

Foundational CSS3 Components

Excerpt from
CSS: The Definitive Guide,
4th Edition

CSS and Documents



O'REILLY®

Eric A. Meyer

www.it-ebooks.info

CSS and Documents

Eric A. Meyer

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

www.it-ebooks.info

CSS and Documents

by Eric A. Meyer

Copyright © 2012 O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Simon St. Laurent and Meghan Blanchette

Cover Designer: Karen Montgomery

Production Editor: Kristen Borg

Interior Designer: David Futato

Copyeditor: Rachel Leach

Illustrator: Robert Romano

Proofreader: O'Reilly Production Services

Revision History for the First Edition:

2012-09-25 First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449342470> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *CSS and Documents*, the image of a salmon, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-34247-0

[LSI]

1348245482

Table of Contents

| | |
|-----------------------------------|----------|
| Preface | v |
| CSS and Documents | 1 |
| A Brief History of (Web) Style | 1 |
| Elements | 2 |
| Replaced and Nonreplaced Elements | 3 |
| Element Display Roles | 3 |
| Bringing CSS and HTML Together | 6 |
| The link Tag | 7 |
| The style Element | 11 |
| The @import Directive | 12 |
| HTTP Linking | 13 |
| Inline Styles | 14 |
| Media Queries | 15 |
| Usage | 15 |
| Media Types | 15 |
| Media Descriptors | 16 |
| Media Feature Descriptors | 18 |
| New Value Types | 20 |
| Style Sheet Contents | 20 |
| Markup | 21 |
| Rule Structure | 21 |
| Whitespace Handling | 22 |
| Media Blocks | 24 |
| CSS Comments | 24 |
| Summary | 25 |

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

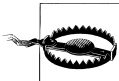
Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples


This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does

require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*CSS and Documents* by Eric A. Meyer (O’Reilly). Copyright 2012 O’Reilly Media, Inc., 978-1-449-34247-0.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

 Safari Books Online (www.safaribooksonline.com) is an on-demand digital library that delivers expert [content](#) in both book and video form from the world’s leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of [product mixes](#) and pricing programs for [organizations](#), [government agencies](#), and [individuals](#). Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O’Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens [more](#). For more information about Safari Books Online, please visit us [online](#).

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://oreil.ly/css-and-documents>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

CSS and Documents

Cascading Style Sheets (CSS) is a powerful tool that transforms the presentation of a document or a collection of documents, and it has spread to nearly every corner of the web as well as into many ostensibly non-web environments. For example, Gecko-based browsers use CSS to affect the presentation of the browser chrome itself, many RSS clients let you apply CSS to feeds and feed entries, and some instant message clients like Adium use CSS to format chat windows. Aspects of CSS can be found in the syntax used by JavaScript frameworks like jQuery. It's everywhere!

A Brief History of (Web) Style

CSS was first proposed in 1994, just as the Web was beginning to really catch on. In fact, the first draft of what would eventually become CSS (titled *Cascading HTML Style Sheets*) was published mere days before the first release of Mozilla (soon to be Netscape Navigator) was announced.

At the time, browsers gave all sorts of styling power to the user—the presentation preferences in Mosaic, for example, permitted all manner of font family, size, and color to be defined by the user on a per-element basis. None of this was available to document authors; all they could do was mark a piece of content as a paragraph, as a heading of some level, as preformatted text, or one of a handful of other element types. If a user configured his browser to make all level-one headings tiny and pink and all level-six headings huge and red, well, that was his lookout.

It was into this milieu that CSS was introduced. Its goal was simple: provide a simple, declarative styling language that was flexible for authors and, most importantly, provided styling power to authors and users alike. By means of the “cascade,” these styles could be combined and prioritized so that both authors and readers had a say—though readers always had the last say.

Work quickly advanced and by late 1996, CSS1 was finished. While the newly established CSS Working Group moved forward with CSS2, browsers struggled to implement CSS1 in an interoperable way. Although each individual piece of CSS was fairly simple on its own, the combination of those pieces created some surprisingly complex

behaviors. There were also some unfortunate missteps in early implementations, such as the now-infamous discrepancy in box model implementations. These problems threatened to derail CSS altogether, but fortunately some clever proposals were implemented, and browsers began to harmonize. Within a few years, thanks to increasing interoperability and high-profile developments such as the CSS-based redesign of *Wired* magazine and the CSS Zen Garden, CSS began to really catch on.

Before all that happened, though, the CSS Working Group had finalized the fairly weighty CSS2 specification in early 1998. Once CSS2 was finished, work immediately began on CSS3 (as well as a clarified version of CSS2 called CSS2.1). In keeping with the spirit of the times, CSS3 was constructed as a series of (theoretically) standalone modules instead of a single monolithic specification. This approach reflected the then-active XHTML specification, which was split into modules for similar reasons.

The rationale for modularizing CSS3 was that each module could be worked on at its own pace, and particularly critical (or popular) modules could be advanced along the W3C's progress track without being held up by others. Indeed, this has turned out to be the case. By early 2012, three CSS3 modules (along with CSS1 and CSS 2.1) had reached full Recommendation status—CSS Color Level 3, CSS Namespaces, and Selectors Level 3. At that same time, seven modules were at Candidate Recommendation status, and several dozen others were in various stages of Working Draft-ness. Under the old approach, colors, selectors, and namespaces would have had to wait for every other part of the specification to be done or cut before they could be part of a completed specification. Thanks to modularization, they didn't have to wait.

The flip side of that advantage is that it's very hard to speak of a single "CSS3 specification." There really isn't any such thing, nor can there be. Even if every other CSS3 module were completed by, say, late 2014, there would already be a Selectors Level 4 in process, and possibly nearing completion. Would we then speak of it as CSS4? What about all the CSS3 features still coming into play?

So while we can't really point to a single tome and say, "There is CSS3," we can talk of features by the module name under which they are introduced. The flexibility modules permit more than makes up for the semantic awkwardness they sometimes create.

Elements

Elements are the basis of document structure. In HTML, the most common elements are easily recognizable, such as `p`, `table`, `span`, `a`, and `div`. Every single element in a document plays a part in its presentation.

Replaced and Nonreplaced Elements

Although CSS depends on elements, not all elements are created equally. For example, images and paragraphs are not the same type of element, nor are `span` and `div`. In CSS, elements generally take two forms: replaced and nonreplaced.

