’s content and an optional marker box. The other list properties allow authors to specify the marker type (image, glyph, or number) and its position with respect to the principal box (outside it or within it before content). They do not allow authors to specify distinct style (colors, fonts, alignment, etc.) for the list marker or adjust its position with respect to the principal box.

Furthermore, when a marker M (created with ’display: marker’) is used with a list item created by the list properties, M replaces the standard list item marker.

With the list properties, the background properties [p. 188] apply to the principal box only; an ’outside’ marker box is transparent. Markers offer more control over marker box style.

’list-style-type’

Value:	disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-alpha | lower-latin | upper-alpha | upper-latin | hebrew | armenian | georgian | cjk-ideographic | hiragana | katakana | hiragana-iroha | katakana-iroha | none | inherit
Initial:	disc
Applies to:	elements with ’display: list-item’
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies appearance of the list item marker if ’list-style-image’ has the value ’none’ or if the image pointed to by the URI cannot be displayed. The value ’none’ specifies no marker, otherwise there are three types of marker: glyphs, numbering systems, and alphabetic systems. **Note.** Numbered lists improve document accessibility by making lists easier to navigate.

Glyphs are specified with **disc**, **circle**, and **square**. Their exact rendering depends on the user agent.

Numbering systems are specified with:

decimal
  Decimal numbers, beginning with 1.
decimal-leading-zero
  Decimal numbers padded by initial zeros (e.g., 01, 02, 03, ..., 98, 99).
lower-roman
  Lowercase roman numerals (i, ii, iii, iv, v, etc.).
upper-roman
  Uppercase roman numerals (I, II, III, IV, V, etc.).
hebrew
  Traditional Hebrew numbering.
georgian
  Traditional Georgian numbering (an, ban, gan, ..., he, tan, in, in-an, ...).
armenian
  Traditional Armenian numbering.
cjk-ideographic
  Plain ideographic numbers
hiragana
  a, i, u, e, o, ka, ki, ...
katakana
  A, I, U, E, O, KA, KI, ...
hiragana-iroha
  i, ro, ha, ni, ho, he, to, ...
katakana-iroha
  I, RO, HA, NI, HO, HE, TO, ...

A user agent that does not recognize a numbering system should use ’decimal’.

Note. *This document does not specify the exact mechanism of each numbering system (e.g., how roman numerals are calculated). A future W3C Note may provide further clarifications.*

Alphabetic systems are specified with:

lower-latin or **lower-alpha**
  Lowercase ascii letters (a, b, c, ... z).
upper-latin or **upper-alpha**
  Uppercase ascii letters (A, B, C, ... Z).
lower-greek
  Lowercase classical Greek alpha, beta, gamma, ... (&#941;, &#942;, &#943;, ...)

This specification does not define how alphabetic systems wrap at the end of the alphabet. For instance, after 26 list items, ’lower-latin’ rendering is undefined. Therefore, for long lists, we recommend that authors specify true numbers.

For example, the following HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
   <HEAD>
     <TITLE>Lowercase latin numbering</TITLE>
     <STYLE type="text/css">
          OL { list-style-type: lower-roman }
     </STYLE>
  </HEAD>
```

```
    <BODY>
      <OL>
        <LI> This is the first item.
        <LI> This is the second item.
        <LI> This is the third item.
      </OL>
    </BODY>
  </HTML>
```

might produce something like this:

```
    i This is the first item.
   ii This is the second item.
  iii This is the third item.
```

Note that the list marker alignment (here, right justified) depends on the user agent.

Note. *Future versions of CSS may provide more complete mechanisms for international numbering styles.*

’list-style-image’

Value:	<uri> | none | inherit
Initial:	none
Applies to:	elements with ’display: list-item’
Inherited:	yes
Percentages:	N/A
Media:	visual

This property sets the image that will be used as the list item marker. When the image is available, it will replace the marker set with the ’list-style-type’ marker.

Example(s):

The following example sets the marker at the beginning of each list item to be the image "ellipse.png".

```
UL { list-style-image: url("http://png.com/ellipse.png") }
```

’list-style-position’

Value:	inside | outside | inherit
Initial:	outside
Applies to:	elements with ’display: list-item’
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies the position of the marker box in the principal block box. Values have the following meanings:

outside

The marker box is outside the principal block box. **Note.** CSS1 did not specify the precise location of the marker box and for reasons of compatibility, CSS2 remains equally ambiguous. For more precise control of marker boxes, please use markers.

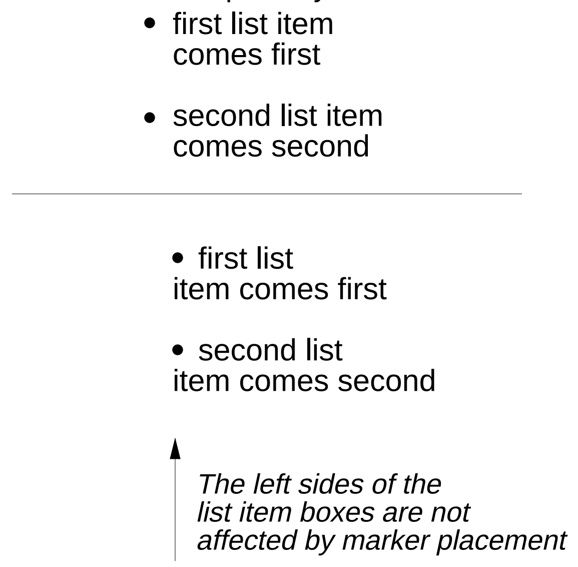
inside

The marker box is the first inline box in the principal block box, after which the element’s content flows.

For example:

```
<HTML>
  <HEAD>
    <TITLE>Comparison of inside/outside position</TITLE>
    <STYLE type="text/css">
      UL         { list-style: outside }
      UL.compact { list-style: inside }
    </STYLE>
  </HEAD>
  <BODY>
    <UL>
      <LI>first list item comes first
      <LI>second list item comes second
    </UL>

    <UL class="compact">
      <LI>first list item comes first
      <LI>second list item comes second
    </UL>
  </BODY>
</HTML>
```

The above example may be formatted as:



first list item comes first
second list item comes second
first list item comes first
second list item comes second
The left sides of the list item boxes are not affected by marker placement

In right-to-left text, the markers would have been on the right side of the box.

’list-style’

Value:	[<’list-style-type’> || <’list-style-position’> || <’list-style-image’>] | inherit
Initial:	not defined for shorthand properties
Applies to:	elements with ’display: list-item’
Inherited:	yes
Percentages:	N/A
Media:	visual

The ’list-style’ property is a shorthand notation for setting the three properties ’list-style-type’, ’list-style-image’, and ’list-style-position’ at the same place in the style sheet.

Example(s):

```
UL { list-style: upper-roman inside }  /* Any UL */
UL > UL { list-style: circle outside } /* Any UL child of a UL */
```

Although authors may specify ’list-style’ information directly on list item elements (e.g., LI in HTML), they should do so with care. The following rules look similar, but the first declares a descendant selector [p. 56] and the second a (more specific) child selector. [p. 57]

```
OL.alpha LI   { list-style: lower-alpha } /* Any LI descendant of an OL */
OL.alpha > LI { list-style: lower-alpha } /* Any LI child of an OL */
```

Authors who use only the descendant selector [p. 56] may not achieve the results they expect. Consider the following rules:

```
<HTML>
  <HEAD>
    <TITLE>WARNING: Unexpected results due to cascade</TITLE>
    <STYLE type="text/css">
      OL.alpha LI  { list-style: lower-alpha }
      UL LI        { list-style: disc }
    </STYLE>
  </HEAD>
  <BODY>
    <OL class="alpha">
      <LI>level 1
      <UL>
         <LI>level 2
      </UL>
    </OL>
  </BODY>
</HTML>
```

The desired rendering would have level 1 list items with ’lower-alpha’ labels and level 2 items with ’disc’ labels. However, the cascading order [p. 73] will cause the first style rule (which includes specific class information) to mask the second. The following rules solve the problem by employing a child selector [p. 57] instead:

```
OL.alpha > LI  { list-style: lower-alpha }
UL LI   { list-style: disc }
```

Another solution would be to specify ’list-style’ information only on the list type elements:

```
OL.alpha  { list-style: lower-alpha }
UL        { list-style: disc }
```

Inheritance will transfer the ’list-style’ values from OL and UL elements to LI elements. This is the recommended way to specify list style information.

Example(s):

A URI value may be combined with any other value, as in:

```
UL { list-style: url("http://png.com/ellipse.png") disc }
```

In the example above, the ’disc’ will be used when the image is unavailable.

A value of ’none’ for the ’list-style’ property sets both ’list-style-type’ and ’list-style-image’ to ’none’:

```
UL { list-style: none }
```

The result is that no list-item marker is displayed.

13 Paged media

Contents

13.1 Introduction to paged media

Paged media (e.g., paper, transparencies, pages that are displayed on computer screens, etc.) differ from continuous media [p. 79] in that the content of the document is split into one or more discrete pages. To handle page breaks, CSS2 extends the visual formatting model [p. 95] as follows:

1. The page box [p. 176] extends the box model [p. 81] to allow authors to specify the size of a page, its margins, etc.
2. The *page model* extends the visual formatting model [p. 95] to account for page breaks. [p. 180]

The CSS2 page model specifies how a document is formatted within a rectangular area -- the page box [p. 176] -- that has a finite width and height. The page box does not necessarily correspond to the real *sheet* where the document will ultimately be rendered (paper, transparency, screen, etc.). The CSS page model specifies formatting in the page box, but it is the user agent's responsibility to transfer the page box to the sheet. Some transfer possibilities include:

- Transferring one page box to one sheet (e.g., single-sided printing).
- Transferring two page boxes to both sides of the same sheet (e.g., double-sided printing).
- Transferring N (small) page boxes to one sheet (called "n-up").

- Transferring one (large) page box to N x M sheets (called "tiling").
- Creating signatures. A signature is a group of pages printed on a sheet, which, when folded and trimmed like a book, appear in their proper sequence.
- Printing one document to several output trays.
- Outputting to a file.

Although CSS2 does not specify how user agents transfer page boxes to sheets, it does include certain mechanisms for telling user agents about the target sheet size and orientation [p. 177] .

13.2 Page boxes: the @page rule

The *page box* is a rectangular region that contains two areas:

- The *page area*. The page area includes the boxes laid out on that page. The edges of the page area act as the initial containing block [p. 96] for layout that occurs between page breaks.
- The margin area, which surrounds the page area.

Note. *In CSS2, the border properties [p. 88] and padding properties [p. 87] do not apply to pages; they may in the future.*

Authors specify the dimensions, orientation, margins, etc. of a page box within an @page rule. An @page rule consists of the keyword "@page", a page selector (followed with no intervening space by an optional page pseudo-class), and a block of declarations (said to be in the *page context*).

The *page selector* specifies for which pages the declarations apply. In CSS2, page selectors may designate the first page, all left pages, all right pages, or a page with a specific name.

The dimensions of the page box are set with the ’size’ property. The dimensions of the page area are the dimensions of the page box minus the margin area.

Example(s):

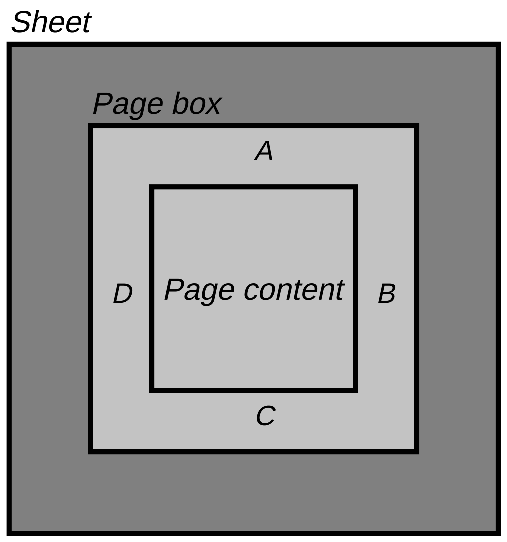
For example, the following @page rule sets the page box size to 8.5 x 11 inches and creates ’2cm’ margin on all sides between the page box edge and the page area:

```
@page { size 8.5in 11in; margin: 2cm }
```

The ’marks’ property in an @page rule specifies crop and cross marks for the page box.

13.2.1 Page margins

The margin properties [p. 85] (’margin-top’, ’margin-right’, ’margin-bottom’, ’margin-left’, and ’margin’) apply within the page context [p. 176] . The following diagram shows the relationships between the sheet, page box, and page margins:



A: margin−top
B: margin−right
C: margin−bottom
D: margin−left

The computed value of box margins at the top or bottom of the page area is ’0’.

The page context [p. 176] has no notion of fonts, so ’em’ and ’ex’ units are not allowed. Percentage values on the margin properties are relative to the dimensions of the page box [p. 176] ; for left and right margins, they refer to page box width while for top and bottom margins, they refer to page box height. All other units associated with the respective CSS2 properties are allowed.

Due to negative margin values (either on the page box or on elements) or absolute positioning [p. 113] content may end up outside the page box, but this content may be "cut" -- by the user agent, the printer, or ultimately, the paper cutter.

13.2.2 Page size: the ’size’ property

’size’

Value:	<length>{1,2} | auto | portrait | landscape | inherit
Initial:	auto
Applies to:	the page context
Inherited:	N/A
Percentages:	N/A
Media:	visual, paged

This property specifies the size and orientation of a page box.

The size of a page box may either be "absolute" (fixed size) or "relative" (scalable, i.e., fitting available sheet sizes). Relative page boxes allow user agents to scale a document and make optimal use of the target size.

Three values for the ’size’ property create a relative page box:

auto
The page box will be set to the size and orientation of the target sheet.
landscape
Overrides the target’s orientation. The page box is the same size as the target, and the longer sides are horizontal.
portrait
Overrides the target’s orientation. The page box is the same size as the target, and the shorter sides are horizontal.

Example(s):
In the following example, the outer edges of the page box will align with the target. The percentage value on the ’margin’ property is relative to the target size so if the target sheet dimensions are 21.0cm x 29.7cm (i.e., A4), the margins are 2.10cm and 2.97cm.

```
@page {
  size: auto;   /* auto is the initial value */
  margin: 10%;
}
```

Length values for the ’size’ property create an absolute page box. If only one length value is specified, it sets both the width and height of the page box (i.e., the box is a square). Since the page box is the initial containing block [p. 96] , percentage values are not allowed for the ’size’ property.
Example(s):
For example:

```
@page {
  size: 8.5in 11in;  /* width height */
}
```

The above example set the width of the page box to be 8.5in and the height to be 11in. The page box in this example requires a target sheet size of 8.5"x11" or larger.
User agents may allow users to control the transfer of the page box to the sheet (e.g., rotating an absolute page box that’s being printed).

Rendering page boxes that do not fit a target sheet

If a page box does not fit the target sheet dimensions, the user agent may choose to:

- Rotate the page box 90° if this will make the page box fit.
- Scale the page to fit the target.

The user agent should consult the user before performing these operations.

Positioning the page box on the sheet

When the page box is smaller than the target size, the user agent is free to place the page box anywhere on the sheet. However, it is recommended that the page box be centered on the sheet since this will align double-sided pages and avoid accidental loss of information that is printed near the edge of the sheet.

13.2.3 Crop marks: the ’marks’ property

’marks’

Value:	[crop || cross] | none | inherit
Initial:	none
Applies to:	page context
Inherited:	N/A
Percentages:	N/A
Media:	visual, paged

In high-quality printing, marks are often added outside the page box. This property specifies whether cross marks or crop marks or both should be rendered just outside the page box [p. 176] edge.

Crop marks indicate where the page should be cut. *Cross marks* (also known as register marks or registration marks) are used to align sheets.

Marks are visible only on absolute page boxes (see the ’size’ property). In relative page boxes, the page box will be aligned with the target and the marks will be outside the printable area.

The size, style, and position of cross marks depend on the user agent.

13.2.4 Left, right, and first pages

When printing double-sided documents, the page boxes [p. 176] on left and right pages should be different. This can be expressed through two CSS pseudo-classes that may be defined in the page context [p. 176] .

All pages are automatically classified by user agents into either the :left or :right pseudo-class.

Example(s):

```
@page :left {
  margin-left: 4cm;
  margin-right: 3cm;
}

@page :right {
  margin-left: 3cm;
  margin-right: 4cm;
}
```

If different declarations have been given for left and right pages, the user agent must honor these declarations even if the user agent does not transfer the page boxes to left and right sheets (e.g., a printer that only prints single-sided).

Authors may also specify style for the first page of a document with the :first pseudo-class:

Example(s):

```
@page { margin: 2cm } /* All margins set to 2cm */

@page :first {
  margin-top: 10cm    /* Top margin on first page 10cm */
}
```

Whether the first page of a document is :left or :right depends on the major writing direction of the document and is outside the scope of this document. However, to force a :left or :right first page, authors may insert a page break before the first generated box (e.g., in HTML, specify this for the BODY element).

Properties specified in a :left (or :right) @page rule override those specified in an @page rule that has no pseudo-class specified. Properties specified in a :first @page rule override those specified in :left (or :right) @page rules.

Note. *Adding declarations to the :left or :right pseudo-class does not influence whether the document comes out of the printer double- or single-sided (which is outside the scope of this specification).*

Note. *Future versions of CSS may include other page pseudo-classes.*

13.2.5 Content outside the page box

When formatting content in the page model, some content may end up outside the page box. For example, an element whose ’white-space’ property has the value ’pre’ may generate a box that is wider than the page box. Also, when boxes are positioned absolutely [p. 113] , they may end up in "inconvenient" locations. For example, images may be placed on the edge of the page box or 100,000 inches below the page box.

A specification for the exact formatting of such elements lies outside the scope of this document. However, we recommend that authors and user agents observe the following general principles concerning content outside the page box:

- Content should be allowed slightly beyond the page box to allow pages to "bleed".
- User agents should avoid generating a large number of empty page boxes to honor the positioning of elements (e.g., you don’t want to print 100 blank pages). Note, however, that generating a small number of empty page boxes may be necessary to honor the ’left’ and ’right’ values for ’page-break-before’ and ’page-break-after’.
- Authors should not position elements in inconvenient locations just to avoid rendering them. Instead:
    - To suppress box generation entirely, set the ’display’ property to ’none’.
    - To make a box invisible, use the ’visibility’ property.
- User agents may handle boxes positioned outside the page box in several ways, including discarding them or creating page boxes for them at the end of the document.

13.3 Page breaks

The following sections explain page formatting in CSS2. Five properties indicate where the user agent may or should break pages, and on what page (left or right) the subsequent content should resume. Each page break ends layout in the current page box [p. 176] and causes remaining pieces of the document tree [p. 30] to be laid out in a new page box.

13.3.1 Break before/after elements: ’page-break-before’, ’page-break-after’, ’page-break-inside’

’page-break-before’

Value:	auto | always | avoid | left | right | inherit
Initial:	auto
Applies to:	block-level elements
Inherited:	no
Percentages:	N/A
Media:	visual, paged

’page-break-after’

Value:	auto | always | avoid | left | right | inherit
Initial:	auto
Applies to:	block-level elements
Inherited:	no
Percentages:	N/A
Media:	visual, paged

’page-break-inside’

Value:	avoid | auto | inherit
Initial:	auto
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual, paged

Values for these properties have the following meanings:

auto
: Neither force nor forbid a page break before (after, inside) the generated box.

always
: Always force a page break before (after) the generated box.

avoid
: Avoid a page break before (after, inside) the generated box.

left
: Force one or two page breaks before (after) the generated box so that the next page is formatted as a left page.

right
: Force one or two page breaks before (after) the generated box so that the next page is formatted as a right page.

A potential page break location is typically under the influence of the parent element’s ’page-break-inside’ property, the ’page-break-after’ property of the preceding element, and the ’page-break-before’ property of the following element. When these properties have values other than ’auto’, the values

’always’, ’left’, and ’right’ take precedence over ’avoid’. See the section on allowed page breaks [p. 183] for the exact rules on how these properties may force or suppress a page break.

13.3.2 Using named pages: ’page’

’page’

Value:	<identifier> | auto
Initial:	auto
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual, paged

The ’page’ property can be used to specify a particular type of page where an element should be displayed.

Example(s):

This example will put all tables on a right-hand side landscape page (named "rotated"):

```
@page rotated {size: landscape}
TABLE {page: rotated; page-break-before: right}
```

The ’page’ property works as follows: If a block box with inline content has a ’page’ property that is different from the preceding block box with inline content, then one or two page breaks are inserted between them, and the boxes after the break are rendered on a page box of the named type. See "Forced page breaks" below. [p. 184]

Example(s):

In this example, the two tables are rendered on landscape pages (indeed, on the same page, if they fit), and the page type "narrow" is not used at all, despite having been set on the DIV:

```
@page narrow {size: 9cm 18cm}
@page rotated {size: landscape}
DIV {page: narrow}
TABLE {page: rotated}
```

with this document::

```
<DIV>
<TABLE>...</TABLE>
<TABLE>...</TABLE>
</DIV>
```

13.3.3 Breaks inside elements: ’orphans’, ’widows’

’orphans’

Value:	<integer> | inherit
Initial:	2
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual, paged

’widows’

Value:	<integer> | inherit
Initial:	2
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual, paged

The ’orphans’ property specifies the minimum number of lines of a paragraph that must be left at the bottom of a page. The ’widows’ property specifies the minimum number of lines of a paragraph that must be left at the top of a page. Examples of how they are used to control page breaks are given below.

For information about paragraph formatting, please consult the section on line boxes [p. 105] .

13.3.4 Allowed page breaks

In the normal flow, page breaks can occur at the following places:

1. In the vertical margin between block boxes. When a page break occurs here, the computed values [p. 70] of the relevant ’margin-top’ and ’margin-bottom’ properties are set to ’0’.
2. Between line boxes [p. 105] inside a block [p. 97] box.

These breaks are subject to the following rules:

- **Rule A:** Breaking at (1) is allowed only if the ’page-break-after’ and ’page-break-before’ properties of all the elements generating boxes that meet at this margin allow it, which is when at least one of them has the value ’always’, ’left’, or ’right’, or when all of them are ’auto’.
- **Rule B:** However, if all of them are ’auto’ and the nearest common ancestor of all the elements has a ’page-break-inside’ value of ’avoid’, then breaking here is not allowed.
- **Rule C:** Breaking at (2) is allowed only if the number of line boxes [p. 105] between the break and the start of the enclosing block box is the value of ’orphans’ or more, and the number of line boxes between the break and the end of the box is the value of ’widows’ or more.
- **Rule D:** In addition, breaking at (2) is allowed only if the ’page-break-inside’ property is ’auto’.

If the above doesn’t provide enough break points to keep content from overflowing the page boxes, then rules B and D are dropped in order to find additional breakpoints.

If that still does not lead to sufficient break points, rules A and C are dropped as well, to find still more break points.

Page breaks cannot occur inside boxes that are absolutely positioned [p. 113] .

13.3.5 Forced page breaks

A page break *must* occur at (1) if, among the ’page-break-after’ and ’page-break-before’ properties of all the elements generating boxes that meet at this margin, there is at least one with the value ’always’, ’left’, or ’right’.

A page break must also occur at (1) if the last line box above this margin and the first one below it do not have the same value for ’page’.

13.3.6 "Best" page breaks

CSS2 does *not* define which of a set of allowed page breaks must be used; CSS2 does not forbid a user agent from breaking at every possible break point, or not to break at all. But CSS2 does recommend that user agents observe the following heuristics (while recognizing that they are sometimes contradictory):

- Break as few times as possible.
- Make all pages that don’t end with a forced break appear to have about the same height.
- Avoid breaking inside a block that has a border.
- Avoid breaking inside a table.
- Avoid breaking inside a floated element

Example(s):

Suppose, for example, that the style sheet contains ’orphans : 4’, ’widows : 2’, and there are 20 lines (line boxes [p. 105]) available at the bottom of the current page:

- If a paragraph at the end of the current page contains 20 lines or fewer, it should be placed on the current page.
- If the paragraph contains 21 or 22 lines, the second part of the paragraph must not violate the ’widows’ constraint, and so the second part must contain exactly two lines
- If the paragraph contains 23 lines or more, the first part should contain 20 lines and the second part the remaining lines.

Now suppose that ’orphans’ is ’10’, ’widows’ is ’20’, and there are 8 lines available at the bottom of the current page:

- If a paragraph at the end of the current page contains 8 lines or fewer, it should be placed on the current page.
- If the paragraph contains 9 lines or more, it cannot be split (that would violate the orphan constraint), so it should move as a block to the next page.

13.4 Cascading in the page context

Declarations in the page context [p. 176] obey the cascade [p. 69] just like normal CSS2 declarations.

Example(s):

Consider the following example:

```
@page {
  margin-left: 3cm;
}

@page :left {
  margin-left: 4cm;
}
```

Due to the higher specificity [p. 73] of the pseudo-class selector, the left margin on left pages will be ’4cm’ and all other pages (i.e., the right pages) will have a left margin of ’3cm’.

14 Colors and Backgrounds

Contents

CSS properties allow authors to specify the foreground color and background of an element. Backgrounds may be colors or images. Background properties allow authors to position a background image, repeat it, and declare whether it should be fixed with respect to the viewport [p. 96] or scrolled along with the document.

See the section on color units [p. 48] for the syntax of valid color values.

14.1 Foreground color: the ’color’ property

’color’

Value:	<color> | inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property describes the foreground color of an element’s text content. There are different ways to specify red:

Example(s):

```
EM { color: red }              /* predefined color name */
EM { color: rgb(255,0,0) }     /* RGB range 0-255   */
```

14.2 The background

Authors may specify the background of an element (i.e., its rendering surface) as either a color or an image. In terms of the box model [p. 81] , "background" refers to the background of the content [p. 81] and the padding [p. 81] areas. Border colors and styles are set with the border properties [p. 88] . Margins are always transparent so the background of the parent box always shines through.

Background properties are not inherited, but the parent box’s background will shine through by default because of the initial ’transparent’ value on ’background-color’.

The background of the box generated by the root element covers the entire canvas [p. 26] .

For HTML documents, however, we recommend that authors specify the background for the BODY element rather than the HTML element. User agents should observe the following precedence rules to fill in the background: if the value of the ’background’ property for the HTML element is different from ’transparent’ then use it, else use the value of the ’background’ property for the BODY element. If the resulting value is ’transparent’, the rendering is undefined.

According to these rules, the canvas underlying the following HTML document will have a "marble" background:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Setting the canvas background</TITLE>
    <STYLE type="text/css">
       BODY { background: url("http://style.com/marble.png") }
    </STYLE>
  </HEAD>
  <BODY>
    <P>My background is marble.
  </BODY>
</HTML>
```

14.2.1 Background properties: ’background-color’, ’background-image’, ’background-repeat’, ’background-attachment’, ’background-position’, and ’background’

’background-color’

Value:	<color> | transparent | inherit
Initial:	transparent
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

This property sets the background color of an element, either a <color> value or the keyword ’transparent’, to make the underlying colors shine through.

Example(s):

```
H1 { background-color: #F00 }
```

’background-image’

Value:	<uri> | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

This property sets the background image of an element. When setting a background image, authors should also specify a background color that will be used when the image is unavailable. When the image is available, it is rendered on top of the background color. (Thus, the color is visible in the transparent parts of the image).

Values for this property are either <uri>, to specify the image, or ’none’, when no image is used.

Example(s):

```
BODY { background-image: url("marble.gif") }
P { background-image: none }
```

’background-repeat’

Value:	repeat | repeat-x | repeat-y | no-repeat | inherit
Initial:	repeat
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

If a background image is specified, this property specifies whether the image is repeated (tiled), and how. All tiling covers the content [p. 81] and padding [p. 81] areas of a box. Values have the following meanings:

repeat
The image is repeated both horizontally and vertically.

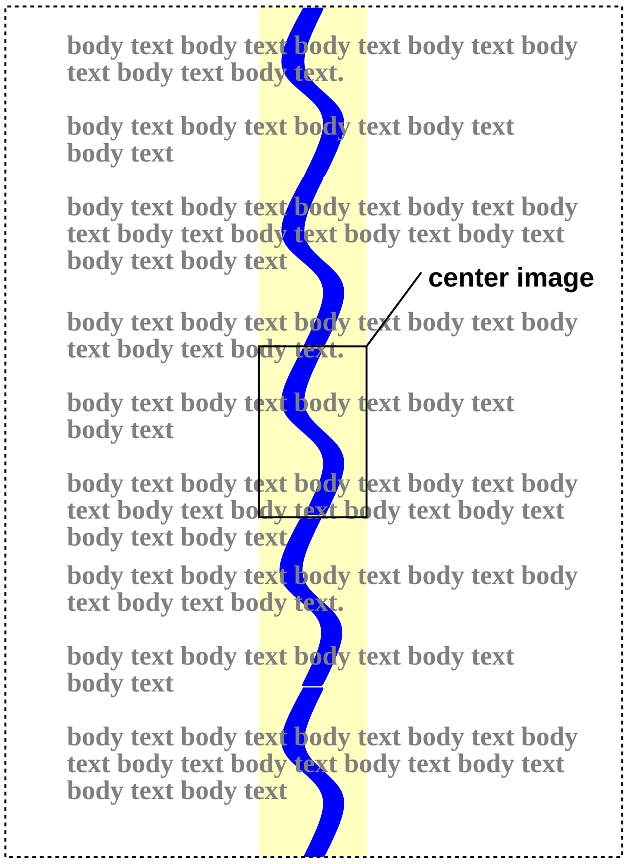
repeat-x
The image is repeated horizontally only.

repeat-y
The image is repeated vertically only.

no-repeat
The image is not repeated: only one copy of the image is drawn.

Example(s):

```
BODY {
  background: white url("pendant.gif");
  background-repeat: repeat-y;
  background-position: center;
}
```


body text body text body text body text body text body text body text.
body text body text body text body text body text
body text body text body text body text body text body text body text body text body text body text body text
center image
body text body text body text body text body text body text body text.
body text body text body text body text body text
body text body text body text body text body text body text body text body text body text body text body text
body text body text body text body text body text body text body text.
body text body text body text body text body text
body text body text body text body text body text body text body text body text body text body text body text

One copy of the background image is centered, and other copies are put above and below it to make a vertical band behind the element.

’background-attachment’

Value:	scroll | fixed | inherit
Initial:	scroll
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

If a background image is specified, this property specifies whether it is fixed with regard to the viewport [p. 96] (’fixed’) or scrolls along with the document (’scroll’).

Even if the image is fixed, it is still only visible when it is in the background or padding area of the element. Thus, unless the image is tiled (’background-repeat: repeat’), it may be invisible.

Example(s):

This example creates an infinite vertical band that remains "glued" to the viewport when the element is scrolled.

```
BODY {
  background: red url("pendant.gif");
  background-repeat: repeat-y;
  background-attachment: fixed;
}
```

User agents may treat ’fixed’ as ’scroll’. However, it is recommended they interpret ’fixed’ correctly, at least for the HTML and BODY elements, since there is no way for an author to provide an image only for those browsers that support ’fixed’. See the section on conformance [p. 32] for details.

’background-position’

Value:	[[<percentage> | <length>]{1,2} | [[top | center | bottom] || [left | center | right]]] | inherit
Initial:	0% 0%
Applies to:	block-level and replaced elements
Inherited:	no
Percentages:	refer to the size of the box itself
Media:	visual

If a background image has been specified, this property specifies its initial position. Values have the following meanings:

<percentage> <percentage>
With a value pair of ’0% 0%’, the upper left corner of the image is aligned with the upper left corner of the box’s padding edge [p. 82] . A value pair of ’100% 100%’ places the lower right corner of the image in the lower right corner of padding area. With a value pair of ’14% 84%’, the point 14% across and 84% down the image is to be placed at the point 14% across and 84% down the padding area.

<length> <length>
With a value pair of ’2cm 2cm’, the upper left corner of the image is placed 2cm to the right and 2cm below the upper left corner of the padding area.

top left and **left top**
Same as ’0% 0%’.

top, **top center**, and **center top**
Same as ’50% 0%’.

right top and **top right**
Same as ’100% 0%’.

left, **left center**, and **center left**
Same as ’0% 50%’.

center and **center center**
Same as ’50% 50%’.

right, **right center**, and **center right**
Same as ’100% 50%’.

bottom left and **left bottom**
Same as ’0% 100%’.

bottom, **bottom center**, and **center bottom**
Same as ’50% 100%’.

bottom right and **right bottom**
Same as ’100% 100%’.

If only one percentage or length value is given, it sets the horizontal position only, the vertical position will be 50%. If two values are given, the horizontal position comes first. Combinations of length and percentage values are allowed, (e.g., ’50% 2cm’). Negative positions are allowed. Keywords cannot be combined with percentage values or length values (all possible combinations are given above).

Example(s):

```
BODY { background: url("banner.jpeg") right top }    /* 100%   0% */
BODY { background: url("banner.jpeg") top center }   /*  50%   0% */
BODY { background: url("banner.jpeg") center }       /*  50%  50% */
BODY { background: url("banner.jpeg") bottom }       /*  50% 100% */
```

If the background image is fixed within the viewport (see the ’background-attachment’ property), the image is placed relative to the viewport instead of the element’s padding area. For example,

Example(s):

```
BODY {
  background-image: url("logo.png");
  background-attachment: fixed;
  background-position: 100% 100%;
  background-repeat: no-repeat;
}
```

In the example above, the (single) image is placed in the lower-right corner of the viewport.