Replaced elements

Replaced elements are those where the element’s content is replaced by something that is not directly represented by document content. Probably the most familiar HTML example is the `img` element, which is replaced by an image file external to the document itself. In fact, `img` has no actual content, as you can see in this simple example:

```

```

This markup fragment contains only an element name and an attribute. The element presents nothing unless you point it to some external content (in this case, an image specified by the `src` attribute). If you point to a valid image file, the image will be placed in the document. If not, it will either display nothing or the browser will show a “broken image” placeholder.

In a like manner, the `input` element is also replaced—by a radio button, checkbox, or text input box, depending on its type.

Nonreplaced elements

The majority of HTML elements are *nonreplaced elements*. This means that their content is presented by the user agent (generally a browser) inside a box generated by the element itself. For example, `hi there` is a nonreplaced element, and the text “hi there” will be displayed by the user agent. This is true of paragraphs, headings, table cells, lists, and almost everything else in HTML.

Element Display Roles

In addition to replaced and nonreplaced elements, CSS2.1 uses two other basic types of elements: *block-level* and *inline-level*. These types will be more familiar to authors who have spent time with HTML markup and its display in web browsers; the elements are illustrated in [Figure 1](#).

h1 (block)

This paragraph (p) element is a block-level element. The strongly emphasized text **is an inline element, and will line-wrap when necessary**. The content outside of inline elements is actually part of the block element. The content inside inline elements *such as this one* belong to the inline element.

Figure 1. Block- and inline-level elements in an HTML document

Block-level elements

Block-level elements generate an element box that (by default) fills its parent element’s content area and cannot have other elements at its sides. In other words, it generates “breaks” before and after the element box. The most familiar block elements from HTML are `p` and `div`. Replaced elements can be block-level elements, but usually they are not.

List items are a special case of block-level elements. In addition to behaving in a manner consistent with other block elements, they generate a marker—typically a bullet for unordered lists and a number for ordered lists—that is “attached” to the element box. Except for the presence of this marker, list items are in all other ways identical to other block elements.

Inline-level elements

Inline-level elements generate an element box within a line of text and do not break up the flow of that line. The best inline element example is the `a` element in HTML. Other candidates are `strong` and `em`. These elements do not generate a “break” before or after themselves, so they can appear within the content of another element without disrupting its display.

Note that while the names “block” and “inline” share a great deal in common with block- and inline-level elements in HTML, there is an important difference. In HTML, block-level elements cannot descend from inline-level elements. In CSS, there is no restriction on how display roles can be nested within each other.

To see how this works, let’s consider a CSS property, `display`:

Values:

none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit

Initial value:

inline

Applies to:

All elements

Inherited:

No

Computed value:

Varies for floated, positioned, and root elements (see CSS2.1, section 9.7); otherwise, as specified

You may have noticed that there are a lot of values, only three of which I’ve even come close to mentioning: `block`, `inline`, and `list-item`.

For the moment, let's just concentrate on **block** and **inline**. Consider the following markup:

```
<body>
<p>This is a paragraph with <em>an inline element</em> within it.</p>
</body>
```

Here we have two block elements (**body** and **p**) and an inline element (**em**). According to the HTML specification, **em** can descend from **p**, but the reverse is not true. Typically, the HTML hierarchy works out so that inlines descend from blocks, but not the other way around.

CSS, on the other hand, has no such restrictions. You can leave the markup as it is but change the display roles of the two elements like this:

```
p {display: inline;}
em {display: block;}
```

This causes the elements to generate a block box inside an inline box. This is perfectly legal and violates no specification in CSS. You would, however, have a problem if you tried to reverse the nesting of the elements in HTML:

```
<em><p>This is a paragraph improperly enclosed by an inline element.</p></em>
```

No matter what you do to the display roles via CSS, this is not legal in HTML.

While changing the display roles of elements can be useful in HTML documents, it becomes downright critical for XML documents. An XML document is unlikely to have any inherent display roles, so it's up to the author to define them. For example, you might wonder how to lay out the following snippet of XML:

```
<book>
  <maintitle>Cascading Style Sheets: The Definitive Guide</maintitle>
  <subtitle>Third Edition</subtitle>
  <author>Eric A. Meyer</author>
  <publisher>O'Reilly and Associates</publisher>
  <pubdate>November 2006</pubdate>
  <isbn type="print">978-0-596-52733-4</isbn>
</book>
<book>
  <maintitle>CSS Pocket Reference</maintitle>
  <subtitle>Third Edition</subtitle>
  <author>Eric A. Meyer</author>
  <publisher>O'Reilly and Associates</publisher>
  <pubdate>October 2007</pubdate>
  <isbn type="print">978-0-596-51505-8</isbn>
</book>
```

Since the default value of **display** is **inline**, the content would be rendered as inline text by default, as illustrated in [Figure 2](#). This isn't a terribly useful display.

You can define the basics of the layout with **display**:

```
book, maintitle, subtitle, author, isbn {display: block;}
publisher, pubdate {display: inline;}
```

Cascading Style Sheets: The Definitive Guide Third Edition Eric A. Meyer O'Reilly and Associates
November 2006 978-0-596-52733-4 CSS Pocket Reference Third Edition Eric A. Meyer O'Reilly and
Associates October 2007 978-0-596-51505-8

Figure 2. Default display of an XML document

You've now set five of the seven elements to be block and two to be inline. This means each of the block elements will be treated much as `div` is treated in HTML, and the two inlines will be treated in a manner similar to `span`.

This fundamental ability to affect display roles makes CSS highly useful in a variety of situations. You could take the preceding rules as a starting point, add a few other styles for greater visual impact, and get the result shown in [Figure 3](#).

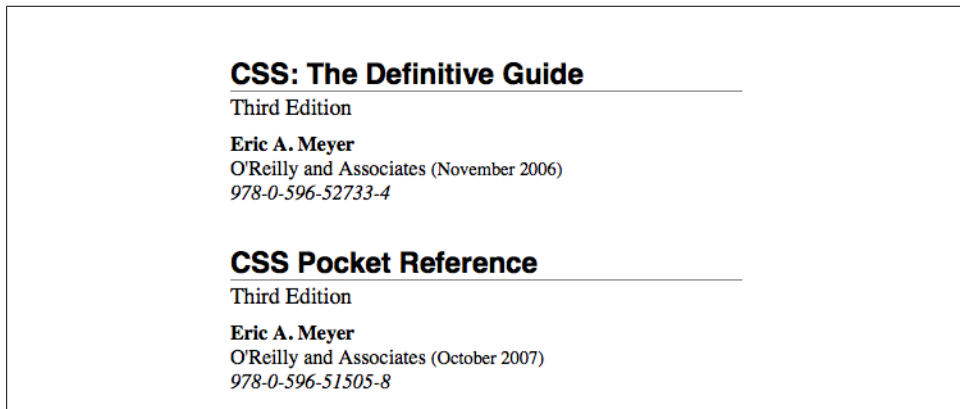


Figure 3. Styled display of an XML document

Before learning how to write CSS in detail, we need to look at how one can associate CSS with a document. After all, without tying the two together, there's no way for the CSS to affect the document. We'll explore this in an HTML setting since it's the most familiar.

Bringing CSS and HTML Together

I've mentioned that HTML documents have an inherent structure, and that's a point worth repeating. In fact, that's part of the problem with web pages of old: too many of us forgot that documents are supposed to have an internal structure, which is altogether different than a visual structure. In our rush to create the coolest-looking pages on the Web, we bent, warped, and generally ignored the idea that pages should contain information with some structural meaning.

That structure is an inherent part of the relationship between HTML and CSS; without it, there couldn't be a relationship at all. To understand it better, let's look at an example HTML document and break it down by pieces:

```

<html>
<head>
<title>Eric's World of Waffles</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all">
<style type="text/css">
/* These are my styles! Yay! */
@import url(sheet2.css);
</style>
</head>
<body>
<h1>Waffles!</h1>
<p style="color: gray;">The most wonderful of all breakfast foods is
the waffle—a ridged and cratered slab of home-cooked, fluffy goodness
that makes every child's heart soar with joy. And they're so easy to make!
Just a simple waffle-maker and some batter, and you're ready for a morning
of aromatic ecstasy!
</p>
</body>
</html>

```

The result of this markup and the applied styles is shown in [Figure 4](#).

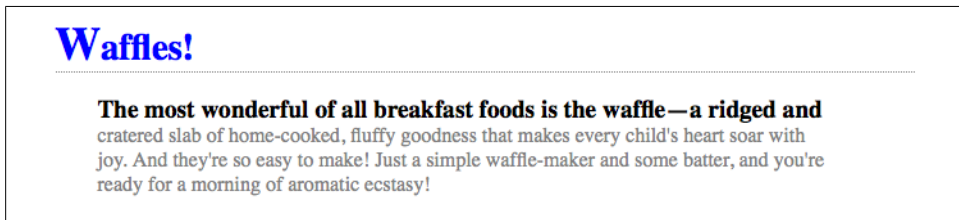


Figure 4. A simple document

Now, let's examine the various ways this document connects to CSS.

The link Tag

First, consider the use of the link tag:

```
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all">
```

The link tag is a little-regarded but nonetheless perfectly valid tag that has been hanging around the HTML specification for years, just waiting to be put to good use. Its basic purpose is to allow HTML authors to associate other documents with the document containing the link tag. CSS uses it to link style sheets to the document; in [Figure 5](#), a style sheet called *sheet1.css* is linked to the document.

These style sheets, which are not part of the HTML document but are still used by it, are referred to as *external style sheets*. This is because they're style sheets that are external to the HTML document. (Go figure.)

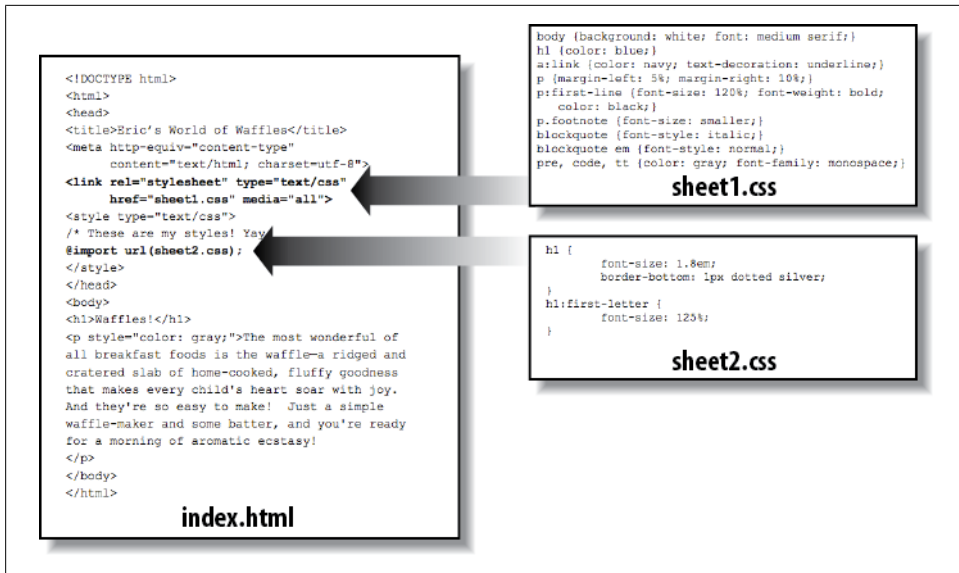


Figure 5. A representation of how external style sheets are applied to documents

To successfully load an external style sheet, `link` must be placed inside the `head` element but may not be placed inside any other element, rather like `title`. This will cause the web browser to locate and load the style sheet and use whatever styles it contains to render the HTML document in the manner shown in Figure 5. Also shown in Figure 5 is the loading of the external *sheet2.css* via the `@import` declaration. Imports must be placed at the beginning of the style sheet that contains them, but they are otherwise unconstrained.

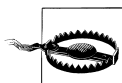
And what is the format of an external style sheet? It's simply a list of rules, just like those we saw in the previous section and in the example HTML document, but in this case, the rules are saved into their own file. Just remember that no HTML or any other markup language can be included in the style sheet—only style rules. Here are the contents of an external style sheet:

```

h1 {color: red;}
h2 {color: maroon; background: white;}
h3 {color: white; background: black;
font: medium Helvetica;}

```

That's all there is to it—no HTML markup or comments at all, just plain-and-simple style declarations. These are saved into a plain-text file and are usually given an extension of *.css*, as in *sheet1.css*.



An external style sheet cannot contain any document markup at all, only CSS rules and CSS comments, both of which are explained later in the chapter. The presence of markup in an external style sheet can cause some or all of it to be ignored.

The filename extension is not required, but some older browsers won't recognize the file as containing a style sheet unless it actually ends with `.css`, even if you *do* include the correct `type` of `text/css` in the `link` element. In fact, some web servers won't hand over a file as `text/css` unless its filename ends with `.css`, though that can usually be fixed by changing the server's configuration files.

Attributes

For the rest of the `link` tag, the attributes and values are fairly straightforward. `rel` stands for "relation," and in this case, the relation is `stylesheet`. The attribute `type` is always set to `text/css`. This value describes the type of data that will be loaded using the `link` tag. That way, the web browser knows that the style sheet is a CSS style sheet, a fact that will determine how the browser deals with the data it imports. After all, there may be other style languages used in the future, so it's important to declare which language you're using.