’background’

Value:	[<’background-color’> || <’background-image’> || <’background-repeat’> || <’background-attachment’> || <’background-position’>] | inherit
Initial:	not defined for shorthand properties
Applies to:	all elements
Inherited:	no
Percentages:	allowed on ’background-position’
Media:	visual

The ’background’ property is a shorthand property for setting the individual background properties (i.e., ’background-color’, ’background-image’, ’background-repeat’, ’background-attachment’ and ’background-position’) at the same place in the style sheet.

The ’background’ property first sets all the individual background properties to their initial values, then assigns explicit values given in the declaration.

Example(s):

In the first rule of the following example, only a value for ’background-color’ has been given and the other individual properties are set to their initial value. In the second rule, all individual properties have been specified.

```
BODY { background: red }
P { background: url("chess.png") gray 50% repeat fixed }
```

14.3 Gamma correction

For information about gamma issues, please consult the the Gamma Tutorial in the PNG specification ([PNG10]).

In the computation of gamma correction, UAs displaying on a CRT may assume an ideal CRT and ignore any effects on apparent gamma caused by dithering. That means the minimal handling they need to do on current platforms is:

PC using MS-Windows
: none

Unix using X11
: none

Mac using QuickDraw
: apply gamma 1.45 [ICC32] (ColorSync-savvy applications may simply pass the sRGB ICC profile to ColorSync to perform correct color correction)

SGI using X
: apply the gamma value from `/etc/config/system.glGammaVal` (the default value being 1.70; applications running on Irix 6.2 or above may simply pass the sRGB ICC profile to the color management system)

NeXT using NeXTStep
: apply gamma 2.22

"Applying gamma" means that each of the three R, G and B must be converted to $R'=R^{gamma}$, $G'=G^{gamma}$, $B'=B^{gamma}$, before being handed to the OS.

This may rapidly be done by building a 256-element lookup table once per browser invocation thus:

```
for i := 0 to 255 do
  raw := i / 255.0;
  corr := pow (raw, gamma);
  table[i] := trunc (0.5 + corr * 255.0)
end
```

which then avoids any need to do transcendental math per color attribute, far less per pixel.

15 Fonts

Contents

15.1 Introduction

When a document’s text is to be displayed visually, characters (abstract information elements) must be mapped to *abstract glyphs*. One or more characters may be depicted by one or more abstract glyphs, in a possibly context-dependent fashion. A *glyph* is the actual artistic representation of an abstract glyph, in some typographic style, in the form of outlines or bitmaps that may be drawn on the screen or paper. A *font* is a set of glyphs, all observing the same basic motif according to design, size, appearance, and other attributes associated with the entire set, and a mapping from characters to abstract glyphs.

A visual user agent must address the following issues before actually rendering a character:

- Is there, directly or by inheritance, a font specified for this character?
- Does the user agent have this font available?
- If so, what glyph(s) does this character or sequence of characters map to?
- If not, what should be done? Should a different font be substituted? Can the font be synthesized? Can it be retrieved from the Web?

In both CSS1 and CSS2, authors specify font characteristics via a series of font properties.

How the user agent handles these properties, when there is no matching font on the client has expanded between CSS1 and CSS2. In CSS1, all fonts were assumed to be present on the client system and were identified solely by name. Alternate fonts could be specified through the properties, but beyond that, user agents had no way to propose other fonts to the user (even stylistically similar fonts that the user agent had available) other than generic default fonts.

CSS2 changes all that, and allows much greater liberty for:

- style sheet authors, to describe the fonts they want to be used
- user agents, in selecting a font when an author’s requested font is not immediately available.

CSS2 improves client-side font matching, enables font synthesis and progressive rendering, and enables fonts to be downloaded over the Web. These enhanced capabilities are referred to as ’WebFonts’

In the CSS2 font model, as in CSS1, each user agent has a "font database" at its disposition. CSS1 referred to this database but gave no details about what was in it. CSS2 defines the information in that database and allows style sheet authors to contribute to it. When asked to display a character with a particular font, the user agent first identifies the font in the database that "best fits" the specified font (according to the font matching algorithm) [p. 231] Once it has identified a font, it retrieves the font data locally or from the Web, and may display the character using those glyphs.

In light of this model, we have organized the specification into two sections. The first concerns the font specification mechanism [p. 197] , whereby authors specify which fonts they would like to have used. The second concerns the font selection mechanism [p. 212] , whereby the client's user agent identifies and loads a font that best fits the author's specification.

How the user agent constructs the font database lies outside the scope of this specification since the database's implementation depends on such factors as the operating system, the windowing system, and the client.

15.2 Font specification

The first phase of the CSS font mechanism concerns how style sheet authors specify which fonts should be used by a user agent. At first, it seem that the obvious way to specify a font is by it's name, a single string - which appears to be separated into distinct parts; for example "BT Swiss 721 Heavy Italic".

Unfortunately, there exists no well-defined and universally accepted taxonomy for classifying fonts based on their names, and terms that apply to one font family name may not be appropriate for others. For example, the term 'italic' is commonly used to label slanted text, but slanted text may also be labeled *Oblique, Slanted, Incline, Cursive*, or *Kursiv*. Similarly, font names typically contain terms that describe the "weight" of a font. The primary role of these names is to distinguish faces of differing darkness within a single font family. There is no accepted, universal meaning to these weight names and usage varies widely. For example a font that you might think of as being bold might be described as being *Regular, Roman, Book, Medium, Semi-* or *Demi-Bold, Bold,* or *Black,* depending on how black the "normal" face of the font is within the design.

This lack of systematic naming makes it impossible, in the general case, to generate a modified font face name that differs in a particular way, such as being bolder.

Because of this, CSS uses a different model [p. 198] . Fonts are requested not through a single font name but through setting a series of font properties. These property values form the basis of the user agent's font selection [p. 212] mechanism. The font properties can be individually modified, for example to increase the boldness, and the new set of font property values will then be used to select from the font database again. The result is an increase in regularity for style sheet authors and implementors, and an increase in robustness.

15.2.1 Font specification properties

CSS2 specifies fonts according to these characteristics:

Font family
: The font family specifies which font family is to be used to render the text. A font family is a group of fonts,designed to be used in combination and exhibiting similarities in design. One member of the family may be italic, another bold, another condensed or using small caps. Font family names include "Helvetica", "New Century Schoolbook", and "Kyokasho ICA L". Font family names are not restricted to Latin characters. Font families may be grouped into different categories: those with or without serifs, those whose characters are or are not proportionally spaced, those that resemble hand-writing, those that are fantasy fonts, etc.

Font style
: The font style specifies whether the text is to be rendered using a normal, italic, or oblique face. *Italic* is a more cursive companion face to the normal face, but not so cursive as to make it a script face. Oblique is a slanted form of the normal face, and is more commonly used as a companion face to sans-serif. This definition avoids having to label slightly slanted normal faces as oblique, or normal Greek faces as italic.

Font variant
: The font variant indicates whether the text is to be rendered using the normal glyphs for lowercase characters or using small-caps glyphs for lower-case characters. A particular font may contain only normal, only small-caps, or both types of glyph; this property is used to request an appropriate font and, if the font contains both variants, the appropriate glyphs.

Font weight
: The font weight refers to the boldness or lightness of the glyphs used to render the text, relative to other fonts in the same font family.

Font stretch
: The font stretch indicates the desired amount of condensing or expansion in the glyphs used to render the text, relative to other fonts in the same font family.

Font size
: The font size refers to the size of the font from baseline to baseline, when set solid (in CSS terms, this is when the ’font-size’ and ’line-height’ proper-ties have the same value).

On all properties except ’font-size’, ’em’ and ’ex’ length values refer to the font size of the current element. For ’font-size’, these length units refer to the font size of the parent element. Please consult the section on length units [p. 43] for more information.

The CSS font properties are used to describe the desired appearance of text in the document. The font descriptors, in contrast, are used to describe the charac-teristics of fonts, so that a suitable font can be chosen to create the desired appearance. For information about the classification of fonts, please consult the section on font descriptors [p. 226] .

15.2.2 Font family: the ’font-family’ property

’font-family’

Value:	[[<family-name> | <generic-family>],]* [<family-name> | <generic-family>] | inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies a prioritized list of font family names and/or generic family names. To deal with the problem that a single font may not contain glyphs to display all the characters in a document, or that not all fonts are available on all systems, this property allows authors to specify a list of fonts, all of the same style and size, that are tried in sequence to see if they contain a glyph for a certain character. This list is called a *font set*.

Example(s):

For example, text that contains English words mixed with mathematical symbols may need a font set of two fonts, one containing Latin letters and digits, the other containing mathematical symbols. Here is an example of a font set suitable for a text that is expected to contain text with Latin characters, Japanese characters, and mathematical symbols:

```
BODY { font-family: Baskerville, "Heisi Mincho W3", Symbol, serif }
```

The glyphs available in the "Baskerville" font (a font that covers only Latin characters) will be taken from that font, Japanese glyphs will be taken from "Heisi Mincho W3", and the mathematical symbol glyphs will come from "Symbol". Any others will come from the generic font family ’serif’.

The generic font family will be used if one or more of the other fonts in a font set is unavailable. Although many fonts provide the "missing character" glyph, typically an open box, as its name implies this should not be considered a match except for the last font in a font set.

There are two types of font family names:

<family-name>
The name of a font family of choice. In the previous example, "Baskerville", "Heisi Mincho W3", and "Symbol" are font families. Font family names containing whitespace [p. 37] should be quoted. If quoting is omitted, any whitespace [p. 37] characters before and after the font name are ignored [p. 42] and any sequence of whitespace characters inside the font name is converted to a single space.

<generic-family>
The following generic families are defined: ’serif’, ’sans-serif’, ’cursive’, ’fantasy’, and ’monospace’. Please see the section on generic font families [p. 209] for descriptions of these families. Generic font family names are keywords, and therefore must not be quoted.

Authors are encouraged to offer a generic font family as a last alternative, for improved robustness.

For example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Font test</TITLE>
    <STYLE type="text/css">
      BODY { font-family: "new century schoolbook", serif }
    </STYLE>
  </HEAD>
  <BODY>
   <H1 style="font-family: ’My own font’, fantasy">Test</H1>
    <P>What’s up, Doc?
  </BODY>
</HTML>
```

Example(s):
The richer selector syntax of CSS2 may be used to create language-sensitive typography. For example, some Chinese and Japanese characters are unified to have the same Unicode codepoint, although the abstract glyphs are not the same in the two languages.

```
*:lang(ja-jp) { font: 900 14pt/16pt "Heisei Mincho W9", serif }
*:lang(zh-tw) { font: 800 14pt/16.5pt "Li Sung", serif }
```

This selects any element that has the given language - Japanese or Traditional Chinese - and requests the appropriate font.

15.2.3 Font styling: the ’font-style’, ’font-variant’, ’font-weight’ and ’font-stretch’ properties

’font-style’

Value:	normal | italic | oblique | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

The ’font-style’ property requests normal (sometimes referred to as "roman" or "upright"), italic, and oblique faces within a font family. Values have the following meanings:

normal
Specifies a font that is classified as ’normal’ in the UA’s font database.
oblique
Specifies a font that is classified as ’oblique’ in the UA’s font database. Fonts with Oblique, Slanted, or Incline in their names will typically be labeled ’oblique’ in the font database. A font that is labeled ’oblique’ in the UA’s font database may actually have been generated by electronically slanting a

normal font.

italic

Specifies a font that is classified as ’italic’ in the UA’s font database, or, if that is not available, one labeled ’oblique’. Fonts with *Italic, Cursive*, or *Kursiv* in their names will typically be labeled ’italic’.

Example(s):

In this example, normal text in an H1, H2, or H3 element will be displayed with an italic font. However, emphasized text (EM) within an H1 will appear in a normal face.

```
H1, H2, H3 { font-style: italic }
H1 EM { font-style: normal }
```

’font-variant’

Value:	normal | small-caps | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

In a small-caps font, the glyphs for lowercase letters look similar to the uppercase ones, but in a smaller size and with slightly different proportions. The ’font-variant’ property requests such a font for bicameral (having two cases, as with Latin script). This property has no visible effect for scripts that are *unicameral* (having only one case, as with most of the world’s writing systems). Values have the following meanings:

normal

Specifies a font that is not labeled as a small-caps font.

small-caps

Specifies a font that is labeled as a small-caps font. If a genuine small-caps font is not available, user agents should simulate a small-caps font, for example by taking a normal font and replacing the lowercase letters by scaled uppercase characters. As a last resort, unscaled uppercase letter glyphs in a normal font may replace glyphs in a small-caps font so that the text appears in all uppercase letters.

Example(s):

The following example results in an H3 element in small-caps, with emphasized words (EM) in oblique small-caps:

```
H3 { font-variant: small-caps }
EM { font-style: oblique }
```

Insofar as this property causes text to be transformed to uppercase, the same considerations as for ’text-transform’ apply.

’font-weight’

Value:	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

The ’font-weight’ property specifies the weight of the font. Values have the following meanings:

100 to 900
These values form an ordered sequence, where each number indicates a weight that is at least as dark as its predecessor.

normal
Same as ’400’.

bold
Same as ’700’.

bolder
Specifies the next weight that is assigned to a font that is darker than the inherited one. If there is no such weight, it simply results in the next darker numerical value (and the font remains unchanged), unless the inherited value was ’900’, in which case the resulting weight is also ’900’.

lighter
Specifies the next weight that is assigned to a font that is lighter than the inherited one. If there is no such weight, it simply results in the next lighter numerical value (and the font remains unchanged), unless the inherited value was ’100’, in which case the resulting weight is also ’100’.

Example(s):

```
P { font-weight: normal }   /* 400 */
H1 { font-weight: 700 }     /* bold */
BODY { font-weight: 400 }
STRONG { font-weight: bolder } /* 500 if available */
```

Child elements inherit the computed value [p. 70] of the weight.

’font-stretch’

Value:	normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

The ’font-stretch’ property selects a normal, condensed, or extended face from a font family. Absolute keyword values have the following ordering, from narrowest to widest :

1. ultra-condensed
2. extra-condensed
3. condensed
4. semi-condensed
5. normal
6. semi-expanded
7. expanded
8. extra-expanded
9. ultra-expanded

The relative keyword ’wider’ sets the value to the next expanded value above the inherited value (while not increasing it above ’ultra-expanded’); the relative keyword ’narrower’ sets the value to the next condensed value below the inherited value (while not decreasing it below ’ultra-condensed’).

15.2.4 Font size: the ’font-size’ and ’font-size-adjust’ properties

’font-size’

Value:	<absolute-size> | <relative-size> | <length> | <percentage> | inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes, the computed value is inherited
Percentages:	refer to parent element’s font size
Media:	visual

This property describes the size of the font when set solid. Values have the following meanings:

<absolute-size>

An <absolute-size> keyword refers to an entry in a table of font sizes computed and kept by the user agent. Possible values are:

[xx-small | x-small | small | medium | large | x-large | xx-large]

On a computer screen a scaling factor of 1.2 is suggested between adjacent indexes; if the ’medium’ font is 12pt, the ’large’ font could be 14.4pt. Different media may need different scaling factors. Also, the user agent should take the quality and availability of fonts into account when computing the table. The table may be different from one font family to another.

Note. *In CSS1, the suggested scaling factor between adjacent indexes was 1.5 which user experience proved to be too large.*

<relative-size>

A <relative-size> keyword is interpreted relative to the table of font sizes and the font size of the parent element. Possible values are:

[larger | smaller]

For example, if the parent element has a font size of ’medium’, a value of ’larger’ will make the font size of the current element be ’large’. If the parent element’s size is not close to a table entry, the user agent is free to interpolate between table entries or round off to the closest one. The user agent may have to extrapolate table values if the numerical value goes beyond the keywords.

<length>

A length value specifies an absolute font size (that is independent of the user agent's font table). Negative lengths are illegal.

<percentage>

A percentage value specifies an absolute font size relative to the parent element's font size. Use of percentage values, or values in 'em's, leads to more robust and cascadable style sheets.

The actual value [p. 70] of this property may differ from the computed value [p. 70] due a numerical value on 'font-size-adjust' and the unavailability of certain font sizes.

Child elements inherit the computed 'font-size' value (otherwise, the effect of 'font-size-adjust' would compound).

Example(s):

```
P { font-size: 12pt; }
BLOCKQUOTE { font-size: larger }
EM { font-size: 150% }
EM { font-size: 1.5em }
```

'font-size-adjust'

Value:	<number> | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

In bicameral scripts, the subjective apparent size and legibility of a font are less dependent on their 'font-size' value than on the value of their 'x-height', or, more usefully, on the ratio of these two values, called the *aspect value* (font size divided by x-height). The higher the aspect value, the more likely it is that a font at smaller sizes will be legible. Inversely, faces with a lower aspect value will become illegible more rapidly below a given threshold size than faces with a higher aspect value. Straightforward font substitution that relies on font size alone may lead to illegible characters.

For example, the popular font Verdana has an aspect value of 0.58; when Verdana's font size 100 units, its x-height is 58 units. For comparison, Times New Roman has an aspect value of 0.46. Verdana will therefore tend to remain legible at smaller sizes than Times New Roman. Conversely, Verdana will often look 'too big' if substituted for Times New Roman at a chosen size.

This property allows authors to specify an aspect value for an element that will preserve the x-height of the first choice font in the substitute font. Values have the following meanings:

none

Do not preserve the font's x-height.

<number>

Specifies the aspect value. The number refers to the aspect value of the first choice font. The scaling factor for available fonts is computed according to the following formula:

```
y(a/a') = c
```

where:

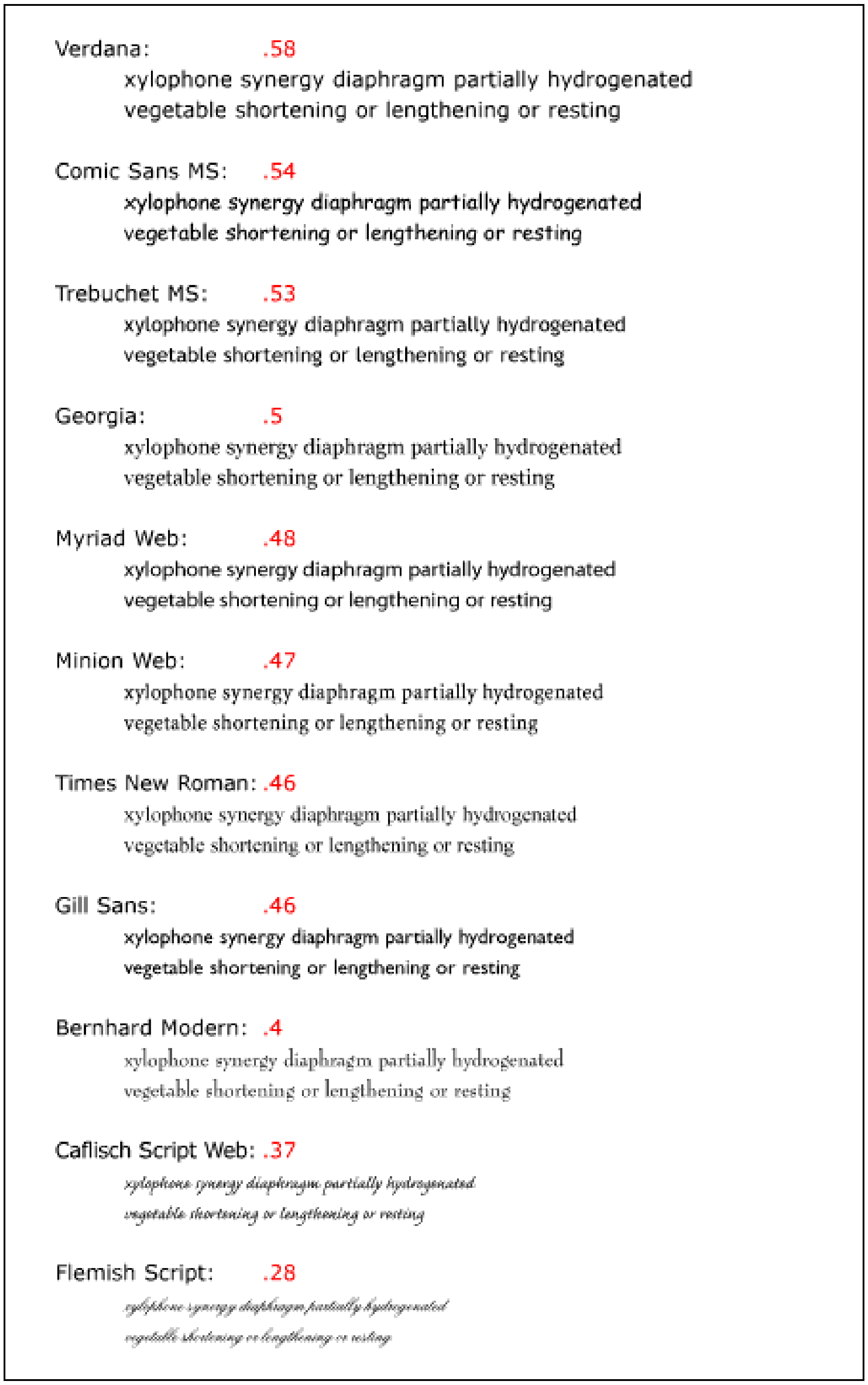
```
y = 'font-size' of first-choice font
a' = aspect value of available font
c = 'font-size' to apply to available font
```

Example(s):

For example, if 14px Verdana (with an aspect value of 0.58) was unavailable and an available font had an aspect value of 0.46, the font-size of the substitute would be 14 * (0.58/0.46) = 17.65px.

Font size adjustments take place when computing the actual value [p. 70] of 'font-size'. Since inheritance is based on the computed value [p. 70] , child elements will inherit unadjusted values.

The first image below shows several typefaces rasterized at a common font size (11pt. at 72 ppi), together with their aspect values. Note that faces with higher aspect values appear larger than those with lower. Faces with very low aspect values are illegible at the size shown.



Verdana: .58
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Comic Sans MS: .54
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Trebuchet MS: .53
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Georgia: .5
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Myriad Web: .48
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Minion Web: .47
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Times New Roman: .46
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

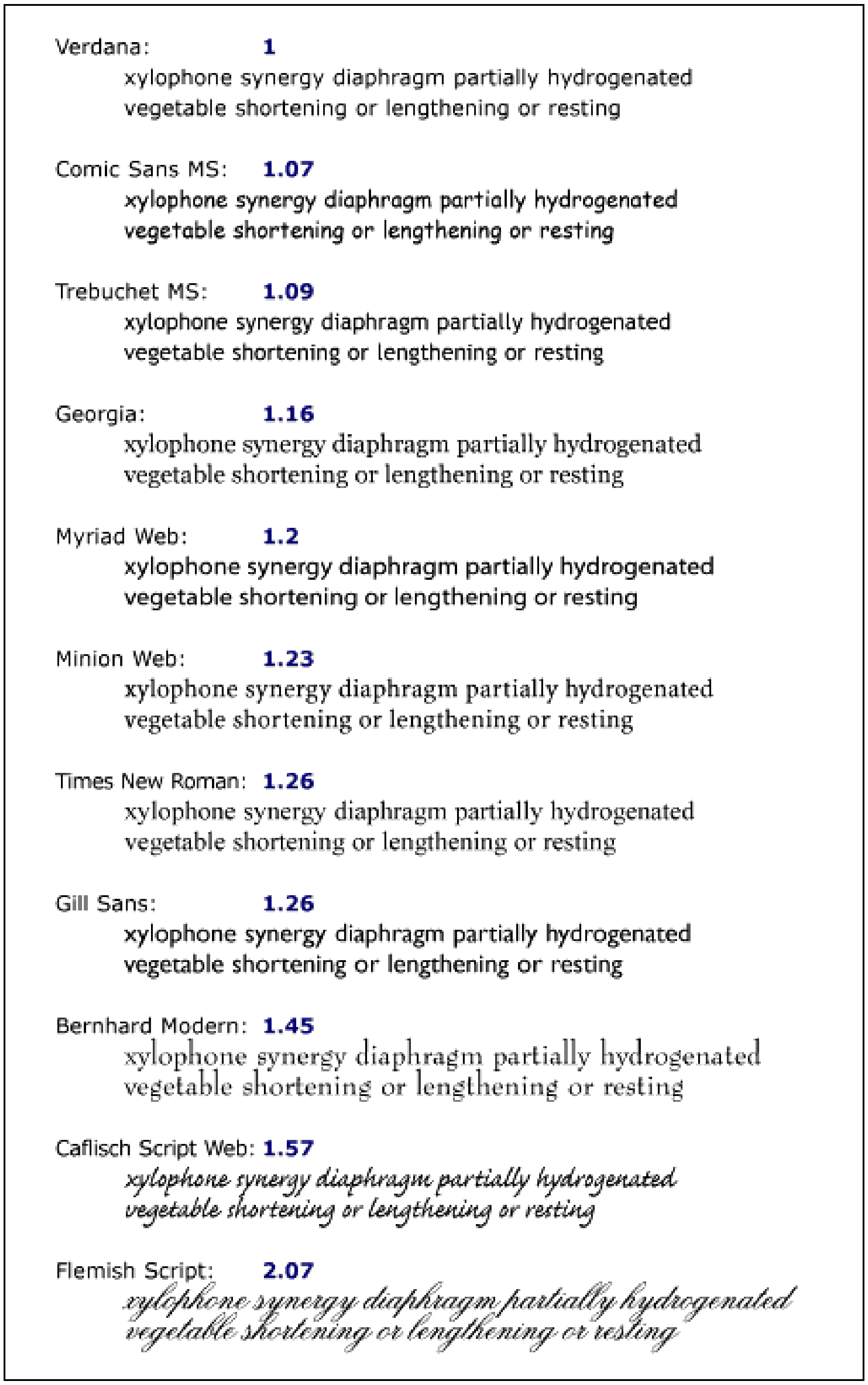
Gill Sans: .46
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Bernhard Modern: .4
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Caflisch Script Web: .37
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

Flemish Script: .28
xylophone synergy diaphragm partially hydrogenated
vegetable shortening or lengthening or resting

The next image shows the results of ’font-size-adjust’ where Verdana has been taken as the"first choice", together with the scaling factor applied. As adjusted, the apparent sizes are nearly linear across faces, though the actual (em) sizes vary by more than 100%. Note that ’font-size-adjust’ tends to stabilize the horizontal metrics of lines, as well.



Verdana:	1	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Comic Sans MS:	1.07	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Trebuchet MS:	1.09	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Georgia:	1.16	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Myriad Web:	1.2	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Minion Web:	1.23	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Times New Roman:	1.26	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Gill Sans:	1.26	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Bernhard Modern:	1.45	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Caflisch Script Web:	1.57	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting
Flemish Script:	2.07	xylophone synergy diaphragm partially hydrogenated vegetable shortening or lengthening or resting

15.2.5 Shorthand font property: the ’font’ property

’font’

Value:	[[<’font-style’> || <’font-variant’> || <’font-weight’>]? <’font-size’> [/ <’line-height’>]? <’font-family’>] | caption | icon | menu | message-box | small-caption | status-bar | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	yes
Percentages:	allowed on ’font-size’ and ’line-height’
Media:	visual

The ’font’ property is, except as described below [p. 209] , a shorthand property for setting ’font-style’, ’font-variant’, ’font-weight’, ’font-size’, ’line-height’, and ’font-family’, at the same place in the style sheet. The syntax of this property is based on a traditional typographical shorthand notation to set multiple properties related to fonts.

All font-related properties are first reset to their initial values, including those listed in the preceding paragraph plus ’font-stretch’ and ’font-size-adjust’. Then, those properties that are given explicit values in the ’font’ shorthand are set to those values. For a definition of allowed and initial values, see the previously defined properties. For reasons of backwards compatibility, it is not possible to set ’font-stretch’ and ’font-size-adjust’ to other than their initial values using the ’font’ shorthand property; instead, set the individual properties.

Example(s):

```
P { font: 12pt/14pt sans-serif }
P { font: 80% sans-serif }
P { font: x-large/110% "new century schoolbook", serif }
P { font: bold italic large Palatino, serif }
P { font: normal small-caps 120%/120% fantasy }
P { font: oblique 12pt "Helvetica Nue", serif; font-stretch: condensed }
```

In the second rule, the font size percentage value (’80%’) refers to the font size of the parent element. In the third rule, the line height percentage (’110%’) refers to the font size of the element itself.

The first three rules do not specify the ’font-variant’ and ’font-weight’ explicitly, so these properties receive their initial values (’normal’). Notice that the font family name "new century schoolbook", which contains spaces, is enclosed in quotes. The fourth rule sets the ’font-weight’ to ’bold’, the ’font-style’ to ’italic’, and implicitly sets ’font-variant’ to ’normal’.

The fifth rule sets the ’font-variant’ (’small-caps’), the ’font-size’ (120% of the parent’s font size), the ’line-height’ (120% of the font size) and the ’font-family’ (’fantasy’). It follows that the keyword ’normal’ applies to the two remaining properties: ’font-style’ and ’font-weight’.

The sixth rule sets the ’font-style’, ’font-size’, and ’font-family’, the other font properties being set to their initial values. It then sets ’font-stretch’ to ’condensed’ since that property cannot be set to that value using the ’font’ shorthand property.

The following values refer to system fonts:

caption
The font used for captioned controls (e.g., buttons, drop-downs, etc.).

icon
The font used to label icons.

menu
  The font used in menus (e.g., dropdown menus and menu lists).
message-box
  The font used in dialog boxes.
small-caption
  The font used for labeling small controls.
status-bar
  The font used in window status bars.

System fonts may only be set as a whole; that is, the font family, size, weight, style, etc. are all set at the same time.These values may then be altered individually if desired. If no font with the indicated characteristics exists on a given platform, the user agent should either intelligently substitute (e.g., a smaller version of the ’caption’ font might be used for the ’smallcaption’ font), or substitute a user agent default font. As for regular fonts, if, for a system font, any of the individual properties are not part of the operating system’s available user preferences, those properties should be set to their initial values.

That is why this property is "almost" a shorthand property: system fonts can only be specified with this property, not with ’font-family’ itself, so ’font’ allows authors to do more than the sum of its subproperties. However, the individual properties such as ’font-weight’ are still given values taken from the system font, which can be independently varied.

Example(s):

```
BUTTON { font: 300 italic 1.3em/1.7em "FB Armada", sans-serif }
BUTTON P { font: menu }
BUTTON P EM { font-weight: bolder }
```

If the font used for dropdown menus on a particular system happened to be, for example, 9-point Charcoal, with a weight of 600, then P elements that were descendants of BUTTON would be displayed as if this rule were in effect:

```
BUTTON P { font: 600 9pt Charcoal }
```

Because the ’font’ shorthand resets to its initial value any property not explicitly given a value, this has the same effect as this declaration:

```
BUTTON P {
  font-style: normal;
  font-variant: normal;
  font-weight: 600;
  font-size: 9pt;
  line-height: normal;
  font-family: Charcoal
}
```

15.2.6 Generic font families

Generic font families are a fallback mechanism, a means of preserving some of the style sheet author’s intent in the worst case when none of the specified fonts can be selected. For optimum typographic control, particular named fonts should be used in style sheets.

All five generic font families are defined to exist in all CSS implementations (they need not necessarily map to five distinct actual fonts). User agents should provide reasonable default choices for the generic font families, which express the characteristics of each family as well as possible within the limits allowed by the underlying technology.

User agents are encouraged to allow users to select alternative choices for the generic fonts.

serif

Glyphs of serif fonts, as the term is used in CSS, have finishing strokes, flared or tapering ends, or have actual serifed endings (including slab serifs). Serif fonts are typically proportionately-spaced. They often display a greater variation between thick and thin strokes than fonts from the ’sans-serif’ generic font family. CSS uses the term ’serif’ to apply to a font for any script, although other names may be more familiar for particular scripts, such as Mincho (Japanese), Sung or Song (Chinese), Totum or Kodig (Korean). Any font that is so described may be used to represent the generic ’serif’ family.

Examples of fonts that fit this description include:

Latin fonts	Times New Roman, Bodoni, Garamond, Minion Web, ITC Stone Serif, MS Georgia, Bitstream Cyberbit
Greek fonts	Bitstream Cyberbit
Cyrillic fonts	Adobe Minion Cyrillic, Excelcior Cyrillic Upright, Monotype Albion 70, Bitstream Cyberbit, ER Bukinst
Hebrew fonts	New Peninim, Raanana, Bitstream Cyberbit
Japanese fonts	Ryumin Light-KL, Kyokasho ICA, Futo Min A101
Arabic fonts	Bitstream Cyberbit
Cherokee fonts	Lo Cicero Cherokee

sans-serif

Glyphs in sans-serif fonts, as the term is used in CSS, have stroke endings that are plain -- without any flaring, cross stroke, or other ornamentation. Sans-serif fonts are typically proportionately-spaced. They often have little variation between thick and thin strokes, compared to fonts from the ’serif’ family. CSS uses the term ’sans-serif’ to apply to a font for any script, although other names may be more familiar for particular scripts, such as Gothic (Japanese), Kai (Chinese), or Pathang (Korean). Any font that is so described may be used to represent the generic ’sans-serif’ family.

Examples of fonts that fit this description include:

Latin fonts	MS Trebuchet, ITC Avant Garde Gothic, MS Arial, MS Verdana, Univers, Futura, ITC Stone Sans, Gill Sans, Akzidenz Grotesk, Helvetica
Greek fonts	Attika, Typiko New Era, MS Tahoma, Monotype Gill Sans 571, Helvetica Greek
Cyrillic fonts	Helvetica Cyrillic, ER Univers, Lucida Sans Unicode, Bastion
Hebrew fonts	Arial Hebrew, MS Tahoma
Japanese fonts	Shin Go, Heisei Kaku Gothic W5
Arabic fonts	MS Tahoma

cursive

Glyphs in cursive fonts, as the term is used in CSS, generally have either joining strokes or other cursive characteristics beyond those of italic typefaces. The glyphs are partially or completely connected, and the result looks more like handwritten pen or brush writing than printed letterwork. Fonts for some scripts, such as Arabic, are almost always cursive. CSS uses the term ’cursive’ to apply to a font for any script, although other names such as Chancery, Brush, Swing and Script are also used in font names.

Examples of fonts that fit this description include:

Latin fonts	Caflisch Script, Adobe Poetica, Sanvito, Ex Ponto, Snell Roundhand, Zapf-Chancery
Cyrillic fonts	ER Architekt
Hebrew fonts	Corsiva
Arabic fonts	DecoType Naskh, Monotype Urdu 507

fantasy

Fantasy fonts, as used in CSS, are primarily decorative while still containing representations of characters (as opposed to Pi or Picture fonts, which do not represent characters). Examples include:

Latin fonts	Alpha Geometrique, Critter, Cottonwood, FB Reactor, Studz

monospace

The sole criterion of a monospace font is that all glyphs have the same fixed width. (This can make some scripts, such as Arabic, look most peculiar.) The effect is similar to a manual typewriter, and is often used to set samples of computer code.

Examples of fonts which fit this description include:

Latin fonts	Courier, MS Courier New, Prestige, Everson Mono
Greek Fonts	MS Courier New, Everson Mono
Cyrillic fonts	ER Kurier, Everson Mono
Japanese fonts	Osaka Monospaced
Cherokee fonts	Everson Mono

15.3 Font selection

The second phase of the CSS2 font mechanism concerns the user agent's selection of a font based on author-specified font properties, available fonts, etc. The details of the font matching algorithm [p. 231] are provided below.

There are four possible font selection actions: name matching, intelligent matching, synthesis, and download.

font name matching

In this case, the user agent uses an existing, accessible font that has the same family name as the requested font. (Note that the appearance and the metrics might not necessarily match, if the font that the document author used and the font on the client system are from different foundries). The matching information is restricted to the CSS font properties, including the family name. This is the only method used by CSS1.

intelligent font matching

In this case, the user agent uses an existing, accessible font that is the closest match in appearance to the requested font. (Note that the metrics might not match exactly). The matching information includes information about the kind of font (text or symbol), nature of serifs, weight, cap height, x height, ascent, descent, slant, etc.

font synthesis

In this case, the user agent creates a font that is not only a close match in appearance, but also matches the metrics of the requested font. The synthesizing information includes the matching information and typically requires more accurate values for the parameters than are used for some matching schemes. In particular, synthesis requires accurate width metrics and character to glyph substitution and position information if all the layout characteristics of the specified font are to be preserved.

font download

Finally, the user agent may retrieve a font over the Web. This is similar to the process of fetching images, sounds, or applets over the Web for display in the current document, and likewise can cause some delay before the page can be displayed.

Progressive rendering is a combination of download and one of the other methods; it provides a temporary substitute font (using name matching, intelligent matching, or synthesis) to allow content to be read while the requested font downloads. Once the real font has been successfully downloaded, it replaces the temporary font, hopefully without the need to reflow.

Note. *Progressive rendering requires metric information about the font in order to avoid re-layout of the content when the actual font has been loaded and rendered. This metric information is sufficiently verbose that it should only be specified at most once per font in a document.*

15.3.1 Font Descriptions and @font-face

The font description provides the bridge between an author’s font specification and the *font data*, which is the data needed to format text and to render the abstract glyphs to which the characters map - the actual scalable outlines or bitmaps. Fonts are *referenced* by style sheet properties.

The font description is added to the font database and then used to select the relevant font data. The font description contains descriptors such as the location of the font data on the Web, and characterizations of that font data. The font descriptors are also needed to match the style sheet font properties to particular font data. The level of detail of a font description can vary from just the name of the font up to a list of glyph widths.

Font descriptors may be classified into three types:

1. those that provide the link between the CSS usage of the font and the font description (these have the same names as the corresponding CSS font properties),
2. the URI for the location of the font data,
3. those that further characterize the font, to provide a link between the font description and the font data.

All font descriptions are specified via a *@font-face* at-rule. The general form is:
`@font-face { <font-description> }`
where the <font-description> has the form:

```
descriptor: value;
descriptor: value;
[...]
descriptor: value;
```

Each @font-face rule specifies a value for every font descriptor, either implicitly or explicitly. Those not given explicit values in the rule take the initial value listed with each descriptor in this specification. These descriptors apply solely within the context of the @font-face rule in which they are defined, and do not apply to document language elements. Thus, there is no notion of which elements the descriptors apply to, or whether the values are inherited by child elements.

The available font descriptors are described in later sections of this specification.

For example, here the font ’Robson Celtic’ is defined and referenced in a style sheet contained in an HTML document.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Font test</TITLE>
    <STYLE TYPE="text/css" MEDIA="screen, print">
      @font-face {
        font-family: "Robson Celtic";
        src: url("http://site/fonts/rob-celt")
      }
      H1 { font-family: "Robson Celtic", serif }
    </STYLE>
  </HEAD>
  <BODY>
    <H1> This heading is displayed using Robson Celtic</H1>
  </BODY>
</HTML>
```

The style sheet (in the STYLE element) contains a CSS rule that sets all H1 elements to use the ’Robson Celtic’ font family.

A CSS1 implementation will search the client for a font whose family name and other properties match ’Robson Celtic’ and, if it fails to find it, will use the UA-specific fallback serif font (which is defined to exist [p. 210]).

A user agent implementing CSS2 will first examine @font-face rules in search of a font description defining ’Robson Celtic’. This example contains a rule that matches. Although this rule doesn’t contain much font data, it does have a URI where the font can be retrieved for rendering this document. Downloaded fonts should not be made available to other applications. If no matching @font-face is found, the user agent will attempt the same match as a user agent implementing CSS1.

Note that if the font ’Robson Celtic’ *had* been installed on the client system, this would have caused the UA to add an entry in the font database for the installed copy as described in the section on the font matching algorithm [p. 231] . The installed copy would have been matched before the downloadable font in the example above.

CSS1 implementations, which do not understand the @font-face rule, will encounter the opening curly brackets and will ignore [p. 42] forward until the matching closing curly brackets. This at-rule conforms with the forward-compatible parsing [p. 35] requirement of CSS. Parsers may ignore [p. 42] these rules without error.

Having the font descriptors separate from the font data has a benefit beyond being able to do font selection and/or substitution. The data protection and replication restrictions on the font descriptors may be much weaker than on the full font data. Thus, it may be possible to install the font definition locally, or at least to have it in a local cache if it occurs in a commonly referenced style sheet; this would not require accessing the full font definition over the Web more than once per named font.

If a font descriptor is duplicated, the last occurring descriptor wins and the rest must be ignored.

Also, any descriptors that are not recognized or useful to the user agent must be ignored [p. 42] . Future versions of CSS may allow additional descriptors for the purpose of better font substitution, matching, or synthesis.

15.3.2 Descriptors for Selecting a Font: ’font-family’, ’font-style’, ’font-variant’, ’font-weight’, ’font-stretch’ and ’font-size’

The following descriptors have the same names as the corresponding CSS2 font properties, and take a single value or comma-separated list of values.

The values within that list are, except as explicitly noted, the same as those for the corresponding CSS2 property. If there is a single value, that is the value that must be matched. If there is a list, any list item constitutes a match. If the descriptor is omitted from the @font-face, the initial value for the descriptor is used.

’font-family’ (Descriptor)

Value:	[<family-name> | <generic-family>] [, [<family-name> | <generic-family>]]*
Initial:	depends on user agent
Media:	visual

This is the descriptor for the font family name [p. 198] of a font and takes the same values as the ’font-family’ property.

’font-style’ (Descriptor)

Value:	all | [normal | italic | oblique] [, [normal | italic | oblique]]*
Initial:	all
Media:	visual

This is the descriptor for the style of a font and takes the same values as the ’font-style’ property, except that a comma-separated list is permitted.

’font-variant’ (Descriptor)

Value:	[normal | small-caps] [,[normal | small-caps]]*
Initial:	normal
Media:	visual

This is the CSS indication of whether this face is the small-caps variant of a font. It takes the same values as the ’font-variant’ property except that a comma-separated list is permitted.

Note. *Cyrillic pryamo&#301; faces may be labeled with a ’font-variant’ of small-caps, which will give better consistency with Latin faces (and the companion kursiv face labeled with ’font-style’ italic for the same reason).*

’font-weight’ (Descriptor)

Value:	all | [normal | bold | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900] [, [normal | bold | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900]]*
Initial:	all
Media:	visual

This is the descriptor for the weight of a face relative to others in the same font family. It takes the same values as the ’font-weight’ property with three exceptions:

1. relative keywords (bolder, lighter) are not permitted.
2. a comma-separated list of values is permitted, for fonts that contain multiple weights.
3. an additional keyword, ’all’ is permitted, which means that the font will match for all possible weights; either because it contains multiple weights, or because that face only has a single weight.

’font-stretch’ (Descriptor)

Value:	all | [normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded] [, [normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded]]*
Initial:	normal
Media:	visual

This is the CSS indication of the condensed or expanded nature of the face relative to others in the same font family. It takes the same values as the ’font-stretch’ property except that:

- relative keywords (wider, narrower) are not permitted
- a comma-separated list is permitted
- The keyword ’all’ is permitted

’font-size’ (Descriptor)

Value:	all | <length> [, <length>]*
Initial:	all
Media:	visual

This is the descriptor for the sizes provided by this font. Only absolute length [p. 43] units are permitted, in contrast to the ’font-size’ property, which allows both relative and absolute lengths and sizes. A comma-separated list of absolute lengths is permitted.

The initial value of ’all’ is suitable for most scalable fonts, so this descriptor is primarily for use in an @font-face for bitmap fonts, or scalable fonts designed to be rasterised at a restricted range of font sizes.

15.3.3 Descriptors for Font Data Qualification: ’unicode-range’

The following descriptor is optional within a font definition, but is used to avoid checking or downloading a font that does not have sufficient glyphs to render a particular character.

’unicode-range’ (Descriptor)

Value:	<urange> [, <urange>]*
Initial:	U+0-7FFFFFFF
Media:	visual

This is the descriptor for the range of ISO 10646 characters [p. 230] covered by the font.

The values of <urange> are expressed using hexadecimal numbers prefixed by "U+", corresponding to character code positions in ISO 10646 ([ISO10646]).

For example, `U+05D1` is the ISO 10646 character ’Hebrew letter bet’. For values outside the Basic Multilingual Plane (BMP), additional leading digits corresponding to the plane number are added, also in hexadecimal, like this: `U+A1234` which is the character on Plane 10 at hexadecimal code position 1234. At the time of writing no characters had been assigned outside the BMP. Leading zeros (for example, 0000004D) are valid, but not required.

The initial value [p. 69] of this descriptor covers not only the entire Basic Multilingual Plane (BMP), which would be expressed as U+0-FFFF, but also the whole repertoire of ISO 10646. Thus, the initial value says that the font may have glyphs for characters anywhere in ISO 10646. Specifying a value for ’unicode-range’ provides information to make searching efficient, by declaring a constrained range in which the font may have glyphs for characters. The font need not be searched for characters outside this range.

Values may be written with any number of digits. For single numbers, the character ’?’ is assumed to mean ’any value’ which creates a *range* of character positions. Thus, using a *single number*:

unicode-range: U+20A7
: no wild cards - it indicates a single character position (the Spanish peseta currency symbol)

unicode-range: U+215?
: one wild card, covers the range 2150 to 215F (the fractions)

unicode-range: U+00??
: two wild cards, covers the range 0000 to 00FF (Latin-1)

unicode-range: U+E??
: two wild cards, covers 0E00 to 0EFF (the Lao script)

A *pair of numbers* in this format can be combined with the dash character to indicate larger ranges. For example:

unicode-range: U+AC00-D7FF
: the range is AC00 to D7FF (the Hangul Syllables area)

Multiple, discontinuous ranges can be specified, separated by a comma. As with other comma-separated lists in CSS, any whitespace [p. 37] before or after the comma is ignored. [p. 42] For example:

unicode-range: U+370-3FF, U+1F??
: This covers the range 0370 to 03FF (Modern Greek) plus 1F00 to 1FFF (Ancient polytonic Greek).

unicode-range: U+3000-303F, U+3100-312F, U+32??, U+33??, U+4E00-9FFF, U+F9000-FAFF, U+FE30-FE4F
: Something of a worst case in terms of verbosity, this very precisely indicates that this (extremely large) font contains only Chinese characters from ISO 10646, without including any characters that are uniquely Japanese or Korean. The range is 3000 to 303F (CJK symbols and punctuation) plus 3100 to 312F (Bopomofo) plus 3200 to 32FF (enclosed CJK letters and months) plus 3300 to 33FF (CJK compatibility zone) plus 4E00 to 9FFF (CJK unified Ideographs) plus F900 to FAFF (CJK compatibility ideographs) plus FE30 to FE4F (CJK compatibility forms).
: A more likely representation for a typical Chinese font would be:
: unicode-range: U+3000-33FF, U+4E00-9FFF

unicode-range: U+11E00-121FF
: This font covers a proposed registration for Aztec pictograms, covering the range 1E00 to 21FF in plane 1.

unicode-range: U+1A00-1A1F
: This font covers a proposed registration for Irish Ogham covering the range 1A00 to 1A1F

15.3.4 Descriptor for Numeric Values: ’units-per-em’

The following descriptor specifies the number of "units" per em; these units may be used by several other descriptors to express various lengths, so ’units-per-em’ is required if other descriptors depend on it.

’units-per-em’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the number of the coordinate units on the em square [p. 227] , the size of the design grid on which glyphs are laid out.

15.3.5 Descriptor for Referencing: ’src’

This descriptor is required for referencing actual font data, whether downloadable or locally installed.

’src’ (Descriptor)

Value: [<uri> [format(<string> [, <string>]*)] | <font-face-name>] [, <uri> [format(<string> [, <string>]*)] | <font-face-name>]*
Initial: undefined
Media: visual

This is a prioritized, comma-separated list of external references and/or locally installed font face names. The external reference points to the font data on the Web. This is required if the WebFont is to be downloaded. The font resource may be a subset of the source font, for example it may contain only the glyphs needed for the current page or for a set of pages.

The external reference consists of a URI, followed by an optional hint regarding the format of font resource to be found at that URI, and this information should be used by clients to avoid following links to fonts in formats they are unable to use. As with any hypertext reference, there may be other formats available, but the client has a better idea of what is likely to be there, in a more robust way than trying to parse filename extensions in URIs.

The format hint contains a comma-separated list of format strings that denote well-known font formats. The user agent will recognize the name of font formats that it supports, and will avoid downloading fonts in formats that it does not recognize.

An initial list of format strings defined by this specification and representing formats likely to be used by implementations on various platforms is:

String	Font Format	Examples of common extensions
"truedoc-pfr"	TrueDoc™ Portable Font Resource	.pfr
"embedded-opentype"	Embedded OpenType	.eot
"type-1"	PostScript™ Type 1	.pfb, .pfa
"truetype"	TrueType	.ttf
"opentype"	OpenType, including TrueType Open	.ttf
"truetype-gx"	TrueType with GX extensions	
"speedo"	Speedo	
"intellifont"	Intellifont	

As with other URIs in CSS [p. 46] , the URI may be partial, in which case it is resolved relative to the location of the style sheet containing the @font-face.

The locally-installed <font-face-name> is the full font name of a locally installed font. The *full font name* is the name of the font as reported by the operating system and is the name most likely to be used in reader style sheets, browser default style sheets or possibly author style sheets on an intranet. Adornments such as bold, italic, and underline are often used to differentiate faces within a font family. For more information about full font names [p. 226] please consult the notes below.

The notation for a <font-face-name> is the full font name, which must be quoted since it may contain any character, including spaces and punctuation, and also must be enclosed in "local(" and ")".

Example(s):

```
src: url("http://foo/bar")
```
a full URI and no information about the font format(s) available there
```
src: local("BT Century 751 No. 2 Semi Bold Italic")
```
references a particular face of a locally installed font
```
src: url("../fonts/bar") format("truedoc-pfr")
```
a partial URI which has a font available in TrueDoc format
```
src: url("http://cgi-bin/bar?stuff") format("opentype",
"intellifont")
```
a full URI, in this case to a script, which can generate two different formats - OpenType and Intellifont
```
src: local("T-26 Typeka Mix"),
url("http://site/magda-extra") format("type-1")
```
two alternatives are given, firstly a locally installed font and secondly a downloadable font available in Type 1 format.

Access to locally installed fonts is via the <font-face-name>. The font face name is not truly unique, nor is it truly platform or font format independent, but at the moment it is the best way to identify locally installed font data. The use of the font face name can be made more accurate by providing an indication of the glyph complement required. This may be done by indicating the range of ISO 10646 character positions for which the font provides some glyphs (see ’unicode-range’).

15.3.6 Descriptors for Matching: ’panose-1’, ’stemv’, ’stemh’, ’slope’, ’cap-height’, ’x-height’, ’ascent’, and ’descent’

These descriptors are optional for a CSS2 definition, but may be used if intelligent font matching or font size adjustment is desired by the author.

’panose-1’ (Descriptor)

Value:	[<integer>]{10}
Initial:	0 0 0 0 0 0 0 0 0 0
Media:	visual

This is the descriptor for the Panose-1 number [p. 230] and consists of ten decimal integers, separated by whitespace [p. 37] . A comma-separated list is not permitted for this descriptor, because the Panose-1 system can indicate that a range of values are matched. The initial value is zero, which means "any", for each PANOSE digit; all fonts will match the Panose number if this value is used. Use of the Panose-1 descriptor is strongly recommended for latin fonts. For further details, see Appendix C [p. 299] .

’stemv’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the vertical stem width [p. 231] of the font. If the value is undefined, the descriptor is not used for matching. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

’stemh’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the horizontal stem width [p. 228] of the font. If the value is undefined, the descriptor is not used for matching. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

’slope’ (Descriptor)

Value:	<number>
Initial:	0
Media:	visual

This is the descriptor for the vertical stroke angle [p. 231] of the font.

’cap-height’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the number of the height of uppercase glyphs [p. 228] of the font. If the value is undefined, the descriptor is not used for matching. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

’x-height’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the height of lowercase glyphs [p. 228] of the font. If the value is undefined, the descriptor is not used for matching. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used. This descriptor can be very useful when using the ’font-size-adjust’ property, because computation of the z value of candidate fonts requires both the font size and the x-height; it is therefore recommended to include this descriptor.

’ascent’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the maximum unaccented height [p. 229] of the font. If the value is undefined, the descriptor is not used for matching. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

’descent’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the Maximum unaccented depth [p. 229] of the font. If the value is undefined, the descriptor is not used for matching. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

15.3.7 Descriptors for Synthesis: ’widths’, ’bbox’ and ’definition-src’

Synthesizing a font means, at minimum, matching the width metrics of the specified font. Therefore, for synthesis, this metric information must be available. Similarly, progressive rendering requires width metrics in order to avoid reflow of the content when the actual font has been loaded. Although the following descriptors are optional for a CSS2 definition, some are required if synthesizing (or reflow-free progressive rendering) is desired by the author. Should the actual font become available, the substitute should be replaced by the actual font. Any of these descriptors that are present will be used to provide a better or faster approximation of the intended font.

Of these descriptors, the most important are the ’widths’ descriptor and ’bbox’ which are used to prevent text reflow should the actual font become available. In addition, the descriptors in the set of descriptors used for matching [p. 220] can be used to provide a better synthesis of the actual font appearance.

’widths’ (Descriptor)

Value:	[<urange>]? [<number>]+ [,[<urange>]? <number>]+]
Initial:	undefined
Media:	visual

This is the descriptor for the glyph widths [p. 228] . The value is a comma-separated list of <urange> values each followed by one or more glyph widths. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

If the <urange> is omitted, a range of U+0-7FFFFFFF is assumed which covers all characters and their glyphs. If not enough glyph widths are given, the last in the list is replicated to cover that urange. If too many widths are provided, the extras are ignored. [p. 42]

Example(s):

For example:

```
widths: U+4E00-4E1F 1736 1874 1692
widths: U+1A?? 1490, U+215? 1473 1838 1927 1684 1356 1792
    1815 1848 1870 1492 1715 1745 1584 1992 1978 1770
```

In the first example a range of 32 characters is given, from 4E00 to 4E1F. The glyph corresponding to the first character (4E00) has a width of 1736, the second has a width of 1874 and the third, 1692. Because not enough widths have been provided, the last width replicates to cover the rest of the specified range. The second example sets a single width, 1490, for an entire range of 256 glyphs and then explicit widths for a range of 16 glyphs.

This descriptor cannot describe multiple glyphs corresponding to a single character, or ligatures of multiple characters. Thus, this descriptor can *only* be used for scripts that do not have contextual forms or mandatory ligatures. It is nevertheless useful in those situations. Scripts that require a one-to-many or many-to-many mapping of characters to glyphs cannot at present use this descriptor to enable font synthesis although they can still use font downloading or intelligent matching.

’bbox’ (Descriptor)

Value:	<number>, <number>, <number>, <number>
Initial:	undefined
Media:	visual

This is the descriptor for the maximal bounding box [p. 229] of the font. The value is a comma-separated list of exactly four numbers specifying, in order, the lower left x, lower left y, upper right x, and upper right y of the bounding box for the complete font.

’definition-src’ (Descriptor)

Value:	<uri>
Initial:	undefined
Media:	visual

The font descriptors may either be within the font definition in the style sheet, or may be provided within a separate *font definition resource* identified by a URI. The latter approach can reduce network traffic when multiple style sheets reference the same fonts.

15.3.8 Descriptors for Alignment: ’baseline’, ’centerline’, ’mathline’, and ’topline’

These optional descriptors are used to align runs of different scripts with one another.

’baseline’ (Descriptor)

Value:	<number>
Initial:	0
Media:	visual

This is the descriptor for the lower baseline [p. 229] of a font. If this descriptor is given a non-default (non-zero) value, the ’units-per-em’ [p. 218] descriptor must also be used.

’centerline’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the central baseline [p. 227] of a font. If the value is undefined, the UA may employ various heuristics such as the midpoint of the ascent and descent values. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

’mathline’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the mathematical baseline [p. 229] of a font. If undefined, the UA may use the center baseline. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

’topline’ (Descriptor)

Value:	<number>
Initial:	undefined
Media:	visual

This is the descriptor for the top baseline [p. 230] of a font. If undefined, the UA may use an approximate value such as the ascent. If this descriptor is used, the ’units-per-em’ [p. 218] descriptor must also be used.

15.3.9 Examples

Example(s):

Given the following list of fonts:

Swiss 721 light	light & light italic
Swiss 721	roman, bold, italic, bold italic
Swiss 721 medium	medium & medium italic
Swiss 721 heavy	heavy & heavy italic
Swiss 721 black	black, black italic, & black #2
Swiss 721 Condensed	roman, bold, italic, bold italic
Swiss 721 Expanded	roman, bold, italic, bold italic

The following font descriptions could be used to make them available for download.

```
@font-face {
    font-family: "Swiss 721";
    src: url("swiss721lt.pfr"); /* Swiss 721 light */
    font-style: normal, italic;
    font-weight: 200;
}
@font-face {
    font-family: "Swiss 721";
    src: url("swiss721.pfr"); /* The regular Swiss 721 */
}
@font-face {
    font-family: "Swiss 721";
    src: url("swiss721md.pfr"); /* Swiss 721 medium */
    font-style: normal, italic;
    font-weight: 500;
}
@font-face {
    font-family: "Swiss 721";
    src: url("swiss721hvy.pfr"); /* Swiss 721 heavy */
    font-style: normal, italic;
    font-weight: 700;
}
@font-face {
    font-family: "Swiss 721";
    src: url("swiss721blk.pfr"); /* Swiss 721 black */
    font-style: normal, italic;
    font-weight: 800,900; /* note the interesting problem that
                               the 900 weight italic doesn’t exist */
}
@font-face {
    font-family: "Swiss 721";
    src: url(swiss721.pfr); /* The condensed Swiss 721 */
    font-stretch: condensed;
}
@font-face {
    font-family: "Swiss 721";
    src: url(swiss721.pfr); /* The expanded Swiss 721 */
    font-stretch: expanded;
}
```

15.4 Font Characteristics

15.4.1 Introducing Font Characteristics

In this section are listed the font characteristics that have been found useful for client-side font matching, synthesis, and download for heterogeneous platforms accessing the Web. The data may be useful for any medium that needs to use fonts on the Web by some other means than physical embedding of the font data inside the medium.

These characteristics are used to characterize fonts. They are not specific to CSS, or to style sheets. In CSS, each characteristic is described by a font descriptor. These characteristics could also be mapped onto VRML nodes, or CGM Application Structures, or a Java API, or alternative style sheet languages. Fonts retrieved by one medium and stored in a proxy cache could be re-used by another medium, saving download time and network bandwidth, if a common system of font characteristics are used throughout.

A non-exhaustive list of examples of such media includes:

- 2-D vector formats
  - Computer Graphics Metafile
  - Simple Vector Format
- 3-D graphics formats
  - VRML
  - 3DMF
- Object embedding technologies
  - Java
  - Active-X
  - Obliq

15.4.2 Full font name

This is the full name of a particular face of a font family. It typically includes a variety of non-standardized textual qualifiers or *adornments* appended to the font family name. It may also include a foundry name or abbreviation, often prepended to the font family name. It is only used to refer to locally installed fonts, because the format of the adorned name can vary from platform to platform. It must be quoted.

For example, the font family name of the TrueType font and the PostScript name may differ in the use of space characters, punctuation, and in the abbreviation of some words (e.g., to meet various system or printer interpreter constraints on length of names). For example, spaces are not allow in a PostScript name, but are common in full font names. The TrueType name table can also contain the PostScript name, which has no spaces.

The name of the font definition is important because it is the link to any locally installed fonts. It is important that the name be robust, both with respect to platform and application independence. For this reason, the name should be one that is not application- or language-specific.

The ideal solution would be to have a name that uniquely identifies each collection of font data. This name does not exist in current practice for font data. Fonts with the same face name can vary over a number of descriptors. Some of these descriptors, such as different complements of glyphs in the font, may be insignificant if the needed glyphs are in the font. Other descriptors, such as different width metrics, make fonts with the same name incompatible. It does not seem possible to define a rule that will always identify incompatibilities, but will not prevent the use of a perfectly suitable local copy of the font data with a given name. Therefore, only the range of ISO 10646 characters will be used to qualify matches for the font face name.

Since a prime goal of the font face name in the font definition is to allow a user agent to determine when there is a local copy of the specified font data, the font face name must be a name that will be in all legitimate copies of the font data. Otherwise, unnecessary Web traffic may be generated due to missed matches for the local copy.

15.4.3 Coordinate units on the em square

Certain values, such as width metrics, are expressed in units that are relative to an abstract square whose height is the intended distance between lines of type in the same type size. This square is called the *em square* and it is the design grid on which the glyph outlines are defined. The value of this descriptor specifies how many units the EM square is divided into. Common values are for example 250 (Intellifont), 1000 (Type 1) and 2048 (TrueType, TrueType GX and OpenType).

If this value is not specified, it becomes impossible to know what any font metrics mean. For example, one font has lowercase glyphs of height 450; another has smaller ones of height 890! The numbers are actually fractions; the first font has 450/1000 and the second has 890/2048 which is indeed smaller.

15.4.4 Central Baseline

This gives the position in the em square [p. 227] of the central baseline. The central baseline is used by ideographic scripts for alignment, just as the bottom baseline is used for Latin, Greek, and Cyrillic scripts.

15.4.5 Font Encoding

Either explicitly or implicitly, each font has a table associated with it, the *font encoding table*, that tells what character each glyph represents. This table is also referred to as an *encoding vector*.

In fact, many fonts contain several glyphs for the same character. Which of those glyphs should be used depends either on the rules of the language, or on the preference of the designer.

In Arabic, for example, all letters have four (or two) different shapes, depending on whether the letter is used at the start of a word, in the middle, at the end, or in isolation. It is the same character in all cases, and thus there is only one character in the source document, but when printed, it looks different each time.

There are also fonts that leave it to the graphic designer to choose from among various alternative shapes provided. Unfortunately, CSS2 doesn’t yet provide the means to select those alternatives. Currently, it is always the default shape that is chosen from such fonts.

15.4.6 Font family name

This specifies the family name portion of the font face name. For example, the family name for Helvetica-Bold is Helvetica and the family name of ITC Stone Serif Semibold Italic is ITC Stone Serif. Some systems treat adornments relating to condensed or expanded faces as if they were part of the family name.

15.4.7 Glyph widths

This is a list of widths, on the design grid, for the glyph corresponding to each character. The list is ordered by ISO10646 code point. Widths cannot usefully be specified when more than one glyph maps to the same character or when there are mandatory ligatures.

15.4.8 Horizontal stem width

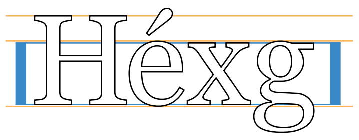
This value refers to the *dominant* stem of the font. There may be two or more designed widths. For example, the main vertical stems of Roman characters will differ from the thin stems on serifed "M" and "N", plus there may be different widths for uppercase and lowercase characters in the same font. Also, either by design or by error, all stems may have slightly different widths.

15.4.9 Height of uppercase glyphs

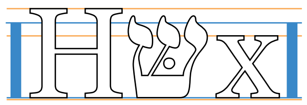
This measurement is the y-coordinate of the top of flat uppercase letters in Latin, Greek, and Cyrillic scripts, measured from the baseline. This descriptor is not necessarily useful for fonts that do not contain any glyphs from these scripts.

15.4.10 Height of lowercase glyphs

This measurement is the y-coordinate of the top of unaccented, non-ascending lowercase letters in Latin, Greek and Cyrillic scripts, measured from the baseline. Flat-topped letters are used, ignoring any optical correction zone. This is usually used as a ratio of lowercase to uppercase heights as a means to compare font families.



This descriptor is not useful for fonts that do not contain any glyphs from these scripts. Since the heights of lowercase and uppercase letters are often expressed as a ratio for comparing different fonts, it may be useful to set both the lowercase and uppercase heights to the same value for unicameral scripts such as Hebrew, where for mixed Latin and Hebrew text, the Hebrew characters are typically set at a height midway between the uppercase and lowercase heights of the Latin font.



15.4.11 Lower Baseline

This gives the position in the em square of the lower baseline. The lower base-line is used by Latin, Greek, and Cyrillic scripts for alignment, just as the upper baseline is used for Sanscrit-derived scripts.

15.4.12 Mathematical Baseline

This gives the position in the em square of the mathematical baseline. The math-ematical baseline is used by mathematical symbols for alignment, just as the lower baseline is used for Latin, Greek, and Cyrillic scripts.

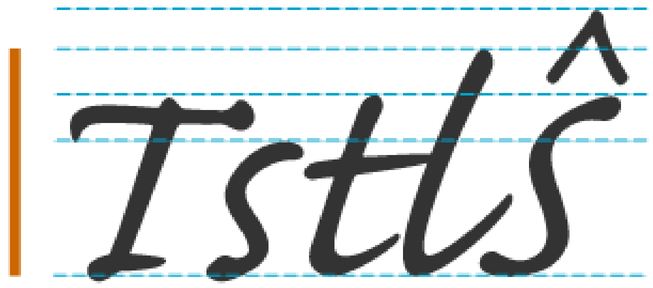
15.4.13 Maximal bounding box

The maximal bounding box is the smallest rectangle enclosing the shape that results if all glyphs in the font are placed with their origins coincident, and then painted.

If a dynamically downloadable font has been generated by subsetting a parent font, the bbox should be that of the parent font.

15.4.14 Maximum unaccented height

This measurement, on the em square, is from the baseline to the highest point reached by any glyph, excluding any accents or diacritical marks.



15.4.15 Maximum unaccented depth

This measurement, on the em square, is from the baseline to the lowest point reached by any glyph, excluding any accents or diacritical marks.

15.4.16 Panose-1 number

Panose-1 is an industry standard TrueType font classification and matching technology. The PANOSE system consists of a set of ten numbers that categorize the key attributes of a Latin typeface, a classification procedure for creating those numbers, and Mapper software that determines the closest possible font match given a set of typefaces. The system *could*, with modification, also be used for Greek and Cyrillic, but is not suitable for unicameral and ideographic scripts (Hebrew, Armenian, Arabic, Chinese/Japanese/Korean).