Next, we find the `href` attribute. The value of this attribute is the URL of your style sheet. This URL can be either absolute or relative, depending on what works for you. In our example, of course, the URL is relative. It just as easily could have been something like `http://meyerweb.com/sheet1.css`.

Finally, we have a `media` attribute. The value of this attribute is one or more *media descriptors*, which are rules regarding media types and the features of those media, with each rule separated by a comma. Thus, for example, you can use a linked style sheet in both screen and projection media:

```
<link rel="stylesheet" type="text/css" href="visual-sheet.css"
      media="screen, projection">
```

Media descriptors can get quite complicated, and are explained in detail later in the chapter. For now, we'll stick with the basic media types shown.

Note that there can be more than one linked style sheet associated with a document. In these cases, only those `link` tags with a `rel` of `stylesheet` will be used in the initial display of the document. Thus, if you wanted to link two style sheets named *basic.css* and *splash.css*, it would look like this:

```
<link rel="stylesheet" type="text/css" href="basic.css">
<link rel="stylesheet" type="text/css" href="splash.css">
```

This will cause the browser to load both style sheets, combine the rules from each, and apply them all to the document. For example:

```
<link rel="stylesheet" type="text/css" href="basic.css">
<link rel="stylesheet" type="text/css" href="splash.css">

<p class="a1">This paragraph will be gray only if styles from the
stylesheet 'basic.css' are applied.</p>
<p class="b1">This paragraph will be gray only if styles from the
stylesheet 'splash.css' are applied.</p>
```

The one attribute that is not in your example markup, but could be, is the `title` attribute. This attribute is not often used, but it could become important in the future and, if used improperly, can have unexpected effects. Why? We will explore that in the next section.

Alternate style sheets

It's also possible to define *alternate style sheets*. These are defined by making the value of the `rel` attribute `alternate stylesheet`, and they are used in document presentation only if selected by the user.

Should a browser be able to use alternate style sheets, it will use the values of the link element's `title` attributes to generate a list of style alternatives. So you could write the following:

```
<link rel="stylesheet" type="text/css"
      href="sheet1.css" title="Default">
<link rel="alternate stylesheet" type="text/css"
      href="bigtext.css" title="Big Text">
<link rel="alternate stylesheet" type="text/css"
      href="zany.css" title="Crazy colors!">
```

Users could then pick the style they want to use, and the browser would switch from the first one, labeled “Default” in this case, to whichever the user picked. [Figure 6](#) shows one way in which this selection mechanism might be accomplished (and in fact was, early in the resurgence of CSS).

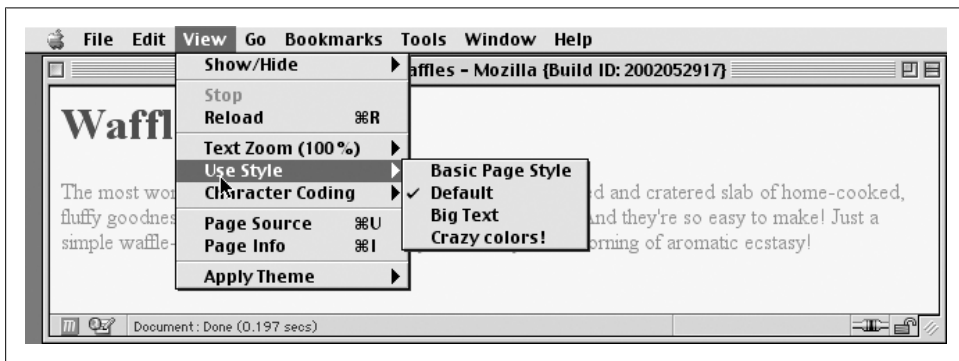


Figure 6. A browser offering alternate style sheet selection



As of early 2012, alternate style sheets were supported in most Gecko-based browsers like Firefox, and in Opera. They could be supported in the Internet Explorer family through the use of JavaScript but are not natively supported by those browsers. The WebKit family did not support selecting alternate style sheets. (Compare this to the age of the browser shown in [Figure 6](#). It's almost shocking.)

It is also possible to group alternate style sheets together by giving them the same `title` value. Thus, you make it possible for the user to pick a different presentation for your site in both screen and print media. For example:

```
<link rel="stylesheet" type="text/css"
      href="sheet1.css" title="Default" media="screen">
<link rel="stylesheet" type="text/css"
      href="print-sheet1.css" title="Default" media="print">
<link rel="alternate stylesheet" type="text/css"
      href="bigtext.css" title="Big Text" media="screen">
<link rel="alternate stylesheet" type="text/css"
      href="print-bigtext.css" title="Big Text" media="print">
```

If a user selects “Big Text” from the alternate style sheet selection mechanism in a conforming user agent, then *bigtext.css* will be used to style the document in the screen medium, and *print-bigtext.css* will be used in the print medium. Neither *sheet1.css* nor *print-sheet1.css* will be used in any medium.

Why is that? Because if you give a link with a `rel` of `stylesheet` a title, then you are designating that style sheet as a *preferred style sheet*. This means that its use is preferred to alternate style sheets, and it will be used when the document is first displayed. Once you select an alternate style sheet, however, the preferred style sheet will *not* be used.

Furthermore, if you designate a number of style sheets as preferred, then all but one of them will be ignored. Consider:

```
<link rel="stylesheet" type="text/css"
      href="sheet1.css" title="Default Layout">
<link rel="stylesheet" type="text/css"
      href="sheet2.css" title="Default Text Sizes">
<link rel="stylesheet" type="text/css"
      href="sheet3.css" title="Default Colors">
```

All three link elements now refer to preferred style sheets, thanks to the presence of a `title` attribute on all three, but only one of them will actually be used in that manner. The other two will be ignored completely. Which two? There’s no way to be certain, as HTML doesn’t provide a method of determining which preferred style sheets should be ignored and which should be used.

If you simply don’t give a style sheet a title, then it becomes a *persistent style sheet* and is always used in the display of the document. Often, this is exactly what an author wants.

The style Element

The style element is one way to include a style sheet, and it appears in the document itself:

```
<style type="text/css">...</style>
```

style should always use the attribute `type`; in the case of a CSS document, the correct value is `"text/css"`, just as it was with the link element.

The `style` element should always start with `<style type="text/css">`, as shown in the preceding example. This is followed by one or more styles and is finished with a closing `</style>` tag. It is also possible to give the `style` element a `media` attribute, which functions in the same manner as previously discussed for linked style sheets.