15.4.17 Range of ISO 10646 characters

This indicates the glyph repertoire of the font, relative to ISO 10646 (Unicode). Since this is sparse (most fonts do not cover the whole of ISO 10646) this descriptor lists blocks or ranges that do have *some* coverage (no promise is made of complete coverage) and is used to eliminate unsuitable fonts (ones that will not have the required glyphs). It does not indicate that the font definitely has the required glyphs, only that it is worth downloading and looking at the font. See [ISO10646] for information about useful documents.

This method is extensible to future allocation of characters in Unicode, without change of syntax and without invalidating existing content.

Font formats that do not include this information, explicitly or indirectly, may still use this characteristic, but the value must be supplied by the document or style sheet author.

There are other classifications into scripts, such as the Monotype system (see [MONOTYPE]) and a proposed ISO script system. These are not readily extensible.

Because of this, classification of glyph repertoires by the range of ISO 10646 characters that may be represented with a particular font is used in this specification. This system is extensible to cover any future allocation.

15.4.18 Top Baseline

This gives the position in the em square of the top baseline. The top baseline is used by Sanscrit-derived scripts for alignment, just as the bottom baseline is used for Latin, Greek, and Cyrillic scripts.

15.4.19 Vertical stem width

This is the width of vertical (or near-vertical) stems of glyphs. This information is often tied to hinting, and may not be directly accessible in some font formats. The measurement should be for the *dominant* vertical stem in the font because there might be different groupings of vertical stems (e.g., one main one, and one lighter weight one as for an uppercase M or N).

15.4.20 Vertical stroke angle

This is the angle, in degrees counterclockwise from the vertical, of the dominant vertical strokes of the font. The value is negative for fonts that slope to the right, as almost all italic fonts do. This descriptor may also be specified for oblique fonts, slanted fonts, script fonts, and in general for any font whose vertical strokes are not precisely vertical. A non-zero value does not of itself indicate an italic font.

15.5 Font matching algorithm

This specification extends the algorithm given in the CSS1 specification. This algorithm reduces down to the algorithm in the CSS1 specification when the author and reader style sheets do not contain any @font-face rules.

Matching of descriptors to font faces must be done carefully. The descriptors are matched in a well-defined order to insure that the results of this matching process are as consistent as possible across UAs (assuming that the same library of font faces and font descriptions is presented to each of them). This algorithm may be optimized, provided that an implementation behaves as if the algorithm had been followed exactly.

1. The user agent makes (or accesses) a database of relevant font-face descriptors of all the fonts of which the UA is aware. If there are two fonts with exactly the same descriptors, one of them is ignored. [p. 42] The UA may be aware of a font because:
    - it has been installed locally
    - it is declared using an @font-face rule in one of the style sheets linked to or contained in the current document
    - it is used in the UA default style sheet, which conceptually exists in all UAs and is considered to have full @font-face rules for all fonts which the UA will use for default presentation, plus @font-face rules for the five special generic font families (see ’font-family’) defined in CSS2
2. At a given element and for each character in that element, the UA assembles the font properties applicable to that element. Using the complete set of properties, the UA uses the ’font-family’ descriptor to choose a tentative font family. Thus, matching on a family name will succeed before matching on some other descriptor. The remaining properties are tested against the family according to the matching criteria described with each descriptor. If there are matches for all the remaining properties, then that is the matching font face for the given element.
3. If there is no matching font face within the ’font-family’ being processed by step 2, *UAs that implement intelligent matching* may proceed to examine

other descriptors such as x-height, glyph widths, and panose-1 to identify a different tentative font family. If there are matches for all the remaining descriptors, then that is the matching font face for the given element. The ’font-family’ descriptor that is reflected into the CSS2 properties is the font family that was requested, not whatever name the intelligently matched font may have. UAs that do not implement intelligent matching are considered to fail at this step.

4. If there is no matching font face within the ’font-family’ being processed by step 3, *UAs that implement font downloading* may proceed to examine the ’src’ descriptor of the tentative font face identified in step 2 or 3 to identify a network resource that is available, and of the correct format. If there are matches for all the remaining descriptors, then that is the matching font face for the given element and the UA may attempt to download this font resource. The UA may choose to block on this download or may choose to proceed to the next step while the font downloads. UAs that do not implement font download, or are not connected to a network, or where the user preferences have disabled font download, or where the requested resource is unavailable for whatever reason, or where the downloaded font cannot be used for whatever reason, are considered to fail at this step.
5. If there is no matching font face within the ’font-family’ being processed by step 3, *UAs that implement font synthesis* may proceed to examine other descriptors such as ’x-height’, glyph widths, and ’panose-1’ to identify a different tentative font family for synthesis. If there are matches for all the remaining descriptors, then that is the matching font face for the given element and synthesis of the faux font may begin. UAs that do not implement font synthesis are considered to fail at this step.
6. If all of steps 3, 4 and 5 fail, and if there is a next alternative ’font-family’ in the font set, then repeat from step 2 with the next alternative ’font-family’.
7. If there is a matching font face, but it doesn’t contain glyph(s) for the current character(s), and if there is a next alternative ’font-family’ in the font sets, then repeat from step 2 with the next alternative ’font-family’. The ’unicode-range’ descriptor may be used to rapidly eliminate from consideration those font faces that do not have the correct glyphs. If the ’unicode-range’ descriptor indicates that a font contains some glyphs in the correct range, it may be examined by the UA to see if it has that particular one.
8. If there is no font within the family selected in 2, then use the inherited or UA-dependent ’font-family’ value and repeat from step 2, using the best match that can be obtained within this font. If a particular character cannot be displayed using this font, the UA should indicate that a character is not being displayed (for example, using the ’missing character’ glyph).
9. UAs that implement progressive rendering and have pending font downloads may, once download is successful, use the downloaded font as a font family. If the downloaded font is missing some glyphs that the temporary progressive font did contain, the downloaded font is not used for that character and the temporary font continues to be used.

Note. *The above algorithm can be optimized to avoid having to revisit the CSS2 properties for each character.*

The per-descriptor matching rules from (2) above are as follows:

1. ’font-style’ is tried first. ’italic’ will be satisfied if there is either a face in the UA’s font database labeled with the CSS keyword ’italic’ (preferred) or ’oblique’. Otherwise the values must be matched exactly or font-style will fail.
2. ’font-variant’ is tried next. ’normal’ matches a font not labeled as ’small-caps’; ’small-caps’ matches (1) a font labeled as ’small-caps’, (2) a font in which the small caps are synthesized, or (3) a font where all lowercase letters are replaced by uppercase letters. A small-caps font may be synthesized by electronically scaling uppercase letters from a normal font.
3. ’font-weight’ is matched next, it will never fail. (See ’font-weight’ below.)
4. ’font-size’ must be matched within a UA-dependent margin of tolerance. (Typically, sizes for scalable fonts are rounded to the nearest whole pixel, while the tolerance for bitmapped fonts could be as large as 20%.) Further computations, e.g., by ’em’ values in other properties, are based on the ’font-size’ value that is used, not the one that is specified.

15.5.1 Mapping font weight values to font names

The ’font-weight’ property values are given on a numerical scale in which the value ’400’ (or ’normal’) corresponds to the "normal" text face for that family. The weight name associated with that face will typically be *Book, Regular, Roman, Normal* or sometimes *Medium*.

The association of other weights within a family to the numerical weight values is intended only to preserve the ordering of weights within that family. User agents must map names to values in a way that preserves visual order; a face mapped to a value must not be lighter than faces mapped to lower values. There is no guarantee on how a user agent will map font faces within a family to weight values. However, the following heuristics tell how the assignment is done in typical cases:

- If the font family already uses a numerical scale with nine values (as e.g., *OpenType* does), the font weights should be mapped directly.
- If there is both a face labeled *Medium* and one labeled *Book, Regular, Roman* or *Normal,* then the *Medium* is normally assigned to the ’500’.
- The font labeled "Bold" will often correspond to the weight value ’700’.
- If there are fewer then 9 weights in the family, the default algorithm for filling the "holes" is as follows. If ’500’ is unassigned, it will be assigned the same font as ’400’. If any of the values ’600’, ’700’, ’800’, or ’900’ remains unassigned, they are assigned to the same face as the next darker assigned keyword, if any, or the next lighter one otherwise. If any of ’300’, ’200’, or ’100’ remains unassigned, it is assigned to the next lighter assigned keyword, if any, or the next darker otherwise.

There is no guarantee that there will be a darker face for each of the ’font-weight’ values; for example, some fonts may have only a normal and a bold face, others may have eight different face weights.

The following two examples show typical mappings.

Assume four weights in the "Rattlesnake" family, from lightest to darkest: *Regular, Medium, Bold, Heavy.*

First example of font-weight mapping

Available faces	Assignments	Filling the holes
"Rattlesnake Regular"	400	100, 200, 300
"Rattlesnake Medium"	500	
"Rattlesnake Bold"	700	600
"Rattlesnake Heavy"	800	900

Assume six weights in the "Ice Prawn" family: *Book, Medium, Bold, Heavy, Black, ExtraBlack.* Note that in this instance the user agent has decided *not* to assign a numeric value to "Example2 ExtraBlack".

Second example of font-weight mapping

Available faces	Assignments	Filling the holes
"Ice Prawn Book"	400	100, 200, 300
"Ice Prawn Medium"	500	
"Ice Prawn Bold"	700	600
"Ice Prawn Heavy"	800	
"Ice Prawn Black"	900	
"Ice Prawn ExtraBlack"	(none)	

15.5.2 Examples of font matching

Example(s):

The following example defines a specific font face, Alabama Italic. A panose font description and source URI for retrieving a truetype server font are also provided. Font-weight and font-style descriptors are provided to describe the font. The declaration says that the weight will also match any request in the range 300 to 500. The font family is Alabama and the adorned font name is Alabama Italic.

```
@font-face {
  src: local("Alabama Italic"),
       url(http://www.fonts.org/A/alabama-italic) format("truetype");
  panose-1: 2 4 5 2 5 4 5 9 3 3;
  font-family: Alabama, serif;
  font-weight:   300, 400, 500;
  font-style:  italic, oblique;
}
```

Example(s):

The next example defines a family of fonts. A single URI is provided for retrieving the font data. This data file will contain multiple styles and weights of the named font. Once one of these @font-face definitions has been dereferenced, the data will be in the UA cache for other faces that use the same URI.

```
@font-face {
  src: local("Helvetica Medium"),
       url(http://www.fonts.org/sans/Helvetica_family) format("truedoc");
  font-family: "Helvetica";
  font-style: normal
}
@font-face {
  src: local("Helvetica Oblique"),
       url("http://www.fonts.org/sans/Helvetica_family") format("truedoc");
  font-family: "Helvetica";
  font-style: oblique;
  slope: -18
}
```

Example(s):

The following example groups three physical fonts into one virtual font with extended coverage. In each case, the adorned font name is given in the src descriptor to allow locally installed versions to be preferentially used if available. A fourth rule points to a font with the same coverage, but contained in a single resource.

```
@font-face {
  font-family: Excelsior;
  src: local("Excelsior Roman"), url("http://site/er") format("intellifont");
  unicode-range: U+??; /* Latin-1 */
}
@font-face {
  font-family: Excelsior;
  src: local("Excelsior EastA Roman"), url("http://site/ear") format("intellifont");
  unicode-range: U+100-220; /* Latin Extended A and B */
}
@font-face {
  font-family: Excelsior;
  src: local("Excelsior Cyrillic Upright"), url("http://site/ecr") format("intellifont");
  unicode-range: U+4??; /* Cyrillic */
}
@font-face {
  font-family: Excelsior;
  src: url("http://site/excels") format("truedoc");
  unicode-range: U+??,U+100-220,U+4??;
}
```

Example(s):

This next example might be found in a UA's default style sheet. It implements the CSS2 generic font family, `serif` by mapping it to a wide variety of serif fonts that might exist on various platforms. No metrics are given since these vary among the possible alternatives.

```
@font-face {
  src: local("Palatino"),
          local("Times New Roman"),
          local("New York"),
          local("Utopia"),
          url("http://somewhere/free/font");
  font-family: serif;
  font-weight: 100, 200, 300, 400, 500;
  font-style: normal;
  font-variant: normal;
  font-size: all
}
```

16 Text

Contents

The properties defined in the following sections affect the visual presentation of characters, spaces, words, and paragraphs.

16.1 Indentation: the ’text-indent’ property

’text-indent’

Value:	<length> | <percentage> | inherit
Initial:	0
Applies to:	block-level elements
Inherited:	yes
Percentages:	refer to width of containing block
Media:	visual

This property specifies the indentation of the first line of text in a block. More precisely, it specifies the indentation of the first box that flows into the block’s first line box [p. 105] . The box is indented with respect to the left (or right, for right-to-left layout) edge of the line box. User agents should render this indentation as blank space.

Values have the following meanings:

<length>
The indentation is a fixed length.

<percentage>
The indentation is a percentage of the containing block width.

The value of ’text-indent’ may be negative, but there may be implementation-specific limits.

Example(s):

The following example causes a ’3em’ text indent.

```
P { text-indent: 3em }
```

16.2 Alignment: the ’text-align’ property

’text-align’

Value:	left | right | center | justify | <string> | inherit
Initial:	depends on user agent and writing direction
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property describes how inline content of a block is aligned. Values have the following meanings:

left, **right**, **center**, and **justify**
    Left, right, center, and double justify text, respectively.
<string>
    Specifies a string on which cells in a table column will align (see the section on horizontal alignment in a column [p. 260] for details and an example). This value applies *only* to table [p. 245] cells. If set on other elements, it will be treated as ’left’ or ’right’, depending on whether ’direction’ is ’ltr’, or ’rtl’, respectively.

A block of text is a stack of line boxes [p. 105] . In the case of ’left’, ’right’ and ’center’, this property specifies how the inline boxes within each line box align with respect to the line box’s left and right sides; alignment is not with respect to the viewport [p. 96] . In the case of ’justify’, the UA may stretch the inline boxes in addition to adjusting their positions. (See also ’letter-spacing’ and ’word-spacing’.)

Example(s):

In this example, note that since ’text-align’ is inherited, all block-level elements inside the DIV element with ’class=center’ will have their inline content centered.

```
DIV.center { text-align: center }
```

***Note.** The actual justification algorithm used is user-agent and written language dependent.*

Conforming user agents [p. 32] may interpret the value ’justify’ as ’left’ or ’right’, depending on whether the element’s default writing direction is left-to-right or right-to-left, respectively.

16.3 Decoration

16.3.1 Underlining, overlining, striking, and blinking: the ’text-decoration’ property

’text-decoration’

Value:	none | [underline || overline || line-through || blink] | inherit
Initial:	none
Applies to:	all elements
Inherited:	no (see prose)
Percentages:	N/A
Media:	visual

This property describes decorations that are added to the text of an element. If the property is specified for a block-level [p. 97] element, it affects all inline-level descendants of the element. If it is specified for (or affects) an inline-level [p. 98] element, it affects all boxes generated by the element. If the element has no content or no text content (e.g., the IMG element in HTML), user agents must ignore [p. 42] this property.

Values have the following meanings:

none
Produces no text decoration.

underline
Each line of text is underlined.

overline
Each line of text has a line above it.

line-through
Each line of text has a line through the middle

blink
Text blinks (alternates between visible and invisible). Conforming user agents [p. 32] are not required to support this value.

The color(s) required for the text decoration should be derived from the ’color’ property value.

This property is not inherited, but descendant boxes of a block box should be formatted with the same decoration (e.g., they should all be underlined). The color of decorations should remain the same even if descendant elements have different ’color’ values.

Example(s):

In the following example for HTML, the text content of all A elements acting as hyperlinks will be underlined:

```
A[href] { text-decoration: underline }
```

16.3.2 Text shadows: the ’text-shadow’ property

’text-shadow’

Value:	none | [<color> || <length> <length> <length>? ,]* [<color> || <length> <length> <length>?] | inherit
Initial:	none
Applies to:	all elements
Inherited:	no (see prose)
Percentages:	N/A
Media:	visual

This property accepts a comma-separated list of shadow effects to be applied to the text of the element. The shadow effects are applied in the order specified and may thus overlay each other, but they will never overlay the text itself. Shadow effects do not alter the size of a box, but may extend beyond its boundaries. The stack level [p. 125] of the shadow effects is the same as for the element itself.

Each shadow effect must specify a shadow offset and may optionally specify a blur radius and a shadow color.

A shadow offset is specified with two <length> values that indicate the distance from the text. The first length value specifies the horizontal distance to the right of the text. A negative horizontal length value places the shadow to the left of the text. The second length value specifies the vertical distance below the text. A negative vertical length value places the shadow above the text.

A blur radius may optionally be specified after the shadow offset. The blur radius is a length value that indicates the boundaries of the blur effect. The exact algorithm for computing the blur effect is not specified.

A color value may optionally be specified before or after the length values of the shadow effect. The color value will be used as the basis for the shadow effect. If no color is specified, the value of the 'color' property will be used instead.

Text shadows may be used with the :first-letter [p. 66] and :first-line [p. 66] pseudo-elements.

Example(s):

The example below will set a text shadow to the right and below the element's text. Since no color has been specified, the shadow will have the same color as the element itself, and since no blur radius is specified, the text shadow will not be blurred:

```
H1 { text-shadow: 0.2em 0.2em }
```

The next example will place a shadow to the right and below the element's text. The shadow will have a 5px blur radius and will be red.

```
H2 { text-shadow: 3px 3px 5px red }
```

The next example specifies a list of shadow effects. The first shadow will be to the right and below the element's text and will be red with no blurring. The second shadow will overlay the first shadow effect, and it will be yellow, blurred, and placed to the left and below the text. The third shadow effect will be placed to the right and above the text. Since no shadow color is specified for the third shadow effect, the value of the element's 'color' property will be used:

```
H2 { text-shadow: 3px 3px red, yellow -3px 3px 2px, 3px -3px }
```

Example(s):
Consider this example:

```
SPAN.glow {
    background: white;
    color: white;
    text-shadow: black 0px 0px 5px;
}
```

Here, the ’background’ and ’color’ properties have the same value and the ’text-shadow’ property is used to create a "solar eclipse" effect:

Note. *This property is not defined in CSS1. Some shadow effects (such as the one in the last example) may render text invisible in UAs that only support CSS1.*

16.4 Letter and word spacing: the ’letter-spacing’ and ’word-spacing’ properties

’letter-spacing’

Value:	normal | <length> | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies spacing behavior between text characters. Values have the following meanings:

normal
The spacing is the normal spacing for the current font. This value allows the user agent to alter the space between characters in order to justify text.

<length>
This value indicates inter-character space *in addition to* the default space between characters. Values may be negative, but there may be implementation-specific limits. User agents may not further increase or decrease the inter-character space in order to justify text.

Character spacing algorithms are user agent-dependent. Character spacing may also be influenced by justification (see the ’text-align’ property).
Example(s):
In this example, the space between characters in BLOCKQUOTE elements is increased by ’0.1em’.

```
BLOCKQUOTE { letter-spacing: 0.1em }
```

In the following example, the user agent is not permitted to alter inter-character space:

```
BLOCKQUOTE { letter-spacing: 0cm }   /* Same as ’0’ */
```

When the resultant space between two characters is not the same as the default space, user agents should not use ligatures.

Conforming user agents [p. 32] may consider the value of the ’letter-spacing’ property to be ’normal’.

’word-spacing’

Value:	normal | <length> | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies spacing behavior between words. Values have the following meanings:

normal
The normal inter-word space, as defined by the current font and/or the UA.

<length>
This value indicates inter-word space *in addition to* the default space between words. Values may be negative, but there may be implementation-specific limits.

Word spacing algorithms are user agent-dependent. Word spacing is also influenced by justification (see the ’text-align’ property).

Example(s):

In this example, the word-spacing between each word in H1 elements is increased by ’1em’.

```
H1 { word-spacing: 1em }
```

Conforming user agents [p. 32] may consider the value of the ’word-spacing’ property to be ’normal’.

16.5 Capitalization: the ’text-transform’ property

’text-transform’

Value:	capitalize | uppercase | lowercase | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property controls capitalization effects of an element’s text. Values have the following meanings:

capitalize
: Puts the first character of each word in uppercase.

uppercase
: Puts all characters of each word in uppercase.

lowercase
: Puts all characters of each word in lowercase.

none
: No capitalization effects.

The actual transformation in each case is written language dependent. See RFC 2070 ([RFC2070]) for ways to find the language of an element.

Conforming user agents [p. 32] may consider the value of ’text-transform’ to be ’none’ for characters that are not from the Latin-1 repertoire and for elements in languages for which the transformation is different from that specified by the case-conversion tables of ISO 10646 ([ISO10646]).

Example(s):

In this example, all text in an H1 element is transformed to uppercase text.

```
H1 { text-transform: uppercase }
```

16.6 Whitespace: the ’white-space’ property

’white-space’

Value:	normal | pre | nowrap | inherit
Initial:	normal
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property declares how whitespace [p. 37] inside the element is handled. Values have the following meanings:

normal
: This value directs user agents to collapse sequences of whitespace, and break lines as necessary to fill line boxes. Additional line breaks may be created by occurrences of "\A" in generated content (e.g., for the BR element in HTML).

pre
: This value prevents user agents from collapsing sequences of whitespace. Lines are only broken at newlines in the source, or at occurrences of "\A" in generated content.

nowrap
: This value collapses whitespace as for ’normal’, but suppresses line breaks within text except for those created by "\A" in generated content (e.g., for the BR element in HTML).

Example(s):

The following examples show what whitespace [p. 37] behavior is expected from the PRE and P elements, and the "nowrap" attribute in HTML.

```
PRE        { white-space: pre }
P          { white-space: normal }
TD[nowrap] { white-space: nowrap }
```

Conforming user agents [p. 32] may ignore [p. 42] the ’white-space’ property in author and user style sheets but must specify a value for it in the default style sheet.

---

17 Tables

Contents

17.1 Introduction to tables

Tables represent relationships between data. Authors specify these relationships in the document language [p. 30] and specify their *presentation* in CSS, in two ways: visually and aurally.

Authors may specify the visual formatting of a table as a rectangular grid of cells. Rows and columns of cells may be organized into row groups and column groups. Rows, columns, row groups, row columns, and cells may have borders drawn around them (there are two border models in CSS2). Authors may align data vertically or horizontally within a cell and align data in all cells of a row or column.

Authors may also specify the aural rendering of a table; how headers and data will be spoken. In the document language, authors may label cells and groups of cells so that when rendered aurally, cell headers are spoken before cell data. In effect, this "serializes" the table: users browsing the table aurally hear a sequence of headers followed by data.

Example(s):
Here is a simple three-row, three-column table described in HTML 4.0:

```
<TABLE>
<CAPTION>This is a simple 3x3 table</CAPTION>
<TR id="row1">
   <TH>Header 1      <TD>Cell 1        <TD>Cell 2
<TR id="row2">
   <TH>Header 2      <TD>Cell 3        <TD>Cell 4
<TR id="row3">
   <TH>Header 3      <TD>Cell 5        <TD>Cell 6
</TABLE>
```

This code creates one table (the TABLE element), three rows (the TR elements), three header cells (the TH elements), and six data cells (the TD elements). Note that the three columns of this example are specified implicitly: there are as many columns in the table as required by header and data cells.
The following CSS rule centers the text horizontally in the header cells and present the data with a bold font weight:

```
TH { text-align: center; font-weight: bold }
```

The next rules align the text of the header cells on their baseline and vertically centers the text in each data cell:

```
TH { vertical-align: baseline }
TD { vertical-align: middle }
```

The next rules specify that the top row will be surrounded by a 3px solid blue border and each of the other rows will be surrounded by a 1px solid black border:

```
TABLE   { border-collapse: collapse }
TR#row1 { border-top: 3px solid blue }
TR#row2 { border-top: 1px solid black }
TR#row3 { border-top: 1px solid black }
```

Note, however, that the borders around the rows overlap where the rows meet. What color (black or blue) and thickness (1px or 3px) will the border between row1 and row2 be? We discuss this in the section on border conflict resolution. [p. 264]
The following rule puts the table caption above the table:

```
CAPTION { caption-side: top }
```

Finally, the following rule specifies that, when rendered aurally, each row of data is to be spoken as a "Header, Data, Data":

```
TH { speak-header: once }
```

For instance, the first row would be spoken "Header1 Cell1 Cell2". On the other hand, with the following rule:

```
TH { speak-header: always }
```

it would be spoken "Header1 Cell1 Header1 Cell2".
The preceding example shows how CSS works with HTML 4.0 elements; in HTML 4.0, the semantics of the various table elements (TABLE, CAPTION, THEAD, TBODY, TFOOT, COL, COLGROUP, TH, and TD) are well-defined. In

other document languages (such as XML applications), there may not be pre-defined table elements. Therefore, CSS2 allows authors to "map" document language elements to table elements via the ’display’ property. For example, the following rule makes the FOO element act like an HTML TABLE element and the BAR element act like a CAPTION element:

```
FOO { display : table }
BAR { display : table-caption }
```

We discuss the various table elements in the following section. In this specification, the term *table element* refers to any element involved in the creation of a table. An "internal" table element is one that produces a row, row group, column, column group, or cell.

17.2 The CSS table model

The CSS table model is based on the HTML 4.0 table model, in which the structure of a table closely parallels the visual layout of the table. In this model, a table consists of an optional caption and any number of rows of cells. The table model is said to be "row primary" since authors specify rows, not columns, explicitly in the document language. Columns are derived once all the rows have been specified -- the first cell of each row belongs to the first column, the second to the second column, etc.). Rows and columns may be grouped structurally and this grouping reflected in presentation (e.g., a border may be drawn around a group of rows).

Thus, the table model consists of tables, captions, rows, row groups, columns, column groups, and cells.

The CSS model does not require that the document language [p. 30] include elements that correspond to each of these components. For document languages (such as XML applications) that do not have pre-defined table elements, authors must map document language elements to table elements; this is done with the ’display’ property. The following ’display’ values assign table semantics to an arbitrary element:

table (In HTML: TABLE)
: Specifies that an element defines a block-level [p. 97] table: it is a rectangular block that participates in a block formatting context [p. 105] .

inline-table (In HTML: TABLE)
: Specifies that an element defines an inline-level [p. 98] table: it is a rectangular block that participates in an inline formatting context [p. 105]).

table-row (In HTML: TR)
: Specifies that an element is a row of cells.

table-row-group (In HTML: TBODY)
: Specifies that an element groups one or more rows.

table-header-group (In HTML: THEAD)
: Like ’table-row-group’, but for visual formatting, the row group is always displayed before all other rows and rowgroups and after any top captions. Print user agents may repeat footer rows on each page spanned by a table.

table-footer-group (In HTML: TFOOT)
: Like ’table-row-group’, but for visual formatting, the row group is always displayed after all other rows and rowgroups and before any bottom

captions. Print user agents may repeat footer rows on each page spanned by a table.

table-column (In HTML: COL)
Specifies that an element describes a column of cells.

table-column-group (In HTML: COLGROUP)
Specifies that an element groups one or more columns.

table-cell (In HTML: TD, TH)
Specifies that an element represents a table cell.

table-caption (In HTML: CAPTION)
Specifies a caption for the table.

Elements with ’display’ set to ’table-column’ or ’table-column-group’ are not rendered (exactly as if they had ’display: none’), but they are useful, because they may have attributes which induce a certain style for the columns they represent.

The default style sheet for HTML 4.0 [p. 291] in the appendix illustrates the use of these values for HTML 4.0:

```
TABLE    { display: table }
TR       { display: table-row }
THEAD    { display: table-header-group }
TBODY    { display: table-row-group }
TFOOT    { display: table-footer-group }
COL      { display: table-column }
COLGROUP { display: table-column-group }
TD, TH   { display: table-cell }
CAPTION  { display: table-caption }
```

User agents may ignore [p. 42] these ’display’ property values for HTML documents, since authors should not alter an element’s expected behavior.

17.2.1 Anonymous table objects

Document languages other than HTML may not contain all the elements in the CSS2 table model. In these cases, the "missing" elements must be assumed in order for the table model to work. The missing elements generate anonymous [p. 98] objects (e.g., anonymous boxes in visual table layout) according to the following rules:

1. Any table element will automatically generate necessary anonymous table objects around itself, consisting of at least three nested objects corresponding to a ’table’/’inline-table’ element, a ’table-row’ element, and a ’table-cell’ element.
2. If the parent P of a ’table-cell’ element T is not a ’table-row’, an object corresponding to a ’table-row’ will be generated between P and T. This object will span all consecutive ’table-cell’ siblings (in the document tree) of T.
3. If the parent P of a ’table-row’ element T is not a ’table’, ’inline-table’, or ’table-row-group’ element, an object corresponding to a ’table’ element will be generated between P and T. This object will span all consecutive siblings (in the document tree) of T that require a ’table’ parent: ’table-row’, ’table-row-group’, ’table-header-group’, ’table-footer-group’, ’table-column’, ’table-column-group’, and ’caption’.
4. If the parent P of a ’table-row-group’ (or ’table-header-group’ or

’table-footer-group’) element T is not a ’table’ or ’inline-table’, an object corresponding to a ’table’ element will be generated between P and T. This object will span all consecutive siblings (in the document tree) of T that require a ’table’ parent: ’table-row’, ’table-row-group’, ’table-header-group’, ’table-footer-group’, ’table-column’, ’table-column-group’, and ’caption’.

5. If a child T of a ’table-row’ element P is not a ’table-cell’ element, an object corresponding to a ’table-cell’ element will be generated between P and T. This object spans all consecutive siblings of T that are not ’table-cell’ elements.

Example(s):
In this XML example, a ’table’ element is assumed to contain the HBOX element:

```
<HBOX>
  <VBOX>George</VBOX>
  <VBOX>4287</VBOX>
  <VBOX>1998</VBOX>
</HBOX>
```

because the associated style sheet is:

```
HBOX { display: table-row }
VBOX { display: table-cell }
```

Example(s):
In this example, three ’table-cell’ elements are assumed to contain the text in the ROWs. Note that the text is further encapsulated in anonymous inline boxes, as explained in visual formatting model [p. 98] :

```
<STACK>
  <ROW>This is the <D>top</D> row.</ROW>
  <ROW>This is the <D>middle</D> row.</ROW>
  <ROW>This is the <D>bottom</D> row.</ROW>
</STACK>
```

The style sheet is:

```
STACK { display: inline-table }
ROW   { display: table-row }
D     { display: inline; font-weight: bolder }
```

HTML user agents are not required to create anonymous objects according to the above rules.

17.3 Column selectors

Table cells may belong to two contexts: rows and columns. However, in the source document cells are descendants of rows, never of columns. Nevertheless, some aspects of cells can be influenced by setting properties on columns.
The following properties apply to column and column-group elements:

’border’
The various border properties apply to columns only if ’border-collapse’ is set to ’collapse’ on the table element. In that case, borders set on columns and column groups are input to the conflict resolution algorithm [p. 264] that

selects the border styles at every cell edge.

’background’
: The background properties set the background for cells in the column, but only if both the cell and row have transparent backgrounds. See "Table layers and transparency." [p. 254]

’width’
: The ’width’ property gives the minimum width for the column.

’visibility’
: If the ’visibility’ of a column is set to ’collapse’, none of the cells in the column are rendered, and cells that span into other columns are clipped. In addition, the width of the table is diminished by the width the column would have taken up. See "Dynamic effects" [p. 261] below. Other values for ’visibility’ have no effect.

Example(s):

Here are some examples of style rules that set properties on columns. The first two rules together implement the "rules" attribute of HTML 4.0 with a value of "cols". The third rule makes the "totals" column blue, the final two rules shows how to make a column a fixed size, by using the fixed layout algorithm [p. 257] .

```
COL   { border-style: none solid }
TABLE { border-style: hidden }
COL.totals { background: blue }
TABLE { table-layout: fixed }
COL.totals { width: 5em }
```

17.4 Tables in the visual formatting model

In terms of the visual formatting model [p. 95] , a table may behave like a block-level [p. 97] or replaced inline-level [p. 98] element. Tables have content, padding, borders, and margins.

In both cases, the table element generates an anonymous [p. 98] box that contains the table box itself and the caption’s box (if present). The table and caption boxes retain their own content, padding, margin, and border areas, and the dimensions of the rectangular anonymous box are the smallest required to contain both. Vertical margins collapse where the table box and caption box touch. Any repositioning of the table must move the entire anonymous box, not just the table box, so that the caption follows the table.

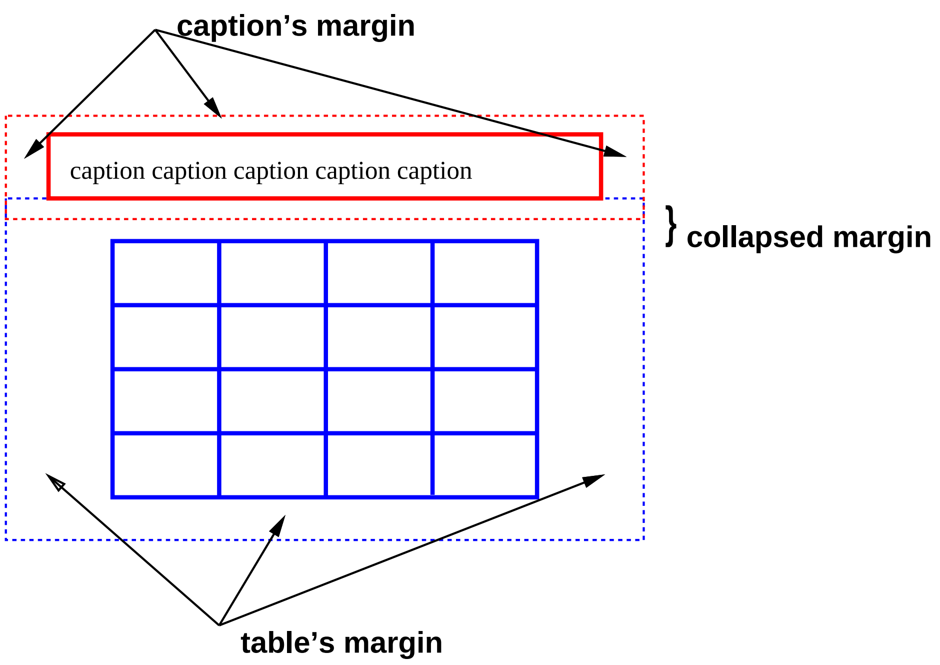


Diagram of a table with a caption above it; the bottom margin of the caption is collapsed with the top margin of the table.

17.4.1 Caption position and alignment

'caption-side'

Value:	top | bottom | left | right | inherit
Initial:	top
Applies to:	'table-caption' elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property specifies the position of the caption box with respect to the table box. Values have the following meanings:

top
Positions the caption box above the table box.

bottom
Positions the caption box below the table box.

left
Positions the caption box to the left of the table box.

right
Positions the caption box to the right of the table box.

Captions above or below a 'table' element are formatted very much as if they were a block element before or after the table, except that (1) they inherit inheritable properties from the table, and (2) they are not considered to be a block box for the purposes of any 'compact' or 'run-in' element that may precede the table.

A caption that is above or below a table box also behaves like a block box for width calculations; the width is computed with respect to the width of the table box’s containing block.

For a caption that is on the left or right side of a table box, on the other hand, a value other than ’auto’ for ’width’ sets the width explicitly, but ’auto’ tells the user agent to chose a "reasonable width". This may vary between "the narrowest possible box" to "a single line", so we recommend that users do not specify ’auto’ for left and right caption widths.

To align caption content horizontally within the caption box, use the ’text-align’ property. For vertical alignment of a left or right caption box with respect to the table box, use the ’vertical-align’ property. The only meaningful values in this case are ’top’, ’middle’, and ’bottom’. All other values are treated the same as ’top’.

Example(s):

In this example, the ’caption-side’ property places captions below tables. The caption will be as wide as the parent of the table, and caption text will be left-justified.

```
CAPTION { caption-side: bottom;
          width: auto;
          text-align: left }
```

Example(s):

The following example shows how to put a caption in the left margin. The table itself is centered, by setting its left and right margins to ’auto’, and the whole box with table and caption is shifted into the left margin by the same amount as the width of the caption.

```
BODY {
    margin-left: 8em
}
TABLE {
    margin-left: auto;
    margin-right: auto
}
CAPTION {
    caption-side: left;
    margin-left: -8em;
    width: 8em;
    text-align: right;
    vertical-align: bottom
}
```

Assuming the width of the table is less than the available width, the formatting will be similar to this:

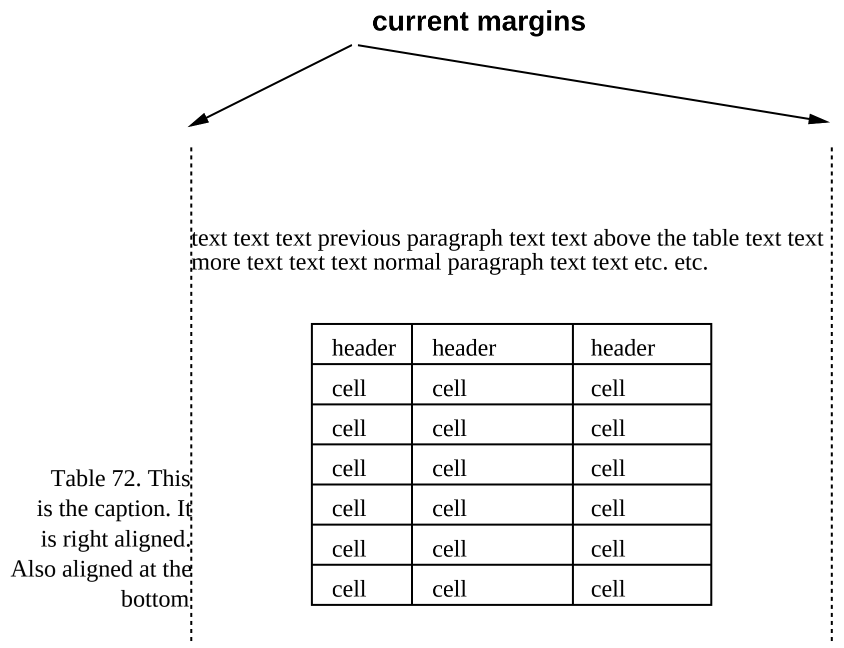

current margins
text text text previous paragraph text text above the table text text more text text text normal paragraph text text etc. etc.
header
header
header
cell
Table 72. This is the caption. It is right aligned. Also aligned at the bottom.

Diagram showing a centered table with the caption extending into the left margin, as a result of a negative 'margin-left' property.

17.5 Visual layout of table contents

Like other elements of the document language [p. 30] , internal table elements generate rectangular boxes [p. 81] with content, padding, and borders. They do not have margins, however.

The visual layout of these boxes is governed by a rectangular, irregular grid of rows and columns. Each box occupies a whole number of grid cells, determined according to the following rules. These rules do not apply to HTML 4.0 or earlier HTML versions; HTML imposes its own limitations on row and column spans.

1. Each row box occupies one row of grid cells. Together, the row boxes fill the table from top to bottom in the order they occur in the source document (i.e., the table occupies exactly as many grid rows as there are row elements).
2. A row group occupies the same grid cells as the rows it contains.
3. A column box occupies one or more columns of grid cells. Column boxes are placed next to each other in the order they occur. The first column box may be either on the left or on the right, depending on the value of the 'direction' property of the table.
4. A column group box occupies the same grid cells as the columns it contains.
5. Cells may span several rows or columns. (Although CSS2 doesn't define how the number of spanned rows or columns is determined, a user agent may have special knowledge about the source document; a future version of CSS may provide a way to express this knowledge in CSS syntax.) Each cell is thus a rectangular box, one or more grid cells wide and high. The top row of this rectangle is in the row specified by the cell's parent. The rectangle must be as far to the left as possible, but it may not overlap with any other cell box, and must be to the right of all cells in the same row that are earlier in the source document. (This constraint holds if the 'direction' property of

the table is 'ltr'; if the 'direction' is 'rtl', interchange "left" and "right" in the previous sentence.)

6. A cell box cannot extend beyond the last row box of a table or row-group; the user agents must shorten it until it fits.

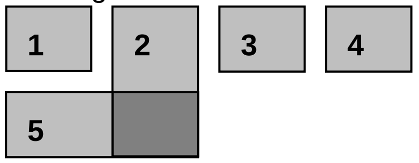
Note. Table cells may be relatively and absolutely positioned, but this is not recommended: positioning and floating remove a box from the flow, affecting table alignment.

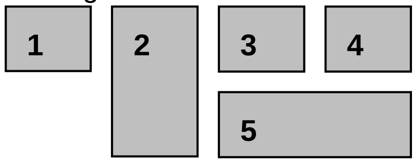
Here are two examples. The first is assumed to occur in an HTML document:

```
<TABLE>
<TR><TD>1 <TD rowspan="2">2 <TD>3 <TD>4
<TR><TD colspan="2">5
</TABLE>
```

```
<TABLE>
<ROW><CELL>1 <CELL rowspan="2">2 <CELL>3 <CELL>4
<ROW><CELL colspan="2">5
</TABLE>
```

The second table is formatted as in the figure on the right. However, the HTML table's rendering is explicitly undefined by HTML, and CSS doesn't try to define it. User agents are free to render it, e.g., as in the figure on the left.

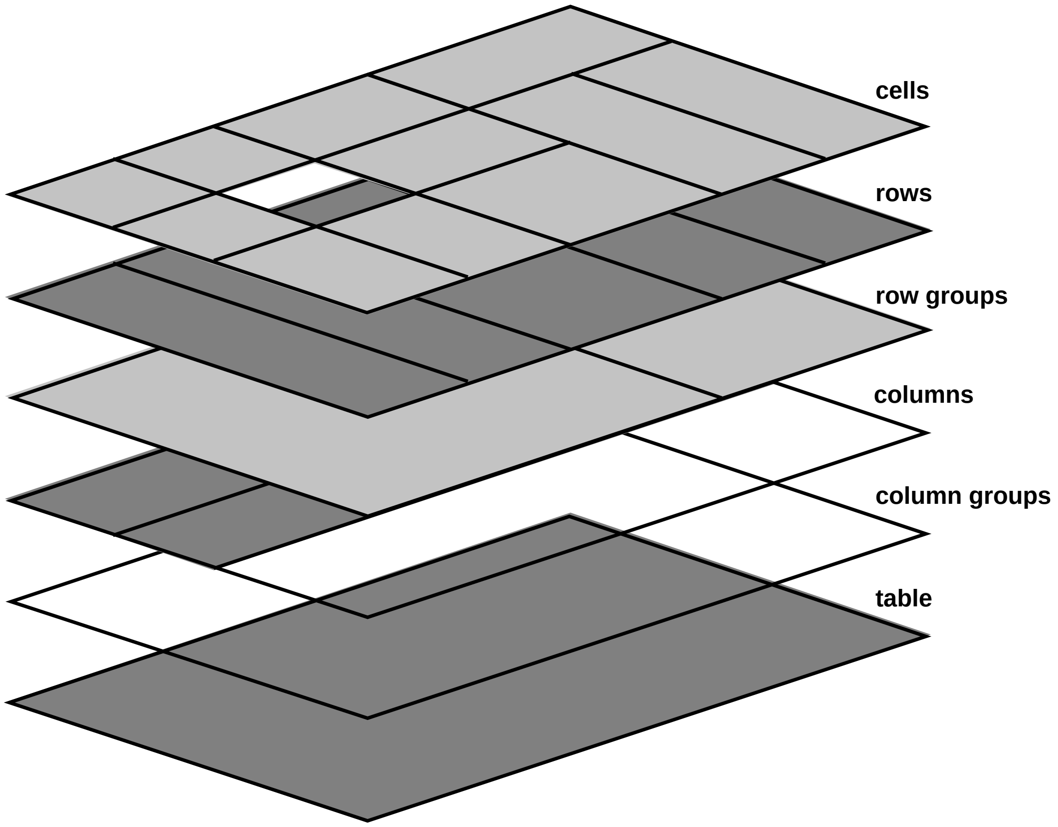




On the left, one possible rendering of an erroneous HTML 4.0 table; on the right, the only possible formatting of a similar, non-HTML table.

17.5.1 Table layers and transparency

For the purposes of finding the background of each table cell, the different table elements may be thought of as being on six superimposed layers. The background set on an element in one of the layers will only be visible if the layers above it have a transparent background.



Schema of table layers.

1. The lowest layer is a single plane, representing the table box itself. Like all boxes, it may be transparent.
2. The next layer contains the column groups. The columns groups are as tall as the table, but they need not cover the whole table horizontally.
3. On top of the column groups are the areas representing the column boxes. Like column groups, columns are as tall as the table, but need not cover the whole table horizontally.
4. Next is the layer containing the row groups. Each row group is as wide as the table. Together, the row groups completely cover the table from top to bottom.
5. The next to last layer contains the rows. The rows also cover the whole table.
6. The topmost layer contains the cells themselves. As the figure shows, although all rows contain the same number of cells, not every cell may have specified content. These "empty" cells are transparent, letting lower layers shine through.

In the following example, the first row contains four cells, but the second row contains no cells, and thus the table background shines through, except where a cell from the first row spans into this row. The following HTML code and style rules

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <STYLE type="text/css">
```

```
      TABLE { background: #ff0; border-collapse: collapse }
      TD    { background: red; border: double black }
    </STYLE>
  </HEAD>
  <BODY>
    <P>
    <TABLE>
      <TR>
        <TD> 1
        <TD rowspan="2"> 2
        <TD> 3
        <TD> 4
      </TR>
      <TR><TD></TD></TR>
    </TABLE>
  </BODY>
</HTML>
```

might be formatted as follows:

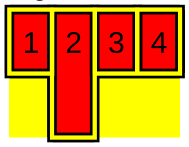


Table with three empty cells in the bottom row.

17.5.2 Table width algorithms: the ’table-layout’ property

CSS does not define an "optimal" layout for tables since, in many cases, what is optimal is a matter of taste. CSS does define constraints that user agents must respect when laying out a table. User agents may use any algorithm they wish to do so, and are free to prefer rendering speed over precision, except when the "fixed layout algorithm" is selected.

’table-layout’

Value:	auto | fixed | inherit
Initial:	auto
Applies to:	’table’ and ’inline-table’ elements
Inherited:	no
Percentages:	N/A
Media:	visual

The ’table-layout’ property controls the algorithm used to lay out the table cells, rows, and columns. Values have the following meaning:

fixed
Use the fixed table layout algorithm

auto
Use any automatic table layout algorithm

The two algorithms are described below.

Fixed table layout

With this (fast) algorithm, the horizontal layout of the table does not depend on the contents of the cells; it only depends on the table's width, the width of the columns, and borders or cell spacing.

The table's width may be specified explicitly with the 'width' property. A value of 'auto' (for both 'display: table' and 'display: inline-table') means use the automatic table layout [p. 257] algorithm.

In the fixed table layout algorithm, the width of each column is determined as follows:

1. A column element with a value other than 'auto' for the 'width' property sets the width for that column.
2. Otherwise, a cell in the first row with a value other than 'auto' for the 'width' property sets the width for that column. If the cell spans more than one column, the width is divided over the columns.
3. Any remaining columns equally divide the remaining horizontal table space (minus borders or cell spacing).

The width of the table is then the greater of the value of the 'width' property for the table element and the sum of the column widths (plus cell spacing or borders). If the table is wider than the columns, the extra space should be distributed over the columns.

In this manner, the user agent can begin to lay out the table once the entire first row has been received. Cells in subsequent rows do not affect column widths. Any cell that has content that overflows uses the 'overflow' property to determine whether to clip the overflow content.

Automatic table layout

In this algorithm (which generally requires no more than two passes), the table's width is given by the width of its columns (and intervening borders [p. 261]). This algorithm reflects the behavior of several popular HTML user agents at the writing of this specification. UAs are not required to implement this algorithm to determine the table layout in the case that 'table-layout' is 'auto'; they can use any other algorithm.

This algorithm may be inefficient since it requires the user agent to have access to all the content in the table before determining the final layout and may demand more than one pass.

Column widths are determined as follows:

1. Calculate the minimum content width (MCW) of each cell: the formatted content may span any number of lines but may not overflow the cell box. If the specified 'width' (W) of the cell is greater than MCW, W is the minimum cell width. A value of 'auto' means that MCW is the minimum cell width.

   Also, calculate the "maximum" cell width of each cell: formatting then content without breaking lines other than where explicit line breaks occur.
2. For each column, determine a maximum and minimum column width from the cells that span only that column. The minimum is that required by the cell with the largest minimum cell width (or the column 'width', whichever is

larger). The maximum is that required by the cell with the largest maximum cell width (or the column ’width’, whichever is larger).

3. For each cell that spans more than one column, increase the minimum widths of the columns it spans so that together, they are at least as wide as the cell. Do the same for the maximum widths. If possible, widen all spanned columns by approximately the same amount.

This gives a maximum and minimum width for each column. Column widths influence the final table width as follows:

1. If the ’table’ or ’inline-table’ element’s ’width’ property has a specified value (W) other than ’auto’, the property’s computed value is the greater of W and the minimum width required by all the columns plus cell spacing or borders (MIN). If W is greater than MIN, the extra width should be distributed over the columns.
2. If the ’table’ or ’inline-table’ element has ’width: auto’, the computed table width is the greater of the table’s containing block width and MIN. However, if the maximum width required by the columns plus cell spacing or borders (MAX) is less than that of the containing block, use MAX.

A percentage value for a column width is relative to the table width. If the table has ’width: auto’, a percentage represents a constraint on the column’s width, which a UA should try to satisfy. (Obviously, this is not always possible: if the column’s width is ’110%’, the constraint cannot be satisfied.)

Note. *In this algorithm, rows (and row groups) and columns (and column groups) both constrain and are constrained by the dimensions of the cells they contain. Setting the width of a column may indirectly influence the height of a row, and vice versa.*

17.5.3 Table height algorithms

The height of a table is given by the ’height’ property for the ’table’ or ’inline-table’ element. A value of ’auto’ means that the height is the sum of the row heights plus any cell spacing or borders. Any other value specifies the height explicitly; the table may thus be taller or shorter than the height of its rows. CSS2 does not specify rendering when the specified table height differs from the content height, in particular whether content height should override specified height; if it doesn’t, how extra space should be distributed among rows that add up to less than the specified table height; or, if the content height exceeds the specified table height, whether the UA should provide a scrolling mechanism. **Note.** Future versions of CSS may specify this further.

The height of a ’table-row’ element’s box is calculated once the user agent has all the cells in the row available: it is the maximum of the row’s specified ’height’ and the minimum height (MIN) required by the cells. A ’height’ value of ’auto’ for a ’table-row’ means the computed row height is MIN. MIN depends on cell box heights and cell box alignment (much like the calculation of a line box [p. 141] height). CSS2 does not define what percentage values of ’height’ refer to when specified for table rows and row groups.

In CSS2, the height of a cell box is the maximum of the table cell’s ’height’ property and the minimum height required by the content (MIN). A value of ’auto’ for ’height’ implies a computed value of MIN. CSS2 does not define what percentage values of 'height' refer to when specified for table cells.

CSS2 does not specify how cells that span more than row affect row height calculations except that the sum of the row heights involved must be great enough to encompass the cell spanning the rows.

The 'vertical-align' property of each table cell determines its alignment within the row. Each cell's content has a baseline, a top, a middle, and a bottom, as does the row itself. In the context of tables, values for 'vertical-align' have the following meanings:

baseline
: The baseline of the cell is put at the same height as the baseline of the first of the rows it spans (see below for the definition of baselines of cells and rows).

top
: The top of the cell box is aligned with the top of the first row it spans.

bottom
: The bottom of the cell box is aligned with the bottom of the last row it spans.

middle
: The center of the cell is aligned with the center of the rows it spans.

sub, super, text-top, text-bottom
: These values do not apply to cells; the cell is aligned at the baseline instead.

The baseline of a cell is the baseline of the first line box [p. 105] in the cell. If there is no text, the baseline is the baseline of whatever object is displayed in the cell, or, if it has none, the bottom of the cell box. The maximum distance between the top of the cell box and the baseline over all cells that have 'vertical-align: baseline' is used to set the baseline of the row. Here is an example:

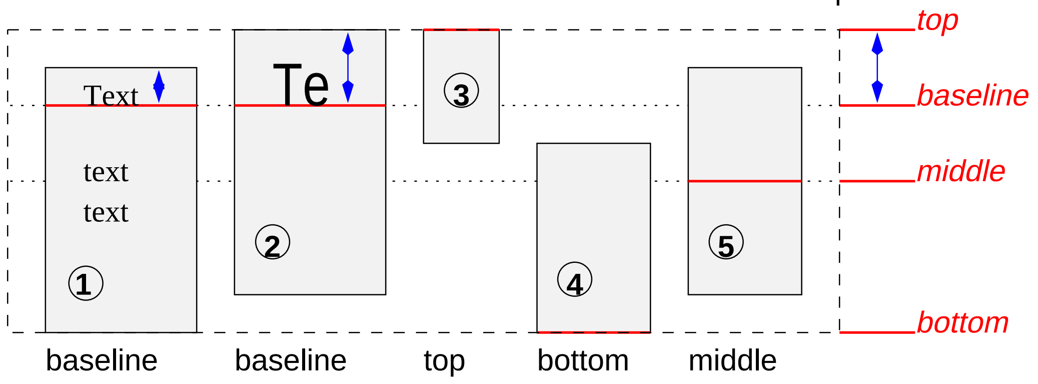


Diagram showing the effect of various values of 'vertical-align' on table cells.

Cell boxes 1 and 2 are aligned at their baselines. Cell box 2 has the largest height above the baseline, so that determines the baseline of the row. Note that if there is no cell box aligned at its baseline, the row will not have (nor need) a baseline.

To avoid ambiguous situations, the alignment of cells proceeds in the following order:

1. First the cells that are aligned on their baseline are positioned. This will establish the baseline of the row. Next the cells with 'vertical-align: top' are positioned.
2. The row now has a top, possibly a baseline, and a provisional height, which is the distance from the top to the lowest bottom of the cells positioned so

far. (See conditions on the cell padding below.)

3. If any of the remaining cells, those aligned at the bottom or the middle, have a height that is larger than the current height of the row, the height of the row will be increased to the maximum of those cells, by lowering the bottom.
4. Finally the remaining cells are positioned.

Cell boxes that are smaller than the height of the row receive extra top or bottom padding.

17.5.4 Horizontal alignment in a column

The horizontal alignment of a cell’s content within a cell box is specified with the ’text-align’ property.

When the ’text-align’ property for more than one cell in a column is set to a <string> value, the content of those cells is aligned along a vertical axis. The beginning of the string touches this axis. Character directionality determines whether the string lies to the left or right of the axis.

Aligning text in this way is only useful if the text fits on one line. The result is undefined if the cell content spans more than one line.

If value of ’text-align’ for a table cell is a string but the string doesn’t occur in the cell content, the end of the cell’s content touches the vertical axis of alignment.

Note that the strings do not have to be the same for each cell, although they usually are.

CSS does not provide a way specify the offset of the vertical alignment axis with respect to the edge of a column box.

Example(s):

The following style sheet:

```
TD { text-align: "." }
TD:before { content: "$" }
```

will cause the column of dollar figures in the following HTML table:

```
<TABLE>
<COL width="40">
<TR> <TH>Long distance calls
<TR> <TD> 1.30
<TR> <TD> 2.50
<TR> <TD> 10.80
<TR> <TD> 111.01
<TR> <TD> 85.
<TR> <TD> 90
<TR> <TD> .05
<TR> <TD> .06
</TABLE>
```

to align along the decimal point. For fun, we have used the :before pseudo-element to insert a dollar sign before each figure. The table might be rendered as follows:

```
 Long distance calls
               $1.30
               $2.50
              $10.80
             $111.01
              $85.
              $90
                $.05
                $.06
```

17.5.5 Dynamic row and column effects

The ’visibility’ property takes the value ’collapse’ for row, row group, column, and column group elements. This value causes the entire row or column to be removed from the display, and the space normally taken up by the row or column to be made available for other content. The suppression of the row or column, however, does not otherwise affect the layout of the table. This allows dynamic effects to remove table rows or columns without forcing a re-layout of the table in order to account for the potential change in column constraints.

17.6 Borders

There are two distinct models for setting borders on table cells in CSS. One is most suitable for so-called separated borders around individual cells, the other is suitable for borders that are continuous from one end of the table to the other. Many border styles can be achieved with either model, so it is often a matter of taste which one is used.

’border-collapse’

Value:	collapse | separate | inherit
Initial:	collapse
Applies to:	’table’ and ’inline-table’ elements
Inherited:	yes
Percentages:	N/A
Media:	visual

This property selects a table’s border model. The value ’separate’ selects the separated borders border model. The value ’collapse’ selects the collapsing borders model. The models are described below.

17.6.1 The separated borders model

’border-spacing’

Value:	<length> <length>? | inherit
Initial:	0
Applies to:	’table’ and ’inline-table’ elements
Inherited:	yes
Percentages:	N/A
Media:	visual

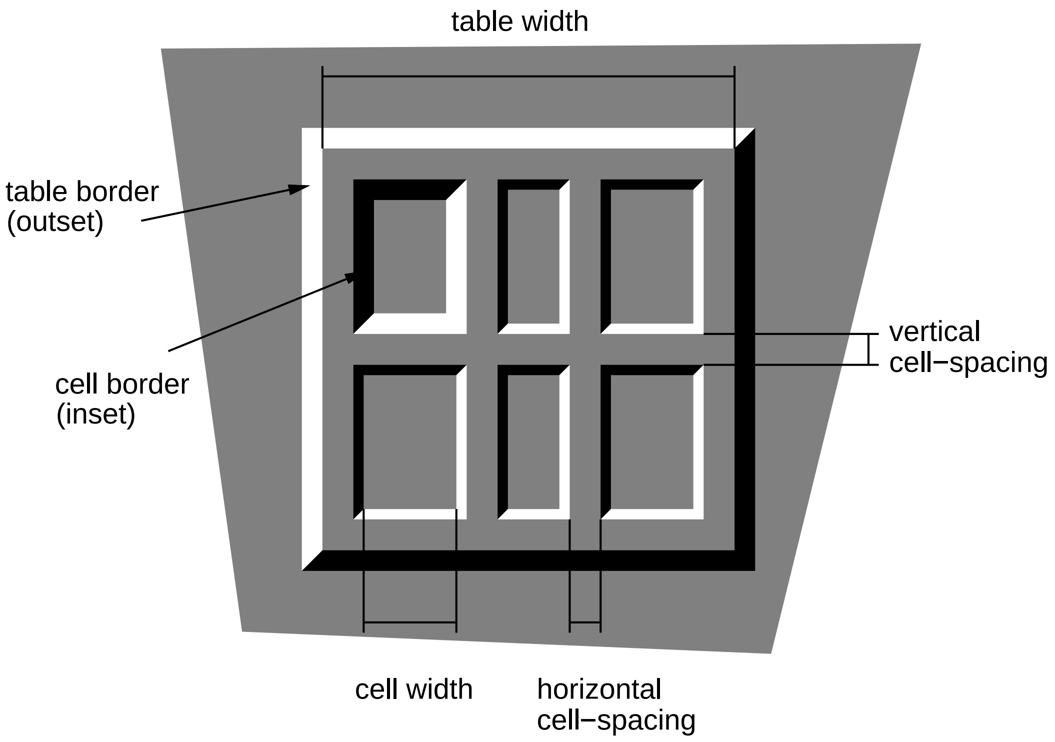
The lengths specify the distance that separates adjacent cell borders. If one length is specified, it gives both the horizontal and vertical spacing. If two are specified, the first gives the horizontal spacing and the second the vertical spacing. Lengths may not be negative.

In this model, each cell has an individual border. The ’border-spacing’ property specifies the distance between the borders of adjacent cells. This space is filled with the background of the table element. Rows, columns, row groups, and column groups cannot have borders (i.e., user agents must ignore [p. 42] the border properties for those elements).

Example(s):

The table in the figure below could be the result of a style sheet like this:

```
TABLE      { border: outset 10pt;
             border-collapse: separate;
             border-spacing: 15pt }
TD         { border: inset 5pt }
TD.special { border: inset 10pt }  /* The top-left cell */
```



A table with ’border-spacing’ set to a length value. Note that each cell has its own border, and the table has a separate border as well.

Borders around empty cells: the ’empty-cells’ property

’empty-cells’

Value:	show | hide | inherit
Initial:	show
Applies to:	’table-cell’ elements
Inherited:	yes
Percentages:	N/A
Media:	visual

In the separated borders model, this property controls the rendering of borders around cells that have no visible content. Empty cells and cells with the 'visibility' property set to 'hidden' are considered to have no visible content. Visible content includes "&nbsp;" and other whitespace except ASCII CR ("\0D"), LF ("\0A"), tab ("\09"), and space ("\20").

When this property has the value 'show', borders are drawn around empty cells (like normal cells).

A value of 'hide' means that no borders are drawn around empty cells. Furthermore, if all the cells in a row have a value of 'hide' and have no visible content, the entire row behaves as if it had 'display: none'.

Example(s):

The following rule causes borders to be drawn around all cells:

```
TABLE { empty-cells: show }
```

17.6.2 The collapsing border model

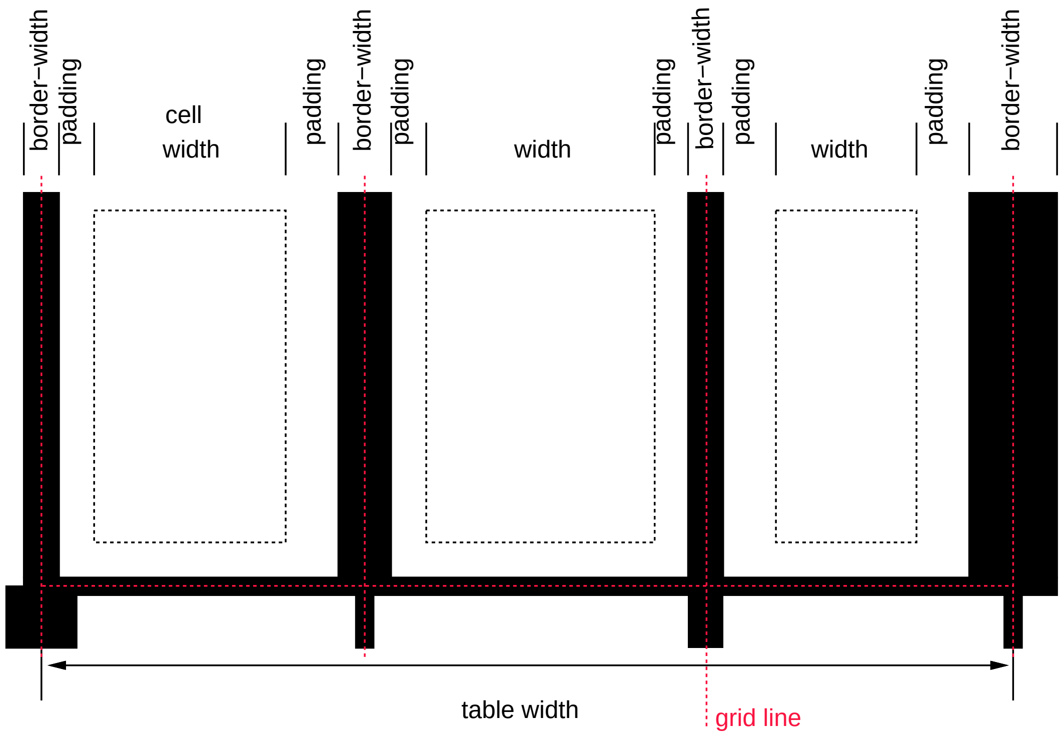
In the collapsing border model, it is possible to specify borders that surround all or part of a cell, row, row group, column, and column group. Borders for HTML's "rule" attribute can be specified this way.

Borders are centered on the grid lines between the cells. User agents must find a consistent rule for rounding off in the case of an odd number of discrete units (screen pixels, printer dots).

The diagram below shows how the width of the table, the widths of the borders, the padding, and the cell width interact. Their relation is given by the following equation, which holds for every row of the table:

$$\textit{row-width} = (0.5 * \textit{border-width}_0) + \textit{padding-left}_1 + \textit{width}_1 + \textit{padding-right}_1 + \textit{border-width}_1 + \textit{padding-left}_2 + ... + \textit{padding-right}_n + (0.5 * \textit{border-width}_n)$$

Here n is the number of cells in the row, and $\textit{border-width}_i$ refers to the border between cells i and $i + 1$. Note only half of the two exterior borders are counted in the table width; the other half of these two borders lies in the margin area.



Schema showing the widths of cells and borders and the padding of cells.

Note that in this model, the width of the table includes half the table border. Also, in this model, a table doesn't have padding (but does have margins).

Border conflict resolution

In the collapsing border model, borders at every edge of every cell may be specified by border properties on a variety of elements that meet at that edge (cells, rows, row groups, columns, column groups, and the table itself), and these borders may vary in width, style, and color. The rule of thumb is that at each edge the most "eye catching" border style is chosen, except that any occurrence of the style 'hidden' unconditionally turns the border off.

The following rules determine which border style "wins" in case of a conflict:

1. Borders with the 'border-style' of 'hidden' take precedence over all other conflicting borders. Any border with this value suppresses all borders at this location.
2. Borders with a style of 'none' have the lowest priority. Only if the border properties of all the elements meeting at this edge are 'none' will the border be omitted (but note that 'none' is the default value for the border style.)
3. If none of the styles is 'hidden' and at least one of them is not 'none', then narrow borders are discarded in favor of wider ones. If several have the same 'border-width' than styles are preferred in this order: 'double', 'solid', 'dashed', 'dotted', 'ridge', 'outset', 'groove', and the lowest: 'inset'.
4. If border styles differ only in color, then a style set on a cell wins over one on a row, which wins over a row group, column, column group and, lastly, table.

Example(s):

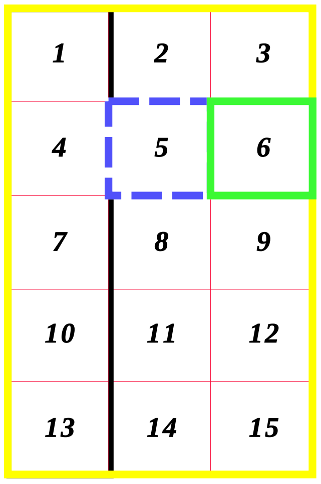
The following example illustrates the application of these precedence rules. This style sheet:

```
TABLE          { border-collapse: collapse;
                 border: 5px solid yellow; }
*#col1         { border: 3px solid black; }
TD             { border: 1px solid red; padding: 1em; }
TD.solid-blue  { border: 5px dashed blue; }
TD.solid-green { border: 5px solid green; }
```

with this HTML source:

```
<P>
<TABLE>
<COL id="col1"><COL id="col2"><COL id="col3">
<TR id="row1">
    <TD> 1
    <TD> 2
    <TD> 3
</TR>
<TR id="row2">
    <TD> 4
    <TD class="solid-blue"> 5
    <TD class="solid-green"> 6
</TR>
<TR id="row3">
    <TD> 7
    <TD> 8
    <TD> 9
</TR>
<TR id="row4">
    <TD> 10
    <TD> 11
    <TD> 12
</TR>
<TR id="row5">
    <TD> 13
    <TD> 14
    <TD> 15
</TR>
</TABLE>
```

would produce something like this:



An example of a table with collapsed borders.

Example(s):

The next example shows a table with horizontal rules between the rows. The top border of the table is set to 'hidden' to suppress the top border of the first row. This implements the "rules" attribute of HTML 4.0 (rules="rows").