The styles between the opening and closing `style` tags are referred to as the *document style sheet* or the *embedded style sheet* (because this kind of style sheet is embedded within the document). It will contain many of the styles that will apply to the document, but it can also contain multiple links to external style sheets using the `@import` directive.

The @import Directive

Now we'll discuss the stuff that is found inside the `style` tag. First, we have something very similar to `link`: the `@import` directive:

```
@import url(sheet2.css);
```

Just like `link`, `@import` can be used to direct the web browser to load an external style sheet and use its styles in the rendering of the HTML document. The only major difference is in the actual syntax and placement of the command. As you can see, `@import` is found inside the `style` container. It must be placed before the other CSS rules or it won't work at all. Consider this example:

```
<style type="text/css">
@import url(styles.css); /* @import comes first */
h1 {color: gray;}
</style>
```

Like `link`, there can be more than one `@import` statement in a document. Unlike `link`, however, the style sheets of every `@import` directive will be loaded and used; there is no way to designate alternate style sheets with `@import`. So, given the following markup:

```
@import url(sheet2.css);
@import url(blueworld.css);
@import url(zany.css);
```

...all three external style sheets will be loaded, and all of their style rules will be used in the display of the document.

As with `link`, you can restrict imported style sheets to one or more media by providing media descriptors after the style sheet's URL:

```
@import url(sheet2.css) all;
@import url(blueworld.css) screen;
@import url(zany.css) projection, print;
```

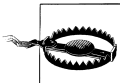
As noted in the section about the `link` element, media descriptors can get quite complicated, and are explained in detail later in the chapter.

`@import` can be highly useful if you have an external style sheet that needs to use the styles found in other external style sheets. Since external style sheets cannot contain any document markup, the `link` element can't be used—but `@import` can. Therefore, you might have an external style sheet that contains the following:

```
@import url(http://example.org/library/layout.css);
@import url(basic-text.css);
@import url(printer.css) print;
body {color: red;}
h1 {color: blue;}
```

Well, maybe not those exact styles, but you get the idea. Note the use of both absolute and relative URLs in the previous example. Either URL form can be used, just as with `link`.

Note also that the `@import` directives appear at the beginning of the style sheet, as they did in our example document. CSS requires the `@import` directive to come before any other rules in a style sheet. An `@import` that comes after other rules (e.g., `body {color: red;}`) will be ignored by conforming user agents.



Older versions of Internet Explorer for Windows do not ignore any `@import` directive, even those that come after other rules. Since other browsers do ignore improperly placed `@import` directives, it is easy to mistakenly place the `@import` directive incorrectly and thus alter the display in other browsers.

HTTP Linking

There is another, far more obscure way to associate CSS with a document: you can link the two together via HTTP headers.

Under Apache, this can be accomplished by adding a reference to the CSS file in a `.htaccess` file. For example:

```
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
```

This will cause supporting browsers to associate the referenced style sheet with any documents served from under that `.htaccess` file. The browser will then treat it as if it were a linked style sheet. Alternatively, and probably more efficiently, you can add an equivalent rule to the server's `httpd.conf` file:

```
<Directory /path/to/ /public/html/directory>
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
</Directory>
```

The effect is exactly the same in supporting browsers. The only difference is in where you declare the linking.

No doubt you noticed the use of the term “supporting browsers.” As of early 2012, the widely used browsers that support HTTP linking of style sheets are the Firefox family and Opera. That restricts this technique mostly to development environments based

on one of those browsers. In that situation, you can use HTTP linking on the test server to mark when you're on the development site as opposed to the public site. It's also an interesting way to hide styles from the WebKit and Internet Explorer families, assuming you have a reason to do so.



There are equivalents to this technique in common scripting languages such as PHP and IIS, both of which allow the author to emit HTTP headers. It's also possible to use such languages to explicitly write a link element into the document based on the server offering up the document. This is a more robust approach in terms of browser support: every browser supports the link element.

Inline Styles

For cases where you want to simply assign a few styles to one individual element, without the need for embedded or external style sheets, employ the HTML attribute `style` to set an *inline style*:

```
<p style="color: gray;">The most wonderful of all breakfast foods is  
the waffle—a ridged and cratered slab of home-cooked, fluffy goodness...  
</p>
```

The `style` attribute can be associated with any HTML tag whatsoever, except for those tags that are found outside of `body` (`head` or `title`, for instance).

The syntax of a `style` attribute is fairly ordinary. In fact, it looks very much like the declarations found in the `style` container, except here the curly braces are replaced by double quotation marks. So `<p style="color: maroon; background: yellow;">` will set the text color to be maroon and the background to be yellow *for that paragraph only*. No other part of the document will be affected by this declaration.

Note that you can only place a declaration block, not an entire style sheet, inside an inline `style` attribute. Therefore, you can't put an `@import` into a `style` attribute, nor can you include any complete rules. The only thing you can put into the value of a `style` attribute is what might go between the curly braces of a rule.

Use of the `style` attribute is not generally recommended. Indeed, it is very unlikely to appear in XML languages other than HTML. Many of the primary advantages of CSS—the ability to organize centralized styles that control an entire document's appearance or the appearance of all documents on a web server—are negated when you place styles into a `style` attribute. In many ways, inline styles are not much better than the `font` tag, although they do have a good deal more flexibility in terms of what visual effects they can apply.

Media Queries

With media queries, an author can define the media environment in which a given style sheet is used by the browser. In the past, this was handled by setting media types via the `media` attribute on the `link` element, on a `style` element, or in the media descriptor of an `@import` or `@media` declaration. Media queries take this concept several steps further by allowing authors to choose style sheets based on the features of a given media type, using what are called media descriptors.

Usage

Media queries can be employed in the following places:

- The `media` attribute of a `link` element.
- The `media` attribute of a `style` element.
- The media descriptor portion of an `@import` declaration.
- The media descriptor portion of an `@media` declaration.

Queries can range from simple media types to complicated combinations of media types and features.

Media Types

The most basic form of media queries are *media types*, which first appeared in CSS2. These are simple labels for different kinds of media. They are:

all

Use in all presentational media.

aural

Use in speech synthesizers, screen readers, and other audio renderings of the document.

braille

Use when rendering the document with a Braille device.

embossed

Use when printing with a Braille printing device.

handheld

Use on handheld devices like personal digital assistants or web-enabled cell phones.

print

Use when printing the document for sighted users and also when displaying a “print preview” of the document.

projection

Use in a projection medium, such as a digital projector used to present a slideshow when delivering a speech.

screen

Use when presenting the document in a screen medium like a desktop computer monitor. All web browsers running on such systems are screen-medium user agents.

tty

Use when delivering the document in a fixed-pitch environment like a teletype printer.

tv

Use when the document is being presented on a television.