```
TABLE[rules=rows] TR { border-top: solid }
TABLE[rules=rows]    { border-collapse: collapse;
                       border-top: hidden }
```

a	b	c
3	4	5
5	12	13

Table with horizontal rules between the rows.

In this case the same effect can also be achieved without setting a 'hidden' border on TABLE, by addressing the first row separately. Which method is preferred is a matter of taste.

```
TR:first-child { border-top: none }
TR { border-top: solid }
```

Example(s):

Here is another example of hidden collapsing borders:

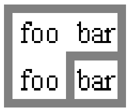


Table with two omitted internal borders.

HTML source:

```
<TABLE style="border-collapse: collapse; border: solid;">
<TR><TD style="border-right: hidden; border-bottom: hidden">foo</TD>
    <TD style="border: solid">bar</TD></TR>
<TR><TD style="border: none">foo</TD>
    <TD style="border: solid">bar</TD></TR>
</TABLE>
```

17.6.3 Border styles

Some of the values of the 'border-style' have different meanings in tables than for other elements. In the list below they are marked with an asterisk.

none
: No border.

***hidden**
: Same as 'none', but in the collapsing border model [p. 263] , also inhibits any other border (see the section on border conflicts [p. 264]).

dotted
: The border is a series of dots.

dashed
: The border is a series of short line segments.

solid
The border is a single line segment.

double
The border is two solid lines. The sum of the two lines and the space between them equals the value of ’border-width’.

groove
The border looks as though it were carved into the canvas.

ridge
The opposite of ’grove’: the border looks as though it were coming out of the canvas.

***inset**
In the separated borders model [p. 261] , the border makes the entire box look as though it were embedded in the canvas. In the collapsing border model [p. 263] , same as ’groove’.

***outset**
In the separated borders model [p. 261] , the border makes the entire box look as though it were coming out of the canvas. In the collapsing border model [p. 263] , same as ’ridge’.

17.7 Audio rendering of tables

When a table is spoken by a speech generator, the relation between the data cells and the header cells must be expressed in a different way than by horizontal and vertical alignment. Some speech browsers may allow a user to move around in the 2-dimensional space, thus giving them the opportunity to map out the spatially represented relations. When that is not possible, the style sheet must specify at which points the headers are spoken.

17.7.1 Speaking headers: the ’speak-header’ property

’speak-header’

Value:	once | always | inherit
Initial:	once
Applies to:	elements that have table header information
Inherited:	yes
Percentages:	N/A
Media:	aural

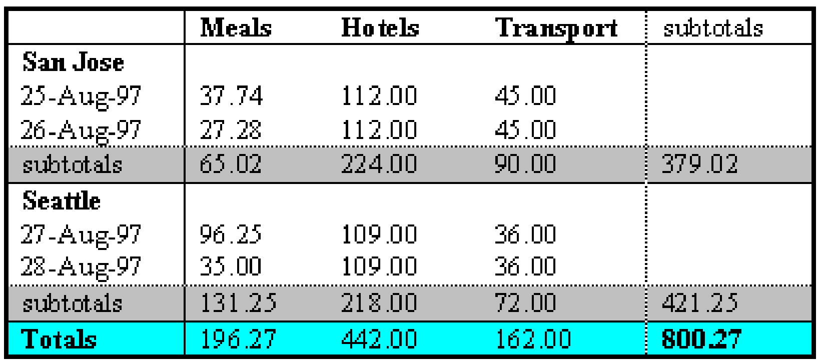
This property specifies whether table headers are spoken before every cell, or only before a cell when that cell is associated with a different header than the previous cell. Values have the following meanings:

once
The header is spoken one time, before a series of cells.

always
The header is spoken before every pertinent cell.

Each document language may have different mechanisms that allow authors to specify headers. For example, in HTML 4.0 ([HTML40]), it is possible to specify header information with three different attributes ("headers", "scope", and "axis"), and the specification gives an algorithm for determining header information when these attributes have not been specified.

Travel Expense Report



	Meals	Hotels	Transport	subtotals
San Jose				
25-Aug-97	37.74	112.00	45.00	
26-Aug-97	27.28	112.00	45.00	
subtotals	65.02	224.00	90.00	379.02
Seattle				
27-Aug-97	96.25	109.00	36.00	
28-Aug-97	35.00	109.00	36.00	
subtotals	131.25	218.00	72.00	421.25
Totals	196.27	442.00	162.00	**800.27**

Image of a table with header cells ("San Jose" and "Seattle") that are not in the same column or row as the data they apply to.

This HTML example presents the money spent on meals, hotels and transport in two locations (San Jose and Seattle) for successive days. Conceptually, you can think of the table in terms of a n-dimensional space. The headers of this space are: location, day, category and subtotal. Some cells define marks along an axis while others give money spent at points within this space. The markup for this table is:

```
<TABLE>
<CAPTION>Travel Expense Report</CAPTION>
<TR>
  <TH></TH>
  <TH>Meals</TH>
  <TH>Hotels</TH>
  <TH>Transport</TH>
  <TH>subtotal</TH>
</TR>
<TR>
  <TH id="san-jose" axis="san-jose">San Jose</TH>
</TR>
<TR>
  <TH headers="san-jose">25-Aug-97</TH>
  <TD>37.74</TD>
  <TD>112.00</TD>
  <TD>45.00</TD>
  <TD></TD>
</TR>
<TR>
  <TH headers="san-jose">26-Aug-97</TH>
  <TD>27.28</TD>
  <TD>112.00</TD>
  <TD>45.00</TD>
  <TD></TD>
</TR>
<TR>
  <TH headers="san-jose">subtotal</TH>
  <TD>65.02</TD>
```

```
    <TD>224.00</TD>
    <TD>90.00</TD>
    <TD>379.02</TD>
  </TR>
  <TR>
    <TH id="seattle" axis="seattle">Seattle</TH>
  </TR>
  <TR>
    <TH headers="seattle">27-Aug-97</TH>
    <TD>96.25</TD>
    <TD>109.00</TD>
    <TD>36.00</TD>
    <TD></TD>
  </TR>
  <TR>
    <TH headers="seattle">28-Aug-97</TH>
    <TD>35.00</TD>
    <TD>109.00</TD>
    <TD>36.00</TD>
    <TD></TD>
  </TR>
  <TR>
    <TH headers="seattle">subtotal</TH>
    <TD>131.25</TD>
    <TD>218.00</TD>
    <TD>72.00</TD>
    <TD>421.25</TD>
  </TR>
  <TR>
    <TH>Totals</TH>
    <TD>196.27</TD>
    <TD>442.00</TD>
    <TD>162.00</TD>
    <TD>800.27</TD>
  </TR>
  </TABLE>
```

By providing the data model in this way, authors make it possible for speech enabled-browsers to explore the table in rich ways, e.g., each cell could be spoken as a list, repeating the applicable headers before each data cell:

```
    San Jose, 25-Aug-97, Meals:  37.74
    San Jose, 25-Aug-97, Hotels:  112.00
    San Jose, 25-Aug-97, Transport:  45.00
   ...
```

The browser could also speak the headers only when they change:

```
  San Jose, 25-Aug-97, Meals: 37.74
      Hotels: 112.00
      Transport: 45.00
    26-Aug-97, Meals: 27.28
      Hotels: 112.00
  ...
```

18 User interface

Contents

18.1 Cursors: the ’cursor’ property

’cursor’

Value:	[[<uri> ,]* [auto | crosshair | default | pointer | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize| text | wait | help]] | inherit
Initial:	auto
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual, interactive

This property specifies the type of cursor to be displayed for the pointing device. Values have the following meanings:

auto
The UA determines the cursor to display based on the current context.

crosshair
A simple crosshair (e.g., short line segments resembling a "+" sign).

default
The platform-dependent default cursor. Often rendered as an arrow.

pointer
The cursor is a pointer that indicates a link.

move
Indicates something is to be moved.

e-resize, ne-resize, nw-resize, n-resize, se-resize, sw-resize, s-resize, w-resize
Indicate that some edge is to be moved. For example, the ’se-resize’ cursor is used when the movement starts from the south-east corner of the box.

text
Indicates text that may be selected. Often rendered as an I-bar.

wait
Indicates that the program is busy and the user should wait. Often rendered as a watch or hourglass.

help

Help is available for the object under the cursor. Often rendered as a question mark or a balloon.

<uri>

The user agent retrieves the cursor from the resource designated by the URI. If the user agent cannot handle the first cursor of a list of cursors, it should attempt to handle the second, etc. If the user agent cannot handle any user-defined cursor, it must use the generic cursor at the end of the list.

Example(s):

```
P { cursor : url("mything.cur"), url("second.csr"), text; }
```

18.2 User preferences for colors

In addition to being able to assign pre-defined color values [p. 48] to text, backgrounds, etc., CSS2 allows authors to specify colors in a manner that integrates them into the user's graphic environment. Style rules that take into account user preferences thus offer the following advantages:

1. They produce pages that fit the user's defined look and feel.
2. They produce pages that may be more accessible as the current user settings may be related to a disability.

The set of values defined for system colors is intended to be exhaustive. For systems that do not have a corresponding value, the specified value should be mapped to the nearest system attribute, or to a default color.

The following lists additional values for color-related CSS attributes and their general meaning. Any color property (e.g., 'color' or 'background-color') can take one of the following names. Although these are case-insensitive, it is recommended that the mixed capitalization shown below be used, to make the names more legible.

ActiveBorder

Active window border.

ActiveCaption

Active window caption.

AppWorkspace

Background color of multiple document interface.

Background

Desktop background.

ButtonFace

Face color for three-dimensional display elements.

ButtonHighlight

Dark shadow for three-dimensional display elements (for edges facing away from the light source).

ButtonShadow

Shadow color for three-dimensional display elements.

ButtonText

Text on push buttons.

CaptionText
Text in caption, size box, and scrollbar arrow box.
GrayText
Grayed (disabled) text. This color is set to #000 if the current display driver does not support a solid gray color.
Highlight
Item(s) selected in a control.
HighlightText
Text of item(s) selected in a control.
InactiveBorder
Inactive window border.
InactiveCaption
Inactive window caption.
InactiveCaptionText
Color of text in an inactive caption.
InfoBackground
Background color for tooltip controls.
InfoText
Text color for tooltip controls.
Menu
Menu background.
MenuText
Text in menus.
Scrollbar
Scroll bar gray area.
ThreeDDarkShadow
Dark shadow for three-dimensional display elements.
ThreeDFace
Face color for three-dimensional display elements.
ThreeDHighlight
Highlight color for three-dimensional display elements.
ThreeDLightShadow
Light color for three-dimensional display elements (for edges facing the light source).
ThreeDShadow
Dark shadow for three-dimensional display elements.
Window
Window background.
WindowFrame
Window frame.
WindowText
Text in windows.

Example(s):
For example, to set the foreground and background colors of a paragraph to the same foreground and background colors of the user’s window, write the following:

```
P { color: WindowText; background-color: Window }
```

18.3 User preferences for fonts

As for colors, authors may specify fonts in a way that makes use of a user’s system resources. Please consult the ’font’ property for details.

18.4 Dynamic outlines: the ’outline’ property

At times, style sheet authors may want to create outlines around visual objects such as buttons, active form fields, image maps, etc., to make them stand out. CSS2 outlines differ from borders [p. 88] in the following ways:

1. Outlines do not take up space.
2. Outlines may be non-rectangular.

The outline properties control the style of these dynamic outlines.

’outline’

Value:	[<’outline-color’> || <’outline-style’> || <’outline-width’>] | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual, interactive

’outline-width’

Value:	<border-width> | inherit
Initial:	medium
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual, interactive

’outline-style’

Value:	<border-style> | inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual, interactive

’outline-color’

Value:	<color> | invert | inherit
Initial:	invert
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual, interactive

The outline created with the outline properties is drawn "over" a box, i.e., the outline is always on top, and doesn’t influence the position or size of the box, or of any other boxes. Therefore, displaying or suppressing outlines does not cause reflow.

The outline is drawn starting just outside the border edge [p. 82] .

Outlines may be non-rectangular. For example, if the element is broken across several lines, the outline is the minimum outline that encloses all the element’s boxes. In contrast to borders [p. 88] , the outline is not open at the line box’s end or start, but is always fully connected.

The ’outline-width’ property accepts the same values as ’border-width’.

The ’outline-style’ property accepts the same values as ’border-style’, except that ’hidden’ is not a legal outline style.

The ’outline-color’ accepts all colors, as well as the keyword ’invert’. ’Invert’ is expected to perform a color inversion on the pixels on the screen. This is a common trick to ensure the focus border is visible, regardless of color background.

The ’outline’ property is a shorthand property, and sets all three of ’outline-style’, ’outline-width’, and ’outline-color’.

Note that the outline is the same on all sides. In contrast to borders, there is no ’outline-top’ or ’outline-left’ property.

This specification does not define how multiple overlapping outlines are drawn, or how outlines are drawn for boxes that are partially obscured behind other elements.

Note. *Since the focus outline does not affect formatting (i.e., no space is left for it in the box model), it may well overlap other elements on the page.*

Example(s):

Here’s an example of drawing a thick outline around a BUTTON element:

```
BUTTON { outline-width : thick }
```

Scripts may be used to dynamically change the width of the outline, without provoking reflow.

18.4.1 Outlines and the focus

Graphical user interfaces may use outlines around elements to tell the user which element on the page has the *focus*. These outlines are in addition to any borders, and switching outlines on and off should not cause the document to reflow. The focus is the subject of user interaction in a document (e.g., for entering text, selecting a button, etc.). User agents supporting the interactive media group [p. 79] must keep track of where the focus lies and must also represent the focus. This may be done by using dynamic outlines in conjunction with the :focus pseudo-class.

Example(s):

For example, to draw a thick black line around an element when it has the focus, and a thick red line when it is active, the following rules can be used:

```
:focus  { outline: thick solid black }
:active { outline: thick solid red }
```

18.5 Magnification

The CSS working group considers that the magnification of a document or portions of a document should not be specified through style sheets. User agents may support such magnification in different ways (e.g., larger images, louder sounds, etc.)

When magnifying a page, UAs should preserve the relationships between positioned elements. For example, a comic strip may be composed of images with overlaid text elements. When magnifying this page, a user agent should keep the text within the comic strip balloon.

---

19 Aural style sheets

Contents

19.1 Introduction to aural style sheets

The aural rendering of a document, already commonly used by the blind and print-impaired communities, combines speech synthesis and "auditory icons." Often such aural presentation occurs by converting the document to plain text and feeding this to a *screen reader* -- software or hardware that simply reads all the characters on the screen. This results in less effective presentation than would be the case if the document structure were retained. Style sheet properties for aural presentation may be used together with visual properties (mixed media) or as an aural alternative to visual presentation.

Besides the obvious accessibility advantages, there are other large markets for listening to information, including in-car use, industrial and medical documentation systems (intranets), home entertainment, and to help users learning to read or who have difficulty reading.

When using aural properties, the canvas consists of a three-dimensional physical space (sound surrounds) and a temporal space (one may specify sounds before, during, and after other sounds). The CSS properties also allow authors to vary the quality of synthesized speech (voice type, frequency, inflection, etc.).

Example(s):

```
H1, H2, H3, H4, H5, H6 {
    voice-family: paul;
    stress: 20;
    richness: 90;
    cue-before: url("ping.au")
}
P.heidi { azimuth: center-left }
P.peter { azimuth: right }
P.goat  { volume: x-soft }
```

This will direct the speech synthesizer to speak headers in a voice (a kind of "audio font") called "paul", on a flat tone, but in a very rich voice. Before speaking the headers, a sound sample will be played from the given URL. Paragraphs with class "heidi" will appear to come from front left (if the sound system is capable of spatial audio), and paragraphs of class "peter" from the right. Paragraphs with class "goat" will be very soft.

19.2 Volume properties: ’volume’

’volume’

Value:	<number> | <percentage> | silent | x-soft | soft | medium | loud | x-loud | inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes
Percentages:	refer to inherited value
Media:	aural

Volume refers to the median volume of the waveform. In other words, a highly inflected voice at a volume of 50 might peak well above that. The overall values are likely to be human adjustable for comfort, for example with a physical volume control (which would increase both the 0 and 100 values proportionately); what this property does is adjust the dynamic range.

Values have the following meanings:

<number>
Any number between ’0’ and ’100’. ’0’ represents the *minimum audible* volume level and 100 corresponds to the *maximum comfortable* level.

<percentage>
Percentage values are calculated relative to the inherited value, and are then clipped to the range ’0’ to ’100’.

silent
No sound at all. The value ’0’ does not mean the same as ’silent’.

x-soft
Same as ’0’.

soft
Same as ’25’.

medium
Same as ’50’.

loud
Same as ’75’.

x-loud
Same as ’100’.

User agents should allow the values corresponding to ’0’ and ’100’ to be set by the listener. No one setting is universally applicable; suitable values depend on the equipment in use (speakers, headphones), the environment (in car, home theater, library) and personal preferences. Some examples:

- A browser for in-car use has a setting for when there is lots of background noise. ’0’ would map to a fairly high level and ’100’ to a quite high level. The speech is easily audible over the road noise but the overall dynamic range is compressed. Cars with better insulation might allow a wider dynamic range.

- Another speech browser is being used in an apartment, late at night, or in a shared study room. ’0’ is set to a very quiet level and ’100’ to a fairly quiet level, too. As with the first example, there is a low slope; the dynamic range is reduced. The actual volumes are low here, whereas they were high in the first example.
- In a quiet and isolated house, an expensive hi-fi home theater setup. ’0’ is set fairly low and ’100’ to quite high; there is wide dynamic range.

The same author style sheet could be used in all cases, simply by mapping the ’0’ and ’100’ points suitably at the client side.

19.3 Speaking properties: ’speak’

’speak’

Value:	normal | none | spell-out | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

This property specifies whether text will be rendered aurally and if so, in what manner (somewhat analogous to the ’display’ property). The possible values are:

none
Suppresses aural rendering so that the element requires no time to render. Note, however, that descendants may override this value and will be spoken. (To be sure to suppress rendering of an element and its descendants, use the ’display’ property).

normal
Uses language-dependent pronunciation rules for rendering an element and its children.

spell-out
Spells the text one letter at a time (useful for acronyms and abbreviations).

Note the difference between an element whose ’volume’ property has a value of ’silent’ and an element whose ’speak’ property has the value ’none’. The former takes up the same time as if it had been spoken, including any pause before and after the element, but no sound is generated. The latter requires no time and is not rendered (though its descendants may be).

19.4 Pause properties: ’pause-before’, ’pause-after’, and ’pause’

’pause-before’

Value:	<time> | <percentage> | inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	no
Percentages:	see prose
Media:	aural

’pause-after’

Value:	<time> | <percentage> | inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	no
Percentages:	see prose
Media:	aural

These properties specify a pause to be observed before (or after) speaking an element’s content. Values have the following meanings:

<time>
: Expresses the pause in absolute time units (seconds and milliseconds).

<percentage>
: Refers to the inverse of the value of the ’speech-rate’ property. For example, if the speech-rate is 120 words per minute (i.e., a word takes half a second, or 500ms) then a ’pause-before’ of 100% means a pause of 500 ms and a ’pause-before’ of 20% means 100ms.

The pause is inserted between the element’s content and any ’cue-before’ or ’cue-after’ content.

Authors should use relative units to create more robust style sheets in the face of large changes in speech-rate.

’pause’

Value:	[[<time> | <percentage>]{1,2}] | inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	no
Percentages:	see descriptions of ’pause-before’ and ’pause-after’
Media:	aural

The ’pause’ property is a shorthand for setting ’pause-before’ and ’pause-after’. If two values are given, the first value is ’pause-before’ and the second is ’pause-after’. If only one value is given, it applies to both properties.

Example(s):

```
H1 { pause: 20ms } /* pause-before: 20ms; pause-after: 20ms */
H2 { pause: 30ms 40ms } /* pause-before: 30ms; pause-after: 40ms */
H3 { pause-after: 10ms } /* pause-before: ?; pause-after: 10ms */
```

19.5 Cue properties: ’cue-before’, ’cue-after’, and ’cue’

’cue-before’

Value:	<uri> | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural

’cue-after’

Value:	<uri> | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural

Auditory icons are another way to distinguish semantic elements. Sounds may be played before and/or after the element to delimit it. Values have the following meanings:

<uri>
: The URI must designate an auditory icon resource. If the URI resolves to something other than an audio file, such as an image, the resource should be ignored and the property treated as if it had the value ’none’.

none
: No auditory icon is specified.

Example(s):

```
A {cue-before: url("bell.aiff"); cue-after: url("dong.wav") }
H1 {cue-before: url("pop.au"); cue-after: url("pop.au") }
```

’cue’

Value:	[<’cue-before’> || <’cue-after’>] | inherit
Initial:	not defined for shorthand properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural

The ’cue’ property is a shorthand for setting ’cue-before’ and ’cue-after’. If two values are given, the first value is ’cue-before’ and the second is ’cue-after’. If only one value is given, it applies to both properties.

Example(s):
The following two rules are equivalent:

```
H1 {cue-before: url("pop.au"); cue-after: url("pop.au") }
H1 {cue: url("pop.au") }
```

If a user agent cannot render an auditory icon (e.g., the user’s environment does not permit it), we recommend that it produce an alternative cue (e.g., popping up a warning, emitting a warning sound, etc.)

Please see the sections on the :before and :after pseudo-elements [p. 153] for information on other content generation techniques.

19.6 Mixing properties: ’play-during’

’play-during’

Value:	<uri> mix? repeat? | auto | none | inherit
Initial:	auto
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural

Similar to the ’cue-before’ and ’cue-after’ properties, this property specifies a sound to be played as a background while an element’s content is spoken. Values have the following meanings:

<uri>
The sound designated by this <uri> is played as a background while the element’s content is spoken.

mix
When present, this keyword means that the sound inherited from the parent element’s ’play-during’ property continues to play and the sound designated by the <uri> is mixed with it. If ’mix’ is not specified, the element’s background sound replaces the parent’s.

repeat
When present, this keyword means that the sound will repeat if it is too short to fill the entire duration of the element. Otherwise, the sound plays once and then stops. This is similar to the ’background-repeat’ property. If the sound is too long for the element, it is clipped once the element has been spoken.

auto
The sound of the parent element continues to play (it is not restarted, which would have been the case if this property had been inherited).

none
This keyword means that there is silence. The sound of the parent element (if any) is silent during the current element and continues after the current element.

Example(s):

```
BLOCKQUOTE.sad { play-during: url("violins.aiff") }
BLOCKQUOTE Q   { play-during: url("harp.wav") mix }
SPAN.quiet     { play-during: none }
```

19.7 Spatial properties: ’azimuth’ and ’elevation’

Spatial audio is an important stylistic property for aural presentation. It provides a natural way to tell several voices apart, as in real life (people rarely all stand in the same spot in a room). Stereo speakers produce a lateral sound stage. Binaural headphones or the increasingly popular 5-speaker home theater setups can generate full surround sound, and multi-speaker setups can create a true three-dimensional sound stage. VRML 2.0 also includes spatial audio, which implies that in time consumer-priced spatial audio hardware will become more widely available.

’azimuth’

Value:	<angle> | [[left-side | far-left | left | center-left | center | center-right | right | far-right | right-side] || behind] | leftwards | rightwards | inherit
Initial:	center
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

Values have the following meanings:

<angle>
: Position is described in terms of an angle within the range ’-360deg’ to ’360deg’. The value ’0deg’ means directly ahead in the center of the sound stage. ’90deg’ is to the right, ’180deg’ behind, and ’270deg’ (or, equivalently and more conveniently, ’-90deg’) to the left.

left-side
: Same as ’270deg’. With ’behind’, ’270deg’.

far-left
: Same as ’300deg’. With ’behind’, ’240deg’.

left
: Same as ’320deg’. With ’behind’, ’220deg’.

center-left
: Same as ’340deg’. With ’behind’, ’200deg’.

center
: Same as ’0deg’. With ’behind’, ’180deg’.

center-right
: Same as ’20deg’. With ’behind’, ’160deg’.

right
: Same as ’40deg’. With ’behind’, ’140deg’.

far-right

Same as ’60deg’. With ’behind’, ’120deg’.

right-side

Same as ’90deg’. With ’behind’, ’90deg’.

leftwards

Moves the sound to the left, relative to the current angle. More precisely, subtracts 20 degrees. Arithmetic is carried out modulo 360 degrees. Note that ’leftwards’ is more accurately described as "turned counter-clockwise," since it *always* subtracts 20 degrees, even if the inherited azimuth is already behind the listener (in which case the sound actually appears to move to the right).

rightwards

Moves the sound to the right, relative to the current angle. More precisely, adds 20 degrees. See ’leftwards’ for arithmetic.

This property is most likely to be implemented by mixing the same signal into different channels at differing volumes. It might also use phase shifting, digital delay, and other such techniques to provide the illusion of a sound stage. The precise means used to achieve this effect and the number of speakers used to do so are user agent-dependent; this property merely identifies the desired end result.

Example(s):

```
H1   { azimuth: 30deg }
TD.a { azimuth: far-right }          /*  60deg */
#12  { azimuth: behind far-right }   /* 120deg */
P.comment { azimuth: behind }        /* 180deg */
```

If spatial-azimuth is specified and the output device cannot produce sounds *behind* the listening position, user agents should convert values in the rearwards hemisphere to forwards hemisphere values. One method is as follows:

- if 90deg < x <= 180deg then x := 180deg - x
- if 180deg < x <= 270deg then x := 540deg - x

’elevation’

Value:	<angle> | below | level | above | higher | lower | inherit
Initial:	level
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

Values of this property have the following meanings:

<angle>

Specifies the elevation as an angle, between ’-90deg’ and ’90deg’. ’0deg’ means on the forward horizon, which loosely means level with the listener. ’90deg’ means directly overhead and ’-90deg’ means directly below.

below
  Same as ’-90deg’.
level
  Same as ’0deg’.
above
  Same as ’90deg’.
higher
  Adds 10 degrees to the current elevation.
lower
  Subtracts 10 degrees from the current elevation.

The precise means used to achieve this effect and the number of speakers used to do so are undefined. This property merely identifies the desired end result.

Example(s):

```
H1   { elevation: above }
TR.a { elevation: 60deg }
TR.b { elevation: 30deg }
TR.c { elevation: level }
```

19.8 Voice characteristic properties: ’speech-rate’, ’voice-family’, ’pitch’, ’pitch-range’, ’stress’, and ’richness’

’speech-rate’

Value:	<number> | x-slow | slow | medium | fast | x-fast | faster | slower | inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

This property specifies the speaking rate. Note that both absolute and relative keyword values are allowed (compare with ’font-size’). Values have the following meanings:

<number>
  Specifies the speaking rate in words per minute, a quantity that varies somewhat by language but is nevertheless widely supported by speech synthesizers.
x-slow
  Same as 80 words per minute.
slow
  Same as 120 words per minute
medium
  Same as 180 - 200 words per minute.

fast
Same as 300 words per minute.
x-fast
Same as 500 words per minute.
faster
Adds 40 words per minute to the current speech rate.
slower
Subtracts 40 words per minutes from the current speech rate.

’voice-family’

Value:	[[<specific-voice> | <generic-voice>],]* [<specific-voice> | <generic-voice>] | inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

The value is a comma-separated, prioritized list of voice family names (compare with ’font-family’). Values have the following meanings:

<generic-voice>
Values are voice families. Possible values are ’male’, ’female’, and ’child’.
<specific-voice>
Values are specific instances (e.g., comedian, trinoids, carlos, lani).

Example(s):

```
H1 { voice-family: announcer, male }
P.part.romeo  { voice-family: romeo, male }
P.part.juliet { voice-family: juliet, female }
```

Names of specific voices may be quoted, and indeed must be quoted if any of the words that make up the name does not conform to the syntax rules for identifiers [p. 35] . It is also recommended to quote specific voices with a name consisting of more than one word. If quoting is omitted, any whitespace [p. 37] characters before and after the font name are ignored and any sequence of whitespace characters inside the font name is converted to a single space.

’pitch’

Value:	<frequency> | x-low | low | medium | high | x-high | inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

Specifies the average pitch (a frequency) of the speaking voice. The average pitch of a voice depends on the voice family. For example, the average pitch for a standard male voice is around 120Hz, but for a female voice, it’s around 210Hz.

Values have the following meanings:

<frequency>
Specifies the average pitch of the speaking voice in hertz (Hz).

x-low, **low**, **medium**, **high**, **x-high**
These values do not map to absolute frequencies since these values depend on the voice family. User agents should map these values to appropriate frequencies based on the voice family and user environment. However, user agents must map these values in order (i.e., ’x-low’ is a lower frequency than ’low’, etc.).

’pitch-range’

Value:	<number> | inherit
Initial:	50
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

Specifies variation in average pitch. The perceived pitch of a human voice is determined by the fundamental frequency and typically has a value of 120Hz for a male voice and 210Hz for a female voice. Human languages are spoken with varying inflection and pitch; these variations convey additional meaning and emphasis. Thus, a highly animated voice, i.e., one that is heavily inflected, displays a high pitch range. This property specifies the range over which these variations occur, i.e., how much the fundamental frequency may deviate from the average pitch.
Values have the following meanings:

<number>
A value between ’0’ and ’100’. A pitch range of ’0’ produces a flat, monotonic voice. A pitch range of 50 produces normal inflection. Pitch ranges greater than 50 produce animated voices.

’stress’

Value:	<number> | inherit
Initial:	50
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

Specifies the height of "local peaks" in the intonation contour of a voice. For example, English is a **stressed** language, and different parts of a sentence are assigned primary, secondary, or tertiary stress. The value of ’stress’ controls the amount of inflection that results from these stress markers. This property is a companion to the ’pitch-range’ property and is provided to allow developers to exploit higher-end auditory displays.

Values have the following meanings:

<number>
: A value, between ’0’ and ’100’. The meaning of values depends on the language being spoken. For example, a level of ’50’ for a standard, English-speaking male voice (average pitch = 122Hz), speaking with normal intonation and emphasis would have a different meaning than ’50’ for an Italian voice.

’richness’

Value:	<number> | inherit
Initial:	50
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

Specifies the richness, or brightness, of the speaking voice. A rich voice will "carry" in a large room, a smooth voice will not. (The term "smooth" refers to how the wave form looks when drawn.)
Values have the following meanings:

<number>
: A value between ’0’ and ’100’. The higher the value, the more the voice will carry. A lower value will produce a soft, mellifluous voice.

19.9 Speech properties: ’speak-punctuation’ and ’speak-numeral’

An additional speech property, speak-header [p. 267] , is described in the chapter on tables [p. 245]

’speak-punctuation’

Value:	code | none | inherit
Initial:	none
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

This property specifies how punctuation is spoken. Values have the following meanings:

code
: Punctuation such as semicolons, braces, and so on are to be spoken literally.

none

Punctuation is not to be spoken, but instead rendered naturally as various pauses.

’speak-numeral’

Value:	digits | continuous | inherit
Initial:	continuous
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural

This property controls how numerals are spoken. Values have the following meanings:

digits

Speak the numeral as individual digits. Thus, "237" is spoken "Two Three Seven".

continuous

Speak the numeral as a full number. Thus, "237" is spoken "Two hundred thirty seven". Word representations are language-dependent.

Appendix A. A sample style sheet for HTML 4.0

This appendix is informative, not normative.

This style sheet describes the typical formatting of all HTML 4.0 ([HTML40]) elements based on extensive research into current UA practice. Developers are encouraged to use it as a default style sheet in their implementations.

The full presentation of some HTML elements cannot be expressed in CSS2, including replaced [p. 30] elements (IMG, OBJECT), scripting elements (SCRIPT, APPLET), form control elements, and frame elements.