The majority of these media types are not supported by any current web browser. The three most widely supported ones are **all**, **screen**, and **print**. As of this writing, some browsers also support **projection**, which allows a document to be presented as a slideshow, whereas several mobile-device browsers support the **handheld** type.

Multiple media types can be specified using a comma-separated list. The following four examples are all equivalent ways of applying a style sheet (or a block of rules) in both screen and projection media.

```
<link type="text/css" href="frobozz.css" media="screen, projection">
<style type="text/css" media="screen, projection">...</style>
@import url(frobozz.css) screen, projection;
@media screen, projection {...}
```

Things get interesting when you add feature-specific descriptors, such as values that describe the resolution or color depth of a given medium, to these media types.

Media Descriptors

The placement of media queries will be very familiar to any author who has ever set a media type on a **link** element or an **@import** declaration. Here are two essentially equivalent ways of applying an external style sheet when rendering the document on a color printer:

```
<link href="print-color.css" type="text/css" media="print and (color)"
rel="stylesheet">

@import url(print-color.css) print and (color);
```

Anywhere a media type can be used, a media query can be used. This means that, following on the examples of the previous section, it is possible to list more than one query in a comma-separated list:

```
<link href="print-color.css" type="text/css"
media="print and (color), projection and (color)" rel="stylesheet">
```

```
@import url(print-color.css) print and (color), projection and (color);
```

In any situation where even one of the media queries evaluates to “true,” the associated style sheet is applied. Thus, given the previous `@import`, `print-color.css` will be applied if rendering to a color printer *or* to a color projection environment. If printing on a black-and-white printer, both queries will evaluate to “false” and `print-color.css` will not be applied to the document. The same holds true in any screen medium, a grayscale projection environment, an aural media environment, and so forth.

Each media descriptor is composed of a media type and one or more listed media features, with each media feature descriptor enclosed in parentheses. If no media type is provided, then it is assumed to be `all`, which makes the following two examples exactly equivalent:

```
@media all and (min-resolution: 96dpi) {...}
```

```
@media (min-resolution: 96dpi) {...}
```

Generally speaking, a media feature descriptor is formatted like a property-value pair in CSS. There are a few differences, most notably that some features can be specified without an accompanying value. Thus, for example, any color-based medium will be matched using `(color)`, whereas any color medium using a 16-bit color depth is matched using `(color: 16)`. In effect, the use of a descriptor without a value is a true/false test for that descriptor: `(color)` means “is this medium in color?”

Multiple feature descriptors can be linked with the `and` logical keyword. In fact, there are two logical keywords in media queries:

`and`

Links together two or more media features in such a way that all of them must be true for the query to be true. For example, `(color) and (orientation: landscape) and (min-device-width: 800px)` means that all three conditions must be satisfied: if the media environment has color, is in landscape orientation, and the device’s display is at least 800 pixels wide, then the style sheet is used.

`not`

Negates the entire query so that if all of the conditions are true, then the style sheet is *not* applied. For example, `not (color) and (orientation: landscape) and (min-device-width: 800px)` means that if the three conditions are satisfied, the statement is negated. Thus, if the media environment has color, is in landscape orientation, and the device’s display is at least 800 pixels wide, then the style sheet is *not* used. In all other cases, it will be used.

Note that the `not` keyword can only be used at the beginning of a media query. It is *not* legal to write something like `(color) and not (min-device-width: 800px)`. In such cases, the query will be ignored. Note also that browsers too old to understand media queries will always skip a style sheet whose media descriptor starts with `not`.

There is no OR keyword for use in media queries. Instead, the commas that separate a list of queries serve the function of an OR—`screen, print` means “apply if the media is screen or print.” Instead of `screen and (max-color: 2) or (monochrome)`, which is invalid and thus ignored, you should write `screen and (max-color: 2), screen and (monochrome)`.

There is one more keyword, `only`, which is designed to create deliberate backwards incompatibility. Yes, really.

`only`

Used to hide a style sheet from browsers too old to understand media queries. For example, to apply a style sheet in all media, but only in those browsers that understand media queries, you write something like `@import url(new.css) only all`. In browsers that do understand media queries, the `only` keyword is ignored and the style sheet is applied. In browsers that do not understand media queries, the `only` keyword creates an apparent media type of `only all`, which is not valid. Thus, the style sheet is not applied in such browsers. Note that the `only` keyword can only be used at the beginning of a media query.

Media Feature Descriptors

So far we’ve seen a number of media feature descriptors in the examples, but not a complete list of the possible descriptors and their values. Let us fix that now!

Note that none of the following values can be negative, and remember that feature descriptors are *always* enclosed in parentheses.

Descriptors: `width`, `min-width`, `max-width`

Values: `<length>`

Refers to the width of the display area of the user agent. In a screen-media web browser, this is the width of the viewport plus any scrollbars. In paged media, this is the width of the page box. Thus, `(min-width: 850px)` applies when the viewport is greater than 850 pixels wide.

Descriptors: `device-width`, `min-device-width`, `max-device-width`

Values: `<length>`

Refers to the width of the complete rendering area of the output device. In screen media, this is the width of the screen. In paged media, this is the width of the page. Thus, `(max-device-width: 1200px)` applies when the device’s output area is less than 1200 pixels wide.

Descriptors: `height`, `min-height`, `max-height`

Values: `<length>`

Refers to the height of the display area of the user agent. In a screen-media web browser, this is the height of the viewport plus any scrollbars. In paged media, this is the height of the page box. Thus, `(height: 567px)` applies when the viewport’s height is precisely 567 pixels tall.

Descriptors: device-height, min-device-height, max-device-height

Values: <length>

Refers to the height of the complete rendering area of the output device. In screen media, this is the height of the screen. In paged media, this is the height of the page. Thus, (max-device-height: 400px) applies when the device's output area is less than 400 pixels tall.

Descriptors: aspect-ratio, min-aspect-ratio, max-aspect-ratio

Values: <ratio>

Refers to the ratio that results from comparing the width media feature to the height media feature (see the definition of <ratio> in the next section). Thus, (min-aspect-ratio: 2/1) applies to any viewport whose width-to-height ratio is at least 2:1.

Descriptors: device-aspect-ratio, min-device-aspect-ratio, max-device-aspect-ratio

Values: <ratio>

Refers to the ratio that results from comparing the device-width media feature to the device-height media feature (see the definition of <ratio> in the next section). Thus, (device-aspect-ratio: 16/9) applies to any output device whose display area width-to-height is exactly 16:9.

Descriptors: color, min-color, max-color

Values: <integer>

Refers to the presence of color-display capability in the output device, with an optional number representing the number of bits used in each color components. Thus, (color) applies to any device with any color depth at all, whereas (min-color: 4) means there must be at least four bits used per color component. Any device that does not support color will return 0.

Descriptors: color-index, min-color-index, max-color-index

Values: <integer>

Refers to the total number of colors available in the output device's color lookup table. Any device that does not use a color lookup table will return 0. Thus, (min-color-index: 256) applies to any device with a minimum of 256 colors available.

Descriptors: monochrome, min-monochrome, max-monochrome

Values: <integer>

Refers to the presence of a monochrome display, with an optional number of bits-per-pixel in the output device's frame buffer. Any device that is not monochrome will return 0. Thus, (monochrome) applies to any monochrome output device, whereas (min-monochrome: 2) means any monochrome output device with a minimum of two bits per pixel in the frame buffer.

Descriptors: resolution, min-resolution, max-resolution

Values: <resolution>

Refers to the resolution of the output device in terms of pixel density, measured in either dots-per-inch (dpi) or dots-per-centimeter (dpcm); see the definition of

<resolution> in the next section for details. If an output device has pixels that are not square, then the least dense axis is used; for example, if a device is 100dpcm along one axis and 120dpcm along the other, then **100** is the value returned. Additionally, in such non-square cases, a bare **resolution** feature query can never match (though **min-resolution** and **max-resolution** can).

Descriptor: orientation

Values: portrait | landscape

Refers to the output device's total output area, where **portrait** is returned if the media feature **height** is equal to or greater than the media feature **width**. Otherwise, the result is **landscape**.

Descriptor: scan

Values: progressive | interlace

Refers to the scanning process used in an output device with a media type of **tv**.

Descriptor: grid

Values: 0 | 1

Refers to the presence (or absence) of a grid-based output device, such as a **tty** terminal. A grid-based device will return **1**; otherwise, **0** is returned.

New Value Types

There are two new value types introduced by media queries that (as of early 2012) are not used in any other context. These types are used in conjunction with specific media features, which are explained in the next section.

<ratio>

A ratio value is two positive <integer> values separated by a solidus (/) and optional whitespace. The first value refers to the width, and the second to the height. Thus, to express a height-to-width ratio of 16:9, you can write **16/9** or **16 / 9**. As of this writing, there is no facility to express a ratio as a single real number, or to use a colon separator instead of a solidus.

<resolution>

A resolution value is a positive <integer> followed by either of the unit identifiers **dpi** (dots per inch) or **dpcm** (dots per centimeter). In CSS terms, a “dot” is any display unit, the most familiar of which is the pixel. As usual, whitespace is not permitted between the <integer> and the identifier. Therefore, a display whose display has exactly 150 pixels (dots) per inch is matched with **150dpi**.

Style Sheet Contents

So after all of that, what about the actual contents of the style sheets? You know, stuff like this:

```
h1 {color: maroon;}
body {background: yellow;}
```

Styles such as these comprise the bulk of any embedded style sheet—simple and complex, short and long. Rarely will you have a document where the `style` element does not contain any rules, although it's possible to have a simple list of `@import` declarations with no actual rules like those shown above.

Before we get going on the rest of the book, though, there are a few top-level things to cover regarding what can or can't go into a style sheet.

Markup

There is no markup in style sheets. This might seem obvious, but you'd be surprised. The one exception is HTML comment markup, which is permitted inside `style` elements for historical reasons:

```
<style type="text/css"><!--  
h1 {color: maroon;}  
body {background: yellow;}  
--></style>
```

That's it.

Rule Structure

To illustrate the concept of rules in more detail, let's break down the structure.

Each rule has two fundamental parts: the *selector* and the *declaration block*. The declaration block is composed of one or more *declarations*, and each declaration is a pairing of a *property* and a *value*. Every style sheet is made up of a series of rules. Figure 7 shows the parts of a rule.

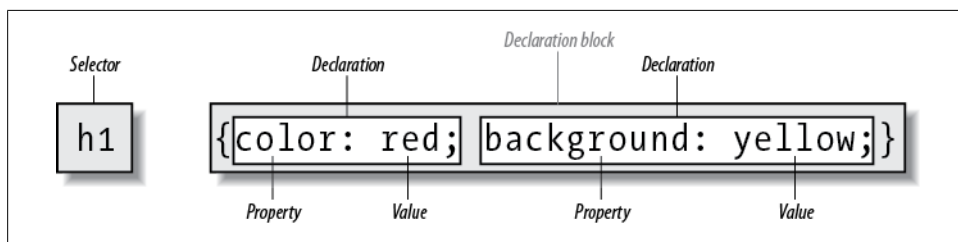


Figure 7. The structure of a rule

The selector, shown on the left side of the rule, defines which piece of the document will be affected. In Figure 7, `h1` elements are selected. If the selector were `p`, then all `p` (paragraph) elements would be selected.

The right side of the rule contains the declaration block, which is made up of one or more declarations. Each declaration is a combination of a CSS property and a value of that property. In Figure 7, the declaration block contains two declarations. The first states that this rule will cause parts of the document to have a **color** of red, and the

second states that part of the document will have a **background** of **yellow**. So, all of the **h1** elements in the document (defined by the selector) will be styled in red text with a yellow background.

Vendor prefixing

Sometimes you'll see pieces of CSS with dashes and labels in front of them, like this: `-o-border-image`. These are called *vendor prefixes*, and are a way for browser vendors to mark properties, values, or other bits of CSS as being experimental or proprietary (or both). As of mid-2012, there were quite a few vendor prefixes in the wild, with the most common being shown in [Table 1](#).

Table 1. Some common vendor prefixes

| Prefix | Vendor |
|----------|--|
| -epub- | International Digital Publishing Forum ePub format |
| -moz- | Mozilla-based browsers (e.g., Firefox) |
| -ms- | Microsoft Internet Explorer |
| -o- | Opera-based browsers |
| -webkit- | WebKit-based browsers (e.g., Safari and Chrome) |

As [Table 1](#) implies, the generally accepted format of a vendor prefix is a dash, a label, and a dash, although a few prefixes erroneously omit the first dash.