```
ADDRESS,
BLOCKQUOTE,
BODY, DD, DIV,
DL, DT,
FIELDSET, FORM,
FRAME, FRAMESET,
H1, H2, H3, H4,
H5, H6, IFRAME,
NOFRAMES,
OBJECT, OL, P,
UL, APPLET,
CENTER, DIR,
HR, MENU, PRE   { display: block }
LI              { display: list-item }
HEAD            { display: none }
TABLE           { display: table }
TR              { display: table-row }
THEAD           { display: table-header-group }
TBODY           { display: table-row-group }
TFOOT           { display: table-footer-group }
COL             { display: table-column }
COLGROUP        { display: table-column-group }
TD, TH          { display: table-cell }
CAPTION         { display: table-caption }
TH              { font-weight: bolder; text-align: center }
CAPTION         { text-align: center }
BODY            { padding: 8px; line-height: 1.33 }
H1              { font-size: 2em; margin: .67em 0 }
H2              { font-size: 1.5em; margin: .83em 0 }
H3              { font-size: 1.17em; margin: 1em 0 }
H4, P,
BLOCKQUOTE, UL,
FIELDSET, FORM,
OL, DL, DIR,
MENU            { margin: 1.33em 0 }
H5              { font-size: .83em; line-height: 1.17em; margin: 1.67em 0 }
H6              { font-size: .67em; margin: 2.33em 0 }
H1, H2, H3, H4,
H5, H6, B,
STRONG          { font-weight: bolder }
BLOCKQUOTE      { margin-left: 40px; margin-right: 40px }
I, CITE, EM,
VAR, ADDRESS    { font-style: italic }
PRE, TT, CODE,
KBD, SAMP       { font-family: monospace }
PRE             { white-space: pre }
BIG             { font-size: 1.17em }
SMALL, SUB, SUP { font-size: .83em }
```

```
SUB             { vertical-align: sub }
SUP             { vertical-align: super }
S, STRIKE, DEL  { text-decoration: line-through }
HR              { border: 1px inset }
OL, UL, DIR,
MENU, DD        { margin-left: 40px }
OL              { list-style-type: decimal }
OL UL, UL OL,
UL UL, OL OL    { margin-top: 0; margin-bottom: 0 }
U, INS          { text-decoration: underline }
CENTER          { text-align: center }
BR:before       { content: "\A" }

/* An example of style for HTML 4.0’s ABBR/ACRONYM elements */

ABBR, ACRONYM   { font-variant: small-caps; letter-spacing: 0.1em }
A[href]         { text-decoration: underline }
:focus          { outline: thin dotted invert }

/* Begin bidirectionality settings (do not change) */
BDO[DIR="ltr"]  { direction: ltr; unicode-bidi: bidi-override }
BDO[DIR="rtl"]  { direction: rtl; unicode-bidi: bidi-override }

*[DIR="ltr"]    { direction: ltr; unicode-bidi: embed }
*[DIR="rtl"]    { direction: rtl; unicode-bidi: embed }

/* Elements that are block-level in HTML4 */
ADDRESS, BLOCKQUOTE, BODY, DD, DIV, DL, DT, FIELDSET,
FORM, FRAME, FRAMESET, H1, H2, H3, H4, H5, H6, IFRAME,
NOSCRIPT, NOFRAMES, OBJECT, OL, P, UL, APPLET, CENTER,
DIR, HR, MENU, PRE, LI, TABLE, TR, THEAD, TBODY, TFOOT,
COL, COLGROUP, TD, TH, CAPTION
                { unicode-bidi: embed }
/* End bidi settings */

@media print {
  @page         { margin: 10% }
  H1, H2, H3,
  H4, H5, H6    { page-break-after: avoid; page-break-inside: avoid }
  BLOCKQUOTE,
  PRE           { page-break-inside: avoid }
  UL, OL, DL    { page-break-before: avoid }
}

@media speech {
  H1, H2, H3,
  H4, H5, H6    { voice-family: paul, male; stress: 20; richness: 90 }
  H1            { pitch: x-low; pitch-range: 90 }
  H2            { pitch: x-low; pitch-range: 80 }
  H3            { pitch: low; pitch-range: 70 }
  H4            { pitch: medium; pitch-range: 60 }
  H5            { pitch: medium; pitch-range: 50 }
  H6            { pitch: medium; pitch-range: 40 }
  LI, DT, DD    { pitch: medium; richness: 60 }
  DT            { stress: 80 }
  PRE, CODE, TT { pitch: medium; pitch-range: 0; stress: 0; richness: 80 }
  EM            { pitch: medium; pitch-range: 60; stress: 60; richness: 50 }
  STRONG        { pitch: medium; pitch-range: 60; stress: 90; richness: 90 }
  DFN           { pitch: high; pitch-range: 60; stress: 60 }
  S, STRIKE     { richness: 0 }
  I             { pitch: medium; pitch-range: 60; stress: 60; richness: 50 }
  B             { pitch: medium; pitch-range: 60; stress: 90; richness: 90 }
  U             { richness: 0 }
  A:link        { voice-family: harry, male }
  A:visited     { voice-family: betty, female }
  A:active      { voice-family: betty, female; pitch-range: 80; pitch: x-high }
}
```

Appendix B. Changes from CSS1

Contents

This appendix is informative, not normative.

CSS2 builds on CSS1 and all valid CSS1 style sheets are valid CSS2 style sheets. The changes between the CSS1 specification (see [CSS1]) and this specification fall into three groups: new functionality, updated descriptions of CSS1 functionality, and changes to CSS1.

B.1 New functionality

In addition to the functionality of CSS1, CSS2 supports:

- The concept of media types [p. 77] .
- The ’inherit’ [p. 71] value for all properties.
- Paged media [p. 175]
- Aural style sheets [p. 277]
- Several internationalization features, including list numbering styles [p. 168] , support for bidirectional text [p. 127] , and support for language-sensitive quotation marks [p. 157] .
- An extended font selection [p. 212] mechanism, including intelligent matching, synthesis, and downloadable fonts. Also, the concept of system fonts has been is introduced, and a new property, ’font-size-adjust’, has been added.
- Tables [p. 245] , including new values on ’display’ and ’vertical-align’.
- Relative [p. 107] and absolute positioning [p. 113] , including fixed positioning [p. 113] .
- New box types (along with block and inline): compact [p. 99] and run-in [p. 100] .
- The ability to control content overflow [p. 145] , clipping [p. 147] , and visibility [p. 149] in the visual formatting model.
- The ability to specify minimum and maximum widths [p. 136] and heights [p. 140] in the visual formatting model.
- An extended selector [p. 53] mechanism, including child selectors, adjacent selectors, and attribute selectors.
- Generated content [p. 153] , counters and automatic numbering [p. 160] , and markers [p. 164] .
- Text shadows, through the new ’text-shadow’ property.
- Several new pseudo-classes [p. 61] , :first-child, :hover, :focus, :lang.
- System colors [p. 272] and fonts [p. 274] .
- Cursors [p. 271] .

- Dynamic outlines [p. 274] .

B.2 Updated descriptions

The CSS1 specification was short and concise. This specification is much more voluminous and more readable. Much of the additional content describes new functionality, but the description of CSS1 features has also been extended. Except in a few cases described below, the rewritten descriptions do not indicate a change in syntax nor semantics.

B.3 Semantic changes from CSS1

While all CSS1 style sheets are valid CSS2 style sheets, there are a few cases where the CSS1 style sheet will have a different meaning when interpreted as a CSS2 style sheet. Most changes are due to implementation experience feeding back into the specification, but there are also some error corrections.

- The meaning of "!important" has been changed. In CSS1, "!important" in an author’s style sheet took precedence over one in a user style sheet. This has been reversed in CSS2.
- In CSS2 color values [p. 48] are clipped with regard to the device gamut, not with regard to the sRGB gamut as in CSS1.
- CSS1 simply said that ’margin-right’ was ignored if the both ’margin-left’ and ’width’ were set. In CSS2 the choice between relaxing ’margin-right’ or ’margin-left’ depends on the writing direction.
- In CSS1, several properties (e.g., ’padding’) had values referring to the width of the parent element. This was an error; the value should always refer to the width of a block-level element and this specification reflects this by introducing the term "containing block".
- The initial value of ’display’ is ’inline’ in CSS2, not ’block’ as in CSS1.
- In CSS1, the ’clear’ property applied to all elements. This was an error, and the property only applies to block-level elements in CSS2.
- In CSS1, ’:link’, ’:visited’ and ’:active’ were mutually exclusive; in CSS2, ’:active’ [p. 63] can occur together with ’:link’ or ’:visited’ [p. 63] .
- The suggested scaling factor between adjacent ’font-size’ indexes in the table of font sizes has been reduced from 1.5 to 1.2.
- The computed value, not the actual value, of ’font-size’ is inherited.
- The CSS1 description of ’inside’ (for ’list-style-position’) allowed the interpretation that the left margin of the text was affected, rather than the position of the bullet. In CSS2 that interpretation is ruled out.
- Please also consult the *normative* section on the differences between the CSS1 and CSS2 tokenizer [p. 312] .

Appendix C. Implementation and performance notes for fonts

Contents

This appendix is informative, not normative.

C.1 Glossary of font terms

DocLock™
: Bitstream's *DocLock™* technology ensures that TrueDoc PFRs can only be used with the site they are published for. A TrueDoc PFR moved to a different site or referenced from another site will not work.

Digital Signature
: Part of a trust management technology, used to provide signed assertions about a resource.

Font Caching
: *Font caching* allows for a temporary copy of fonts on the client system. They are often stored on disk with other cached items such as graphics specifically for the UA.

Font Face
: A "handle" that refers to a specific face of a font, excluding the font size.

Font Matching
: *Font matching* is a process of selecting a similar font based on using one or more attributes of the primary font. Common attribute include serif, sans-serif, weight, cap height, x-height, spacing, language, and posture. Font matching is dependent on the algorithm and the variety of candidate fonts.

Glyph Representation Sub-setting
: *Glyph Representation sub-setting* is the process by which unwanted glyph representations (together with their side bearings and kerning information) are removed from a primary font to produce a smaller subset font that covers a particular document or set of documents. This is a particular win for documents that use ideographic scripts, where the glyph complement of the base font can be very large. Glyph representation sub-setting for documents using scripts that require ligatures, such as Arabic, is difficult without knowing the ligature formation rules of the final display system.

Intellifont
: Intellifont technology was developed by Agfa and is the native format for Hewlett-Packard and other printers that use the PCL5 language. It is also

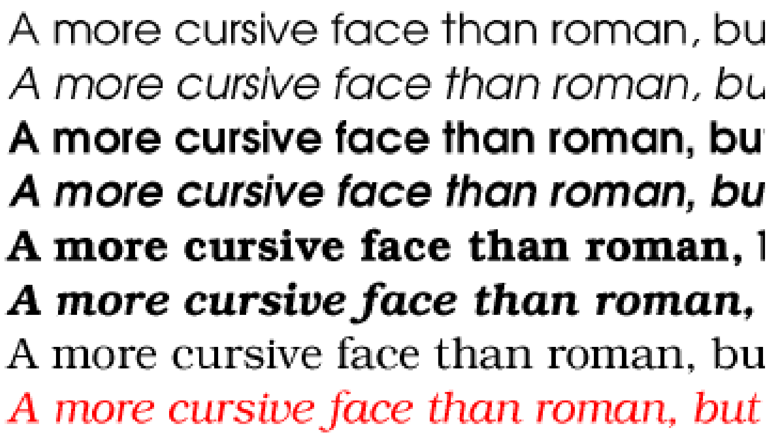
the native font format on the Amiga computers.

Infinifont

A font synthesis technique which, given a Panose-1 number (and, optionally, additional font description data) can generate a faux font without extrapolating from a single master outline or interpolating between two or more outlines (see [INFINIFONT]).

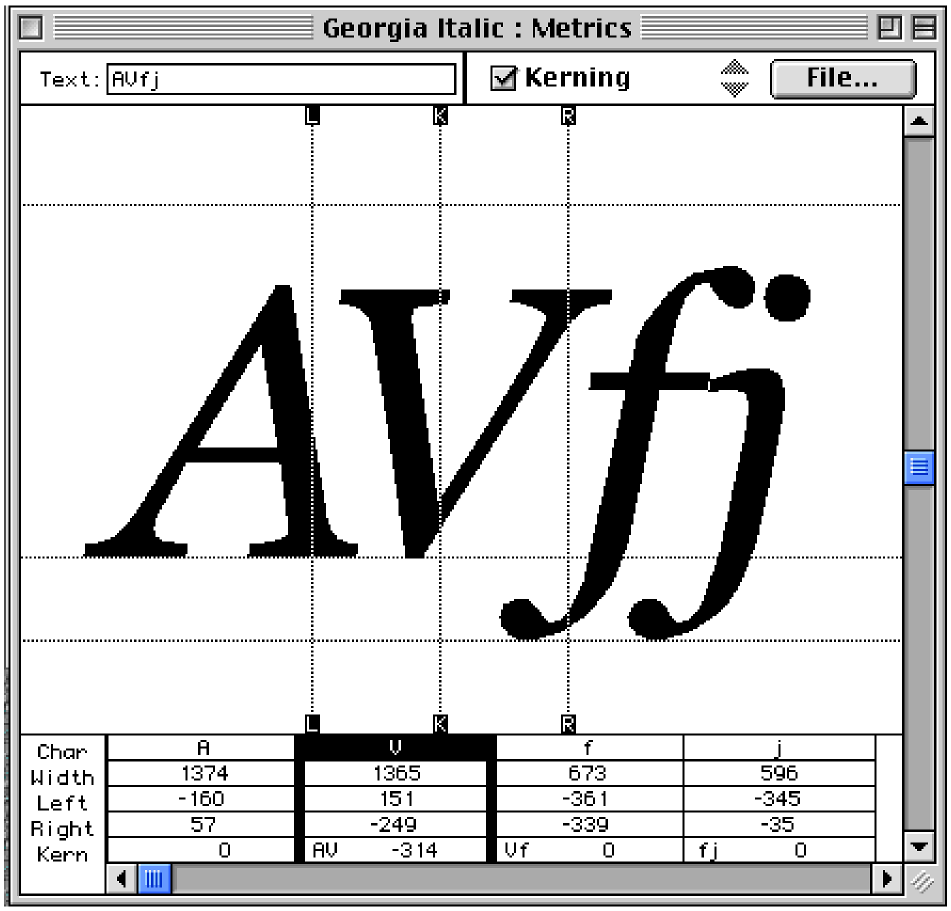
Italic

A class of letter forms for Latin scripts, that are more cursive than roman letter forms but less so than script forms. Often, a pair of fonts are designed to be used together; one is a serifed roman and one is italic. Other terms to describe this class of letter forms include cursive and, for Cyrillic scripts, kursiv. For sans-serif faces, the companion face is often a slanted or oblique variant rather than a different class of letter form.


A more cursive face than roman, bu
A more cursive face than roman, bu
A more cursive face than roman, bu
A more cursive face than roman, bu
A more cursive face than roman,
A more cursive face than roman,
A more cursive face than roman, bu
A more cursive face than roman, but

Kerning

Altering the spacing between selected glyph representations, which would otherwise appear to be too close or too far apart, to obtain a more even typographical color.



Text: AVfj

☑ Kerning

Char	A	V	f	j
Width	1374	1365	673	596
Left	-160	151	-361	-345
Right	57	-249	-339	-35
Kern	0	AV -314	Vf 0	fj 0

Multiple Master Font

A *Multiple Master Font* contain two primary fonts that are used with special rendering software to provide an interpolated result. Adobe Systems provides a mechanism that allows for parameters to be used to control the output or the interpolated output font. These parameters generally describe the characteristics of an original font and the multiple master result is referred to as a "synthesized font."

Open Type

Open Type is an extension to the TrueType font format that contains additional information that extends the capabilities of the fonts to support high-quality international typography. Open Type can associate a single character with multiple glyph representations, and combinations of characters with a single glyph representation (ligature formation). Open Type includes two-dimensional information to support features for complex positioning and glyph attachment. TrueType Open and OpenType contain explicit script and language information, so a text-processing application can adjust its behavior accordingly (see [OPENTYPE]).

Server Font

A *Server Font* is a font resource located on the web server that is referenced by the WebFont definition. The user agent may use this resource for rendering the page.

Speedo

Speedo font technology was developed by Bitstream and is the native font format on the Atari ST and Falcon computers. It is also used by computers running the X window system.

TrueDoc

TrueDoc technology was developed by Bitstream for the creation, transport, and imaging of platform independent scalable font objects on the web. Creation of font objects is done by the TrueDoc character shape recorder (CSR) and the rendering of the font objects is done by TrueDoc's character shape player (CSP). The technology is intended to be used on the web for viewing and printing.

TrueDoc Portable Font Resource

A *TrueDoc Portable font resource* (or, **PFR**) is a platform-independent scalable font object that is produced by a character shape player. Input may be either TrueType or Type 1 of any flavor on either Windows, Mac, or Unix. TrueDoc Portable Font Resources provide good compression ratios, are platform independent, and because they are not in an native font format (TrueType or Type 1) they can not be easily installed.

TrueType

TrueType is a font format developed by Apple and licensed to Microsoft. TrueType is the native operating system font format for Windows and Macintosh. TrueType contains a hierarchical set of tables and glyph representations. Characters can be hinted on a per character and point size basis yielding excellent quality at screen resolutions. TrueType fonts for Windows and Mac have few differences, though they can be different enough to prevent cross platform usage.

TrueType Collection

A *TrueType Collection* (or **TTC**) is an extension to the TrueType format that includes tables that allow for multiple TrueType fonts to be contained within a single TrueType font file. TrueType collection files are relatively rare at this time.

TrueType GX Fonts

TrueType GX Fonts contain extensions to the standard TrueType format that allow for mutable fonts, similar to Multiple Master fonts. There may be several mutation axis such as weight, height, and slant. The axis can be defined to obtain almost any effect. TrueType GX can also supports alternate glyph representation substitution for ligatures, contextual forms, fractions, etc. To date, TrueType GX is available only on the Mac (see [TRUETYPEGX]).

Type 1 font

Type 1 fonts, developed by Adobe Systems, were one of first scalable formats available. Type 1 fonts generally contain 228 characters with the glyph representations described using third degree bezier curves. Mac, Windows, and X have similar but separate formats; Adobe provides Adobe Type Manager for all three platforms. Type1c is a more recent losslessly-compressed storage form for Type 1 glyph representations.

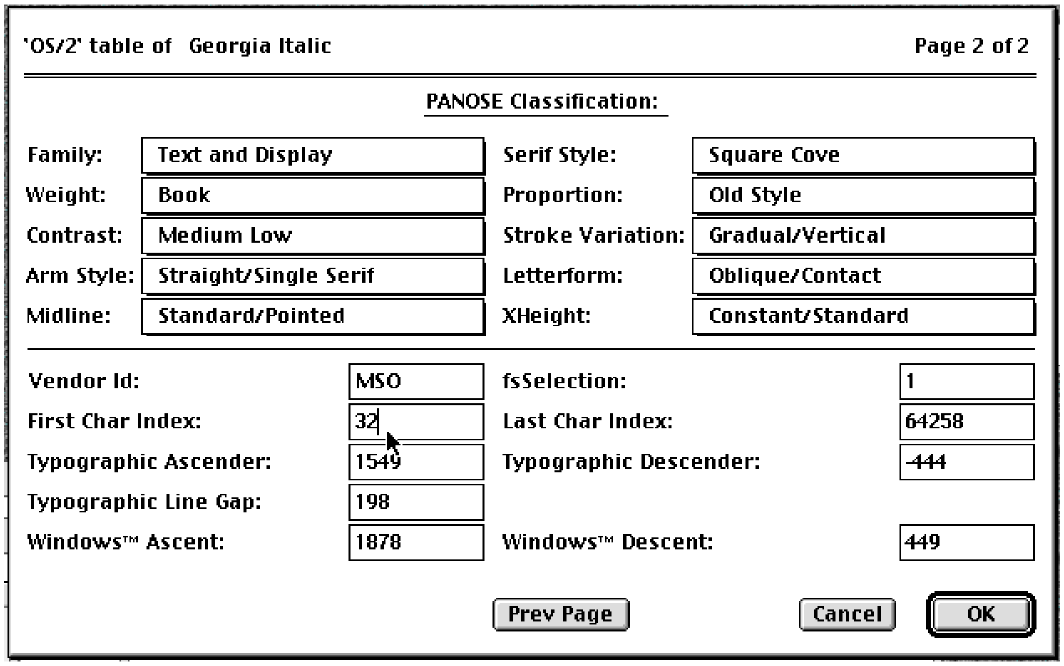
URI Binding

A process of locking a particular font resource to a given Web site by embedding an encrypted URI or a digitally signed usage assertion into the font resource.

C.2 Font retrieval

There are many different font formats in use by many different platforms. To select a preferred font format, transparent content negotiation is used (see [NEGOT]). It is always possible to tell when a font is being dereferenced, because the URI is inside a font description. A given implementation will know which downloadable font formats it supports and can thus use the format hint to avoid downloading fonts in an unsupported format.

C.3 Meaning of the Panose Digits


'OS/2' table of Georgia Italic

Page 2 of 2

PANOSE Classification:

Family:	Text and Display	Serif Style:	Square Cove
Weight:	Book	Proportion:	Old Style
Contrast:	Medium Low	Stroke Variation:	Gradual/Vertical
Arm Style:	Straight/Single Serif	Letterform:	Oblique/Contact
Midline:	Standard/Pointed	XHeight:	Constant/Standard

Vendor Id:	MSO	fsSelection:	1
First Char Index:	32	Last Char Index:	64258
Typographic Ascender:	1549	Typographic Descender:	-444
Typographic Line Gap:	198		
Windows™ Ascent:	1878	Windows™ Descent:	449

The Family, Serif Style and Proportion numbers are used by Windows95 for font selection and matching.

The meaning of the ten numbers and the allowable values (given in parentheses) are given below for the most common case, where the "family" digit is `2, Text and Display`. (If the first digit has a different value, the remaining nine digits have different meanings). For further details on Panose-1, see [PANOSE].

Family
- Any (0)
- No Fit (1)
- [PANOSE] **Latin Text and Display** (2)
- [PANOSE] **Latin Script** (3)
- [PANOSE] **Latin Decorative** (4)
- [PANOSE] **Latin Pictorial** (5)

Serif Style
- Any (0)
- No Fit (1)
- Cove (2)
- Obtuse Cove (3)

- Square Cove (4)
- Obtuse Square Cove (5)
- Square (6)
- Thin (7)
- Bone (8)
- Exaggerated (9)
- Triangle (10)
- Normal Sans (11)
- Obtuse Sans (12)
- Perp Sans (13)
- Flared (14)
- Rounded (15)

Weight

- Any (0)
- No Fit (1)
- Very Light (2)[100]
- Light (3) [200]
- Thin (4) [300]
- Book (5) [400] *same as CSS1 'normal'*
- Medium (6) [500]
- Demi (7) [600]
- Bold (8) [700] *same as CSS1 'bold'*
- Heavy (9) [800]
- Black (10) [900]
- Extra Black / Nord (11) [900] *force mapping to CSS1 100-900 scale*

Proportion

- Any (0)
- No Fit (1)
- Old Style (2)
- Modern (3)
- Even Width (4)
- Expanded (5)
- Condensed (6)
- Very Expanded (7)
- Very Condensed (8)
- Monospaced (9)

Contrast

- Any (0)
- No Fit (1)
- None (2)
- Very Low (3)
- Low (4)
- Medium Low (5)
- Medium (6)
- Medium High (7)
- High (8)
- Very High (9)

Stroke Variation

- Any (0)
- No Fit (1)
- No Variation (2)
- Gradual/Diagonal (3)
- Gradual/Transitional (4)
- Gradual/Vertical (5)
- Gradual/Horizontal (6)
- Rapid/Vertical (7)
- Rapid/Horizontal (8)
- Instant/Horizontal (9)
- Instant/Vertical (10)

Arm Style

- Any (0)
- No Fit (1)
- Straight Arms/Horizontal (2)
- Straight Arms/Wedge (3)
- Straight Arms/Vertical (4)
- Straight Arms/Single Serif (5)
- Straight Arms/Double Serif (6)
- Non-Straight Arms/Horizontal (7)
- Non-Straight Arms/Wedge (8)
- Non-Straight Arms/Vertical 90)
- Non-Straight Arms/Single Serif (10)
- Non-Straight Arms/Double Serif (11)

Letterform

- Any (0)
- No Fit (1)
- Normal/Contact (2)
- Normal/Weighted (3)
- Normal/Boxed (4)
- Normal/Flattened (5)
- Normal/Rounded (6)
- Normal/Off Center (7)
- Normal/Square (8)
- Oblique/Contact (9)
- Oblique/Weighted (10)
- Oblique/Boxed (11)
- Oblique/Flattened (12)
- Oblique/Rounded (13)
- Oblique/Off Center (14)
- Oblique/Square (15)

Midline

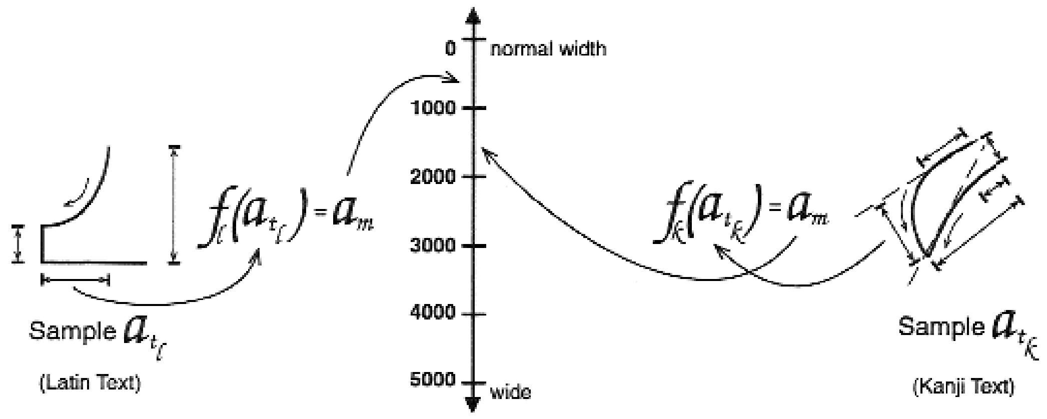
- Any (0)
- No Fit (1)
- Standard/Trimmed (2)
- Standard/Pointed (3)

- Standard/Serifed (4)
- High/Trimmed (5)
- High/Pointed (6)
- High/Serifed (7)
- Constant/Trimmed (8)
- Constant/Pointed (9)
- Constant/Serifed (10)
- Low/Trimmed (11)
- Low/Pointed (12)
- Low/Serifed (13)

XHeight

- Any (0)
- No Fit (1)
- Constant/Small (2)
- Constant/Standard (3)
- Constant/Large (4)
- Ducking/Small (5)
- Ducking/Standard (6)
- Ducking/Large (7)

Panose-2 (see [PANOSE2]) is a specification for a more comprehensive font classification and matching technology which is not limited to Latin typefaces. For example, the serif characteristics of a Latin face may be compared with the stroke terminations of a Kanji face.



The Panose-2 value is not stored inside any known font formats, but may be measured.

C.4 Deducing Unicode Ranges for TrueType

This information is available in the font by looking at the ’ulUnicodeRange’ bits in the ’OS/2’ table (if it has one), which holds a bitfield representation of the set. This table is defined in revision 1.66 of the TrueType specification, from Microsoft. Considering this information as a set, each element corresponds to a Unicode 1.1 character block, and the presence of that element in the set indicates that the font has one or more glyph representations to represent at least one character in that block. The set has 128 elements as described below. The order generally follows that in the Unicode 1.1 standard. This table may be used to convert the information in a TrueType font into a CSS ’unicode-range’ descriptor.

Block	**Add**	**Block name**	**Unicode range**
0	1	Basic Latin	U+0-7F
1	2	Latin-1 Supplement	U+80-FF
2	4	Latin-1 Extended-A	U+100-17F
3	8	Latin Extended-B	U+180-24F
4	1	IPA Extensions	U+250-2AF
5	2	Spacing Modifier Letters	U+2B0-2FF
6	4	Combining Diacritical Marks	U+300-36F
7	8	Greek	U+370-3CF
8	1	*Greek Symbols and Coptic*	U+3D0-3EF
9	2	Cyrillic	U+400-4FF
10	4	Armenian	U+530-58F
11	8	Hebrew	U+590-5FF
12	1	*Hebrew Extended-A* *Hebrew Extended-B*	?? what ranges ??
13	2	Arabic	U+600-69F
14	4	*Arabic Extended*	U+670-6FF
15	8	Devanagari	U+900-97F
16	1	Bengali	U+980-9FF
17	2	Gurmukhi	U+A00-A7F
18	4	Gujarati	U+A80-AFF
19	8	Oriya	U+B00-B7F
20	1	Tamil	U+B80-BFF
21	2	Telugu	U+C00-C7F
22	4	Kannada	U+C80-CFF
23	8	Malayalam	U+D00-D7F

Block	Add	Block name	Unicode range
24	1	Thai	U+E00-E7F
25	2	Lao	U+E80-EFF
26	4	Georgian	U+10A0-10EF
27	8	*Georgian Extended*	U+10F0-10FF ??
28	1	Hangul Jamo	U+1100-11FF
29	2	Latin Extended Additional	-
30	4	Greek Extended	U+1F00-1FFF
31	8	General Punctuation	U+2000-206F
32	1	Superscripts and Subscripts	-
33	2	Currency Symbols	U+20A0-20CF
34	4	Combining Marks for Symbols	U+20D0-20FF
35	8	Letterlike Symbols	U+2100-214F
36	1	Number Forms	U+2150-218F
37	2	Arrows	U+2190-21FF
38	4	Mathematical Operators	U+2200-22FF
39	8	Miscellaneous Technical	U+2300-23FF
40	1	Control Pictures	U+2400-243F
41	2	Optical Character Recognition	U+2440-245F
42	4	Enclosed Alphanumerics	U+2460-24FF
43	8	Box Drawing	U+2500-257F
44	1	Block Elements	U+2580-259F
45	2	Geometric Shapes	U+25A0-25FF
46	4	Miscellaneous Symbols	U+2600-26FF
47	8	Dingbats	U+2700-27BF
48	1	CJK Symbols and Punctuation	U+3000-303F
49	2	Hiragana	U+3040-309F

Block	Add	Block name	Unicode range
50	4	Katakana	U+30A0-30FF
51	8	Bopomofo	U+3100-312F
52	1	Hangul Compatibility Jamo	U+3130-318F
53	2	CJK Miscellaneous	??
54	4	Enclosed CJK Letters and Months	U+3200-32FF
55	8	CJK compatibility	U+3300-33FF
56	1	Hangul	U+AC00-D7FF
59	8	CJK Unified Ideographs	U+4E00-9FFF
60	1	Private Use Area	U+E000-F8FF
61	2	CJK Compatibility Ideographs	U+F900-FAFF
62	4	Alphabetic Presentation Forms	U+FB00-FB4F
63	8	Arabic Presentation Forms-A	U+FB50-FDFF
64	1	Combining Half Marks	U+FE20-FE2F
65	2	CJK compatibility Forms	U+FE30-FE4F
66	4	Small Form Variants	U+FE50-FE6F
67	8	Arabic Presentation Forms-B	U+FE70-FEFF
68	1	Halfwidth and Fullwidth Forms	U+FF00-FFEF
69	2	Specials	U+FFF0-FFFD

The TrueType bitfield system has the problem that it is tied to Unicode 1.1 and is unable to cope with Unicode expansion - it is unable to represent Tibetan for example, or other scripts introduced with Unicode 2.0 or later revisions.

C.5 Automatic descriptor generation

Authoring tools should allow style sheet authors to add and edit font descriptors. In some cases, however, authoring tools can help by examining locally installed fonts and automatically generating descriptors for fonts referenced in the style sheet. This is also a function that can be carried out by tools which subset or convert fonts ready for dynamic download.

This table suggests where such information can be found, for common font formats.

Descriptor	Type 1	TrueType and OpenType	TrueType GX [TRUETYPEGX]
'ascent'	`'Ascender'` in AFM/PFM file	`'Ascender'` in `'hhea'` table or (preferably) `'sTypoAscender'` in `'OS/2'` table	`'horizontalBefore'` in `'fmtx'` table
'baseline'			`bsln` table, see note [p. 306]
'bbox'	`FontBBox`, font dictionary	`'xMin'`, `'xMax'`, `'yMin'` and `'yMax'` entries of the `'head'` table	
'cap-height'	`CapHeight` in AFM/PFM file		
'descent'	`'Descender'` in the AFM/PFM file.		
'mathline'			`bsln` table
'font-family'	`FamilyName`, fontinfo dictionary	`name` table	
'stemh'	`StdHW`, private dictionary of AFM/PFM file		
'stemv'	`/StdVW`, private dictionary	`cvt` table	
'topline'			`bsln` table
'unicode-range'	cmap file	`OS/2` table, see Appendix C [p. 302]	
'units-per-em'	`FontMatrix`, font dictionary	`unitsPerEm`, head table.	
'widths'		`hmtx` table	

- Within the `bsln` table, the `ideographic centered baseline` may be used for stretches of predominantly ideographic characters and the `ideographic low baseline` is more suitable for ideographic characters in a run of predominantly Latin, Greek or Cyrillic characters.

Appendix D. The grammar of CSS2

Contents

This appendix is normative.

The grammar below defines the syntax of CSS2. It is in some sense, however, a superset of CSS2 as this specification imposes additional semantic constraints not expressed in this grammar. A conforming UA must also adhere to the forward-compatible parsing rules [p. 35] , the property and value notation [p. 14] , and the unit notation. In addition, the document language may impose restrictions, e.g. HTML imposes restrictions on the possible values of the "class" attribute.

D.1 Grammar

The grammar below is LL(1) (but note that most UA's should not use it directly, since it doesn't express the parsing conventions [p. 42] , only the CSS2 syntax). The format of the productions is optimized for human consumption and some shorthand notation beyond Yacc (see [YACC]) is used:

- *****: 0 or more
- **+**: 1 or more
- **?**: 0 or 1
- **|**: separates alternatives
- **[ ]**: grouping

The productions are:

```
stylesheet
  : [ CHARSET_SYM S* STRING S* ';' ]?
    [S|CDO|CDC]* [ import [S|CDO|CDC]* ]*
    [ [ ruleset | media | page | font_face ] [S|CDO|CDC]* ]*
  ;
import
  : IMPORT_SYM S*
    [STRING|URI] S* [ medium [ ',' S* medium]* ]? ';' S*
  ;
media
  : MEDIA_SYM S* medium [ ',' S* medium ]* '{' S* ruleset* '}' S*
  ;
medium
  : IDENT S*
  ;
page
  : PAGE_SYM S* IDENT? pseudo_page? S*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
```

```
pseudo_page
  : ':' IDENT
  ;
font_face
  : FONT_FACE_SYM S*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
operator
  : '/' S* | ',' S* | /* empty */
  ;
combinator
  : '+' S* | '>' S* | /* empty */
  ;
unary_operator
  : '-' | '+'
  ;
property
  : IDENT S*
  ;
ruleset
  : selector [ ',' S* selector ]*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
selector
  : simple_selector [ combinator simple_selector ]*
  ;
simple_selector
  : element_name? [ HASH | class | attrib | pseudo ]* S*
  ;
class
  : '.' IDENT
  ;
element_name
  : IDENT | '*'
  ;
attrib
  : '[' S* IDENT S* [ [ '=' | INCLUDES | DASHMATCH ] S*
    [ IDENT | STRING ] S* ]? ']'
  ;
pseudo
  : ':' [ IDENT | FUNCTION S* IDENT S* ')' ]
  ;
declaration
  : property ':' S* expr prio?
  | /* empty */
  ;
prio
  : IMPORTANT_SYM S*
  ;
expr
  : term [ operator term ]*
  ;
term
  : unary_operator?
    [ NUMBER S* | PERCENTAGE S* | LENGTH S* | EMS S* | EXS S* | ANGLE S* |
      TIME S* | FREQ S* | function ]
  | STRING S* | IDENT S* | URI S* | RGB S* | UNICODERANGE S* | hexcolor
  ;
function
  : FUNCTION S* expr ')' S*
  ;
/*
 * There is a constraint on the color that it must
 * have either 3 or 6 hex-digits (i.e., [0-9a-fA-F])
 * after the "#"; e.g., "#000" is OK, but "#abcd" is not.
 */
hexcolor
  : HASH S*
  ;
```

D.2 Lexical scanner

The following is the tokenizer, written in Flex (see [FLEX]) notation. The tokenizer is case-insensitive.

The two occurrences of "\377" represent the highest character number that current versions of Flex can deal with (decimal 255). They should be read as "\4177777" (decimal 1114111), which is the highest possible code point in Unicode/ISO-10646.