The uses and abuses of vendor prefixes are long, tortuous, and beyond the scope of this book. Suffice to say that they started out as a way for vendors to test out new features, thus helping speed interoperability without worrying about being locked into legacy behaviors that were incompatible with other browsers. This avoided a whole class of problems that nearly strangled CSS in its infancy. Unfortunately, prefixed properties were then publicly deployed by web authors and ended up causing a whole new class of problems. As of this writing, the future of vendor prefixes is in serious doubt; it is possible that within a few years they will have been abandoned.

Please always remember that if you see prefixed CSS in the wild, treat whatever you find with caution and test across multiple browsers—each, possibly, with its own prefix—before using what you have found in your own designs.

Whitespace Handling

CSS is basically insensitive to whitespace between rules, and largely insensitive to whitespace within rules, although there are a few exceptions.

In general, CSS treats whitespace just like HTML does: any sequence of whitespace characters is collapsed to a single space for parsing purposes. Thus, you can format the hypothetical `rainbow` rule in the following ways:

```
rainbow:infrared red orange yellow green blue indigo violet ultraviolet;
rainbow:
  infrared red orange yellow green blue indigo violet ultraviolet;
rainbow:
  infrared
  red
  orange
  yellow
  green
  blue
  indigo
  violet
  ultraviolet
;
```

...as well as any other separation patterns you can think up. The only restriction is that the separating characters be whitespace: an empty space, a tab, or a newline, alone or in combination, as many as you like.

Similarly, you can format series of rules with whitespace in any fashion you like. These are just five of an effectively infinite number of possibilities:

```
html{color:black;}
body {background: white;}
p {
  color: gray;}
h2 {
  color : silver ;
}
ol
{
  color
  :
  silver
  ;
}
```

As you can see from the first rule, whitespace can be largely omitted. Indeed, this is usually the case with “minified” CSS, which is CSS that’s had every last possible bit of extraneous whitespace removed. The rules after the first two use progressively more extravagant amounts of whitespace until, in the last rule, pretty much everything that can be separated onto its own line has been.

Any of these approaches are valid, so you should pick the formatting that makes the most sense—that is, is easiest to read—in your eyes, and stick with it.

There are some places where the presence of whitespace is actually required. The most common example is when separating a list of keywords in a value, as in the hypothetical rainbow examples. Those must always be whitespace-separated.

Media Blocks

In cases where you want to embed media-specific rules into a style sheet (as opposed to making entire style sheets media-specific using the `media` attribute or a media-described `@import` declaration), you can use an `@media` block. It looks like this:

```
h1 {color: maroon;}
@media projection {
  body {background: yellow;}
}
```

In this example, `h1` elements will be colored maroon in all media, but the `body` element will get a yellow background only in a projection medium.

You can have as many `@media` blocks as you like in a given style sheet, each with its own set of media descriptors (see later in this chapter for details). You could even encapsulate all of your rules in an `@media` block if you chose:

```
@media all {
  h1 {color: maroon;}
  body {background: yellow;}
}
```

However, since this is exactly the same as if you stripped off the first and last line shown, there isn't a whole lot of point to doing so.



The indentation shown in this section was solely for purposes of clarity. You do not have to indent the rules found inside an `@media` block, but you're welcome to do so if it makes your CSS easier for you to read.

CSS Comments

CSS does allow for comments. These are very similar to C/C++ comments in that they are surrounded by `/*` and `*/`:

```
/* This is a CSS1 comment */
```

Comments can span multiple lines, just as in C++:

```
/* This is a CSS1 comment, and it
can be several lines long without
any problem whatsoever. */
```

It's important to remember that CSS comments cannot be nested. So, for example, this would not be correct:

```
/* This is a comment, in which we find
another comment, which is WRONG
  /* Another comment */
and back to the first comment */
```

Of course, this is hardly ever desirable to nest comments, so this limitation is no big deal.



One way to create “nested” comments accidentally is to temporarily comment out a large block of a style sheet that already contains a comment. Since CSS doesn’t permit nested comments, the “outside” comment will end where the “inside” comment ends.

Unfortunately, there is no “rest of the line” comment pattern such as `//` or `#` (the latter of which is reserved for ID selectors anyway). The only comment pattern in CSS is `/* */`. Therefore, if you wish to place comments on the same line as markup, then you need to be careful about how you place them. For example, this is the correct way to do it:

```
h1 {color: gray;} /* This CSS comment is several lines */
h2 {color: silver;} /* long, but since it is alongside */
p {color: white;} /* actual styles, each line needs to */
pre {color: gray;} /* be wrapped in comment markers. */
```

Given this example, if each line isn’t marked off, then most of the style sheet will become part of the comment and thus will not work:

```
h1 {color: gray;} /* This CSS comment is several lines
h2 {color: silver;} long, but since it is not wrapped
p {color: white;} in comment markers, the last three
pre {color: gray;} styles are part of the comment. */
```

In this example, only the first rule (`h1 {color: gray;}`) will be applied to the document. The rest of the rules, as part of the comment, are ignored by the browser’s rendering engine.



CSS comments are treated by the CSS parser as if they do not exist at all, and so do not count as whitespace for parsing purposes. This means you can put them into the middle of rules—even right inside declarations!

Summary

With CSS, it is possible to completely change the way elements are presented by a user agent. This can be executed at a basic level with the `display` property, and in a different way by associating style sheets with a document. The user will never know whether this is done via an external or embedded style sheet, or even with an inline style. The real importance of external style sheets is the way in which they allow authors to put all of a site’s presentation information in one place, and point all of the documents to that place. This not only makes site updates and maintenance a breeze, but it helps to save bandwidth since all of the presentation is removed from documents.

To make the most of the power of CSS, authors need to know how to associate a set of styles with the elements in a document. To fully understand how CSS can do all of this, authors need a firm grasp of the way CSS selects pieces of a document for styling, which is the subject of another title, “Selectors, Specificity, and the Cascade.”

About the Author

Eric A. Meyer has been working with the Web since late 1993 and is an internationally recognized expert on the subjects of HTML, CSS, and web standards. A widely read author, he is a past member of the CSS&FP Working Group and was the primary creator of the W3C's CSS1 Test Suite. In 2006, Eric was inducted into the International Academy of Digital Arts and Sciences for “international recognition on the topics of HTML and CSS” and helping to “inform excellence and efficiency on the Web.”

Eric is currently the principal founder at Complex Spiral Consulting, which counts among its clients a wide variety of corporations, educational institutions, and government agencies. He is also, along with Jeffrey Zeldman, co-founder of An Event Apart (“The design conference for people who make websites”), and he speaks regularly at that conference as well as many others. Eric lives with his family in Cleveland, Ohio, which is a much nicer city than you've been led to believe. A historian by training and inclination, he enjoys a good meal whenever he can and considers almost every form of music to be worthwhile.