```
%option case-insensitive

h               [0-9a-f]
nonascii        [\200-\377]
unicode         \\{h}{1,6}[ \t\r\n\f]?
escape          {unicode}|\\[ -~\200-\377]
nmstart         [a-z]|{nonascii}|{escape}
nmchar          [a-z0-9-]|{nonascii}|{escape}
string1         \"([\t !#$%&(-~]|\\{nl}|\’|{nonascii}|{escape})*\"
string2         \’([\t !#$%&(-~]|\\{nl}|\"|{nonascii}|{escape})*\’

ident           {nmstart}{nmchar}*
name            {nmchar}+
num             [0-9]+|[0-9]*"."[0-9]+
string          {string1}|{string2}
url             ([!#$%&*-~]|{nonascii}|{escape})*
w               [ \t\r\n\f]*
nl              \n|\r\n|\r|\f
range           \?{1,6}|{h}(\?{0,5}|{h}(\?{0,4}|{h}(\?{0,3}|{h}(\?{0,2}|{h}(\??|{h})))))

%%

[ \t\r\n\f]+            {return S;}

\/\*[^*]*\*+([^/][^*]*\*+)*\/   /* ignore comments */

"<!--"                  {return CDO;}
"-->"                   {return CDC;}
"~="                    {return INCLUDES;}
"|="                    {return DASHMATCH;}

{string}                {return STRING;}

{ident}                 {return IDENT;}

"#"{name}               {return HASH;}

"@import"               {return IMPORT_SYM;}
"@page"                 {return PAGE_SYM;}
"@media"                {return MEDIA_SYM;}
"@font-face"            {return FONT_FACE_SYM;}
"@charset"              {return CHARSET_SYM;}
"@"{ident}              {return ATKEYWORD;}

"!{w}important"         {return IMPORTANT_SYM;}

{num}em                 {return EMS;}
{num}ex                 {return EXS;}
{num}px                 {return LENGTH;}
{num}cm                 {return LENGTH;}
{num}mm                 {return LENGTH;}
{num}in                 {return LENGTH;}
{num}pt                 {return LENGTH;}
{num}pc                 {return LENGTH;}
{num}deg                {return ANGLE;}
```

```
{num}rad               {return ANGLE;}
{num}grad              {return ANGLE;}
{num}ms                {return TIME;}
{num}s                 {return TIME;}
{num}Hz                {return FREQ;}
{num}kHz               {return FREQ;}
{num}{ident}           {return DIMEN;}
{num}%                 {return PERCENTAGE;}
{num}                  {return NUMBER;}

"url("{w}{string}{w}")" {return URI;}
"url("{w}{url}{w}")"    {return URI;}
{ident}"("             {return FUNCTION;}

U\+{range}             {return UNICODERANGE;}
U\+{h}{1,6}-{h}{1,6}   {return UNICODERANGE;}

.                      {return *yytext;}
```

D.3 Comparison of tokenization in CSS2 and CSS1

There are some differences in the syntax specified in the CSS1 recommendation ([CSS1]), and the one above. Most of these are due to new tokens in CSS2 that didn’t exist in CSS1. Others are because the grammar has been rewritten to be more readable. However, there are some incompatible changes, that were felt to be errors in the CSS1 syntax. They are explained below.

- CSS1 style sheets could only be in 1-byte-per-character encodings, such as ASCII and ISO-8859-1. CSS2 has no such limitation. In practice, there was little difficulty in extrapolating the CSS1 tokenizer, and some UAs have accepted 2-byte encodings.
- CSS1 only allowed four hex-digits after the backslash (\) to refer to Unicode characters, CSS2 allows six [p. 38] . Furthermore, CSS2 allows a whitespace character to delimit the escape sequence. E.g., according to CSS1, the string "\abcdef" has 3 letters (\abcd, e, and f), according to CSS2 it has only one (\abcdef).
- The tab character (ASCII 9) was not allowed in strings. However, since strings in CSS1 were only used for font names and for URLs, the only way this can lead to incompatibility between CSS1 and CSS2 is if a style sheet contains a font family that has a tab in its name.
- Similarly, newlines (escaped with a backslash [p. 50]) were not allowed in strings in CSS1.
- CSS2 parses a number immediately followed by an identifier as a DIMEN token (i.e., an unknown unit), CSS1 parsed it as a number and an identifier. That means that in CSS1, the declaration ’font: 10pt/1.2serif’ was correct, as was ’font: 10pt/12pt serif’; in CSS2, a space is required before "serif". (Some UAs accepted the first example, but not the second.)
- In CSS1, a class name could start with a digit (".55ft"), unless it was a dimension (".55in"). In CSS2, such classes are parsed as unknown dimensions (to allow for future additions of new units). To make ".55ft" a valid class, CSS2 requires the first digit to be escaped (".\55ft")

Appendix E. References

Contents

E.1 Normative references

[COLORIMETRY]
"Colorimetry, Second Edition", CIE Publication 15.2-1986, ISBN 3-900-734-00-3.
Available at http://www.hike.te.chiba-u.ac.jp/ikeda/CIE/publ/abst/15-2-86.html.

[CSS1]
"Cascading Style Sheets, level 1", H. W. Lie and B. Bos, 17 December 1996. Available at http://www.w3.org/TR/REC-CSS1-961217.html.

[FLEX]
"Flex: The Lexical Scanner Generator", Version 2.3.7, ISBN 1882114213.

[HTML40]
"HTML 4.0 Specification", D. Raggett, A. Le Hors, I. Jacobs, 8 July 1997. Available at http://www.w3.org/TR/REC-html40/. The Recommendation defines three document type definitions: Strict, Transitional, and Frameset, all reachable from the Recommendation.

[IANA]
"Assigned Numbers", STD 2, RFC 1700, USC/ISI, J. Reynolds and J. Postel, October 1994.
Available at ftp://ftp.internic.net/rfc/rfc1700.txt.

[ICC32]
"ICC Profile Format Specification, version 3.2", 1995.
Available at ftp://sgigate.sgi.com/pub/icc/ICC32.pdf.

[ISO8879]
ISO 8879:1986 "Information Processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML)", ISO 8879:1986.
For the list of SGML entities, consult ftp://ftp.ifi.uio.no/pub/SGML/ENTITIES/.

[ISO10646]
"Information Technology - Universal Multiple- Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", ISO/IEC 10646-1:1993. The current specification also takes into consideration the first five amendments to ISO/IEC 10646-1:1993. Useful roadmap of the BMP and roadmap of plane 1 documents show which scripts sit at which numeric ranges.

[PNG10]
"PNG (Portable Network Graphics) Specification, Version 1.0 specification", T. Boutell ed., 1 October 1996.
Available at http://www.w3.org/pub/WWW/TR/REC-png-multi.html.

[RFC1808]
"Relative Uniform Resource Locators", R. Fielding, June 1995.
Available at ftp://ds.internic.net/rfc/rfc1808.txt.

[RFC2045]
"Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", N. Freed and N. Borenstein, November 1996.
Available at ftp://ftp.internic.net/rfc/rfc2045.txt. Note that this RFC obsoletes RFC1521, RFC1522, and RFC1590.

[RFC2068]
"HTTP Version 1.1 ", R. Fielding, J. Gettys, J. Mogul, H. Frystyk Nielsen, and T. Berners-Lee, January 1997.
Available at ftp://ftp.internic.net/rfc/rfc2068.txt.

[RFC2070]
"Internationalization of the HyperText Markup Language", F. Yergeau, G. Nicol, G. Adams, and M. Dürst, January 1997.
Available at ftp://ds.internic.net/rfc/rfc2070.txt.

[RFC2119]
"Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997.
Available at ftp://ds.internic.net/rfc/rfc2119.txt.

[RFC2318]
"The text/css Media Type", H. Lie, B. Bos, C. Lilley, March 1998.
Available at ftp://ds.internic.net/rfc/rfc2318.txt.

[RFC1738]
"Uniform Resource Locators", T. Berners-Lee, L. Masinter, and M. McCahill, December 1994.
Available at ftp://ds.internic.net/rfc/rfc1738.txt.

[SRGB]
"Proposal for a Standard Color Space for the Internet - sRGB", M. Anderson, R. Motta, S. Chandrasekar, M. Stokes.
Available at http://www.w3.org/Graphics/Color/sRGB.html.

[UNICODE]
"The Unicode Standard: Version 2.0", The Unicode Consortium, Addison-Wesley Developers Press, 1996. For bidirectionality, see also the corrigenda at http://www.unicode.org/unicode/uni2errata/bidi.htm. For more information, consult the Unicode Consortium’s home page at http://www.unicode.org/.
The latest version of Unicode. For more information, consult the Unicode Consortium’s home page at http://www.unicode.org/.

[URI]
"Uniform Resource Identifiers (URI): Generic Syntax and Semantics", T. Berners-Lee, R. Fielding, L. Masinter, 18 November 1997.
Available at http://www.ics.uci.edu/pub/ietf/uri/draft-fielding-uri-syntax-01.txt. This is a work in progress that is expected to update [RFC1738] [p. 314] and [RFC1808] [p. 314] .

[XML10]
"Extensible Markup Language (XML) 1.0" T. Bray, J. Paoli, C.M. Sperberg-McQueen, editors, 10 February 1998.
Available at http://www.w3.org/TR/REC-xml/.

[YACC]
"YACC - Yet another compiler compiler", S. C. Johnson, Technical Report, Murray Hill, 1975.

E.2 Informative references

[CHARSETS]
Registered charset values. Download a list of registered charset values from ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

[DOM]
"Document Object Model Specification", L. Wood, A. Le Hors, 9 October 1997.
Available at http://www.w3.org/TR/WD-DOM/

[ISO10179]
ISO/IEC 10179:1996 "Information technology -- Processing languages -- Document Style Semantics and Specification Language (DSSSL)"
Available at http://occam.sjf.novell.com:8080/dsssl/dsssl96.

[GAMMA]
"Gamma correction on the Macintosh Platform", C. A. Poynton.
Available at
ftp://ftp.inforamp.net/pub/users/poynton/doc/Mac/Mac_gamma.pdf.

[HTML32]
"HTML 3.2 Reference Specification", Dave Raggett, 14 January 1997.
Available at http://www.w3.org/TR/REC-html32.html.

[INFINIFONT]
See http://www.fonts.com/hp/infinifont/moredet.html.

[ISO9899]
ISO/IEC 9899:1990 Programming languages -- C.

[MONOTYPE]
See http://www.monotype.com/html/oem/uni_scrmod.html.

[NEGOT]
"Transparent Content Negotiation in HTTP", K. Holtman, A. Mutz, 9 March, 1997.
Available at http://gewis.win.tue.nl/~koen/conneg/draft-ietf-http-negotiation-01.html.

[OPENTYPE]
See http://www.microsoft.com/OpenType/OTSpec/tablist.htm.

[PANOSE]
For information about PANOSE classification metrics, consult http://www.fonts.com/hp/panose/greybook and the following chapters: Latin Text, Latin Script, Latin Decorative, and Latin Pictorial.
Panose numbers for some fonts are available online and may be queried.

[PANOSE2]
See http://www.w3.org/Fonts/Panose/pan2.html Panose-2 is not limited to Latin typefaces.

[POSTSCRIPT]
"The PostScript Language Reference Manual", Second Edition, Adobe Systems, Inc., Addision-Wesley Publishing Co., December 1990.

[RFC1630]
"Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", T. Berners-Lee, June 1994.
Available at ftp://ftp.internic.net/rfc/rfc1630.txt.

[RFC1766]
"Tags for the Identification of Languages", H. Alvestrand, March 1995.
Available at ftp://ftp.internic.net/rfc/rfc1766.txt.

[RFC1866]
"HyperText Markup Language 2.0", T. Berners-Lee and D. Connolly, November 1995.
Available at ftp://ds.internic.net/rfc/rfc1866.txt.

[RFC1942]
"HTML Tables", Dave Raggett, May 1996.
Available at ftp://ds.internic.net/rfc/rfc1942.txt.

[TRUETYPEGX]
See http://fonts.apple.com/TTRefMan/index.html for details about TrueType GX from Apple Computer, including descriptions of the added tables and font quality specifications.

[W3CSTYLE]
W3C resource page on web style sheets.
Examine at http://www.w3.org/pub/WWW/Style.

[WAI-PAGEAUTH]
"WAI Accesibility Guidelines: Page Authoring" for designing accessible documents are available at:
http://www.w3.org/TR/WD-WAI-PAGEAUTH.

Appendix F. Property index

Name	Values	Initial value	Applies to (Default: all)	Inherited?	Percentages (Default: N/A)	Media groups
’azimuth’ [p. 283]	<angle> | [[left-side | far-left | left | center-left | center | center-right | right | far-right | right-side] || behind] | leftwards | rightwards | inherit	center		yes		aural [p. 79]
’background’ [p. 192]	[’background-color’ || ’background-image’ || ’background-repeat’ || ’background-attachment’ || ’background-position’] | inherit	XX		no	allowed on ’back-ground-position’	visual [p. 79]
’background-attachment’ [p. 190]	scroll | fixed | inherit	scroll		no		visual [p. 79]
’background-color’ [p. 188]	<color> | transparent | inherit	transparent		no		visual [p. 79]
’background-image’ [p. 188]	<uri> | none | inherit	none		no		visual [p. 79]
’background-position’ [p. 191]	[[<percentage> | <length>]{1,2} | [[top | center | bottom] || [left | center | right]]] | inherit	0% 0%	block-level and replaced elements	no	refer to the size of the box itself	visual [p. 79]
’background-repeat’ [p. 189]	repeat | repeat-x | repeat-y | no-repeat | inherit	repeat		no		visual [p. 79]
’border’ [p. 92]	[’border-width’ || ’border-style’ || <color>] | inherit	see individ-ual proper-ties		no		visual [p. 79]
’border-collapse’ [p. 261]	collapse | separate | inherit	collapse	’table’ and ’inline-table’ elements	yes		visual [p. 79]
’border-color’ [p. 90]	<color>{1,4} | transpar-ent | inherit	see individ-ual proper-ties		no		visual [p. 79]
’border-spacing’ [p. 261]	<length> <length>? | inherit	0	’table’ and ’inline-table’ elements	yes		visual [p. 79]
’border-style’ [p. 91]	<border-style>{1,4} | inherit	see individ-ual proper-ties		no		visual [p. 79]
’border-top’ [p. 92] ’border-right’ [p. 92] ’border-bottom’ [p. 92] ’border-left’ [p. 92]	[’border-top-width’ || ’border-style’ || <color>] | inherit	see individ-ual proper-ties		no		visual [p. 79]
’border-top-color’ [p. 89] ’border-right-color’ [p. 89] ’border-bottom-color’ [p. 89] ’border-left-color’ [p. 89]	<color> | inherit	the value of the ’color’ property		no		visual [p. 79]
’border-top-style’ [p. 91] ’border-right-style’ [p. 91] ’border-bottom-style’ [p. 91] ’border-left-style’ [p. 91]	<border-style> | inherit	none		no		visual [p. 79]

Name	Values	Initial value	Applies to (Default: all)	Inherited?	Percentages (Default: N/A)	Media groups
’border-top-width’ [p. 88] ’border-right-width’ [p. 88] ’border-bottom-width’ [p. 88] ’border-left-width’ [p. 88]	<border-width> | inherit	medium		no		visual [p. 79]
’border-width’ [p. 89]	<border-width>{1,4} | inherit	see individ- ual proper- ties		no		visual [p. 79]
’bottom’ [p. 104]	<length> | <percentage> | auto | inherit	auto	positioned elements	no	refer to height of containing block	visual [p. 79]
’caption-side’ [p. 251]	top | bottom | left | right | inherit	top	’table-caption’ elements	yes		visual [p. 79]
’clear’ [p. 112]	none | left | right | both | inherit	none	block-level elements	no		visual [p. 79]
’clip’ [p. 147]	<shape> | auto | inherit	auto	block-level and replaced elements	no		visual [p. 79]
’color’ [p. 187]	<color> | inherit	depends on user agent		yes		visual [p. 79]
’content’ [p. 155]	[<string> | <uri> | <counter> | attr(X) | open-quote | close-quote | no-open-quote | no-close-quote]+ | inherit	empty string	:before and :after pseudo-elements	no		all [p. 79]
’counter-increment’ [p. 161]	[<identifier> <integer>?]+ | none | inherit	none		no		all [p. 79]
’counter-reset’ [p. 160]	[<identifier> <integer>?]+ | none | inherit	none		no		all [p. 79]
’cue’ [p. 281]	[’cue-before’ || ’cue-after’] | inherit	XX		no		aural [p. 79]
’cue-after’ [p. 281]	<uri> | none | inherit	none		no		aural [p. 79]
’cue-before’ [p. 281]	<uri> | none | inherit	none		no		aural [p. 79]
’cursor’ [p. 271]	[[<uri> ,]* [auto | crosshair | default | pointer | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize| text | wait | help]] | inherit	auto		yes		visual [p. 79] , interactive [p. 79]
’direction’ [p. 128]	ltr | rtl | inherit	ltr	all elements, but see prose	yes		visual [p. 79]
’display’ [p. 101]	inline | block | list-item | run-in | compact | marker | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none | inherit	inline		no		all [p. 79]
’elevation’ [p. 284]	<angle> | below | level | above | higher | lower | inherit	level		yes		aural [p. 79]
’empty-cells’ [p. 262]	show | hide | inherit	show	’table-cell’ elements	yes		visual [p. 79]

Name	Values	Initial value	Applies to (Default: all)	Inherited?	Percentages (Default: N/A)	Media groups
’float’ [p. 111]	left | right | none | inherit	none	all but positioned elements and generated content	no		visual [p. 79]
’font’ [p. 207]	[[’font-style’ || ’font-variant’ || ’font-weight’]? ’font-size’ [/ ’line-height’]? ’font-family’] | caption | icon | menu | message-box | small-caption | status-bar | inherit	see individ-ual proper-ties		yes	allowed on ’font-size’ and ’line-height’	visual [p. 79]
’font-family’ [p. 199]	[[<family-name> | <generic-family>],]* [<family-name> | <generic-family>] | inherit	depends on user agent		yes		visual [p. 79]
’font-size’ [p. 203]	<absolute-size> | <rela-tive-size> | <length> | <percentage> | inherit	medium		yes, the computed value is inherited	refer to parent element’s font size	visual [p. 79]
’font-size-adjust’ [p. 204]	<number> | none | inherit	none		yes		visual [p. 79]
’font-stretch’ [p. 202]	normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded | inherit	normal		yes		visual [p. 79]
’font-style’ [p. 200]	normal | italic | oblique | inherit	normal		yes		visual [p. 79]
’font-variant’ [p. 201]	normal | small-caps | inherit	normal		yes		visual [p. 79]
’font-weight’ [p. 201]	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit	normal		yes		visual [p. 79]
’height’ [p. 137]	<length> | <percentage> | auto | inherit	auto	all elements but non-replaced inline elements, table columns, and column groups	no	see prose	visual [p. 79]
’left’ [p. 104]	<length> | <percentage> | auto | inherit	auto	positioned elements	no	refer to width of containing block	visual [p. 79]
’letter-spacing’ [p. 241]	normal | <length> | inherit	normal		yes		visual [p. 79]
’line-height’ [p. 142]	normal | <number> | <length> | <percentage> | inherit	normal		yes	refer to the font size of the element itself	visual [p. 79]
’list-style’ [p. 171]	[’list-style-type’ || ’list-style-position’ || ’list-style-image’] | inherit	XX	elements with ’display: list-item’	yes		visual [p. 79]
’list-style-image’ [p. 170]	<uri> | none | inherit	none	elements with ’display: list-item’	yes		visual [p. 79]
’list-style-position’ [p. 170]	inside | outside | inherit	outside	elements with ’display: list-item’	yes		visual [p. 79]

Name	Values	Initial value	Applies to (Default: all)	Inherited?	Percentages (Default: N/A)	Media groups
’list-style-type’ [p. 168]	disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-alpha | lower-latin | upper-alpha | upper-latin | hebrew | armenian | georgian | cjk-ideographic | hira-gana | katakana | hira-gana-iroha | katakana-iroha | none | inherit	disc	elements with ’display: list-item’	yes		visual [p. 79]
’margin’ [p. 85]	<margin-width>{1,4} | inherit	XX		no	refer to width of containing block	visual [p. 79]
’margin-top’ [p. 85] ’margin-right’ [p. 85] ’margin-bottom’ [p. 85] ’margin-left’ [p. 85]	<margin-width> | inherit	0		no	refer to width of containing block	visual [p. 79]
’marker-offset’ [p. 167]	<length> | auto | inherit	auto	elements with ’display: marker’	no		visual [p. 79]
’marks’ [p. 179]	[crop || cross] | none | inherit	none	page context	N/A		visual [p. 79] , paged [p. 79]
’max-height’ [p. 140]	<length> | <percentage> | none | inherit	none	all elements except non-replaced inline elements and table elements	no	refer to height of containing block	visual [p. 79]
’max-width’ [p. 137]	<length> | <percentage> | none | inherit	none	all elements except non-replaced inline elements and table elements	no	refer to width of containing block	visual [p. 79]
’min-height’ [p. 140]	<length> | <percentage> | inherit	0	all elements except non-replaced inline elements and table elements	no	refer to height of containing block	visual [p. 79]
’min-width’ [p. 136]	<length> | <percentage> | inherit	UA depen-dent	all elements except non-replaced inline elements and table elements	no	refer to width of containing block	visual [p. 79]
’orphans’ [p. 182]	<integer> | inherit	2	block-level elements	yes		visual [p. 79] , paged [p. 79]
’outline’ [p. 274]	[’outline-color’ || ’outline-style’ || ’outline-width’] | inherit	see individ-ual proper-ties		no		visual [p. 79] , interactive [p. 79]
’outline-color’ [p. 274]	<color> | invert | inherit	invert		no		visual [p. 79] , interactive [p. 79]
’outline-style’ [p. 274]	<border-style> | inherit	none		no		visual [p. 79] , interactive [p. 79]
’outline-width’ [p. 274]	<border-width> | inherit	medium		no		visual [p. 79] , interactive [p. 79]

Name	Values	Initial value	Applies to (Default: all)	Inherited?	Percentages (Default: N/A)	Media groups
'overflow' [p. 145]	visible | hidden | scroll | auto | inherit	visible	block-level and replaced elements	no		visual [p. 79]
'padding' [p. 87]	<padding-width>{1,4} | inherit	XX		no	refer to width of containing block	visual [p. 79]
'padding-top' [p. 87] 'padding-right' [p. 87] 'padding-bottom' [p. 87] 'padding-left' [p. 87]	<padding-width> | inherit	0		no	refer to width of containing block	visual [p. 79]
'page' [p. 182]	<identifier> | auto	auto	block-level elements	yes		visual [p. 79] , paged [p. 79]
'page-break-after' [p. 181]	auto | always | avoid | left | right | inherit	auto	block-level elements	no		visual [p. 79] , paged [p. 79]
'page-break-before' [p. 181]	auto | always | avoid | left | right | inherit	auto	block-level elements	no		visual [p. 79] , paged [p. 79]
'page-break-inside' [p. 181]	avoid | auto | inherit	auto	block-level elements	yes		visual [p. 79] , paged [p. 79]
'pause' [p. 280]	[[<time> | <percent-age>]{1,2}] | inherit	depends on user agent		no	see descriptions of 'pause-before' and 'pause-after'	aural [p. 79]
'pause-after' [p. 280]	<time> | <percentage> | inherit	depends on user agent		no	see prose	aural [p. 79]
'pause-before' [p. 279]	<time> | <percentage> | inherit	depends on user agent		no	see prose	aural [p. 79]
'pitch' [p. 286]	<frequency> | x-low | low | medium | high | x-high | inherit	medium		yes		aural [p. 79]
'pitch-range' [p. 287]	<number> | inherit	50		yes		aural [p. 79]
'play-during' [p. 282]	<uri> mix? repeat? | auto | none | inherit	auto		no		aural [p. 79]
'position' [p. 102]	static | relative | abso-lute | fixed | inherit	static	all elements, but not to generated content	no		visual [p. 79]
'quotes' [p. 157]	[<string> <string>]+ | none | inherit	depends on user agent		yes		visual [p. 79]
'richness' [p. 288]	<number> | inherit	50		yes		aural [p. 79]
'right' [p. 104]	<length> | <percentage> | auto | inherit	auto	positioned elements	no	refer to width of containing block	visual [p. 79]
'size' [p. 177]	<length>{1,2} | auto | portrait | landscape | inherit	auto	the page context	N/A		visual [p. 79] , paged [p. 79]
'speak' [p. 279]	normal | none | spell-out | inherit	normal		yes		aural [p. 79]
'speak-header' [p. 267]	once | always | inherit	once	elements that have table header informa-tion	yes		aural [p. 79]
'speak-numeral' [p. 289]	digits | continuous | inherit	continuous		yes		aural [p. 79]
'speak-punctuation' [p. 288]	code | none | inherit	none		yes		aural [p. 79]

Name	Values	Initial value	Applies to (Default: all)	Inherited?	Percentages (Default: N/A)	Media groups
'speech-rate' [p. 285]	<number> | x-slow | slow | medium | fast | x-fast | faster | slower | inherit	medium		yes		aural [p. 79]
'stress' [p. 287]	<number> | inherit	50		yes		aural [p. 79]
'table-layout' [p. 256]	auto | fixed | inherit	auto	'table' and 'inline-table' elements	no		visual [p. 79]
'text-align' [p. 238]	left | right | center | justify | <string> | inherit	depends on user agent and writing direction	block-level elements	yes		visual [p. 79]
'text-decoration' [p. 239]	none | [underline || overline || line-through || blink] | inherit	none		no (see prose)		visual [p. 79]
'text-indent' [p. 237]	<length> | <percentage> | inherit	0	block-level elements	yes	refer to width of containing block	visual [p. 79]
'text-shadow' [p. 239]	none | [<color> || <length> <length> <length>? ,]* [<color> || <length> <length> <length>?] | inherit	none		no (see prose)		visual [p. 79]
'text-transform' [p. 242]	capitalize | uppercase | lowercase | none | inherit	none		yes		visual [p. 79]
'top' [p. 104]	<length> | <percentage> | auto | inherit	auto	positioned elements	no	refer to height of containing block	visual [p. 79]
'unicode-bidi' [p. 128]	normal | embed | bidi-override | inherit	normal	all elements, but see prose	no		visual [p. 79]
'vertical-align' [p. 143]	baseline | sub | super | top | text-top | middle | bottom | text-bottom | <percentage> | <length> | inherit	baseline	inline-level and 'table-cell' elements	no	refer to the 'line-height' of the element itself	visual [p. 79]
'visibility' [p. 149]	visible | hidden | collapse | inherit	inherit		no		visual [p. 79]
'voice-family' [p. 286]	[[<specific-voice> | <generic-voice>],]* [<specific-voice> | <generic-voice>] | inherit	depends on user agent		yes		aural [p. 79]
'volume' [p. 278]	<number> | <percent-age> | silent | x-soft | soft | medium | loud | x-loud | inherit	medium		yes	refer to inherited value	aural [p. 79]
'white-space' [p. 243]	normal | pre | nowrap | inherit	normal	block-level elements	yes		visual [p. 79]
'widows' [p. 183]	<integer> | inherit	2	block-level elements	yes		visual [p. 79] , paged [p. 79]
'width' [p. 133]	<length> | <percentage> | auto | inherit	auto	all elements but non-replaced inline elements, table rows, and row groups	no	refer to width of containing block	visual [p. 79]
'word-spacing' [p. 242]	normal | <length> | inherit	normal		yes		visual [p. 79]
'z-index' [p. 126]	auto | <integer> | inherit	auto	positioned elements	no		visual [p. 79]

Appendix G. Descriptor index

Name	Values	Initial value
’ascent’ [p. 222]	<number>	undefined
’baseline’ [p. 224]	<number>	0
’bbox’ [p. 223]	<number>, <number>, <number>, <number>	undefined
’cap-height’ [p. 221]	<number>	undefined
’centerline’ [p. 224]	<number>	undefined
’definition-src’ [p. 223]	<uri>	undefined
’descent’ [p. 222]	<number>	undefined
’font-family’ [p. 215]	[<family-name> | <generic-family>] [, [<family-name> | <generic-family>]]*	depends on user agent
’font-size’ [p. 216]	all | <length> [, <length>]*	all
’font-stretch’ [p. 216]	all | [normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded] [, [normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded]]*	normal
’font-style’ [p. 215]	all | [normal | italic | oblique] [, [normal | italic | oblique]]*	all
’font-variant’ [p. 215]	[normal | small-caps] [,[normal | small-caps]]*	normal
’font-weight’ [p. 215]	all | [normal | bold | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900] [, [normal | bold | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900]]*	all

Name	Values	Initial value
’mathline’ [p. 224]	<number>	undefined
’panose-1’ [p. 220]	[<integer>]{10}	0 0 0 0 0 0 0 0 0 0
’slope’ [p. 221]	<number>	0
’src’ [p. 218]	[<uri> [format(<string> [, <string>]*)] | <font-face-name>] [, <uri> [format(<string> [, <string>]*)] | <font-face-name>]*	undefined
’stemh’ [p. 221]	<number>	undefined
’stemv’ [p. 221]	<number>	undefined
’topline’ [p. 224]	<number>	undefined
’unicode-range’ [p. 217]	<urange> [, <urange>]*	U+0-7FFFFFFF
’units-per-em’ [p. 218]	<number>	undefined
’widths’ [p. 222]	[<urange>]? [<number>]+ [,[<urange>]? <number>]+]	undefined
’x-height’ [p. 221]	<number>	undefined

Appendix H. Index